

BASE 3.0.1 Documentation

BASE 3.0.1 Documentation

Last modified:

<lastmodified></lastmodified>

Version:

3.0.1 (build #5859)

Table of Contents

I. Overview	1
1. Why use BASE	2
1.1. Case I: The SCAN-B BASE installation at Department of Oncology, Lund Uni- versity	2
1.2. Case II: The BASE installation at SCIBLU, Department of Oncology, Lund Uni- versity	3
2. BASE features	4
2.1. Web interface	4
2.2. Information and annotation management	4
2.3. Data sharing and privacy	4
2.4. File and directory structure	5
2.5. Plugin and extension infrastructure	5
2.6. Batch upload and download of data	5
2.7. Supported array platforms and raw data formats	6
2.7.1. Vendor specific and custom printing array platforms	6
2.7.2. Available raw data types	7
2.8. Supported sequencing applications	8
2.9. Repository and standards	8
3. Resources	9
3.1. BASE project site	9
3.1.1. Download	9
3.1.2. Ticket system	9
3.1.3. Roadmap	9
3.1.4. Documentation	10
3.2. BASE plug-ins site	10
3.3. Demo server	10
3.4. Mailing lists	10
3.5. BASE-hacks	11
II. User documentation	12
4. Overview of user documentation	13
4.1. Working environment	13
4.2. Start working with BASE	13
4.2.1. Administrative tasks	14
4.2.2. User tasks	14
5. Using the web client	16
5.1. Introduction	16
5.1.1. Logging in	16
5.1.2. Forgotten password	16
5.1.3. The home page	16
5.1.4. Using the menu bar	18
5.1.5. Getting help	18
5.2. Configuring your account	19
5.2.1. Contact information	19
5.2.2. Other information	19
5.2.3. Changing password	20
5.2.4. Preferences	20
5.3. Working with items	22
5.3.1. Create a new item	23
5.3.2. Edit an existing item	23
5.3.3. Delete items	24
5.3.4. Restore deleted items	24
5.3.5. Share items to other users	24
5.3.6. Change owner of items	27
5.4. Listing items	27
5.4.1. Ordering the list	28

5.4.2. Filtering the list	29
5.4.3. Configuring which columns to show	30
5.4.4. Presets	32
5.5. Trashcan	34
5.5.1. Delete items permanently	35
5.5.2. View dependencies of a trashed item	35
5.6. Item overview	36
5.6.1. Validation options	37
5.6.2. Fixing validation failures	39
6. Projects and the permission system	40
6.1. The permission system	40
6.1.1. Permission levels	40
6.1.2. Getting access to an item	41
6.1.3. Plug-in permissions	41
6.2. Projects	41
6.2.1. Creating a project	42
6.2.2. The active project	43
6.2.3. How to give other users access to your project	44
6.2.4. Default items	46
6.2.5. Working with the items in the project	47
6.3. Permission templates	49
7. File management	50
7.1. File system	50
7.1.1. Disk space quota	51
7.2. Handling files	51
7.2.1. Upload a new file	51
7.2.2. External files	54
7.2.3. File servers	55
7.2.4. Edit a file	56
7.2.5. Move files	57
7.2.6. Viewing and downloading files	58
7.2.7. Directories	59
8. Jobs	61
9. Reporters	65
9.1. Reporter types	65
9.2. Reporters	66
9.2.1. Import/update reporter from files	66
9.2.2. Manual management of reporters	66
9.2.3. Deleting reporters	69
9.3. Reporter lists	69
9.3.1. Merging reporter lists	71
10. Annotations	73
10.1. Annotation Types	73
10.1.1. Properties	74
10.1.2. Options	75
10.1.3. Item types	77
10.1.4. Units	78
10.1.5. Categories	79
10.2. Annotating items	80
10.2.1. Inheriting annotations from other items	82
10.2.2. Mass annotation import plug-in	84
11. Experimental platforms and data file types	85
11.1. Platforms	85
11.2. Platform variants	88
11.3. Data file types	88
11.4. Selecting files for an item	90
12. Item subtypes	92
12.1. Item subtype properties	93

12.2. File types	94
13. Protocols	96
13.1. Protocol parameters	97
14. Hardware and software	99
14.1. Hardware	99
14.2. Software	100
15. Array LIMS	101
15.1. Array designs	101
15.1.1. Properties	102
15.1.2. Importing features to an array design	103
15.2. Array batches	103
15.3. Array slides	104
15.3.1. Creating array slides	104
15.3.2. Multiple slides wizard	106
16. Biomaterial LIMS	109
16.1. Biosources	109
16.2. Samples	110
16.2.1. Create sample	111
16.2.2. Sample properties	112
16.2.3. Sample parents	114
16.3. Extracts	115
16.3.1. Create extract	115
16.3.2. Extract properties	116
16.3.3. Extract parents	118
16.4. Tags	119
16.5. Bioplates	119
16.5.1. Bioplate properties	120
16.5.2. Biowells	121
16.5.3. Bioplate types	122
16.5.4. Bioplate events	123
16.6. Biomaterial lists	132
16.7. Physical bioassays	132
16.7.1. Create physical bioassays	132
16.7.2. Bioassay properties	133
16.7.3. Parent extracts	135
17. Experiments and analysis	136
17.1. Derived bioassays	136
17.1.1. Create derived bioassays	136
17.1.2. Derived bioassay properties	137
17.2. Raw bioassays	138
17.2.1. Create raw bioassays	138
17.2.2. Raw bioassay properties	139
17.2.3. Import raw data	140
17.2.4. Raw data types	141
17.2.5. Spot images	141
17.3. Experiments	143
17.3.1. Experiment properties	144
17.3.2. Experimental factors	147
17.4. Analysing data within BASE	148
17.4.1. Transformations and bioassay sets	148
17.4.2. Filtering data	148
17.4.3. Normalizing data	148
17.4.4. Other analysis plug-ins	148
17.4.5. The plot tool	148
17.4.6. Experiment explorer	149
18. Import of data	150
18.1. General import procedure	150
18.1.1. Select plug-in and file format	150

18.1.2. Specify plug-in parameters	153
18.1.3. Add the import job to the job queue	154
18.2. Batch import of data	155
18.2.1. File format	155
18.2.2. Running the item batch importer	156
18.2.3. Comments on the item batch importers	157
19. Export of data	159
19.1. Select plug-in and configuration	159
19.2. Specify plug-in parameters	159
19.3. The table exporter plug-in	161
III. Admin documentation	164
20. Installation and upgrade instructions	165
20.1. Upgrade instructions	165
20.2. Installation instructions	167
20.3. Installing job agents	171
20.3.1. BASE application server side setup	171
20.3.2. Database server setup	172
20.3.3. Job agent client setup	172
20.3.4. Configuring the job agent	173
20.4. Server configurations	174
20.4.1. Sending a broadcast message to logged in users	176
21. Plug-ins and extensions	178
21.1. Managing plug-ins and extensions	178
21.1.1. Automatic installation wizard	178
21.1.2. Manual plug-in registration	181
21.1.3. BASE version 1 plug-ins	183
21.1.4. Installing the X-JSP compiler	183
21.1.5. Disable/enable plug-ins and extensions	184
21.1.6. Plug-in permissions	184
21.2. Plug-in configurations	186
21.2.1. Configuring plug-in configurations	187
21.2.2. Importing and exporting plug-in configurations	189
21.2.3. The Test with file function	189
22. Account administration	194
22.1. Users administration	194
22.1.1. Edit user	194
22.1.2. Default group and role membership	201
22.2. Groups administration	201
22.2.1. Edit group	202
22.3. Roles administration	204
22.3.1. Pre-defined system roles	204
22.3.2. Edit role	205
22.4. Disk space/quota	208
22.4.1. Edit quota	209
22.4.2. Disk usage	211
IV. Developer documentation	212
23. Migrating code from BASE 2 to BASE 3	213
23.1. Compiling the code against BASE 3	213
23.2. Core API changes	213
23.3. Packaging your plug-in so that it installs in BASE 3	214
24. Developer overview of BASE	216
24.1. Fixed vs. dynamic database	217
24.2. Hibernate and the DbEngine	218
24.3. The Batch API	218
24.4. Data classes vs. item classes	219
24.5. The Query API	219
24.6. The Controller API	220
24.7. The Extensions API	220

24.8. Plug-ins	220
24.9. Client applications	221
25. Plug-in developer	222
25.1. How to organize your plug-in project	222
25.1.1. Using Ant	222
25.1.2. Make the plug-in compatible with the auto-installation wizard	224
25.2. The Plug-in API	225
25.2.1. The main plug-in interfaces	225
25.2.2. How the BASE core interacts with the plug-in when... ..	235
25.2.3. Abort a running a plug-in	237
25.2.4. Using custom JSP pages for parameter input	239
25.3. Import plug-ins	240
25.3.1. Autodetect file formats	240
25.3.2. The AbstractFlatFileImporter superclass	242
25.4. Export plug-ins	247
25.4.1. Immediate download of exported data	247
25.4.2. The AbstractExporterPlugin class	248
25.5. Analysis plug-ins	249
25.5.1. The AbstractAnalysisPlugin class	252
25.5.2. The AnalysisFilterPlugin interface	253
25.6. Other plug-ins	253
25.6.1. Authentication plug-ins	253
25.6.2. Secondary file storage plug-ins	255
25.6.3. File unpacker plug-ins	257
25.6.4. File packer plug-ins	258
25.6.5. Logging plug-ins	259
25.7. How BASE load plug-in classes	261
25.8. Example plug-ins (with download)	263
26. Extensions developer	264
26.1. Overview	264
26.1.1. Download code examples	264
26.1.2. Terminology	264
26.2. Hello world as an extension	265
26.2.1. Extending multiple extension points with a single extension	267
26.3. Custom action factories	267
26.4. Custom images, JSP files, and other resources	270
26.4.1. Javascript and stylesheets	271
26.4.2. X-JSP files	272
26.5. Custom renderers and renderer factories	273
26.6. Extension points	274
26.6.1. Error handlers	276
26.7. Custom servlets	277
26.8. Extension points defined by BASE	279
26.8.1. Menu: extensions	279
26.8.2. Toolbars	279
26.8.3. Edit dialogs	279
26.8.4. Bioassay set: Tools	280
26.8.5. Bioassay set: Overview plots	280
26.8.6. Services	280
26.8.7. Connection managers	280
26.8.8. Fileset validators	281
27. Web services	282
27.1. Available services	282
27.1.1. Services	282
27.2. Client development	283
27.2.1. Receiving files	283
27.3. Services development	285
27.3.1. Generate WSDL-files	286

27.4. Example web service client (with download)	286
28. The BASE API	287
28.1. The Public API of BASE	287
28.1.1. What is backwards compatibility?	287
28.2. The Data Layer API	288
28.2.1. Basic classes and interfaces	290
28.2.2. User authentication and access control	293
28.2.3. Reporters	296
28.2.4. Quota and disk usage	298
28.2.5. Client, session and settings	299
28.2.6. Files and directories	301
28.2.7. Experimental platforms and item subtypes	304
28.2.8. Parameters	306
28.2.9. Annotations	308
28.2.10. Protocols, hardware and software	311
28.2.11. Plug-ins, jobs and job agents	312
28.2.12. Biomaterial LIMS	316
28.2.13. Array LIMS - plates	319
28.2.14. Array LIMS - arrays	321
28.2.15. Bioassays and raw data	323
28.2.16. Experiments and analysis	325
28.2.17. Other classes	329
28.3. The Core API	330
28.3.1. Authentication and sessions	330
28.3.2. Access permissions	330
28.3.3. Data validation	330
28.3.4. Transaction handling	330
28.3.5. Create/read/write/delete operations	330
28.3.6. Batch operations	330
28.3.7. Quota	330
28.3.8. Plugin execution / job queue	330
28.3.9. Using files to store data	330
28.3.10. Sending signals (to plug-ins)	336
28.4. The Query API	338
28.5. The Dynamic API	338
28.6. The Extensions API	338
28.6.1. The core part	338
28.6.2. The web client part	344
28.7. Other useful classes and methods	347
29. Write documentation	348
29.1. User, administrator and developer documentation with Docbook	348
29.1.1. Documentation layout	348
29.1.2. Getting started	348
29.1.3. Docbook tags to use	353
29.2. Create UML diagrams with MagicDraw	358
29.2.1. Organisation	358
29.2.2. Classes	359
29.2.3. Diagrams	364
29.3. Javadoc	364
29.3.1. Writing Javadoc	365
30. Core developer reference	367
30.1. Publishing a new release	367
30.2. Subversion / building BASE	367
30.3. Coding rules and guidelines	367
30.3.1. Development process and other important procedures	367
30.3.2. General coding style guidelines	368
30.3.3. API changes and backwards compatibility	374
30.3.4. Data-layer rules	375

30.3.5. Item-class rules	389
30.3.6. Batch-class rules	407
30.3.7. Test-class rules	407
V. FAQ	408
31. Frequently Asked Questions with answers	409
31.1. Reporter related questions with answers	409
31.2. Array design related questions with answers	409
31.3. Biomaterial, Protocol, Hardware, Software related questions with answers	410
31.4. Data Files and Raw Data related questions with answers	411
31.5. Analysis related questions with answers	412
VI. Appendix	413
A. Core plug-ins shipped with BASE	414
A.1. Core analysis plug-ins	414
A.2. Core export plug-ins	415
A.3. Core import plug-ins	415
A.3.1. Core batch import plug-ins	417
A.4. Core intensity plug-ins	417
A.5. Uncategorized core plug-ins	417
B. base.config reference	418
C. extended-properties.xml reference	426
D. Platforms and raw-data-types.xml reference	430
D.1. Default platforms and variants installed with BASE	430
D.2. raw-data-types.xml reference	430
E. web.xml reference	434
F. jobagent.properties reference	436
G. jobagent.sh reference	440
H. Other configuration files	442
H.1. mysql-queries.xml and postgres-queries.xml	442
H.2. log4j.properties	442
H.3. hibernate.cfg.xml	442
H.4. ehcache.xml	442
I. API changes that may affect backwards compatibility	443
I.1. BASE 3.0 release	443
I.2. All BASE 2.x releases	443
J. Things to consider when updating an existing BASE installation	444
J.1. All BASE 2.x releases	444
K. File formats	445
K.1. The BFS (BASE File Set) format	445
K.1.1. The basics of BFS	445
K.1.2. Using BFS for spotdata to and from external plug-ins	447
K.2. The BASEfile format	450
K.2.1. To be done	450

Part I. Overview

BASE is a freely available software solution designed for laboratories looking for a single point of storage for all information related to their experimentation and for storing, managing, and analysing genomics/transcriptomics data. BASE offers a multi-user local data repository featuring a web browser user interface, laboratory information management system (LIMS) for biomaterials and array production, annotations such as clinical information, hierarchical overview of analysis, and integrates tools like MultiExperiment Viewer¹.

BASE is GPLv3 licensed² and can be freely downloaded and installed by anyone. Once installed and configured, the system comprise a database layer and a web server coupled to layers of software for managing and analysing data in the database.

¹ <http://www.tm4.org/mev/>

² <http://www.gnu.org/licenses/gpl-3.0.html>

Chapter 1. Why use BASE

BASE was initially developed to manage array-based data but is now extended to support storage and analysis of sequencing data. The first sequencing application is RNAseq.

We outline two different uses of BASE to give a flavour why you should consider to use BASE. The first example describes a research project involving sequencing based gene expression analysis and the second example describes a microarray service facility use of BASE.

1.1. Case I: The SCAN-B BASE installation at Department of Oncology, Lund University

SCAN-B¹ is a project and network of researchers and clinicians that was initialised in the fall 2009. The project is centred on a prospective study where all new breast cancer patients in southern Sweden are asked to enrol. Within the covered region approximately 1500 patients are diagnosed with breast cancer annually. The overall aim is to continuously collect and analyse the consecutive, population-based, breast cancer cohort. Analyses include generation of gene expression and sequencing data with the ultimate goal to build an infrastructure for future real-time clinical implementation.

SCAN-B uses BASE to store and manage all information related to enrolled patients and collected sample material including clinical information and experimental data. Analysis and execution of standard analysis pipelines for sequencing data will be performed through BASE.

The SCAN-B BASE installation consists of three main parts; first, the hardware on which the system runs; secondly, the BASE software and database, as well as configured analysis plugins; thirdly, an external file system for storage of sequencing data that are referenced from BASE. In addition, maintenance of the hardware and configured database/software is required. The server hardware comprises one main computer and RAIDed hard drive system. It also includes a backup solution configured to backup the entire database 2 times per week. Computational nodes are connected to the main computer and used to run configured analysis plugins in a seamless integrated fashion. Maintenance includes managing the backup-schedule, updating the main BASE software, developing, configuring, and maintaining analysis plugins, and maintaining the underlying database and external storage file systems.

Whereas the BASE software itself is freely available to anyone, a particular BASE installation at a research site is in general not freely accessible. Although BASE can be downloaded and installed on a regular off-the-shelf personal computer with relative ease by anyone, considerable effort and costs are associated with maintaining a BASE installation of the size and scope of the SCAN-B BASE installation. A pristine BASE installation includes generic features and functionality to support a framework of procedures to manage data collection in large projects. Within SCAN-B large effort is spent on defining the required procedures where laboratory work is mirrored in BASE. This implies interplay with adopting the BASE software (the Reggie extension² is an example of adaptation on BASE to specific needs in SCAN-B) and the laboratory work to achieve efficient data collection. To achieve high quality data production, measures for continuous quality assurance and collection of data associated with patients, samples, and laboratory processing must also be implemented.

¹ http://www.med.lu.se/english/klinvetlund/canceromics/consortia/scan_b

² <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.reggie>

1.2. Case II: The BASE installation at SCIBLU, Department of Oncology, Lund University

In the spring of 2004, Lund University created Swegene Centre for Integrative Biology at Lund University (SCIBLU), which comprise the merger of five of the most successful Swegene resource and development platforms into one unit, located in the Lund University Biomedical Centre (BMC). SCIBLU offers integrated service within the main -omics areas. The DNA microarray technology within SCIBLU was initially established in 2000 as a cancer research resource at the department of Oncology and in conjunction with this the development of BASE was initiated.

At SCIBLU a BASE installation is maintained and used as a production installation that manages information surrounding array fabrication (array LIMS) as well as array data generated by the SCIBLU provided services. This particular BASE installation was initially set up in 2003 and to date manage array data from more than 13 000 hybridisation covering a variety of technical platforms such as cDNA, oligo, and BeadChip expression arrays, as well as BAC and oligo aCGH arrays.

The SCIBLU BASE installation consists of two main parts; first, the hardware on which the system runs; secondly, the BASE software and database, as well as configured analysis plugins. Regular maintenance of the hardware and configured database/software is also required. The hardware comprises one main computer and raided hard drive system. It also includes a backup solution configured to backup the entire database 2 times per week. Finally, the hardware includes 2 computational servers connected to the main computer and used to run configured analysis plugins in a seamless integrated fashion. The software used for the SCIBLU BASE installation is freely available from the BASE project site. Maintenance include managing the backup-schedule, updating the main BASE software, updating and managing probe annotations, management of user accounts, configuring and maintaining analysis plugins, and maintaining the underlying database.

Users of the microarray services offered by SCIBLU, e.g., expression analysis or aCGH, are provided access to the SCIBLU production BASE installation as part of the included services. The access comprises user account, access to array LIMS (when in-house produced arrays are utilised), and hard drive space to cover space needed for storing the data generated through the SCIBLU provided service. Additional disk space can be acquired and is associated with an additional cost for the user. Examples of when additional disk space is needed include scenarios where users want to perform extensive data analysis within BASE and decide to store the analysis results within BASE, e.g., many parallel analysis branches or extensive generation of data plots and figures. Other examples include when users want to import data from third party providers (public data repositories or alternative array data providers) to perform meta-analysis with their data generated within SCIBLU.

Chapter 2. BASE features

The BASE application features many components; MIAME compliance, multi-user, data sharing, data access management, array and biomaterial LIMS, multiple array platforms, RNAseq sequencing support, extensibility, configurable plug-ins, annotation customisation, streamlined access to analysis tools, integration of MultiExperiment Viewer (MeV)¹, web services API, and more. To support all components the underlying relational database has grown to become very large and complex, especially since BASE itself works with objects posing additional database tables to keep track of objects stored in a relational database. Thus, rather than trying to describe every feature in detail here, we highlight some of the more important features.

2.1. Web interface

The entire system is accessed through a web-interface over the Internet using a standard web browser, such as Firefox, Safari, Opera, or Internet Explorer. Access privileges to a particular BASE installation are managed by personal accounts through the web-interface. A local administrator creates new user accounts with specific roles and access privileges and has an overall managerial responsibility for an individual BASE installation. With exception for the administrator with global data access, individual users have sole access to and control their inputted data. Users have the possibility to share data they own (or have share credentials for) to other users of the same BASE installation.

2.2. Information and annotation management

BASE features a biomaterial LIMS tracking biological material from its source to hybridisation/sequencing and ultimately to raw data and analysis. All events throughout sample handling are tracked and information on used and remaining quantities, physical sample locations, quality control information, and sample relations is stored in BASE. Racks or boxes holding biomaterials can be created as BioPlates and plate events are easily performed for extraction or labelling events. Although becoming less commonly used, the array production LIMS of previous BASE versions is retained to support researchers with spotting facilities, e.g., protein array production and BAC array printing that may not be commercially available.

Events in biomaterial and array LIMS are annotable with protocols and event dates, and most items can be annotated with customisable annotation types such as floats, integers, dates, and Boolean flags. Change history for biomaterial items is available if configured and can be used to track modifications in the database. Annotations are either free form or from a preset list of values, and can be marked as required for MIAME compliance. The annotation system is searchable and the user can select any annotations to be an experimental factors in analysis whereby it becomes available to analysis plugins and plot-tools.

2.3. Data sharing and privacy

One of the important features of BASE is its capabilities as a local data repository. The repository functionality is amended with data grouping, sharing, and privacy policies. A BASE project is used to group items (biomaterial, raw data, and experiments) into a logical entity, and a BASE experiment is a collection of bioassays, e.g., array data, grouped logically together for further analysis. All items can co-exist in several projects and experiments without any unnecessary copying of information.

Data privacy is guarded by the data owner and BASE allows the owner to set data access rules. To this end, each item in BASE is owned by a user enabling him to share data with colleagues. The

¹ <http://www.tm4.org/mev/>

grouping of data in projects allows the data owner to simply include other users in a project in order to share data. Each item can have different access levels even within a project, and project members can have different privileges. The data access rules are very flexible and can be overwhelming since access levels on almost any item can be individually set. However, using projects, the proper access levels can be set at a single point of interaction.

2.4. File and directory structure

BASE has an integrated file system to provide the possibility for researchers to collect all data files related to a project in one single storage location. Data files are uploaded using a web browser or an ftp client. The file storage is an integral part of a strategy to store all experiment relevant data in BASE, even data types not already supported in analysis. Collecting all data allows future reuse of the data as more data are produced, and new analysis tools becomes available.

2.5. Plugin and extension infrastructure

BASE features a hierarchically organised analysis interface that allows data filtering, normalisation, transformation, and other analyses. Parameters and settings are automatically stored for each step in the analysis. The selection of analysis tools depends on array type and available plug-ins where a wide range of tools are pre-installed with BASE, and optional plug-ins can be downloaded from the BASE plug-in site . BASE capitalise from other software tools, such as MEV, by integrating them into the user interface. Such integration provide streamlined access to analysis modules in external tools. BASE even features a rudimentary manual transform creator that enables researchers to add analysis steps within the hierarchical overview of analysis performed independently of BASE. The transform creator enables storage of result files and parameter information for archival, tracking, and sharing purposes.

The analysis of genomics data is continuously evolving with new methods and techniques. To this end BASE provides extensions and plug-in programming interfaces (APIs) to enable straightforward additions of new analysis tools. The use of the APIs is well documented and there are numerous examples on how to create extensions. The MEV and ftp-server integration all utilise the extension mechanism, and the automatically generated overview plots available in the experimental analysis view are also extensions. The plug-in API is used for all data imports and exports, and most analysis tools, providing new developers a lot of example code to examine when they create BASE plug-ins.

2.6. Batch upload and download of data

File, annotation, and item upload can be done asynchronously as data are generated or information becomes available. To relieve researchers from the tedious task of entering data one by one a set of batch import were created; the information generated throughout the experimental work is uploaded to BASE in plain tab-separated files. These files are supplied to batch importer plug-ins that parse the files and create items and associations according to the information in the files. The same plug-ins can be used to batch update many items. Similarly, annotating items is done by creating tab-separated files with annotation information, uploading these to BASE, and loading the file content into the database using annotation importers. If needed, annotations are easily updated with the same mechanism.

Files uploaded to BASE are stored in the directory structure within BASE and multiple files are easily transferred to BASE either packaged in compressed files with a single upload action, or by using an ftp client supporting transfer of file structures. Similarly, downloading multiple files is straightforward either using an ftp client or by a single click in the BASE web interface. Download of items is done through item listing views enabling users to filter and select what information should be downloaded.

2.7. Supported array platforms and raw data formats

There are many types of microarrays, techniques, and brands available for researchers; one- or two-channel hybridizations, spotted cDNA/oligo arrays, Affymetrix (GeneChip), Illumina (SNP, DASL, WGEX, microRNA), aCGH, SNP, tiling arrays, and many more. In addition expression data can be derived from sequencing data, i.e., RNASeq. Data is produced in different file formats that must be treated differently depending on type.

Many platforms and experimental setups are supported in downstream analysis but some microarray techniques cannot currently be analysed within BASE simply because lack of support in available plug-ins. The problem is resolved by creating new, or extending available, plug-ins that add analysis capabilities of platforms and techniques not readily supported in analysis. Extending analysis capabilities to new technologies is only a matter of local needs and resources. We add support for platforms in use at the Lund University microarray facility and make our tools freely available to the community.

For two channel array platforms it is straightforward to customise BASE for a specific array platform, the platform simply needs to be adapted to the (BASE) Generic platform. The adaptation is to create a raw data format definition and to configure raw data importers, or make use of already available raw data formats. However, it is not always possible to make a natural mapping of a platform to the Generic platform. Platforms such as Affymetrix and Illumina platforms cannot naturally be mapped on to the Generic two channel platform. For Affymetrix, BASE comes with a specific Affymetrix platform and Illumina can be supported by customising BASE (go to the [Illumina package²](#) web site for more information on adding Illumina support to BASE).

How to adapt new array platforms to the Generic platform format or how to create a new platform type in BASE can be read elsewhere in this document. Here we list different array platforms used in BASE and also list raw data types supported by BASE. However, not all platforms nor raw data types listed below are available out-of-the box and a BASE administrator must customise his local BASE installation for their specific need. What comes pre-configured when BASE is installed is indicated in the lists below.

2.7.1. Vendor specific and custom printing array platforms

Not all array platforms listed below are available by default. The comments to specific platforms explain how to enable the use of the array platform in BASE. In some cases there is no confirmed usage of a platform but we believe it has been tested by anonymous users.

Affymetrix

The Affymetrix platform comes pre-configured with a new BASE installation. Affymetrix platform in this context are the Affymetrix expression arrays. So far there has been no reason for expanding the Array platform to other chip-types. In principle any Affymetrix chip type can be stored in BASE but current plug-ins will always assume that expression data is stored and analysed. This can be resolved by adding variants of the Affymetrix platform but the Lund BASE team currently has no plans to create Affymetrix variants.

Agilent

Custom printing

The array layout options are endless and imagination is the only limitation ... almost. BASE can import many in-house array designs and platforms. The custom arrays usually fall back on one of the raw data types already available such as GenePix.

² <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.illumina>

Illumina

There are several variants of the Illumina platform. Using several variants allows BASE to adapt its handling of different Illumina chip types. Illumina platform support is not included in a standard BASE installation but there is a `Illumina` package³ available for seamless integration of the Illumina array platform to BASE.

ImaGene

No successful use confirmed but ImaGene raw data is available in BASE.

Sequencing

Expression data from sequencing experiments. Cufflinks raw-data type is available for expression values from sequencing experiments.

Unlisted

In principle any platform generating a matrix of data can be imported into BASE. Simply utilise the available raw data formats and data importers.

2.7.2. Available raw data types

Raw data comes in many different formats. These formats are usually defined by scanner software vendors and BASE must keep track of the different formats for analysis and plotting. BASE supports many formats out the box, but some formats need to be added manually by the BASE administrator (indicated in the list below).

Affymetrix**AIDA****Agilent****BZScan****ChipSkipper****Cufflinks****GenePix****GeneTAC****Illumina**

The Illumina array platform usage is recommended to be based on the *Illumina Bead Summary (IBS)* raw data format below.

Illumina Bead Summary (IBS)

Not available in BASE directly but it is added with the `Illumina` plug-in⁴ that adds Illumina array platform support to BASE.

ImaGene**QuantArray Biotin****QuantArray Cy****SpotFinder**

³ <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.illumina>

⁴ <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.illumina>

2.8. Supported sequencing applications

BASE was originally developed for management and analysis of array based data. Recent version, starting at version 3, have been adopted to support sequencing based data. Being a newly developed feature it is not as mature as the array part of BASE.

For sequencing data in general, BASE can be used for data management and sharing. BASE currently has extended support for sequencing applications such as RNAseq where data is transformed to gene expression measurements. For such applications array designs can be created based on gene structure defined in GTF formatted files⁵. For example, GTF files for all RefSeqs or known genes.

2.9. Repository and standards

The Microarray Gene Expression Data Society (MGED) develops and maintains standards for data acquisition, representation, and interchange such as the MIAME guidelines, the MAGE-TAB interchange format, and the MGED Ontology for microarray experiments. BASE does not enforce the use of the MGED standards but support storage of information required by MIAME. BASE has an experiment item overview functionality useful for validating information related to experiments. The validation level is user selectable of which the option regarding MIAME compliance is most relevant here. When users or server administrators create annotation types in BASE these annotation values can be marked as required by MIAME and optionally defined to be a list of pre-defined values from a controlled vocabulary. Validation will check for inconsistencies and report errors, and give the user an opportunity to fix issues immediately or later. After resolving the issues raised by the validation, data can be exported for submission to public repositories such as ArrayExpress, Gene Expression Omnibus (GEO), and CIBEX.

⁵ http://en.wikipedia.org/wiki/Gene_transfer_format

Chapter 3. Resources

There are several resources available for those who are using BASE or have some other kind of interested in the BASE project. This chapter contains information about those resources and also some short instructions on how to use each one of them.

3.1. BASE project site

The BASE project site is located at <http://base.thep.lu.se>. Here is a lot of useful information about the project and the program, e.g. documentation/manuals, download-pages, contact information and much more. The most important parts of the page are covered in this section.

3.1.1. Download

The download page is accessed from the download section, on the home page, by following the link to **Download Page**¹. From this page you can download BASE releases as packaged tar.gz files or checkout the source code directly from the Subversion repository. See the separate parts on the web page to get more information how to proceed with each one of them.

Packaged BASE releases

Both source-packages and binary-packages are available for each release of the program.

Repository access

With this option the visitor can get the source code directly from Subversion. There are at least three different version that are available to checkout from the repository.

- The latest production release. This will give you the same source code as one of the packaged releases.
- The latest non-released bugfix branch. Use this if you are affected by a bug that has been fixed but not yet released.
- Bleeding edge of the software, which is the latest revision of the program. The code is not guaranteed to work correctly and it is recommended to backup important data in the database before updating. Use this at your own risk, we cannot guarantee that you will be able to upgrade the installation to another version or release.

3.1.2. Ticket system

A ticket is a note about a bug or a new feature that has not yet been implemented. To show the list of outstanding tickets use the **View Tickets**² button on BASE web site. It is a good idea to have a look at this list before reporting a bug or requesting a new feature. Perhaps someone has registered the issue as a ticket already. This list can also be used to see how the BASE development is proceeding and when some particular request is planned to be fixed.

To report bugs, add feature requests, and comment an existing ticket, you needs to be logged in to the trac environment. This is done by clicking on the **login**-link to the right in the upper corner on the home page. The **Feedback**-section, also on the home page, contains more information how to proceed.

3.1.3. Roadmap

The roadmap of BASE is accessed from the **Roadmap**³ button on the home page. This page contains information about the plans for future development, including the tickets that should be fixed for

³ <http://base.thep.lu.se/roadmap>

each milestone. Usually, only the next upcoming release has a date set. The **BASE Future Release** milestone is used to collected tickets that we think should be fixed but are less important or require too much work. Contributions are welcome.

3.1.4. Documentation

All documentation that are associated with the project can be found in the Documentation-section on the start page.

Manuals

Useful information for the common user and the administrator, like user documentation, installation instructions and administration guide. These different documents will eventually be replaced with this document when it includes the corresponding texts.

Specifications

This part contains specification for the separate divisions of the project, such as core specification, web-client specification and more.

Developer information

Information for those who are interested to be involved in the development of BASE.

Future development

Link to a list of ideas for future development that are not covered and monitored in the milestones on the road map page. In other words - ideas that are not planned to be done within nearest 6 to 12 month.

3.2. BASE plug-ins site

Plug-ins which are not included in the installation of BASE, have their own site, called BASE plug-ins web site⁴ which includes a download page for submitted none-core plug-ins. Here is also information how to become a plug-in developer, with commit access to the repository, and how to submit a plug-in to the download page. You will also be able to find example code for plug-ins, extensions, web services, etc.

3.3. Demo server

There is a demo server up running for anyone who wants to explore BASE without having to install it. Follow the link on BASE web site to the demo server or go directly to <http://base2.thep.lu.se/demo/>⁵

Use **base2** as login and **base2** as password to login to the demo server. The base2 user account has read privileges to all data on the demo server and can view almost every list page. If extended privileges are wanted, please contact the administrator of the demo server (see the bottom of the browser when visiting the demo server).

3.4. Mailing lists

BASE project has three different mailing lists available for subscription. Visit the mailing list page⁶ to get more information about each one of the mailing lists. All posted mails are saved in an archive for each list, it can therefore be a good idea to have a look here before posting a new thread.

These are the available mailing lists, more details about each one of them can be found on the mailing list page.

⁴ <http://baseplugins.thep.lu.se/>

⁵ <http://base2.thep.lu.se/demo/>

⁶ <http://base.thep.lu.se/wiki/MailingLists>

- basedb-users
- basedb-devel
- basedb-announce

3.5. BASE-hacks

There is a trac/subversion server keeping track of changes made to third party projects that are changed to make them work with BASE. Open source project usually have a requirement that changes are made public. On the BASE-hacks web site⁷ you will find a list of modified packages and information about the changes performed.

⁷ <http://dev.thep.lu.se/basehacks>

Part II. User documentation

Chapter 4. Overview of user documentation

The 'User documentation' part is quite extensive and covers everything from how to Log in on a BASE server and find your way through the program, to working with experiments and doing some useful analysis. The intention with this chapter is to give an overview of the following chapters so it will be easier for you to know where to look for certain information in case you don't want to read the whole part from the beginning to the end.

4.1. Working environment

Before you start working with any big experiment or project in BASE it could be a good idea to get to know the environment and perhaps personalize some behavior and appearance of the program. When this is done your daily work in BASE will be much easier and you will feel more comfortable working with the program.

Most of the things that have to do with the working environment are gathered in one chapter, where the first subsection, Section 5.1, "Introduction" (page 16) , gives a good guidance how to start using BASE including a general explanation how to navigate your way through the program.

The second subsection, Section 5.2, "Configuring your account"(page 19) , describes how to personalize BASE with contact information, preferences and changing password. The preferences are for instance some appearance like date format, text size or the look of the toolbar buttons.

The last two subsections, Section 5.3, "Working with items"(page 22) and Section 5.4, "Listing items" (page 27) , in the web client chapter explains how to work with BASE. No matter what you are going to do the user interface contains a lot of common functions that works the same everywhere. For example, how to list and search for items, how to create new items and modify and delete existing items. Subsequent chapters with detailed information about each type of item will usually not include descriptions of the common functionality.

4.2. Start working with BASE

There are some working principles that need to be understood by all users in BASE. These concern the permission system and how to get the workflow to move on without any disturbance caused by insufficient permissions. The key is to work in projects, which is covered in detail in Chapter 6, *Projects and the permission system* (page 40) .

Understanding the permission system and how to work in projects will not only make it more simple for you to work in BASE but also for your co-workers who want access to your data.

The next thing to do is to add some relevant data to work with. Most of the different items can be created manually from the web client, but some items and data must be imported from files. Before importing a file, it has to be uploaded on the BASE-server's file system. Chapter 7, *File management* (page 50) gives you information about the server's file system and how to upload the files.

Chapter 18, *Import of data* (page 150) explains how the import is done and Chapter 19, *Export of data* (page 159) covers how data later on can be exported from the database back into files, often simple text files or xml files.

Each different item has it's own section in this part of the documentation, where more specific information and also some screen shots can be found. Go back to the table of contents for this part and look up the item you want to know more about.

4.2.1. Administrative tasks

Most of the tasks in this section require more privileges than the normal user credentials. As always, there are many ways to do things so steps presented here is the path to get going with BASE as fast as possible without creating havoc in future use of the BASE server.

1. Log in as `root` using the password you set during BASE initialization. Create an account and give it the administrator-role. Switch user to the new admin account and use this for all future administrative tasks.

Note

The root-account should only be used to create the first administrator account and nothing else.

2. First thing to do, when logged in as administrator, is to create other user-accounts and give them appropriate roles, most of them should be assigned to the User-role.

Information related to user-accounts can be found at Chapter 22, *Account administration* (page 194).

3. Next step for you as an administrator is to import reporter-map and corresponding reporters to BASE. For import of Genepix data you can use the `Reporter importer` plug-in and `Reporter map importer` plug-in that come with BASE. Go to Array LIMS `Array designs` or `View Reporters` respectively and start the import from there. You can read more about data-import in Chapter 18, *Import of data* (page 150)

4.2.2. User tasks

A normal user is not allowed to add array design, reporter information, and a lot of other information to BASE. The reason for this is that a lot of information should only exist as one copy in the database. For example, reporters should only exist in one copy because everyone uses the same reporters. There is no need to store several copies of the same array design.

A user will normally upload experimental data to BASE for import into the database. To be able to import the data, the array design which is used, must be available in BASE at import time. If the array design is not available, a user with the proper credential must add the array design to BASE.

1. The first thing for an user to do is creating a project to work in and set this as *the active project*. This should be done before any other items are created. Section 6.2, “Projects” (page 41) tell you more about how working in projects can help you and your co-workers.
2. Next step is to create raw bioassays and up-load raw data to BASE. This is done in the raw bioassay section. (View `Raw bioassays`) . For more information see Section 17.2, “Raw bioassays” (page 138).
3. Now when there are data to work with, you can create your first experiment. You reach the experiment section through the menu `View Experiments`. Further reading in Section 17.3, “Experiments” (page 143).
4. a. The analysis often starts with the creation of a root bioassay set. Open the recently created experiment and go to the **Bioassay sets** tab. Click on the **New root bioassay set** button to start the creation.

b. With a root bioassay set you can now continue your analysis with different kinds of analysis plug-in. To the right of the each listed bioassay set is a set of icons for the actions that can be performed. Section 17.4, “Analysing data within BASE”(page 148) goes to the bottom of analysis in BASE.

This concludes the short step-by-step get going text. Far from all functionality in BASE has been covered here. E.g. nothing about LIMS or biomaterials have been mentioned. But you should now at least be familiar with getting to that point when it is possible to do some analysis.

Chapter 5. Using the web client

5.1. Introduction

5.1.1. Logging in

There are three things that you need to know before you can use BASE:

1. The address (URL) to a BASE server
2. A username to login with
3. A password

You may, for example, try the BASE demo server. Go to the URL `http://base2.thep.lu.se/demo/` and enter **base2** for the login and **base2** for the password.

You need to get all three things from an administrator of the BASE server. If you know only the address to the BASE server, you may check the front page if the administrator has added any information about how to get a username/password there. Look for the **Get an account!** link on the front page.

Logging in is simple, just enter your **login** and **password** in the form on the front page and click the **Login** button.

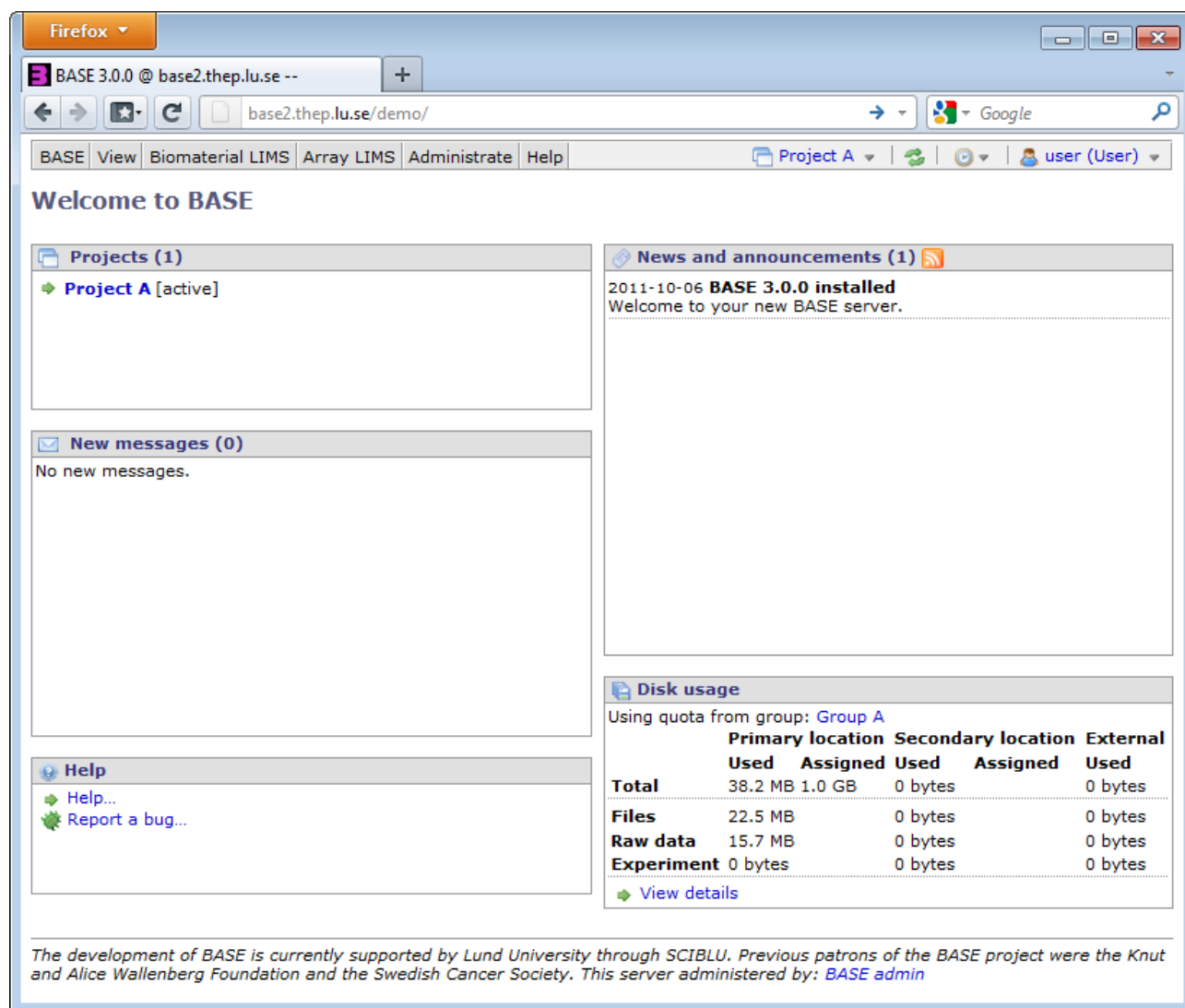
5.1.2. Forgotten password

If you forget your password you will need to get a new one. BASE stores the passwords in an encrypted form that does not allow anyone, not even the server administrator, to find out the un-encrypted password.

To get a new password you will have to contact the server administrator. There may be a **Forgot your password?** link on the front page where the server administrator has entered information about how to get a new password.

5.1.3. The home page

When you have been logged in the home page will be displayed. It displays some useful information. You can also go to the home page using the View Home menu.

Figure 5.1. The home page

New messages

Messages are, for example, sent by plug-ins to notify you about finished jobs. In the future, you may get messages from other sources as well. As of today, messages are not used for communication between users.

Projects

A list of projects that you are a member of. Projects are an important part of BASE and are the best way to share data when you are cooperating with other users. We recommend that you always use a project when working with BASE. For more information read Chapter 6, *Projects and the permission system* (page 40). The list displays the most recently used projects first and then fills up with the rest sorted by name.

Disk usage

An overview of how much disk space you have been assigned and how much you are using.


Help

Links for getting help and reporting bugs. The number of links displayed here may vary depending on the server configuration.

News and announcements

A list of important news and announcements from the server administrator. Here you may, for example, find information about server upgrades and maintenance.

Subscribe to the RSS news feed.

Click on the RSS icon  in the News and announcements title bar. This allows you to subscribe to the news feed from the BASE server so that you don't miss anything interesting.

5.1.4. Using the menu bar

On the top of the home page is the Menu bar. This is the main navigation tool in BASE. It works the same way as the regular menu system found in most other applications. Use the mouse to click and select an item from the menu.

Most of the menu is in two levels, ie. clicking on a top-level menu will open a submenu just below it. Clicking on something in the submenu will take you to another page or open a pop-up dialog window. For example, the Biomaterial LIMS Samples menu will take you to the page listing samples and BASE Contact information opens a dialog where you can modify your contact information details.

The menu bar also contains shortcuts to some often-used actions:

**Projects**

A list of all projects you are a member of. The most recently used projects are listed first, then the list is filled with the rest of your projects up to a maximum of 15. If you have more projects an option to display the remaining projects is activated. Selecting a project in the list will make that project the active project.

Tip

The sort order in the menu of non-recent projects is the same as the sort order on the projects list page. If you, for example, want to sort the newest project first (after the most recently used ones), select to sort by the **Registered** column in descending order on the list page. The menu will automatically use the same order.

**Refresh page**

Refresh/reload the current page. This is useful when you add or modify items in BASE. Most of the time the page is refreshed automatically, but in some cases you will have to use this button to refresh the page.

Warning

Do not use your browser's **Refresh** button. Most browsers will take you to the login page again.

**Recent items**

Shortcut to the most recently viewed items. The number of items are configurable and you can also make some item types *sticky*. This will for example keep the shortcut to the last experiment even if you have viewed lots of other items more recently. See the section called “The Recent items tab” (page 22) for configuration information.


**Logged in user**

Displays the name of the currently logged in user and allows you to quickly log out and switch to another user.

5.1.5. Getting help

Besides reading this document there are more ways to get help:

On-line context-sensitive help

Whenever you find a small help icon  or button you may click it to get help about the part of the page you are currently viewing. The icon is located in the title bar in most pop-up dialog windows and in the toolbar in most other pages.

Using the Help menu

The Help menu contains links for getting on-line help. These links may be configured by a server administrator, so they may be different from server to server. By default links for reporting a bug and accessing this document are installed.

Mailing lists and other resources

See Chapter 3, *Resources* (page 9).

5.2. Configuring your account

5.2.1. Contact information

Use the BASE Contact information menu to bring up the user information dialog.

This dialog has three tabs, **Contact information** (selected), **Password** and **Other information**. The logged in user can update the following contact information details.

Multi-user accounts

If you are using a multi-user account, for example a demo-account, you do not have permission to change the contact information.

Full name

Your full name. You are not allowed to change this. If it is not correct, contact an administrator to do it for you.

Email

Your email address (optional). If an email has been specified and if the server administrator has enabled email notifications, you also have the option to select if messages should be sent as emails. This can be useful to keep track of jobs that take a long time to complete.

Organisation

The name of the organisation you work for or represent (optional).

Address

Your postal address as it should be printed on letters to you (optional).

Phone

Your phone number (optional). You may enter multiple phone numbers, for example your work phone number and a mobile number.

Fax

Your fax number (optional).

Url

An URL to your home page or your organisation's home page (optional).

Press **Save** to save the changes or **Cancel** to abort.

5.2.2. Other information

Use the BASE Other information... menu to bring up the other information dialog.

This dialog has three tabs, **Contact information**, **Password** and **Other information** (selected).

The look of the **Other information** tab can differ a bit between different servers, depending on what settings the server is installed with. There are three inputs in a fresh BASE installation but it is only the **Description** text area that is static, the others can be removed or more fields can be added (managed by the server administrator). The three fields, included in a the BASE installation, are

Mobile

Your mobile number(Optional).

Skype

Your Skype contact information(Optional).

Description

Text area where you can put useful information that couldn't be stored anywhere else(Optional).

Press **Save** to save the changes or **Cancel** to abort.

5.2.3. Changing password

Use the BASE Change password menu to bring up the change password dialog.

This dialog has three tabs, **Contact information**, **Password** (selected) and **Other information**.

New password

Enter the new password.

Retype password

Retype the same password. You must do this to avoid spelling mistakes.

Multi-user accounts

If you are using a multi-user account, for example a demo-account, you do not have permission to change the password.

Empty passwords

If you leave both fields empty the password will not be changed. It is not possible to have an empty password.

5.2.4. Preferences

Use the BASE Preferences menu to bring up the preferences dialog. This dialog has three tabs, **Appearance**, **Plugins** and **Recent items**.

The Appearance tab

This tab contains settings that affect the appearance of the web client.

Font size

Select a basic font size. You can choose between five sizes: extra small (XS), small (S), medium (M), large (L) and extra large (XL). The default font size is medium.

Scale factor

The scale factor affects the size of pop-up windows. This setting exists because different browsers render pages differently. If you often find that pop-up windows are too small you can change this setting to make them bigger.

Note

The scale factor is automatically changed if the font size is changed.

Display long texts

This setting is used to control how long description texts are displayed in tables and other places with limited space. There are three settings:

- **Always:** The full text is always displayed. This may cause tables, etc. to become hard to read since cells will automatically grow to be able to display the full text.
- **On hover:** A short version of the text is displayed and the full text is automatically displayed when the mouse is moved over the text. Texts that are not fully visible are indicated with a dotted line to the right.
- **On click:** A short version of the text is displayed and the full text is displayed when the mouse is clicked somewhere on the short text. Texts that are not fully visible are indicated with a grey line to the right.

Warning

The 'On click' mode may not perform so well if lots of items are displayed in a single list. This is particularly so with Internet Explorer (version 7) which is 5-10 times slower than Firefox to render the page. If you experience problems with this mode you should either use a different mode or display less items on a single page.

Toolbar

You may choose if the toolbar buttons should have only images, only text or both images and text. The default is that they have both images and text.

Ratio color range

Select three colors to use when displaying data that is suitable for color coding, for example the intensity ratio in two-color experiments. The default setting is blue-white-yellow. The list of presets contains other useful color combinations (for example, the BASE version 1 red-yellow-green) and the most recently used color combinations.

Date format

A format string describing how dates should be displayed. We support all formatting options supported by the Java language. For more information see: SimpleDateFormat documentation¹ The most useful format patterns are:

- yy: two-digit year
- yyyy: four-digit year
- MM: two-digit month
- MMM: month name (short)
- MMMM: month name (full)
- dd: two-digit day in month

The list of presets contains the most commonly/recently used date formats.

Date-time format

A format string describing how dates with times should be displayed. We support all formatting options supported by the Java language. For more information see: SimpleDateFormat documentations² The most useful time-format patterns are:

- HH: two-digit hour (0-23)
- hh: two-digit hour (1-12)
- a: AM/PM marker
- mm: two-digit minute

¹ <http://download.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html>

² <http://download.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html>

- ss: two-digit second

Decimals

The number of decimals to display for numerical floating point values. The default is 2.

The Plugins tab

This tab contains settings that affect plug-in execution.

Messages

Mark the checkbox if you want to have a message sent to you when a plug-in completes execution. This setting can be overridden each time you start a plug-in. You'll receive the message as a notification in BASE, but it may also be possible to get the message as an email.

Remove jobs

This checkbox should be marked if you want the jobs, done by import or export plug-ins, to be marked as removed if they finished successfully. This setting can be overridden each time you start a plug-in.

Show warnings

This checkbox should be marked if you want to show warning messages from plug-ins in the **Select plug-in** dialog. Warning-level messages usually originates from plug-ins that are unrelated to the current task and are only of interest to plug-in developers. Error messages that are related to the current task are always shown.

The Recent items tab

This tab contains settings that affect the **Recent items** menu and selection lists in many edit dialogs.

Recently viewed items

The number of recently *viewed* items to remember. The default is to remember 6 items. The remembered items will be displayed in the **Recent items** menu in the menu bar.

Recently used items

The number of recently *used* items to remember. The default is to remember 4 items. The remembered items will be displayed in edit dialogs where they have been used before. Each type of edit operation has its own list of remembered items. For example, there is one list that remembers the most recently used protocols when creating a sample, and there is another list that remembers the most recently used scanners when creating a scan.

Load the names of all items

If checked, the names of the items will be loaded and displayed in the menu, otherwise only the ID and type of item is displayed.

Sticky items

Always remember the last viewed item of the selected types. For example, if you have selected *Experiment* as a sticky item, the last viewed experiment will be remembered even if you view hundreds of other items. Use the arrow buttons to move item types between the lists and sort the sticky items list. Sticky items will be displayed in the **Recent items** menu in the menu bar.

5.3. Working with items

No matter what you are doing in BASE some things work more or less in the same way. This section covers things that are common for most parts of BASE.

You mostly work with a single type of item at a time. This is reflected in the menu system. For example, use **Biomaterial LIMS** Samples to work with samples, and **View** Experiments to work with experiments. In most cases the list view for that type of item is displayed. The list view, as

the name says, is used to list all items. There are two more standard views, the single-item view and the edit view.

List view

This view lists all items of a certain type. The view allows you to search and it is possible to configure which information to show for each item. It also contains functions that can be used on multiple items at the same time, for example, delete, share and export. See Section 5.4, “Listing items” (page 27) for more information.

Single-item view

Displays information about a single item. For some items it is very little, but for some it is very much and the information may be divided into multiple tabs.

Edit view

This view is used for editing the information about a single item. It is always displayed as a pop-up window. Quite often the popup has multiple tabs, but the most important information is found on the first tab. Information that is required is always found on the first tab.

5.3.1. Create a new item

New items are mostly created from the list view. For example, to create a new experiment go to the View Experiments page. Here you will find a **New...** button in the toolbar. The button is disabled if you do not have permission to create new experiments. Otherwise, click on it and enter any required information in the pop-up dialog. Sometimes there are multiple tabs in this dialog. In the case of experiments there are three tabs: **Experiment**, **Publication** and **Experimental factors**. As a general rule, only the first tab has information that is required. The information in all other tabs are optional.

In some places you will also find actions that create items directly in the list. For example in the list of samples or on the single-item view for a sample you can create an extract using that sample as the parent. If you use such links the parent item will in most cases be selected automatically, which saves you a few clicks when creating new items.

Click on the **Save** button to save the new item to the database or on the **Cancel** button to abort.

Note

To speed up data entry when adding multiple new items there are a few tricks you can use to make the web client supply default values for most properties. To find a default value the following checklist is used in this order:

1. If the list have an active filter the filter values are used as default property values for the new item. For example, if you are listing experiments with **Genepix** raw data type the new experiment will automatically have **Genepix** selected. This trick should work for all properties except annotations, if it does not report it as a bug to the development team.
2. When you link to other items the same item will be used the next time. For example, if you create an extract and selects an extraction protocol the same protocol is used the next time you create another extract. In fact, BASE will remember as many items as specified by the **Recently used items** setting (default is 4), allowing you to quickly select one of those protocols. the section called “The Recent items tab”(page 22) contains more information about the setting.
3. If you have a project active and that project has specified default values those values will be used for new items. A project can specify defaults for protocols, hardware and software and a few other settings.

5.3.2. Edit an existing item

On all single-item views there is an **Edit...** button in the toolbar that opens a pop-up dialog for editing the properties of the item. This button is disabled if the logged in user does not have write permission for the item.

You can also open the edit pop-up in most other places where the item appears, for example, in lists or the single-item view of a related item. Press and hold one of the **CTRL**, **ALT** or **SHIFT** keys while clicking on the link and the edit window will open in a pop-up. If you do not have write permission on the item there is no meaning to open the edit pop-up and you will be taken to the single-item view page instead.

Click on the **Save** button to save the changes to the database or on the **Cancel** button to abort.

5.3.3. Delete items

You can delete items either from the list view or from a single-item view. In both cases, deleted items are only moved to the trashcan. No information is removed from the database. This allows you to restore items if you later find out that you need them again. In fact, there is nothing special about a removed item. It can still be used for the same things as any non-removed item can.

Important

To really delete items from the database you have two options:

1. Go to the trashcan View Trashcan and delete it from there. From the trashcan you can delete several items in one go. See Section 5.5, “Trashcan” (page 34).
2. Click on the small trashcan icon in the list or single-item view. You can only delete one item at a time.

To delete items from the list view you must first mark the checkbox for each item you want to delete. Then, click on the **Delete** button. The list should refresh itself automatically. If you want to confirm that the items have been removed use the **view / presets** dropdown and select the **Removed** option. The removed items should now be displayed in the list with a small trashcan icon to indicate that they are located in the trashcan.

To delete items from the single-item view, click on the **Delete** button in the toolbar. The page will refresh itself automatically and a small trashcan icon should be displayed. If you do not have permission to delete the item the delete button is disabled.

5.3.4. Restore deleted items

You can restore deleted items either from the trashcan, from the list view, or from the single-item view. This section only covers the last two cases. The trashcan is described in Section 5.5, “Trashcan” (page 34).

To delete items from the list view you must first make the deleted items appear in the list. This is easy, just use the **view / presets** dropdown and select the **Removed** option. The list should refresh itself automatically. The removed items are displayed in the list with a small trashcan icon to indicate that they are located in the trashcan. Then, mark the checkbox for each item that you want to restore and click the **Restore** button. The list should refresh itself automatically and the trashcan icon should be gone from the restored items.

To restore items from the single-item view, click on the **Restore** button in the toolbar. The page will refresh itself automatically and the small trashcan icon should be gone. If you do not have permission to restore the item the restore button is disabled.

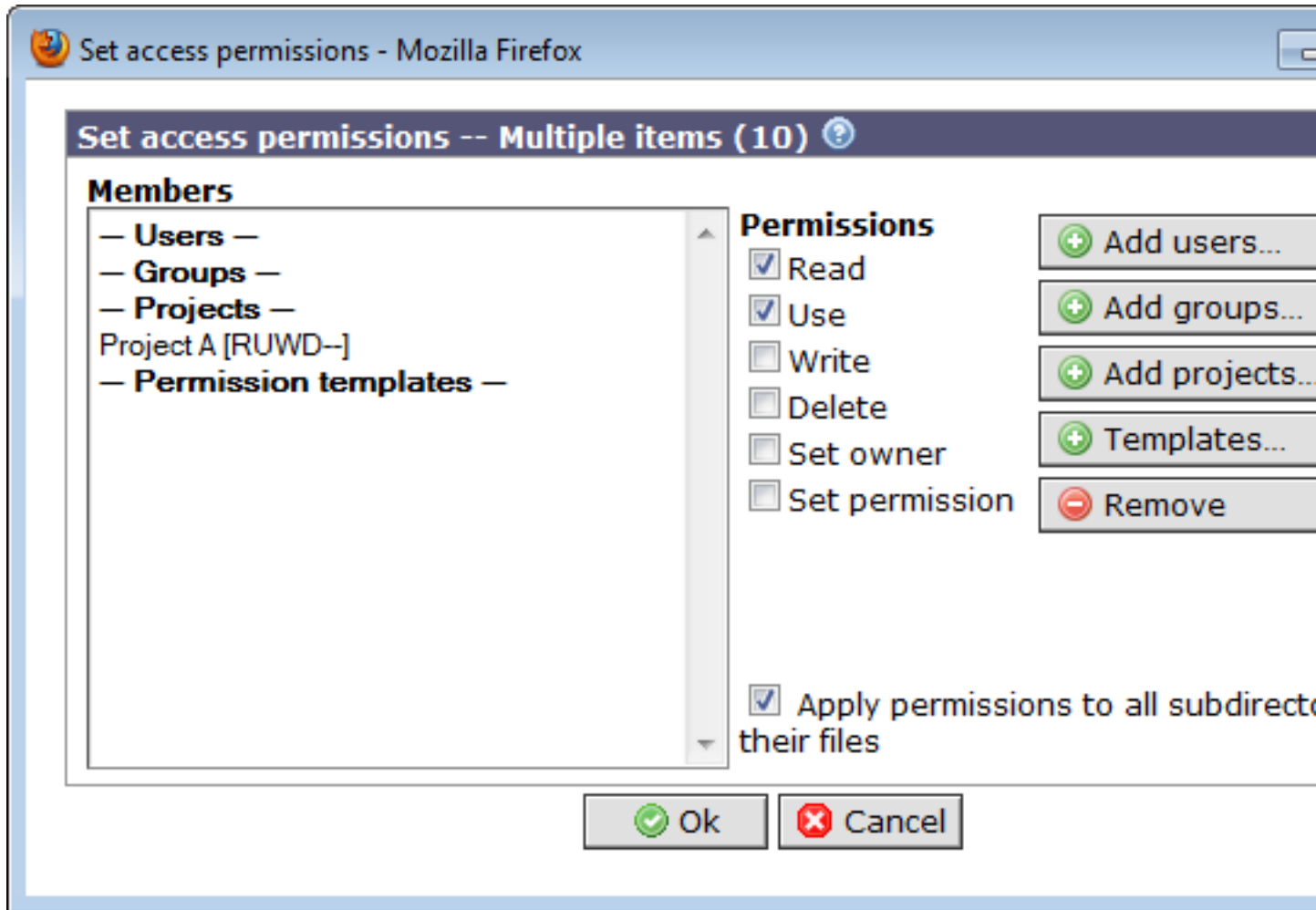
5.3.5. Share items to other users

Sharing data with other users is an important feature of BASE, which allows you cooperate in teams. If you follow the instructions in Chapter 6, *Projects and the permission system* (page 40) you will find that you almost never have to share items manually to other users. This is because whenever you work with an active project each new item you create will automatically be shared according to the settings of that project. In most cases, this is all you need.

If you still need to manually share your data with other users, here is how to do it.

From a list view, mark the checkbox for each item you want to share. Then, click on the **Share...** button. If you are on a single-item page, click on the **Share...** button on that page. In both cases, this will open the **Set access permissions** dialog window.

Figure 5.2. Sharing items to other users



Members

The list displays the users, groups and projects that already has access to the items you selected. The list shows the name and the permission level. The permission level uses a one-letter code as follows:

- **R** = Read
- **U** = Use
- **W** = Write
- **D** = Delete
- **O** = Set owner
- **P** = Set permission

Instead of a permission code, the word **varying** may be displayed. This happens if the items you selected have been shared with different permission.

The **Permission templates** part of the list is always empty to begin with.

Permissions

When you select a user, group or project in the list, the checkboxes will change to indicate the current permissions. The exception is if the permissions are varying, in which case no checkboxes are checked. To change the permissions just check the permissions you want to grant or uncheck the permissions you want to revoke. You can select more than one user, group or project and change the permissions for all of them at once.

The permission boxes are disabled if a permission template is selected. The permissions are already part of the template and can't be changed here.

Add users

Opens a pop-up window that allows you to select users to share the items to. In the pop-up window, mark one or more users and click on the **Ok** button. The pop-up window will only list users that you have permission to read. Unless you are an administrator, this usually means that you can only see users that:

- you share group memberships with (the *Everyone* group or groups with hidden members doesn't count)
- are members of the currently active project, if any.

Users that already have access to the item are not included in the list. If you don't see a user that you want to share an item to, you'll need to talk to an administrator for setting up the proper group membership.

Add groups

Opens a pop-up window that allows you to select groups to share the items to. In the pop-up window, mark one or more groups and click on the **Ok** button. Unless you are an administrator, the pop-up window will only list groups where you are a member. It will not list groups that already have access to the items. The *Everyone* groups is normally not visible unless have a specific permission to share items with this group.

Add projects

Opens a pop-up window that allows you to select projects to share the items to. In the pop-up window, mark one or more projects and click on the **Ok** button. Unless you are an administrator, the pop-up window will only list projects where you are a member. It will not list projects that already have access to the items.

Templates

Opens a pop-up window that allows you to select permission templates. In the pop-up window, mark one or more templates and click on the **Ok** button. Unless you are an administrator, the pop-up window will only list templates that you are allowed to use. It will not list templates that have already been added.

Note

The permissions from the selected templates are *copied* to the items when the access permissions are saved. If you re-open the share dialog, the actual permissions are shown and the permission templates section is empty. Modifying the permission template later doesn't affect the permissions on existing items. See Section 6.3, "Permission templates" (page 49) for more information about permission templates.

Remove

Click on this button to revoke access permissions from the selected users, groups and projects.

Apply permissions to all sub-directories and their files

This option shows up if at least one of the selected items is a directory. If this option is selected the permissions given to the directory will recursively be copied to all files and sub-directories. Existing permissions on those items will be overwritten with the new permissions.

Use the **Save** button to save your changes or the **Cancel** button to close the pop-up without saving.

5.3.6. Change owner of items

Sometimes it may be necessary to change the owner of an item. This can be done by everyone with *Set owner* permission on the item. For a user to have the rights to change owner of an item, the item must either be owned by or shared with *Set owner* permission to the user. See Section 5.3.5, “Share items to other users” (page 24).

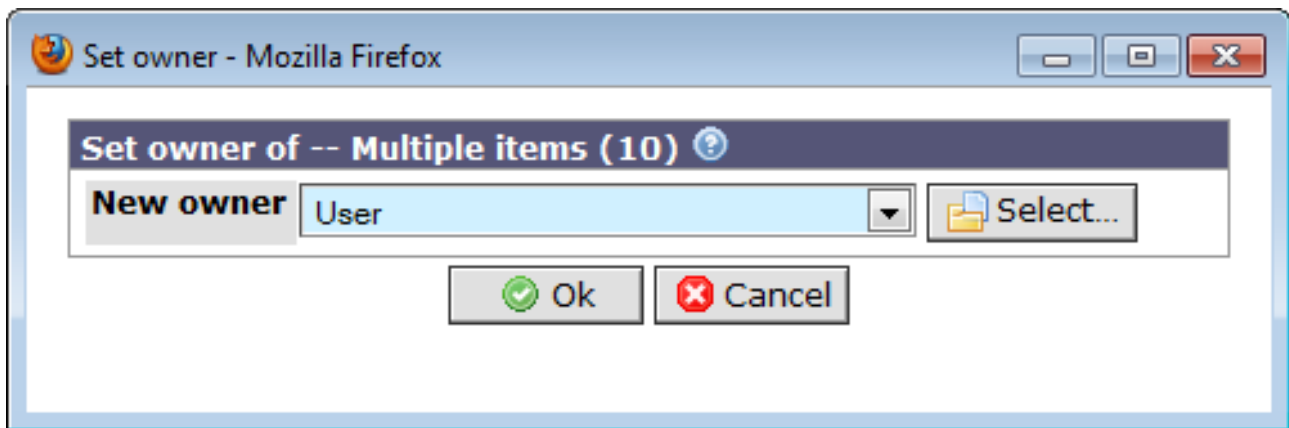
An user with *Set owner* permission can go to a list view (or the single-item view), mark the checkboxes for the items to change owner of, and click on the **Set owner** button. A dialog window, like the screen-shot below, will appear.

New owner

The user to be the new owner of selected item(s). By default the current user will be selected but other users can be picked from the *currently used* part of the drop-down list or by clicking on **Select**.

Use the **Save** button to set the new owner or the **Cancel** button to close the pop-up without saving.

Figure 5.3. Select a new owner



Warning

If you are the original owner of the items, you should be aware of that after the change you may no longer have access to the items. If you make a mistake you may have to talk to an administrator to correct it.

5.4. Listing items

All pages that lists items are very similar in their appearance and functionality. In this section we will describe the things that are common for most (if not all) list pages.

Use the menu to open a page listing items. Most list pages can only list one type of items. For example: use the View Samples menu to list samples and the View Experiments menu to list experiments.

Tip

An example of a list page that can list items of several types is found by going to View All items. This page lists all items that you are the owner of. It has a few limitations:

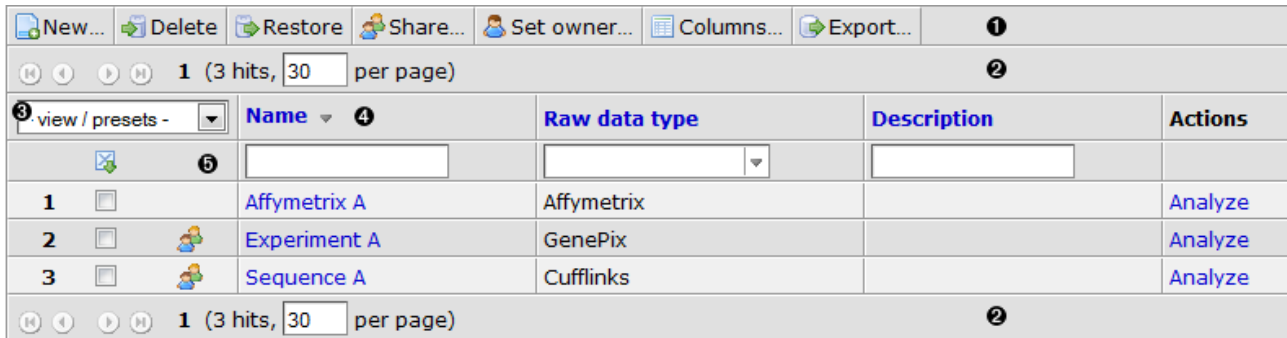
- It support only a limited set of columns (id, item type, name and description) since these are the only properties that are common among all items. It is also possible to display sharing information.

- The list may not have full support for filtering and sorting. This is due to a limitation in the query system used to generate the list.

There are also several similarities:

- It supports all of the regular multi-item operations such as delete, restore, share and change owner.
- Clicking on the name of the item will take you to the single-item view of that item. Holding down **CTRL**, **ALT** or **SHIFT** while clicking, will open the edit pop-up.

Figure 5.4. A typical list page



The typical list page contains the following important elements:

1. Toolbar

A toolbar with buttons for various actions such as **New...** for creating a new item, **Delete** for deleting items and **Columns...** for configuring columns. Depending on the permissions of the logged in user some buttons may be disabled (greyed out) or not shown at all.

2. Navigation bar

If there are many items the list will be divided into pages, each one showing a limited number of items. The navigation bar allows you to move to other pages and specify how many items each page should display. The navigation bar is repeated at the bottom of the list so you do not have to scroll back to the top of a long list just to get to another page.

3. List of presets

A list with preconfigured settings which allows you to quickly switch between different layouts (sort order, visible columns, filter settings, etc).

4. Column headers

The columns headers can be used for selecting sort order.

5. Filter bar

The filter bar allows you to search for items.

5.4.1. Ordering the list

Most lists are by default sorted by the name of the item. This can be changed by clicking on the column header of another column. If you click on the same column twice the sort order is reversed. A downwards or upwards pointing arrow is displayed next to the column header in the column that is currently used for sorting. Column headers that are black cannot be used for sorting.

It is possible to use more than one column for sorting. Press and hold one of the **CTRL**, **ALT** or **SHIFT** keys while clicking on another column header. The original sorting is kept and the new column is used for sub-sorting the list. The procedure can be repeated with more columns if you need to sort on three or more columns. To revert to sort by only one column again click a column header without holding down any key.

5.4.2. Filtering the list

If the list contains many items you may need to use a filter to be able to find the item you are looking for. The input boxes on the line below the column headers are used for filtering. Most columns are filtered using a free-text input box, but some columns that can only take a few distinct values use a selection list or radio buttons instead. The selection list and radio buttons are very simple to use. Just select the alternative that you want to filter on. The list will be automatically updated when the selection has been made.

The free-text filter is a bit more complex. By default, an exact match is required, use % as a wildcard character that matches any character. For example, the filter

Experiment A

only matches the same exact string, but the filter

Exp%

matches

Experiment A, Experiment B, etc.

If you want to filter on several values at the same time, separate the values in the filter input box with the “|” character. For example, a filter text like

Experiment A|C%

matches both “Experiment A” and values that begin with “C”.

You can also use operators to find items which has a value that is greater than, less than or not equal to a specific value. This is mostly useful on numeric or date columns but also works on text columns. The operator must be entered first in the free-text box, for example

<=10

to find items which has a value less than or equal to 10. Here is a list of the supported operators:

List of operators supported by the free-text filter

<
Less than

<=
Less than or equal to

>
Greater than

>=
Greater than or equal to

=
Equal to (useful to find items with a null value). Supports filtering on more then one value.

<>, !=
Not equal to (useful to find items with a non-null value). Supports filtering on more then one value.

==

Same as = but interprets “|”, “%” and other special characters literally. Use this when you need an exact string match.

><

Within a range. Two values separated by “|” are required. For example, `><10|20` to find values between 10 and 20 (inclusive).

Units

Some (numeric) columns have values with units. There are, for example, the *Original quantity* and *Remaining quantity* columns for biomaterials, which have values in micrograms (µg), and annotations which may have any unit.

When filtering on a column that has a unit, numeric values without units are interpreted as the default unit for that column. But it is also possible to add a unit to the filter value. The examples below are filtering on the original quantity column of a biomaterial:

`>=0.5mg`

matches biomaterials with an original quantity `>=500µg`.

`=100|200|300µg`

matches biomaterials with exactly 100, 200 or 300 micrograms.

It is also possible to mix units in a single filter:

`=100|200|300µg|0.5|1mg`

which matches 100, 200, 300, 500 and 1000 micrograms.

Be aware of rounding errors

All filter values with a unit that is different from the default unit are converted to the default unit before being applied. Since numeric conversions are never exact down to the last decimal, this may result in problems to filter with an exact match. The last example above could, for example, be converted to: 100, 200, 300, 500.000001 and 999.99999998.

Hard-to-type characters

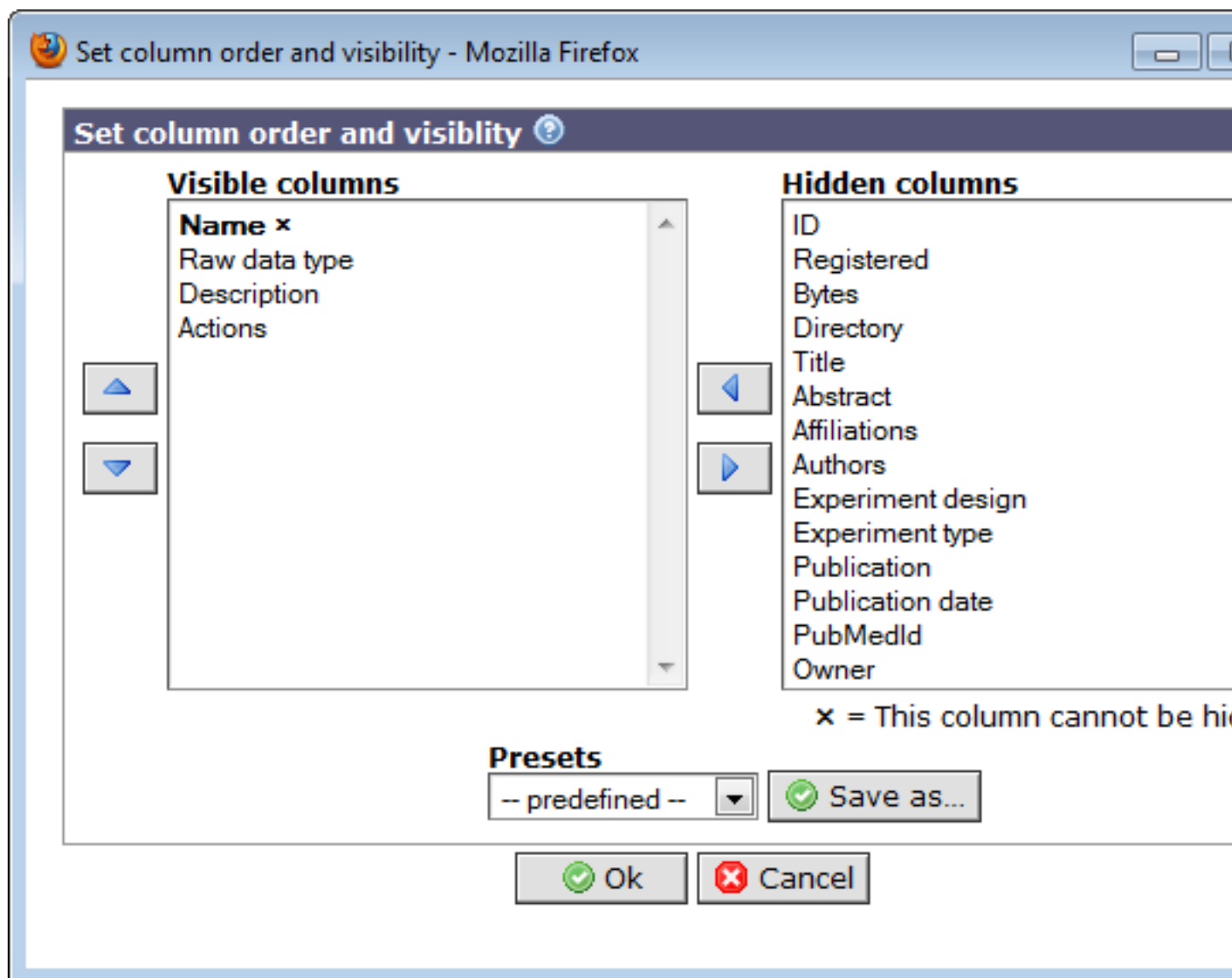
Some units contains hard-to-type characters. For example, the greek letter µ in µg, and mℓ and mℓ for areas and volumes. In all those cases it is also possible to use ug, m2 and m3, respectively.

Units are case-sensitive

All units are case sensitive. The main reason for this is that it must be possible to tell the difference between *milli* (*m*) and *mega* (*M*) prefixes, for example, *mJ* and *MJ*.

5.4.3. Configuring which columns to show

Most lists show only a small subset of the columns it is capable of showing. Use the **Columns...** button to open a dialog that allows you to select which columns to show and the order in which they are shown.

Figure 5.5. Configuring which columns to show**Visible columns**

Shows the columns that are currently visible. Use the up/down arrow buttons to arrange the order of the visible columns. The topmost column is shown to the left. Use the right arrow button to move columns from this list to the hidden columns list. Columns marked with an **x** are required and cannot be hidden. In most lists the **Name** column is the only column that is required.

Hidden columns

Shows columns that are not currently visible in the list. Use the left arrow button to move columns from this list to the visible columns list.

Presets

A dropdown list that allows you to select a set of preconfigured columns. You may also create your own preset if you often need to switch between different configurations. The list of presets is the same as the one described below, but if used from this dialog the presets only affects the visible columns and not filters or sort order.

Use the **Save** button to apply your changes or the **Cancel** button to close the pop-up without saving.

5.4.4. Presets

The **view / presets** dropdown has three main functions:

1. Switch between different configuration presets. The top of the dropdown contains user-defined presets (**Saved preset #1** and **#2**) and a few preconfigured presets. The user-defined presets are used to store a complete table configuration, including:

- Which columns are visible and their order
- The column (or columns) used for sorting
- Filter settings
- The number of items per page and the current page

The preconfigured presets only affects the visible columns as follows:

- **All columns** - Show all columns.
- **Required columns** - Show only the required columns. Usually only the **Name** column is required.
- **Default columns** - Show the default set of columns.
- **Other...** - Open the configure columns dialog box, described in Section 5.4.3, “Configuring which columns to show” (page 30).

2. Filter items by the removed status and the access permissions to an item.

- **Removed** - If checked, items that have been moved to the trashcan are shown, otherwise they are hidden.
- **Owned by me** - If checked, items that the logged in user owns are displayed, otherwise they are hidden.
- **Shared to me** - If checked, items that are owned by other users but shared to the logged in user are displayed, otherwise they are hidden.
- **In current project** - If checked, items that are linked with the current project are displayed, otherwise they are hidden. It does not matter if the logged in user is the owner or not. This option is only available if a project is active.
- **Owned by others** - This option is only available to administrators and will display items that are owned by other users.

The default is to display item that the current user owns and, if a project is active, items in that project.

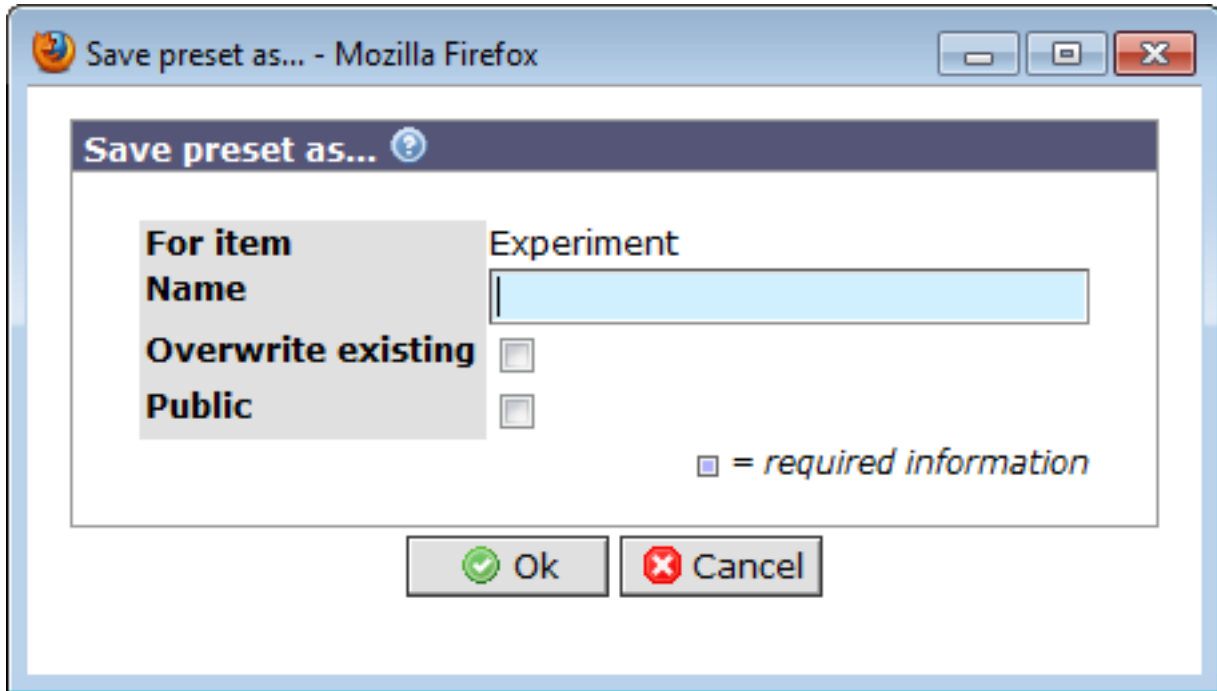
3. Administrate the presets

- **Clear filter** - Clears all filters.
- **Save as...** - Save the current configuration as a preset.
- **Manage...** - Opens a dialog where you can remove saved presets. You can also load saved presets from the dialog, but it is quicker to just use the dropdown list for this.

Save a preset

If you select the **Save as...** option from the **view / presets** dropdown the **Save preset as** dialog is opened.

Figure 5.7. Save preset as



The screenshot shows a web browser window titled "Save preset as... - Mozilla Firefox". Inside the browser, a dialog box titled "Save preset as..." is open. The dialog contains the following elements:

- A header bar with the title "Save preset as..." and a help icon.
- A section with four labels: "For item", "Name", "Overwrite existing", and "Public".
- Next to "For item" is a dropdown menu currently showing "Experiment".
- Next to "Name" is a text input field.
- Next to "Overwrite existing" and "Public" are checkboxes, both of which are currently unchecked.
- Below the checkboxes is a legend: a blue square icon followed by the text "= required information".
- At the bottom of the dialog are two buttons: "Ok" (with a green checkmark icon) and "Cancel" (with a red X icon).

For item

The type of item the preset is saved for.

Name

The name of the preset. The name must be unique.

Overwrite existing

If a preset with the same name already exists, it is overwritten if this checkbox is checked.

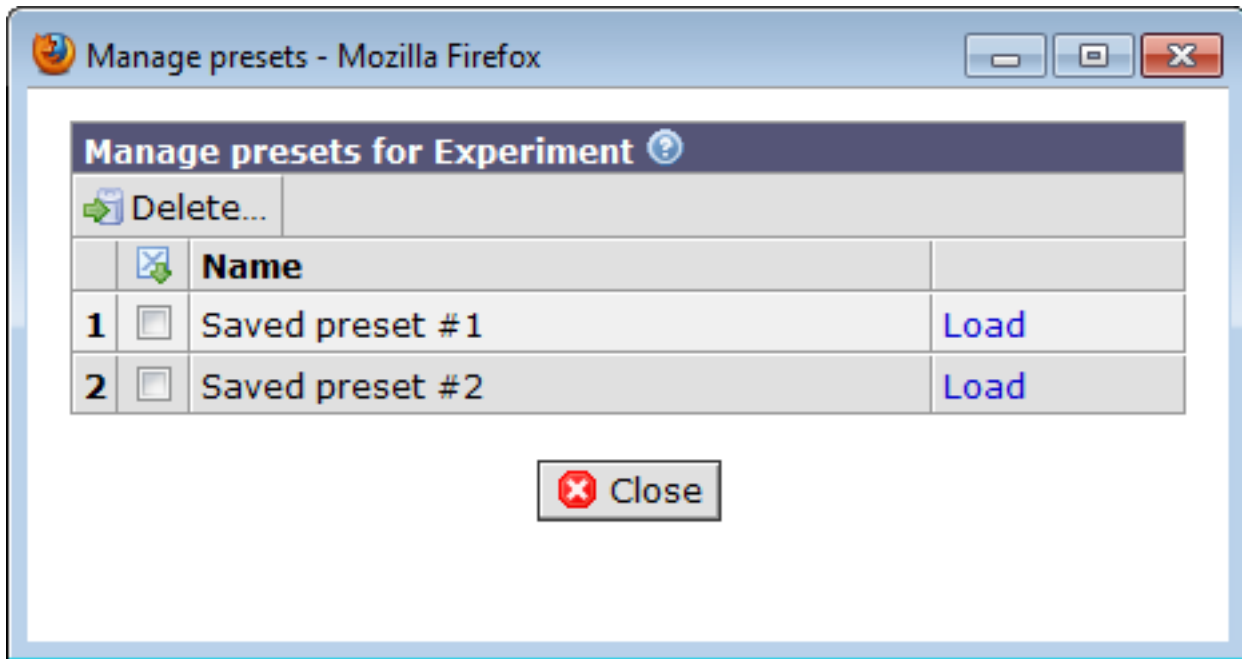
Public

This options is only available for users which has the `SHARE_TO_EVERYONE` permission. If checked the preset is visible to all users.

Use the **Ok** button to save the preset or the **Cancel** button to close the pop-up without saving.

Manage presets

If you select the **Manage...** option from the **view / presets** dropdown the **Manage presets** dialog is opened.

Figure 5.8. Manage presets

From this dialog you can delete or load presets.

To delete presets, first mark the checkbox in front of each preset you want to delete. Then, click on the **Delete...** button. You will get a warning about that the action cannot be undone. Unlike other items, the presets are not moved to the trashcan. Click on **Ok** to delete the preset.

Edit a preset

It is not possible to edit a preset directly. To change an existing preset you must:

1. Load the preset.
2. Use the interface to change column settings, filter, sort order, etc.
3. Save the preset with the same name.

Use the **Close** button to close the pop-up.

5.5. Trashcan


All items that have been deleted, and are owned by you, are listed in your trashcan. This list page is accessed with View Trashcan and it differs a bit from the other common list pages. The most significant difference is that the trashcan page can contain more than one item type, actually all removable item types in BASE can be listed in the trashcan. Items that neither can be removed or deleted, *i.e.*, items like sessions, nor clients' help texts since these are deleted from the database immediately in list/item view will be shown in the trashcan page.

Warning

Some item types do not have any owner and these are listed in the trashcans for everyone with delete permission on that specific item type.

Things that the trashcan page have in common with other list pages are the possibility to restore and view/edit items, see Section 5.3.4, "Restore deleted items"(page 24) and Section 5.3.2, "Edit an existing item" (page 23) . If an item is restored, it will of course disappear from the trashcan.

5.5.1. Delete items permanently

Items can be permanently deleted from BASE only if they are not used by other items. Items that are used have the icon  in the first column and by clicking on it you can get more information about the dependencies, see Section 5.5.2, “View dependencies of a trashed item” (page 35) .

Note

This view is NOT the same view page as when clicking on the item's name, which brings you to the item's view page.

To delete one or several items permanently from the trashcan you first have to select them and then to click on the **Delete** button. Press then on either **Ok** (completes the deletion) or **Cancel** (no items will be deleted) in the dialog window that appears.

Empty trashcan

If all items in the trashcan should be deleted permanently the **Empty trash** button can be used. This function will remove all items that are listed in your trashcan, except those items which other items, not marked for deletion or cannot be deleted, are dependent on.

5.5.2. View dependencies of a trashed item

This view can only be accessed from trashed items that are linked together with other items. Beside the item's **item type**, **name**, and **description** there is a list at the bottom of the view page with those items that are using the current item in some way.

Figure 5.9. Item view of a trashed item.

Trashcan ▶ Extract: Extract A.ref

Properties


Edit...


Restore


Share...

Help...

Permissions on this item: *Read, Use, Write, Delete, Take ownership, Change permission*

 This item has been flagged for deletion

 This item is shared to other user, groups and/or projects

 This item is used by other items and can't be permanently deleted

Type

Name

Description

Extract

Extract A.ref

Items using Extract: Extract A.ref

Delete

Restore

	Name/ID	Type	Description
1	Labeled extract A.ref	Extract	
2	Labeled extract A.ref (dye-swap)	Extract	
3	Library A.ref	Extract	

Shared to

Item type	Name	Permissions
Project	Project A	RUWD

1. This icon indicates that the item cannot be deleted permanently cause of some dependencies, listed below.
2. Common properties for all removable items.

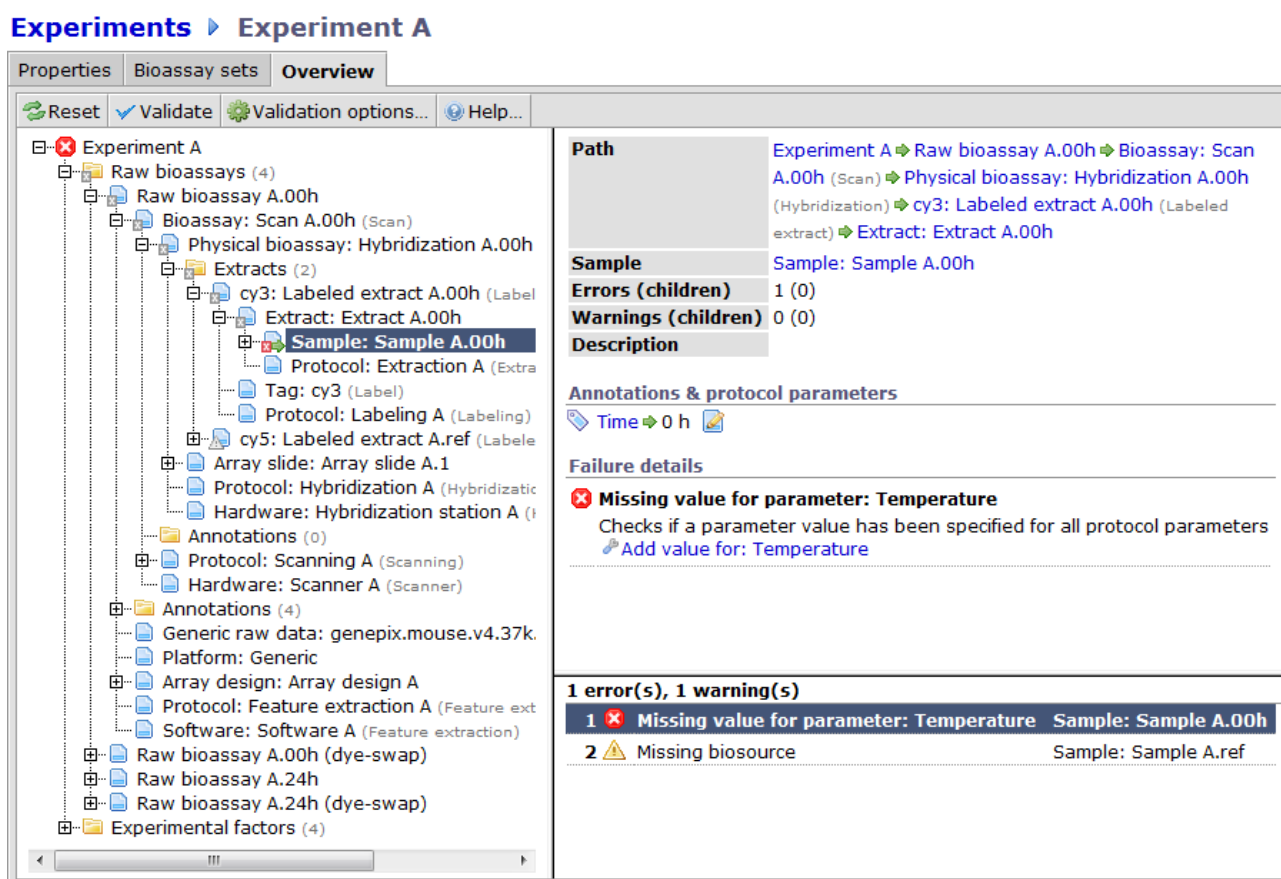
3. A list of other items that are using the current item, blocking permanent removal.

5.6. Item overview

With the **Item overview** function you can get an overview of all bioassays, extracts, samples, annotations, raw data sets, etc. that are related to a given item. In the overview you can also validate the data to find possibly missing or incorrect information.

You can access the overview for an item by navigating to the single-item view of the item you are interested in. Then, switch to the **Overview** tab that is present on that page. Here is an example of what is displayed:

Figure 5.10. The item overview



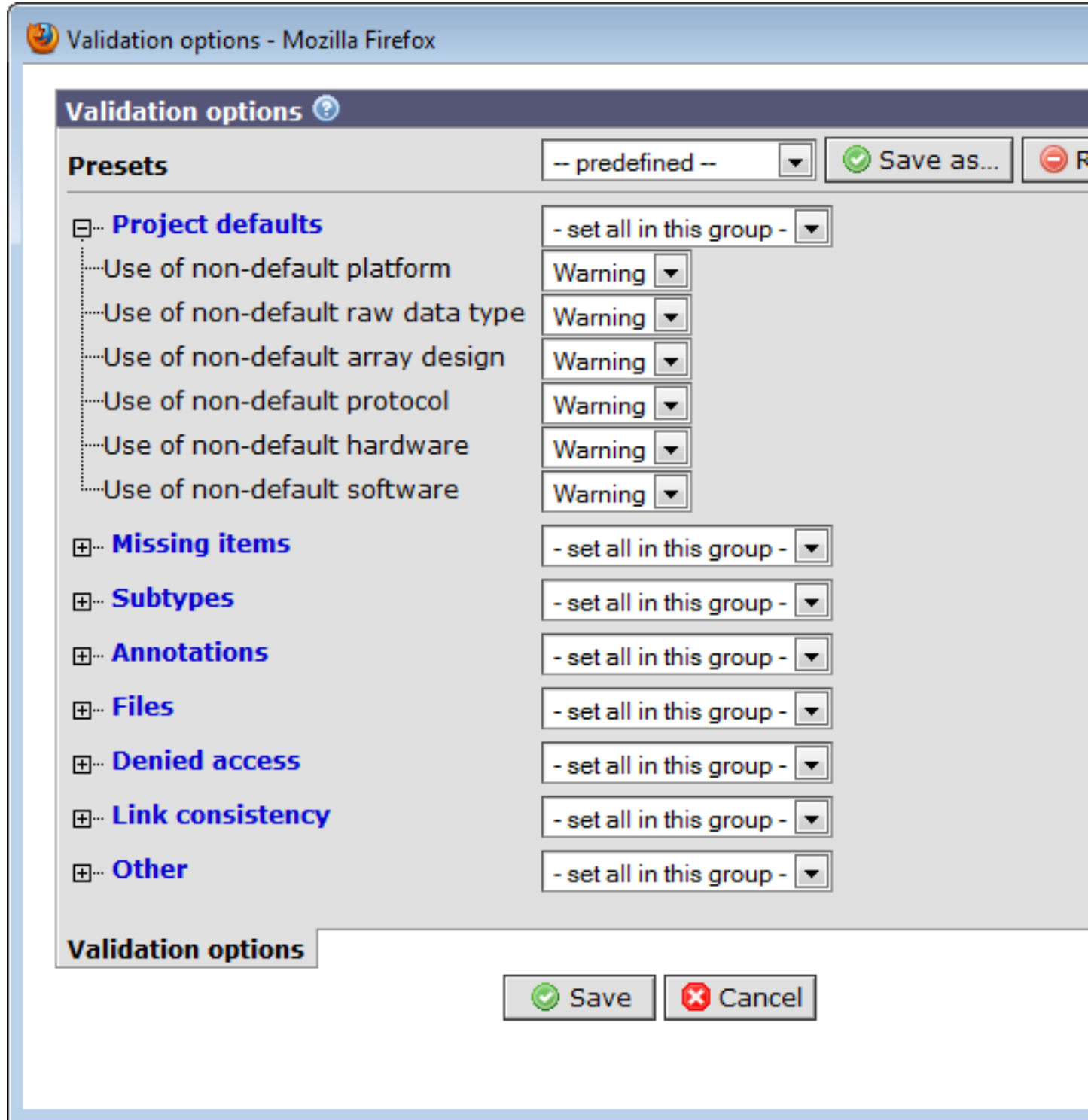
The page is divided into three sections:

- To the left is a tree displaying items that are related to the current item. The tree is loaded gradually when you click your way through the sublevels. The only exception is after a validation has been done, in this case the whole tree is loaded through the validation-process.
- The lower right shows a list of warnings and error messages that was found when validating the data. This section is empty if no validation has been done. Click on the **Validate** button to validate the data and load errors and warnings. In the example you can see that we have failed to specify a value for the **Temperature** protocol parameter for one of the samples.
- The upper right shows information about the currently selected item in the tree. This part will also contain more information about errors or warnings for this item, but only if a validation has been done. It may also present you with one or more suggestions about how to fix the problem and with a link that takes you to the most probable location where you can fix the error or warning.

5.6.1. Validation options

Click on the **Validation options** button in the toolbar to open the **Validation options** dialog.

Figure 5.11. Validation options



The validation procedure is highly configurable and you can select what you want to ignore, and what should be displayed as an error or warning.

Presets

The list contains predefined and user defined validation options. Use the **Save as...** button to save the current options as a user defined preset. The **Remove...** button is used to remove the currently selected preset. Predefined presets cannot be deleted.

Project defaults

The options in this section are used to check if your experiment uses the same values as set by the project default values of the currently active project (see Section 6.2, “Projects” (page 41)). If no project is active these options are ignored. The validation is only performed if the project has at least one item with a matching type. For example, if no default array design has been added to the project, all array designs are allowed. For items that can have a subtype, the subtype is also considered. For example, if only a default sampling protocol has been selected, no warnings are generated for extraction protocols.

Missing items

The options in this section are used to check if you have specified values for optional items. For example, there is an option that warns you if you have not specified a protocol. For items that can have a subtype, the rule here is to use the information about related subtypes before reporting a missing item. For example, the *labeled extract* subtype is related to the *label* subtype and if a label is missing it is reported as a warning. If there is no related subtype no warning is generated.

Subtypes

The options in this section are used to check that related items have a subtype that matches the subtype of the main item. For example, if we have an *extract* which is a *labeled extract* subtype the subtype for the related tag should be *label*, but if the extract is a *library* the subtype of the tag should be *barcode*.

Annotations

The options in this section are used to check problems related to annotations. The most important ones are listed here:

- *Missing MIAME annotation value*: Checks that you have specified values for all annotations marked as **Required for MIAME**.
- *Missing factor value*: Checks that you have specified values for all annotations used as experimental factors in the experiment. This is only checked when an experiment is selected as the root item.
- *Missing parameter value*: Checks that you have specified values for all protocol parameters.
- *Annotation is protocol parameter*: Checks if an item has been annotated with a an annotation that is actually a protocol parameter.
- *Annotation has invalid value*: Checks if annotation values are correct with respect to the rules given by the annotation type. This might include numeric values that are outside the valid range, or values not in the list of allows values for an enumerated annotation type.
- *Inheriting annotation from non-parent*: Checks if inherited annotations really comes from a parent item. This might happen if you rearrange parent-child relationship because you found that they were incorrectly linked.

Files

The options in this section are related to the validity of data files that can be attached to some items, for example, raw bioassays and array designs. The data files are usually validated immediately when they are used and the result is saved to the database. The options in this dialog can be used to find (or ignore) problems with data files.

Denied access

The options in this section are used to check if you do not have access (read permission) to an item in the experiment hierarchy. If this happens the validation cannot proceed in that branch. This might mask other validation problems.

Link consistency

The options in this section are used to check that links between (multiple) items are consistent with each other. The most important options are:

- *Array design mismatch*: Checks if the array design specified for a raw bioassay is the same array design specified for the physical bioassay.
- *Multiple array designs*: Checks if all raw bioassays in an experiment use the same array design or not. This is only checked when the root item is an experiment.
- *Circular reference to pooled item*: If you have used pooling, checks that no circular references have been created.
- *Multiple extracts with same tag+position*: Checks if the extracts on a physical bioassay have a unique combination of tag+position. This is usually required to be able to assign measured data correctly downstreams.

Other

This section collects options that does not fit into any of the other sections. The most important options are:

- *Non-unique name*: Checks if two items of the same type have the same name. It is usually a good idea to have unique names within an experiment if the data is going to be exported and in other circumstances.

Click on the **Save** button to use the current settings. The display will automatically refresh itself.

5.6.2. Fixing validation failures

The overview includes a function that allows you to quickly fix most of the problems found during the validation. The easiest way to use the function is:

1. Click on an error or warning in the list of failures in the lower right pane. The tree in the left pane and the item overview in the top right pane will automatically be updated to show the exact location of the faulty item.
2. The upper right pane should contain a list labeled **Failure details** with more information about each failure and also one or more suggestions for fixing the problem. For example, a failure due to a missing item should suggest that you add or select an item.
3. The suggestions should also have links that takes you to an edit view where you can do the changes.
4. After saving the changes you must click on the **Validate** button to update the interface. If you want, you can fix more than one failure before clicking on the button.

Inactive links?

If you do not have permission to fix a problem the links will be inactive and you'll have to talk to someone with more powers.

Chapter 6. Projects and the permission system

6.1. The permission system

BASE is a multi-user environment that supports cooperation between users while protecting all data against unauthorized access or modification. To make this possible an elaborate permission system has been developed that allows a user to specify exactly the permission to give to other users and at the same time makes it easy to handle the permissions of multiple items with just a few interactions. For this to work smoothly there are a few recommendations that all users should follow. The first and most important recommendation is:

Always use a project!

By collecting items in a project the life will be a lot easier when you want to share your data with others. This is because you can always treat all items in a project as one collection and grant or revoke access to the project as a whole.

6.1.1. Permission levels

Whenever you try to create or access existing items in BASE the core will check that you have the proper permission to do so. There are several permission levels:

Read

Permission to read information about the item, such as the name and description.

Use

Permission to use the information. In most cases this means linking with other items. For example, if you have permission to use a protocol you may specify that protocol as the extraction protocol when creating an extract from a sample. In the case of plug-ins, you need this permission to be able to execute them.

Write

Permission to change information about the item.

Delete

Permission to delete the item.

Change owner

Permission to change the owner of an item. This is implemented as a **Set owner** function in the web client (Section 5.3.6, “Change owner of items” (page 27)), where you can change the owner of items that you have permission to do so on.

Change permissions

Permission to change the permissions on the item.

Create

Permission to create new items. This permission can only be given to roles.

Deny

Deny all access to the item. This permission can only be given to roles.

Note

A user's permissions need to be reloaded for the permissions that have been changed should take effect. This is done either manually with the menu choice BASE Reload permissions or automatically next time the user logs in to BASE.

6.1.2. Getting access to an item

There are several ways that permission to access an item can be granted to you. The list below is a description of how the permission checks are implemented in the BASE core:

1. Check if you are the *root user*. The root user has full permission to everything and the permission check stops here.
2. Check if you are a *member of a role* that gives you access to the item. Role-based permissions can only be specified based on generic item types and is valid for all items of that type. The role-based permissions also include a special deny permission that prevents a user from accessing any item. In that case, the permission check stops here.
3. Check if you are the *owner of the item*. As the owner you have full permission to the item and the permission check stops here. This step is not done for items that doesn't have an owner.
4. Check if you have been granted access to the item by the *sharing system* (cf. Section 5.3.5, "Share items to other users" (page 24)). The sharing system can grant access to individual users, groups of users and to projects. We recommend that you always use projects to share your items. This step is not done for items that can't be shared.
5. Some items implement special permission checks. For example:
 - **News:** You always have read access to news if today's date falls between the start and end date of the news item.
 - **Groups:** You have read access to all groups where you are a member.
 - **Users:** You have read permission to all users that share group membership with, excluding the *Everyone* group. When a project is active, you also have read permission to all users that are members of that project.

There are more items with special permission checks but we do not list those here.

6.1.3. Plug-in permissions

Another aspect of the permission system is that plug-ins may also have permissions of their own. The default is that plug-ins run with the same permissions as the user that invoked the plug-in. Sometimes this can be seen as a security risk if the plug-in is not trusted. A malicious plug-in can, for example, delete the entire database if invoked by the root user.

An administrator can choose to give a plug-in only those permissions that is required to complete it's task. If the plug-in permission system is enabled for a plug-in the default is to deny all actions. Then, the administrator must assign permissions to the plug-in. There are two variants:

- A permission can be granted regardless of if the user that invoked the plug-in had the permission or not. This makes it possible to develop a plug-in that allows users to do things that they normally do not have permission to do directly in the web interface.
- A permission can be granted only if the user also has the permission. This is the same as not using the plug-in permission system, except that unspecified permissions are always denied.

Note

Plug-in developers can supply information about the wanted permissions making it easy for the administrator to just check the permissions and accept them with just a single click if they make sense. See Section 21.1.6, "Plug-in permissions" (page 184) for more information.

6.2. Projects

Projects are an important part of BASE and the permission system for several reasons:

- They do not require an administrator to setup and use. All regular users may create a project, add items to it and share it with other users. You are in complete control of who gets access to the project, the items it contains and which permission levels to use.
- All items in a project are treated as one collection. If a new member joins the team, just give the new person access to the project and that person will be able to access all items in the project.
- When you create new items, they are automatically shared using the settings from the active project. There is almost no need to share items manually. All you have to remember is to set an active project, and this is easy accessible from the menu bar.
- Filter out items that you do not want to see. When you have set an active project you may choose to only see items that are part of that project and no other items (Section 5.4.4, “Presets” (page 32)).
- It's easy to share multiple items between projects. Items may be part of more than one project. If you create a new project that builds on a previous one you can easily share some or all of the existing items to the new project from one central place, the **Items** tab on the project's single-item view.
- It is possible to assign default protocols, software, hardware and other items to a project. This makes it easier when creating new items since BASE will automatically suggest, for example the extraction protocol used when creating a new extract. The default items are also used by the **item overview** validation functionality, which makes it possible to spot mistakes. See Section 5.6, “Item overview” (page 36).

6.2.1. Creating a project

You can list and manage all of your projects by going to View Projects. Use the **New...** button to create a new projects.

Figure 6.1. Projects properties

Figure 6.1 shows the 'Edit project -- Project A' dialog box. The dialog is titled 'Edit project -- Project A' and contains the following fields and controls:

- Name:** A text field containing 'Project A'.
- Default permissions:** A dropdown menu showing '- select a template or specify below -', a 'Select...' button, and a list of checkboxes for 'Read', 'Use', 'Write', 'Delete', 'Set owner', and 'Set permission'. The first four are checked.
- Description:** A large empty text area.
- Buttons:** 'Save' (with a green checkmark) and 'Cancel' (with a red X).
- Tabs:** 'Project', 'Members', and 'Default items'.
- Legend:** A blue square icon represents 'required information'.

This tab allows users to enter essential information about a project.

Name

The name of the project. We recommend that project names are unique, since at some times it may need to be referenced by name.

Default permissions

This setting specifies the permissions to give to new items that are created while this project is the active project. The recommended setting is **delete** permission. Optionally, a permission template may be selected, in which case the permissions are copied from the template to the new item. See Section 6.3, "Permission templates" (page 49) for more information.

Description

An optional description of the project.

6.2.2. The active project

The active project concept is central to the sharing system. You should always, with few exceptions, have a project active when you work with BASE. The most important reason is that new items will


automatically be shared using the settings in the active project. This considerably reduces the time needed for managing access permissions. Without an active project you would have to manually set the permission on all items you create. If you have hundreds of items this is a time-consuming and boring task best to be avoided.


If you work with multiple projects you will probably find the filtering function that hides items that are not part of the active project to be useful. As a matter of fact, if you try to access an item that is part of another (not active) project you will get an error message saying that you do not have permission to access the item (unless you are the owner).

Selecting an active project

Since it's important to always have an active project there are several ways to make a project the active one.

- The easiest way and the one you will probably use most of the time is to use the menu bar shortcut.

Look in the menu for the project icon . Next to it, the name of the active project is displayed.

If you see  - **no active project** - here, it means that you have not selected a project to work in. Click on the icon or project name to open a drop-down menu and select a project to set as the active project. If another project is already active it will automatically be inactivated.

The most recently used projects are listed first, then the list is filled with the rest of your projects up to a maximum of 15. If you have more projects an option to display the remaining projects is activated.

Tip

The sort order of the non-recent projects is the same as the sort order on the projects list page. If you, for example, want to sort the newest project first, select to sort by the **Registered** column in descending order on the list page. The menu will automatically use the same order.

- Use the **BASE** Select project menu and select the project from the submenu that opens up.
- Go to the homepage using the **View** Home menu and select a project from the list displayed there.

Note

Only one project can be active at a time.

Caution

If you change the active project while viewing an item that you no longer has access to in the context of the new project an error message about missing permission will be displayed. Unfortunately, this is all that is displayed and it may be difficult to navigate to a working page again. In the worst case, you may have to go to the login page and login again.

Default permissions for the active project

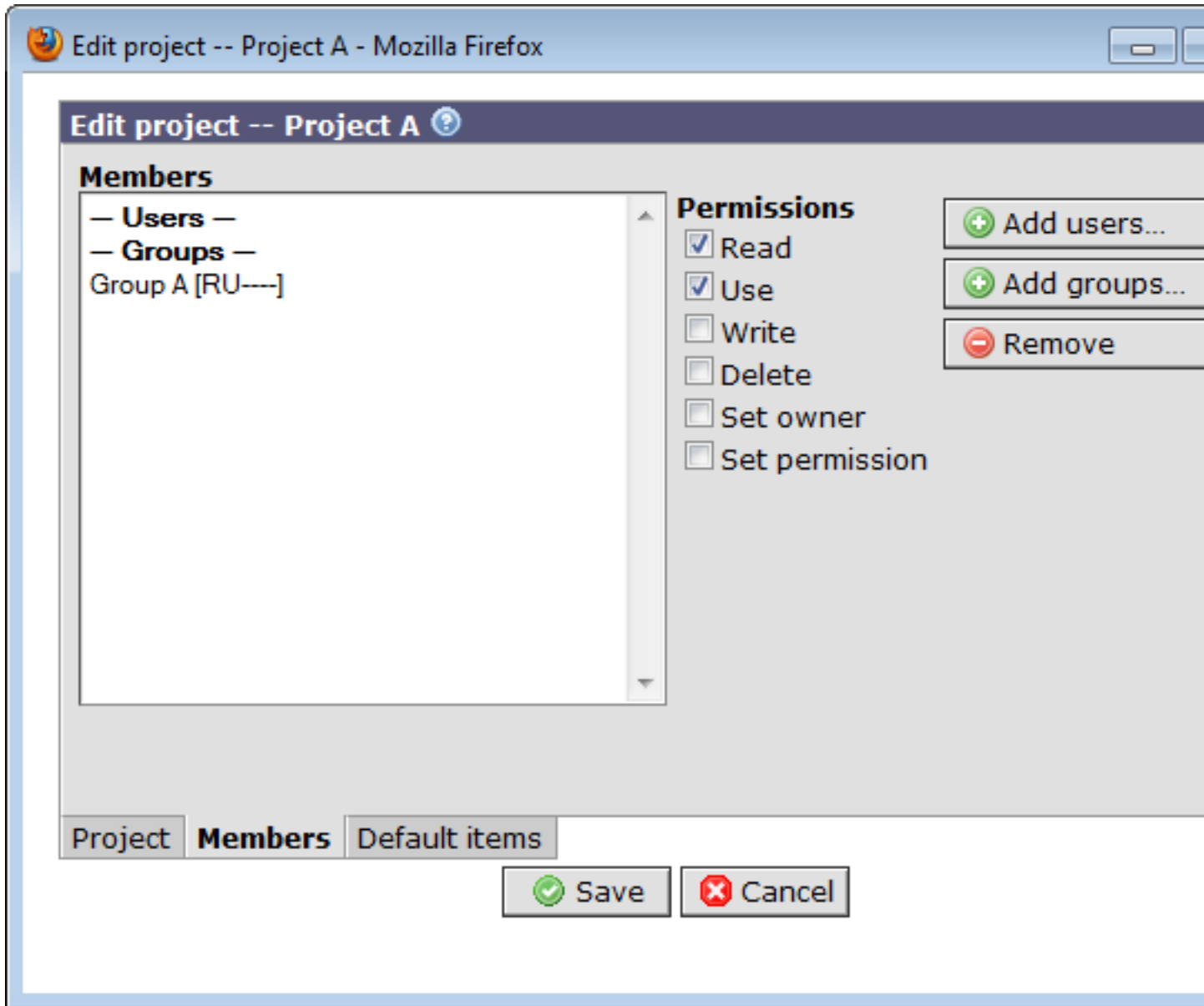
When a project is active all new items you create are automatically shared using the settings from the active project. If the active project has a permission template the permissions from the template are copied to the new item. If the project doesn't have a permission template, the new item is shared to the active project with the configured default level. By default, projects doesn't have a permission template and the default permissions are set to *read*, *use*, *write* and *delete*. It is possible to change the default permission level by modifying the settings for the project. Simply open the edit-view page for the project and select the permissions you want and save. From now on, all new items will be shared with the specified permissions. Items that are already in the project are not affected by the change.

6.2.3. How to give other users access to your project

First, you will need to open the **Edit project** dialog. Here is how to do that:

1. Navigate to the single-item view of your project from the View Projects list.
2. Click on the **Edit...** button to open the **Edit project** dialog.
3. Switch to the **Members** tab. From this page you can add and remove users and change the access levels of existing ones.

Figure 6.2. Manage members of a project



Members

The members list contains users and groups that are already members of the project. The list shows the name and the permission level. The permission level uses a one-letter code as follows:

- **R** = Read
- **U** = Use
- **W** = Write
- **D** = Delete

- **O** = Set owner
- **P** = Set permission

Permissions

When you select an user or group in the list the current permission will be checked. To change the permissions just check the permissions you want to grant or uncheck the permissions you want to revoke. You may select more than one user and/or group and change the permissions for all of them at once.

Note

In most cases, you should give the project members **use** permission. This will allow a user to use all items in the project as well as add new items to it. If you give them **write** or **delete** permission they will be able to modify or delete all items including those that they do not own.

This rule is valid for all items that are shared to the project with the default **delete** permission. Items that are shared with a lower permission, for example, **use**, can be accessed with at most that permission.

Add users

Opens a popup window that allows you to add users to the project. In the popup window, mark one or more users and click on the **Ok** button. The popup window will only list users that you have permission to read. Unless you are an administrator, this usually means that you can only see users that:

- you share group memberships with (the *Everyone* group and groups with hidden members doesn't count)
- are members of the currently active project, if any.

Users that already have access to the project are not included in the list. If you don't see a user that you want to add to the project, you'll need to talk to an administrator for setting up the proper group membership.

Add groups

Opens a popup window that allows you to add groups to the project. In the popup window, mark one or more groups and click on the **Ok** button. Unless you are an administrator, the popup window will only list groups that you are a member of. It will not list groups that are already part of the project.

Remove

Click on this button to remove the selected users and/or groups from the project.

Use the **Save** button to save your changes or the **Cancel** button to close the popup without saving.

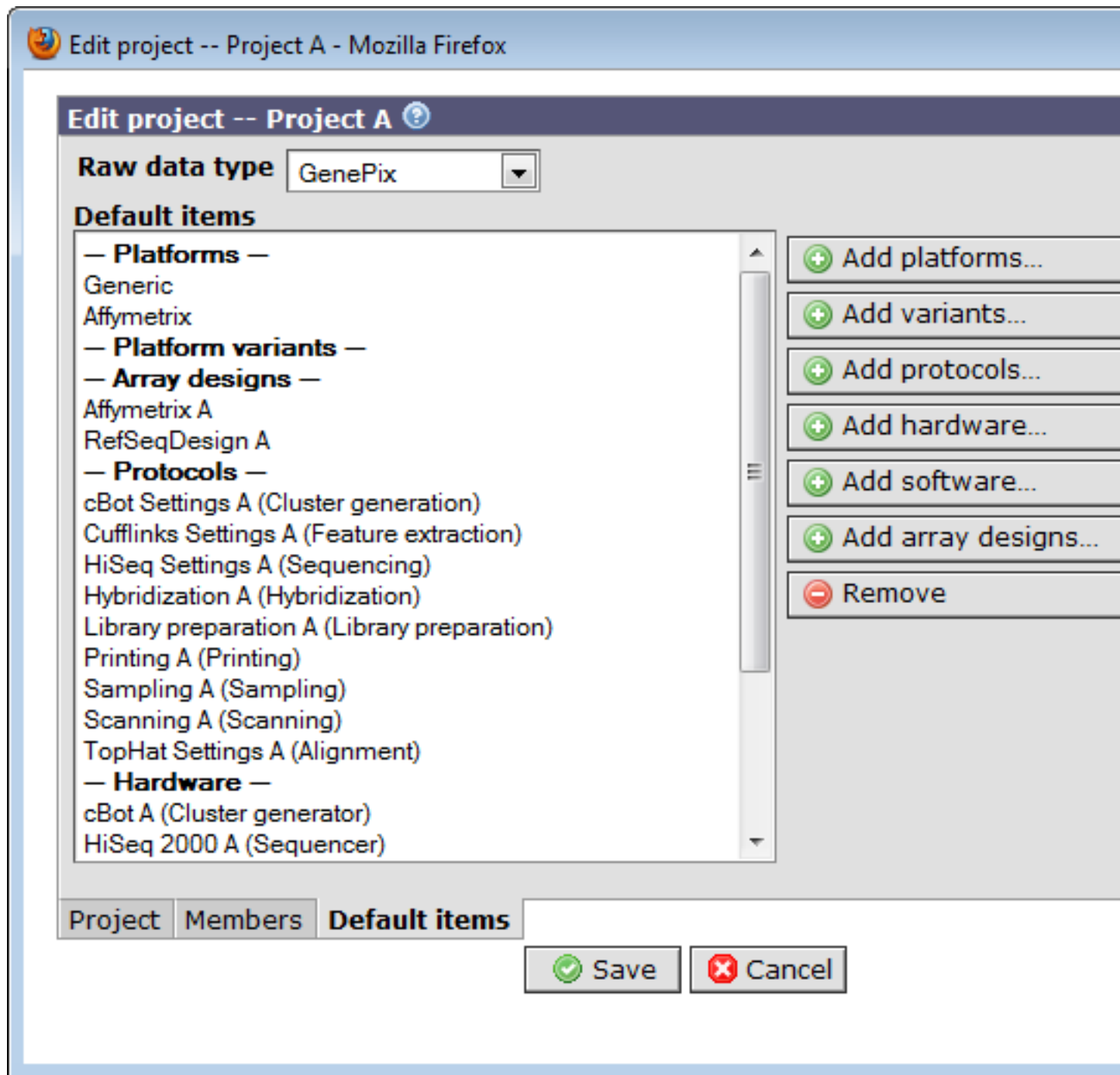
6.2.4. Default items

A number of default item can be assigned to a project. It is possible to select one raw data type and any number of platforms, variants, protocols, hardware, software and array designs. The default items are used by BASE to suggest default values. The subtype (see Chapter 12, *Item subtypes* (page 92)) of each item is used as a filter so that, for example, an extraction protocol is suggested when creating an extract, and a hybridization protocol when creating a hybridization. Use the various **Add** buttons to add items to the project and the **Remove** button to remove them.

Note

Make sure that the items that are selected as default items also are shared to the project with at least **use** permission. Otherwise the default items will not show up for other members of the project, which may result in registering incorrect data.

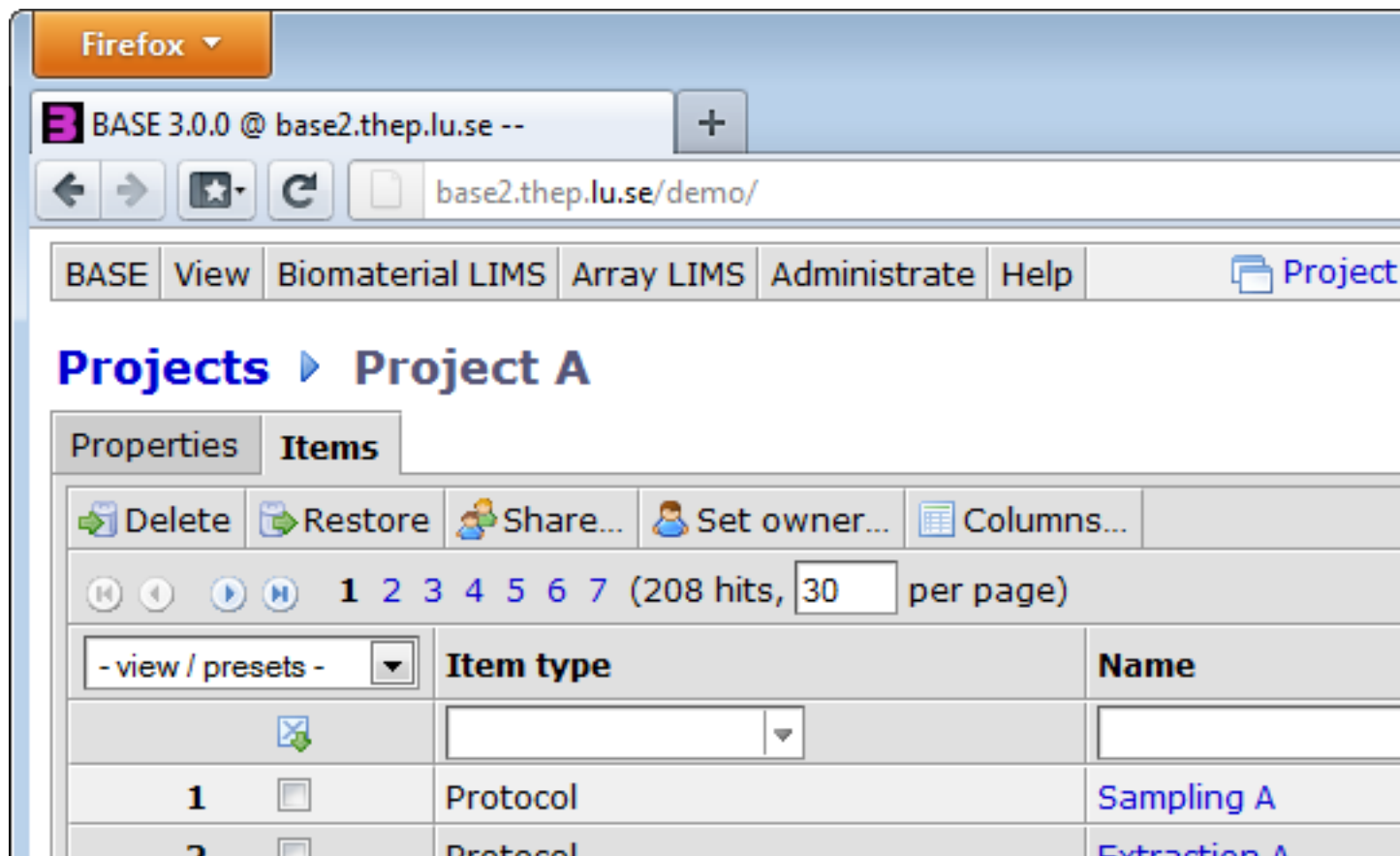
Figure 6.3. Project default items



6.2.5. Working with the items in the project

If you go to the single-item view for a project you will find that there is an extra tab, **Items**, on that page.

Figure 6.4.



Clicking on that tab will display a page that is similar to a list view. However there are some differences:

- The list is not limited to one type of item. It can display all items that are part of the project.
- It support only a limited set of columns (id, name, description, owner and a few more) since these are the only properties that are common among all items.
- The list cannot be sorted. This is due to a limitation in the query system used to generate the list.

Note

The list only works for the active project. For all other projects it will only display items that are owned by the logged in user.

There are also several similarities:

- It supports all of the regular multi-item operations such as delete, restore, share and change owner.
- Clicking on the name of the item will take you to the single-item view of that item. Holding down **CTRL**, **ALT** or **SHIFT** while clicking, will open the edit popup.

Tip

This list is very useful when you are creating a new project, in which you want to reuse items from an old project.

- Activate the old project and go to this view.
- Mark the checkbox for all items that you want to use in the new project.
- Click on the **Share...** button and share the items to the new project.

| If you have more than one old project, repeat the above procedure.

6.3. Permission templates

A *permission template* is a pre-defined set of permissions for users, groups and/or projects. The template makes it easy to quickly share items to multiple users, groups and projects, possible with different permissions for everyone. There are three major use-cases where permission templates are useful:

- A permission template can be associated with project. When the project is selected as the active project, the permissions from the template are copied to any new items that are created. Note that the new items may or may not be shared with the active project, depending on the settings in the permission template.
- Permission templates can be selected in the share dialog, making it easier to manually share items to multiple users, groups and projects in just a few clicks.
- Permission templates can be used with some batch item importers, making it easier for administrators which only needs a single data file even if the data belong to different projects.

Permission templates are managed from the View Permission templates menu. The template is a very simple item that only has a name (required) and a description (optional). We recommend that the names of the templates are kept unique, but this is not enforced by BASE. To assign permissions to the template use the **Set permissions** button. This is the same dialog as the share dialog.

Permissions are copied

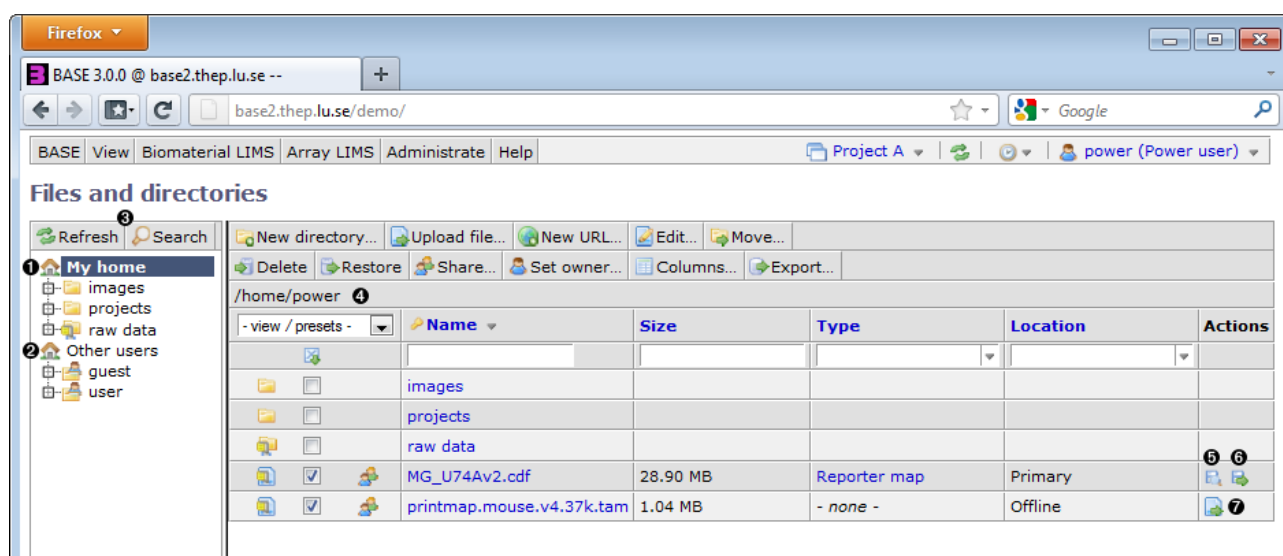
When a permission template is used the permissions are *copied* to the items. Modifications to the template that are made afterwards doesn't affect the permissions for the items on which the template was used.

Chapter 7. File management

7.1. File system

Files in BASE are managed from the page at View Files. The basic layout on the page is the same as for all the other list pages in BASE but there are some differences e.g. there is a navigation tree to the left, used to browse the directory structure, and there are some buttons in the toolbar, that are special for files and directories. The figure below is a representation of the files and directories-page and is followed with a short description to some of the special functions.

Figure 7.1. The file page



1. Home directory for current user

This is the logged in user's home directory with sub directories. It is visible if the current user has a home directory and is then always located at the top of the navigation tree. Click on a directory to display it's contents to the right or click on the plus sign to expand the directory and view the sub directories. Note that the directory tree is lazily loaded. Subdirectories that hasn't yet been opened have light gray icon which is removed if they turn out to be empty.

Note

A directory can only be opened from the navigation tree or from the directory icon in the list. A click on a directory's name in the list will open the directory's edit window.

2. Other users

The other users' home directories that the current user has permission to read are listed here.

Missing directory?

Not all users have a home directory connected to their accounts. If a user is missing, it most certainly depends on that the user account have not got a home directory. Home directories are managed by the administrator of the BASE server.

3. Button toolbar

The button toolbar contains functions that are relevant for the navigation tree. Use the **Refresh** button to update the directory tree, for example, after creating a new subdirectory. Use the **Search** button to search for files and directories no matter where they are located. The search form is displayed to the right and is the same as the usual file and directory listing, except that it will not show any files if there is no filter.

4. Current directory

Shows the full BASE path to current directory.

5. View a file's contents

A click on this icon will open the file's contents in a new window. If the browser does not has support to view the file there will be a dialog window to download the file instead.

6. Download file

Download the file to a local computer with this icon. The download will start in a new dialog window.

7. Re-upload a file

This icon is only visible for those files that have been moved offline and can be used to re-upload the file to the BASE. Start to upload the file to the same position by clicking on the icon.

Replace an existing file

It is possible to re-upload file that are on-line, but this has to be done from the single-item view.

7.1.1. Disk space quota

Normally, a user is assigned limited disk space for files. More information about how much quota the current account has and how much of it that is occupied can be found at the account's home page, described in Section 5.1.3, "The home page" (page 16). See also Section 22.4, "Disk space/quota" (page 208) for more information about the quota system.

7.2. Handling files

7.2.1. Upload a new file

Uploading a file is started by clicking on **Upload file...** in the toolbar. The uploaded file will be placed in current directory.

Figure 7.2. Upload new file

Upload new file ?

Directory /home/power

File

Replace existing ☐

Write protected ☐

Store compressed - auto -

Type - none -

Character set - n/a -

Description

Max transfer rate 100.0 MB/s (approx.)

Compressed file

☐ Unpack file

☐ Overwrite existing files

☒ Keep the compressed file

☐ = required

File

Directory

Shows the current directory, where the file will be uploaded. This property cannot be changed and is only for information.

File

This field is required and needs to have a valid file path for the local computer before the upload is started. Use **Browse...** to choose which file to upload.

Replace existing

Tick this checkbox if you want to overwrite an existing file that has the same name as the one you are going to upload. In contrast with other items, file names must be unique since they are often referenced by name.

Write protected

Mark this checkbox if you want the file to be write protected. A write protected file cannot be deleted, moved offline or replaced with another file. It is still possible to change other metadata, such as the name, description, file type, MIME type, etc.

Store compressed

You can select if you want BASE to store your file in a compressed format or in its normal format. Compressing the file may save a lot of disk space and it also uses less quota. There are three options:

- **auto:** Let BASE automatically decide if the file should be compressed or not. The file is compressed if (1): it is uploaded to a directory that has the *compress files* flag set or (2): if the matching MIME type has the *compress files* flag set.
- **yes:** Store the file in a compressed format.
- **no:** Store the file in its normal format.

Type

This is the file-type that the uploaded file should get. Select – **none** – if the file should not be associated with any file type.

Character set

If you are uploading a text file, it may be a good idea to select the character set used for the file. The most common character set is UTF-8 or ISO-8859-1. It may be important to get this setting correct if you are going to import data from the file later.

Description

A description about the uploaded file can be put into this text area. Use the magnifying glass to edit the text in a pop-up window with a larger text area.

Max transfer rate

This shows the maximum transfer rate that the upload will approximately reach. The transfer rate is set by the server admin and cannot be changed.

Compressed file

These settings are only active if you select a compressed file format that BASE knows how to unpack. BASE ships with support for some of the most common compressed file formats, such as zip and tar, but this can be extended by the use of plug-ins. See Section 25.6.3, “File unpacker plug-ins” (page 257) for more information.

- **Unpack file:** Mark this checkbox if the compressed file should be unpacked after it had been uploaded. The files will be unpacked with the same sub-directory structure as in the compressed file.
- **Overwrite existing files:** Mark this checkbox if the unpacking is allowed to overwrite existing files.
- **Keep the compressed file:** Mark this checkbox if you want to keep the compressed file after upload. Otherwise, only the unpacked files are kept.

Finish the configuration by clicking on either **Upload**, which will start uploading the selected file, or **Cancel** to abort the upload procedure.

Replace an existing file

It is possible to replace an existing file. This is done by clicking on the **replace** link on the single-item view for the file you want to replace. If the file has been moved offline, you can also use the icon

in the actions-column, see number 7 in Figure 7.1, “The file page”(page 50). The procedure to upload the file is the same as when uploading a new file, except that compressed files cannot be unpacked. There is also an extra option, **Validate MD5**, that tells BASE to check that the file is the same as the one it is replacing. This option is useful when you are re-uploading a file that has been moved offline and want to be certain that it is the same file as the original.

Note

You cannot replace a file which has been marked as *write protected*.

7.2.2. External files

Files doesn't have to be stored on the BASE server. It is possible to register an external file by giving the URL to it. In most cases, BASE will be able to use the external file in the same way as a file that has been uploaded to the BASE server. To create an external file reference, use the **New URL...** button.

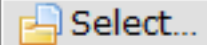
Figure 7.3. Create external file

Create file ?

Path /home/power

URL http://base.thep.lu.se

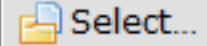
☒ Load metadata


Server base.thep.lu.se 

Name base.thep.lu.se


Write protected ☐

Type -none-


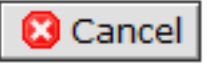
MIME type 

Character set 

Description

 = required information

File

The dialog is more or less the same as the **Edit file** dialog, but has additional fields for the **URL** and an optional **File server**.

URL

The full URL to the referenced file. BASE ships with support *http* and *https* URLs, but the administrator may have installed support for other protocols.

Load metadata

Check this box if you want BASE to try to load metadata such as MIME type, size, last modification date etc. for the file. This will also verify that the file actually exists.

Server

Select a file server for this file. This is optional, but is needed to access password-protected files or for some https connections.

7.2.3. File servers

Figure 7.4. Fileserver properties

Edit file server -- base.thep.lu.se - Mozilla Firefox

Edit file server -- base.thep.lu.se ?

Name base.thep.lu.se

Connection manager - auto - • = Supports auto-detection

Automatically select a connection manager among those that supports auto-detection.

Host

Username

Password

(leave empty to keep the password)

Description

□ = required input

File server Certificates

Save Cancel

File server are used for external files that are password protected and for files that are using the https protocol and require certificates to connect to the server.

Name

The name of the file server.

Connection manager

The **auto** setting allows BASE to automatically select a manager based on the file URL. It is possible to force a specific connection manager from this list. The list of available connection managers can be extended by plug-ins. See Section 26.8.7, "Connection managers" (page 280) for more information.

Host

If specified, overrides the host (and/or port) part of the file URL with the value in this field. This is useful, if for example, a file can be accessed using a "public" path that is entered as the URL for the file and an "internal" path that is used from the BASE server.

Username/password

If the file server requires authorization to access the files you should add a username and password. This will be used by BASE to access the files. Currently, BASE supports *Basic* and *Digest* authentication.

Description

Enter a description of the file server.

Certificates

On this tab you may specify server and client certificates. A **server certificate** may be needed to access files with the https protocol on servers that use certificates that can't automatically be trusted, for example, a self-signed certificate. The server certificate is uploaded as a file and must be a X.509 certificate in either binary or base64-encoded DER format.

A **client certificate** may be needed to access files with the https protocol on servers that require that clients authenticate themselves with a certificate. The certificate is typically issued by the owner of the server and may be password-protected. The client certificate is uploaded as a file and must be in PKCS #12 format.

Use the **Remove existing...** checkboxes to remove previously uploaded certificates. Leave everything empty to keep things as they are.

7.2.4. Edit a file

The edit window to set a file's property in can either be open with **Edit...** that is located in the toolbar at the file's view page or by holding down **CTRL**, **ALT** or **SHIFT** when clicking on the file's name in the list. It requires that the current user has write permission on the file to be able to edit and set the properties.

Path

This is the path where the file is located. This can only be changed by moving the file. Read more about how this is done in Section 7.2.5, "Move files" (page 57).

URL, Server

See *External files* above.

Name

The file's name, which cannot be left empty and must be unique in current directory. The maximum length of the file name is 255 characters and it can contain blank spaces but not any of ~, \, /, :, ;, *, ?, <, > or |.

Write protected

Mark this checkbox if you want the file to be write protected. A write protected file cannot be deleted, moved offline or replaced with another file. It is still possible to change other metadata, such as the name, description, file type, MIME type, etc.

Type

Sets which kind of type the file is. Select the file type to use from the drop down list with available types. The option **-none-** should be used if the file should not be associated with any kind of file type.

MIME type

The file's content/media type. This is normally set automatically when uploading the file into BASE but it can be changed by an user, that has write permissions, at any time.

Description

This text area can be used to store relevant information about file and it's contents. Use the magnifying glass, located to the right under the text area, to edit the text in a larger window.

Finish the editing process by pressing either **Save** to save the properties to the database or **Cancel** to abort and discard the changes.

7.2.5. Move files

These functions are used to manage the location of the files on the server. They are all accessed from the **Move** button on the list view or from the single-item view. On the list view, you must first select one or more files / directories.

Write protect your files!

If you mark a file as *write protected* it will not be possible to delete, move or replace the file. Use this options for important data files that you do not want to loose by accident.

To another directory

Files and directories can be moved to other directories for re-organization. The user need write permission on the target directory to be able to move the files/directories to it.

First, select all files and directories in the current path that should be moved and then click on **Move...** To another directory in the toolbar to open a window with the directory tree where the target directory can be picked.

Choose a directory which the selected items should be moved to. It is possible to create new sub-directories with the **New...** button.

Click on **Ok** to carry out the move or **Cancel** to abort.

Offline

Moving a file offline means that the actual file contents is deleted from the server's disk space but information about the file will still exist as an item in the database. This makes it possible to save disk space but still be able to associated the file with other items in BASE.

First, select all files in current path that should be moved offline and then click on **Move...** Offline in the toolbar.

Warning

Be careful! The selected files will be removed from the server. The only way to recover the contents again is to re-upload the files.

To the secondary storage

This option is only available if the server administrator has enabled it.

The secondary storage is a kind of storage where it is appropriate to store files that have been used and no longer requires immediate access. Moving a file to and from the secondary storage is the job of a plug-in, which is usually executed once or twice a day.

First, select all files in the current path that should be moved and then click on **Move...** To secondary location in the toolbar. The only thing that will happen is that BASE sets a flag on each file. The next time the secondary storage plug-in is executed, the files will be moved to the secondary storage. The actual file contents is deleted from the server's disk.

While the file is in the secondary storage BASE behaves in the same way as if the file is offline. The file cannot be used to import data from, or other things. To use the file again, the file must be moved back to the primary storage.

To bring files back from the secondary storage, select the files and then click on **Move...** To primary location in the toolbar. The files will be moved back the next time the secondary storage plug-in is executed.

Do not forget to set quota for the secondary storage

The default installation does not assign quota for the secondary storage. Unless the administrator assigns quota the move will silently fail.

7.2.6. Viewing and downloading files

In **Actions** column in the list view there are icons you can click on to perform different kinds of actions on a file, like downloading the file and viewing the file. The same icons appear on the single-item view and in most other places where files are used. You cannot view or download files that have been moved offline or to the secondary storage.

Download a file

This will let the user to download the contents of a file to a path on a local computer. The window that opens contains the selected file's name, size e.t.c. and it will also open a download dialog window where the user can choose what to do with the file locally.

Download does not start

Click on the file's path name in the pop-up window if the download dialog window does not appear.

Close the pop-up window and return to file page with **Close**.

View the contents of file

A file's contents can be displayed directly in the web browser if the browser supports displaying that kind of files. Typically all HTML, text files and images are supported. Click on the icon to view the contents in a new window. If the type is not supported by the browser there will be a dialog-window to download the file instead.

Download/compress multiple files

You can download multiple files/directories at the same time. First, from the file browser, select one or more files/directories. Then, click on the **Export** button. Select the **Packed file exporter** plug-in and choose one of the file formats below it. On the **Next** page you can specify other options for the download:

- **Save as:** The path to a file on the BASE file system where the selected files and directories should be packed. Leave this field empty to download the files to your own computer.
- **Overwrite:** If you are saving to the BASE file system you may select if it is allowed to overwrite an existing file or not.

- **Remove files/directories:** If you select this option the selected files and directories will be marked as removed. You must still go to the **Trashcan** and remove the items permanently.

7.2.7. Directories

Directories in BASE are folders where files can be organized into. Click on **New directory...** in the toolbar to create a directory in current path and edit the properties as described below.

Figure 7.5. Directory properties

The screenshot shows a web browser window titled "Edit directory -- raw data - Mozilla Firefox". Inside, there is a dialog box titled "Edit directory -- raw data" with a help icon. The dialog contains the following fields and options:

- Path:** /home/power
- Name:** raw data (highlighted in blue)
- Compress new files:** ☐ no ☒ yes ☐ Recursive
- Share new files and sub-directories automatically:** ☒ no ☐ yes ☐ Recursive
- Description:** Place your raw data files here (text area)

At the bottom right of the dialog, there is a legend: ☐ = required information. At the bottom of the browser window, there are "Save" and "Cancel" buttons.

Properties

Path

This property is read-only in the edit window but can be changed by moving the directory, described in Section 7.2.5, "Move files" (page 57).

Name

The directory's name to identify it with in the list. This field must have a value and it has to be an unique name for the current directory.

Compress new files

Enable this option to let BASE store files that are uploaded to this directory in a compressed format. This option only affect files that are uploaded later, it doesn't affect already existing files

or files that are moved between directories. Check the **Recursive** button to apply this setting to all subdirectories.

Share new files and sub-directories automatically

Enable this option to let BASE automatically share new files and directories with the same permissions as have been specified on this directory. This option is useful when you have assigned a specific directory as a common area for a group of users and you want to make sure that all users has access to all files. Some restrictions apply:

- Permissions for the *Everyone* group are not inherited if the logged in user doesn't have the *SHARE_TO_EVERYONE* permission.
- If a project is active the new file/directory will be shared to the active project as well.

Check the **Recursive** button to apply this setting to all subdirectories.

Description

Any relevant information about the directory can be written in this text area. The magnifying glass down to the right can be used to edit the description text in a larger text area, just click on the icon to open it in a separate pop-up window.

The editing process is completed with either **Save**, to save the properties into the database, or with **Cancel** to discard the changes. Both of the buttons will close the edit window and if the directory is updated the list will be reloaded with the directory's new properties.

Note

The new directory does not appear in the navigation tree to the left automatically. You must click on the **Refresh** button.

Chapter 8. Jobs

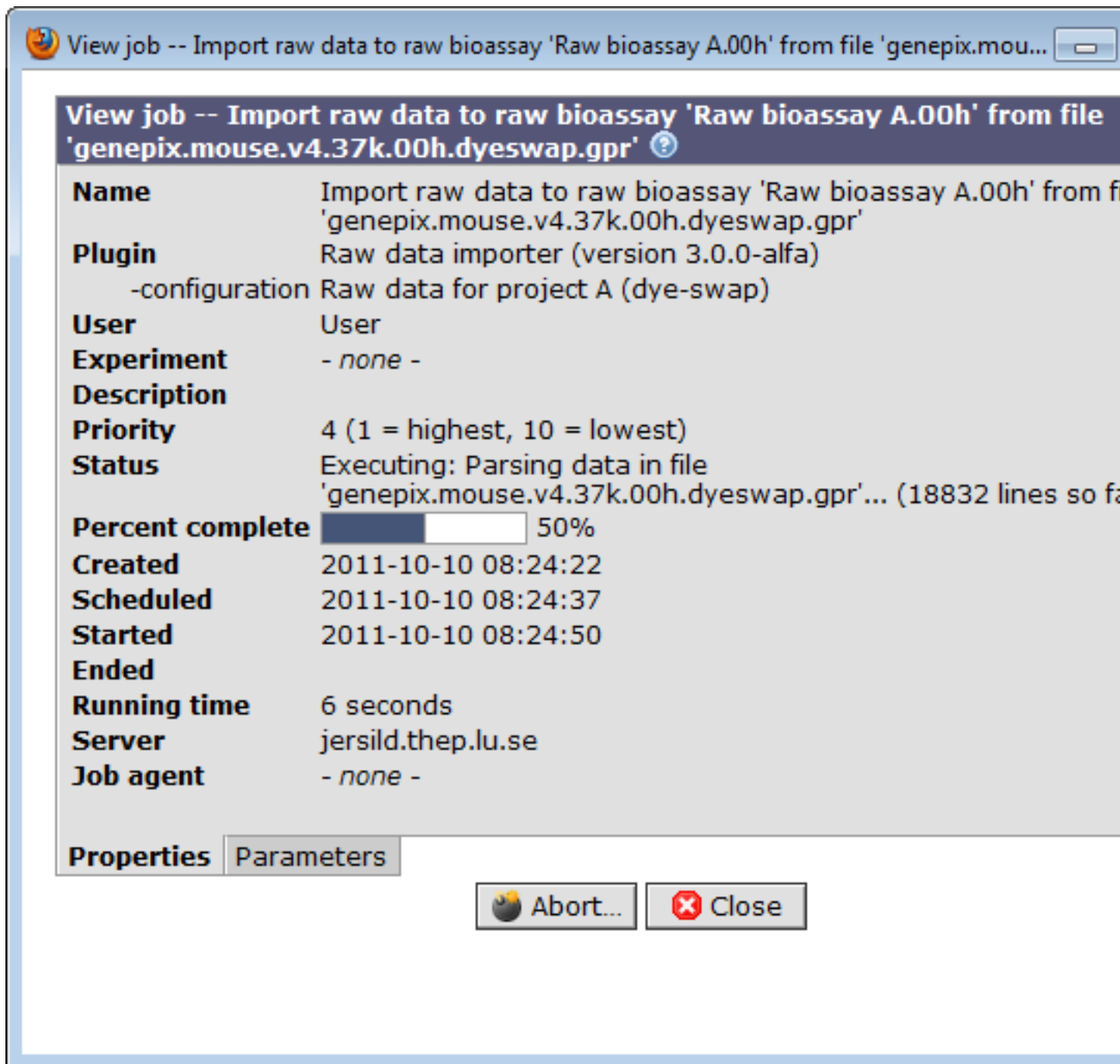
A *job* is a task that is performed by a plug-in. Some jobs can be configured to execute immediately (for a few plug-ins it is mandatory that the job runs immediately), but normally jobs are placed in the job queue after they have been configured. The jobs will then be picked out for execution depending on their priority and waiting time in the queue.

Creating and configuring a job is made from a context, eg. a particular list page or single-item page, which is supported by the plug-in you want to use. If no context is needed, the job configuration is started from the plug-in's single item view. Typically, there are three main types of jobs, *import*, *export* and *other* jobs. They usually show up as **Import**, **Export** and **Run plug-in** buttons in the gui. In the experiment analysis section two more types of plug-ins are used, *intensity* and *analysis*.

Click on View Jobs to list your jobs. Details of a single job is displayed in a pop-up window. This window opens either if you click on a job's name from the list page of jobs or when a job configuration is finished. The window contain two tabs, one with information about the job and another with the parameters for the plug-in used in the job.

The values listed on the **Parameters** tab depends on what the plug-in needs from the user. If a specific plug-in configuration was used, those parameters are also listed here.

Figure 8.1. Job properties



The properties are set either when configuring the job or by the system. No parameters can be edit after a job is created.

Name

The name of the job is set in the last step of a job configuration.

Plugin

The plug-in to use in the job.

Configuration

Name of the plug-in configuration that is used.

User

The user who created/configured the job.

Experiment

Name of the experiment which the job was configured within.

Description

A description of the job. Like the name-property it can be set in the job configuration.

Priority

Priority the job has in the job queue.

Status

Shows the status of the job. A job can have one of following status.

- *Not configured* - The plug-in has not been configured properly and is not placed in the job queue.
- *Waiting* - The job is waiting in the job queue.
- *Preparing* - The job queue is preparing to executed the job.
- *Executing* - The job is being executed.
- *Done* - Indicates that the job has finished successfully.
- *Error* - The job has finsished with an error.
- *Aborting* - The job has received an abort signal from the user and tries to abort the work. This field will also display any messages from the plug-in while it is running and the final message after it's completion.

Percent complete

Progress of the job. How detailed this is depends on how often the plug-in reports it's progress.

Created

Date and time when the job was created and registered in the database.

Scheduled

Date and time when the job was added to the job queue. This is usually the same as the *created* date and time, but can be different if the job has been restarted after an error.

Started

Date and time when execution of the job started.

Ended

Date and time when the job stopped running. Either because it was finished, aborted or interrupted by an error.

Running time

Time the job has been running.

Server

Name of the server, where the job was performed.

Job agent

The job agent the job is/was running on. It is also possible to set this value before a job is executed. If that has been done only the selected job agent will accept the job. This options is normally only given to powers users and needs the *Select job agent* permission. See Section 22.3.2, "Edit role" (page 205).

Depending on the status of the job, there may also be one or more buttons on the form.

Abort

Aborts a job that is running or hasn't started yet. Jobs that hasn't started can always be aborted. Jobs that are already executing can only be aborted if the plug-in supports it. The button will not be visible if the plug-in doesn't supports being aborted.

Restart job

Retry a failed job with the same parameters. Sometimes the reason that a job failed can be fixed. For example, by changing the permissions on items the job needs to access. Use this button to place the job in the job queue again. It is not possible to change job parameters with one exception, if the job uses a plug-in configuration and the configuration has been changed it is possible to select if the old or new configuration values should be used.

Re-configure job

Retry a failed job with different parameters. This feature is supported by most but not all plug-ins. It is very useful when the failure is due to a misconfiguration and the job may succeed if it was configured differently.

Really run

Some plug-ins have support for a *dry-run* mode that executes the job but doesn't save any changes to the database. If the dry run completes successfully, this button can be used to run the job for real.

Close

Close the window.

Chapter 9. Reporters

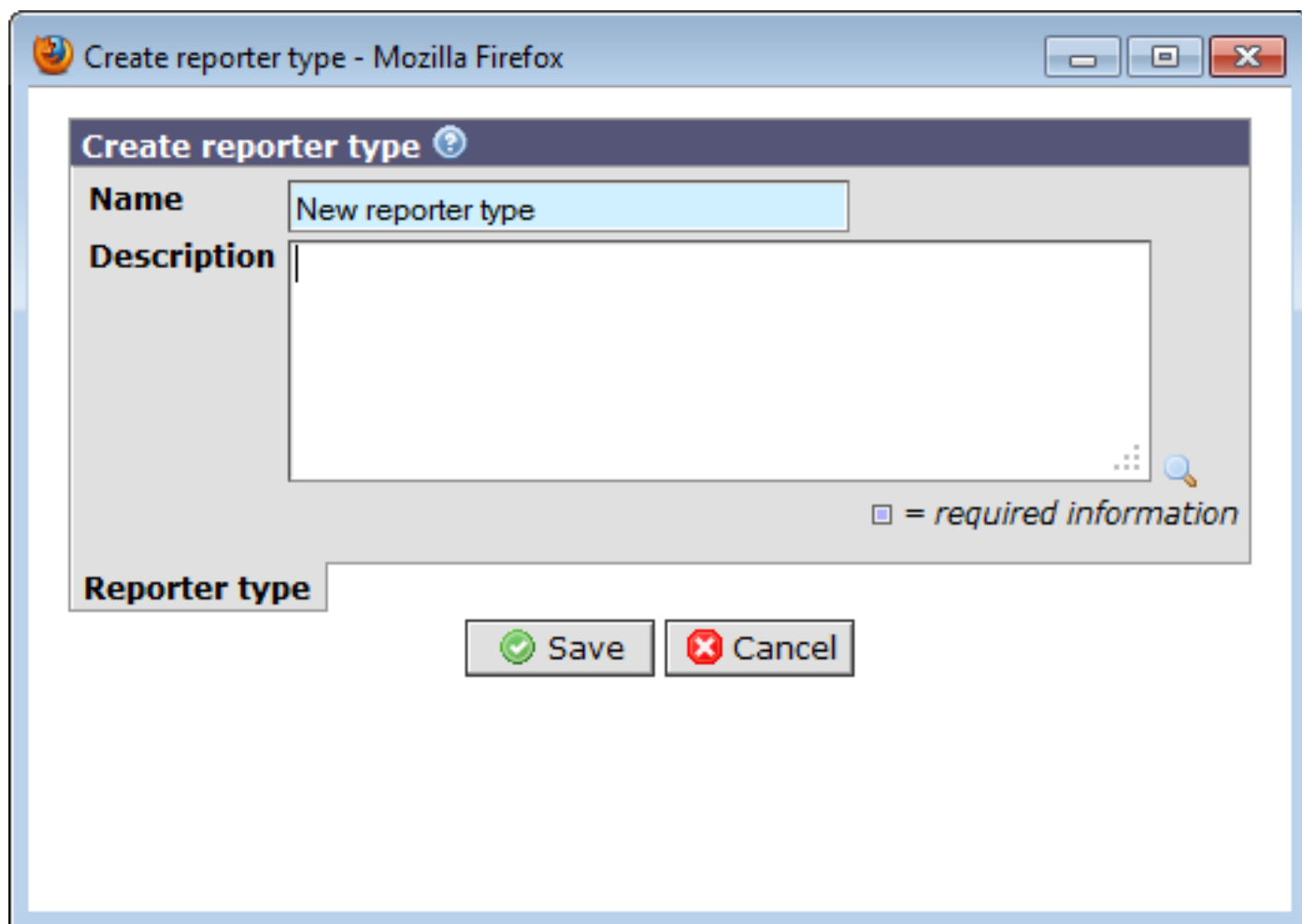
Reporter, a term coined by the MAGE object model refers to spotted DNA sequence on a microarray. Reporters are therefore usually described by a sequence and a series of database identifiers qualifying that sequence. Reporters are generally understood as the thing biologists are interested in when carrying out DNA microarray experiments.

In BASE, reporters also refer to Affymetrix Probeset ID but reporters can be used to describe genes, transcripts, exons or any other sequence entity of biological relevance.

9.1. Reporter types

Reporter Type allows classification of reporters based on their usage and qualification defined during the array design specification. You can manage the reporter types by going to Administrative Types > Reporter types.

Figure 9.1. Reporter type properties



The screenshot shows a web browser window titled "Create reporter type - Mozilla Firefox". Inside the browser, there is a form titled "Create reporter type ?". The form has two main input fields: "Name" and "Description". The "Name" field contains the text "New reporter type". The "Description" field is a large text area that is currently empty. Below the "Description" field, there is a legend indicating that a blue square icon represents "required information". At the bottom of the form, there are two buttons: "Save" (with a green checkmark icon) and "Cancel" (with a red X icon).

Name

The name of the reporter type. It is advised to define the name so that it is compatible with the MIAME requirements¹ and recommendations issues by microarray data repositories. Alternately, the local reporter type could be submitted to those repositories for term inclusion.

¹ <http://www.mged.org/Workgroups/MIAME/miame.html>

Description

A description of the reporter type.

9.2. Reporters

Go to [View Reporters](#) to view and manage the reporters.

9.2.1. Import/update reporter from files

Reporters are used to represent genes, transcripts, exons and therefore come in their thousands. To solve this problem, BASE relies on *Reporter import plug-ins*. Those need to be specifically configured to deal with a particular input file format. This input file can be typically be an Axon GAL file or an Affymetrix CSV file which both provide information about reporters and their annotations. See Chapter 18, *Import of data*(page 150) for more information about importing and Section 21.2, “Plug-in configurations” (page 186) for more information about configuring file formats.

Dealing with Affymetrix probesets

In BASE, Affymetrix probesets should be treated as reporters. The probeset ID could be stored in both the **Name** and the **External ID** fields of the reporter table. Storing the probeset ID should be enough as most analysis tools allow retrieval of updated information based on the probeset ID from web resources.

For some Affymetrix chips the associated CSV file does not list all reporters on the actual chip. This will lead to problems in later use of the affected chip types. Simply use the associated CDF file to import the missing probesets into BASE, make sure not to upgrade existing reporters when starting the plug-in.

9.2.2. Manual management of reporters

Reporters can also be created or edited manually one-by-one. This follows the same pattern as for all other items and is described in general terms in Section 5.3, “Working with items” (page 22).

Figure 9.2. Reporter properties

Edit reporter -- M200000001 ?

Name M200000001

External ID M200000001

Type - none -

Gene symbol Hoxa10

Description Homeobox protein Hox-A10 (Hox-1.8).
[Source:Uniprot/SWISSPROT;Acc:P31310]

☐ = required information

Reporter Extended properties

☒ Save ☐ Cancel

This tab shows core information that would be common to all BASE instances.

Name

The name of the reporter. This is often the same as the **External ID**.

External ID

The external ID of the reporter as it is defined in some database. The ID must be unique within BASE. The external ID is what plug-ins uses to match reporter information found in raw data files, array design files, etc.

Type

Optionally select a reporter type.

Gene symbol

The gene this reporter represents.

Figure 9.3. Extended reporter properties

Edit reporter -- M200000001

Species	<input type="text"/>
Cluster ID	<input type="text"/>
Length	<input type="text"/>
Sequence	<input type="text" value="TGGAAGCCTAGGTGGGCTGGGGCAAGCAGAAATAAAAATGAG
AGAAGGGAGATATTGTTTGGATTTCCT"/>
Vector	<input type="text"/>
Tissue	<input type="text"/>
Library	<input type="text"/>
Accession	<input type="text"/>
NID	<input type="text"/>
Chromosome	<input type="text"/>
Cytoband	<input type="text"/>
Markers	<input type="text"/>
Antibiotics	<input type="text"/>

Reporter **Extended properties**

Reporters belong to a special class whose properties can be defined and extended by system administrators. This is done by modifying the `extended-properties.xml` file during database configuration or upgrade. All fields on this tab are automatically generated based on this configuration and can be different from one server to the next. See Section 20.2, “Installation instructions” (page 167) and Appendix C, *extended-properties.xml reference* (page 426) for more information.

Note

It is possible to configure the extended properties so that links to the primary external databases can be made. For example, the **Cluster ID** is linked to the UniGene database at NCBI².

9.2.3. Deleting reporters

Deleted reporters cannot be restored

Reporters are treated differently from other items (e.g biosources or protocols) since they does not use the trashcan mechanism (see Section 5.5, “Trashcan” (page 34)). The deletion happens immediately and is an unrecoverable event. BASE will always show a warning message which you must confirm before the reporters are deleted.

Reporters which has been referenced to from reporter lists, raw data, array designs, plates or any other item cannot be deleted.

Batch deletion

A common problem is to delete reporters that has been accidentally created. The regular web interface is usually no good since it only allows you to delete a limited number of reporters at a time. To solve this problem the reporter import plug-in can be used in delete mode. You can use the same file as you used when importing. Just select the **delete** option for the **mode** parameter in the configuration wizard and continue as usual. If the plug-in is used in delete mode from a reporter list it will only remove the reporters from the reporter list. The reporters are not deleted from the database.

Note

It may be a bit confusing to delete things from an import plug-in. But since plug-ins can only belong to one category and we wanted to re-use existing file format definitions this was our only option.

9.3. Reporter lists

BASE allows for defining sets of reporters for a particular use, for instance to define a list of reporters to be used on an array. There are several ways to do so:

- Use the **New reporter list** button on the View Reporters page. This creates a reporter list with the current selection or filtered out reporters.
- Use the **New** button on the View Reporter lists page. This creates an initially empty reporter list that can be filled later.
- Use the **New reporter list** button on the **Features** tab on the single-item view page for an array design. This creates a reporter list with all or some of the reporters used on the array design.
- Use the **New reporter list** button on the **Raw data** tab on the single-item view page for a raw bioassay. This creates a reporter list with all or some of the reporters used by the raw bioassay.
- Use the **New reporter list** button on the **Spot data** tab on the single-item view page for a bioassay set in the experiment analysis section. This creates a reporter list with all or some of the reporters used by the current bioassay set.
- Use the **New reporter list** button on the **Reporter search** tab in experiment explorer. This creates a reporter list with all or some of the reporters used by the current bioassay set.

² <http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=unigene>

Figure 9.4. The Create reporter list called from the reporters page.

Name

The name of the reporter list.

External ID

An optional external ID. This value is useful, for example, for a tool that automatically updates the reporter list from some external source. It is not used by BASE.

Which reporters

Select one of the options for specifying which reporters should be included in the list. This option is only available when creating a new reporter list, not when editing an existing list. The default is to create a list with *all* reporters that are in the current list.

Description

A description of the reporter list.

Tip

To add or remove reporters to the list use the **Reporters** tab on the single-item view page of a reporter list. This tab lists all reporters in the list and there are functions for removing, adding and importing reporters to the list.

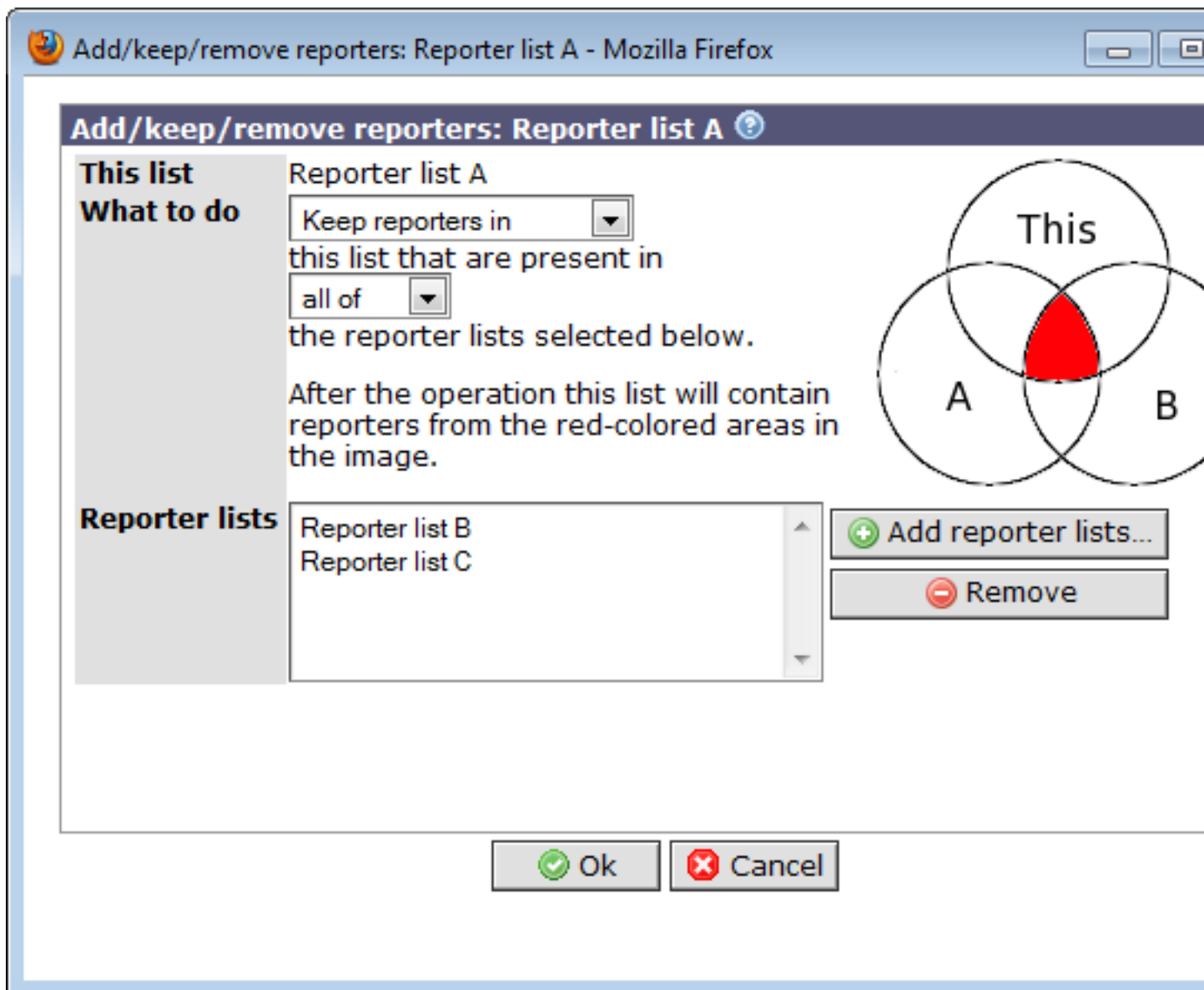
9.3.1. Merging reporter lists

It is possible to modify a reporter list by merging it with other reporter lists. Go to the single-item view page for a reporter list. There should be three buttons in the toolbar corresponding to three main merge variants:

- **Union:** Add all reporters from the source lists to the destination list.
- **Intersection:** Keep only reporters that are present in all selected lists.
- **Complement:** Keep only reporters that are unique for the destination list.

All three buttons open up the same dialog with different default selections.

Figure 9.5. Merge reporter lists



This list

Shows the name of the current reporter list.

What to do

In this drop-down list you can select if you want to *add*, *keep* or *remove* reporters from the selected list based on if a reporter is present in one or all of some other reporter lists (selected

below). The image to the right gives an overview what will happen for the currently selected alternatives. The red-colored areas indicate which reporters that are included in the final list. The white-colored areas indicate reporters that doesn't pass the filter and are excluded from the final list.

Reporter lists

You will need to select at least one more list to merge with.

Chapter 10. Annotations

10.1. Annotation Types

BASE has been engineered to closely map the MIAME concepts¹. However, since MIAME is focused on microarray processing workflow, information about the description biological samples themselves was left out. BASE users are free to annotate biomaterials (and most BASE items) as they wish, from basic free text description to more advanced ontology based terms. To accommodate the annotation needs of users eager with detailed sample annotations and also the needs of very different communities, BASE provides a mechanism that allows a high level of annotation customization. BASE allows to create descriptive elements for both quantitative annotation and qualitative annotation of Biomaterials via the *Annotation Type mechanism*. Actually, annotation types can be used to annotate not only *Biomaterials* but almost all BASE items, from *Plates* to *Protocols* and *Bioassay sets*.

Go to [Administrate Types](#) Annotation types to manage your annotation types.

To create a new annotation type, click on the **New...** button. This behaves differently than other buttons found elsewhere and you must select one of the 9 different types which can be split in 4 main groups.

- **Integer**, **Long**, **Float** and **Double** for numerical annotation types.
- **String** and **Text** for textual annotation types. The difference is that **String**:s can have a maximum length of 255 characters and can have an attached list of predefined value. **Text** annotation types have no practical limit and are always free-text.
- **Boolean** for declaring annotation types that can take one TRUE/FALSE values.
- **Date** and **Timestamp** for declaring annotation types used as calendar/time stamps.

Note

These distinctions matter essentially to database administrators who need fine tuning of database settings. Therefore, creation of annotation type should be supervised by system administrators.

The **Edit annotation type** window is opened in a pop-up. It contains several tabs.

¹ <http://www.mged.org/Workgroups/MIAME/miame.html>

10.1.1. Properties

Figure 10.1. Annotation type properties

Edit annotation type -- Time ?

Value type Integer

Name Time

External ID

Multiplicity 1 0 or empty = unlimited

Default value

Required for MIAME ☐

Protocol parameter ☐

Description

☐ = *required information*

Annotation type Options Item types Units Categories

Save Cancel

Name

The name of the annotation type.

External ID

An ID identifying this annotation type in an external database. This value can be used by tools that need to update annotation types in BASE from external sources. The value does not have to be unique and is not used by BASE.

Multiplicity

The maximum number of values that can be entered for this annotation type. The default is 1. A value of 0, means that any number of values can be used.

Default value

A value that can be used as the default when adding values.

Required for MIAME

If a value must be specified for this annotation type in order for the experiment to be compliant with MIAME.

Protocol parameter

If the annotation type is a protocol parameter. As a protocol parameter an item can only be annotated if a protocol that includes this parameter has been used. See Section 13.1, “Protocol parameters” (page 97) for more information.

Description

A short textual description to clarify the usage.

10.1.2. Options

Figure 10.2. Annotation type options

The screenshot shows a web browser window titled "Edit annotation type -- Time - Mozilla Firefox". Inside the browser is a dialog box titled "Edit annotation type -- Time" with a help icon. The dialog box contains the following elements:

- Interface:** Three radio buttons: "text box" (selected), "selection list", and "radiobuttons/checkboxes".
- Min value:** A text input field with the note "empty = no limit".
- Max value:** A text input field with the note "empty = no limit".
- Input box width:** A text input field containing the value "40".
- Values:** A large text area for entering values.
- Instruction:** "One enumeration value per line" below the text area.
- Tabs:** A row of tabs: "Annotation type", "Options" (selected), "Item types", "Units", and "Categories".
- Buttons:** "Save" (with a green checkmark icon) and "Cancel" (with a red X icon) buttons at the bottom.

The available options in this tab depends on the type of annotation type, eg. if is a string, numeric or another type.

Interface

Select the type of graphical element to use for entering values for the annotation type. You can select between three different options:

- **text box**: The user must enter the value in a free-text box.
- **selection list**: The user must select values from a list of predefined values.
- **radiobuttons/checkboxes**: The user must select values by marking checkboxes or radiobuttons.

The last two options requires that a list of values are available. Enter possible values in the **Values** which will be activated automatically.

Tip

In term of usability, a drop-down list can be more easily navigated especially when the number of possible values is large, and because selection-list and drop-down list allow use of arrow and tab for selection.

Min/max value

Available for numerical annotation types only. Specifies the minimum and maximum allowed value. If left empty, the bound(s) are undefined and any value is allowed.

Max length

The maximum allowed length of a string annotation value. If empty, 255 is the maximum length. If you need longer values than that, use a *text* annotation type.

Input box width/height

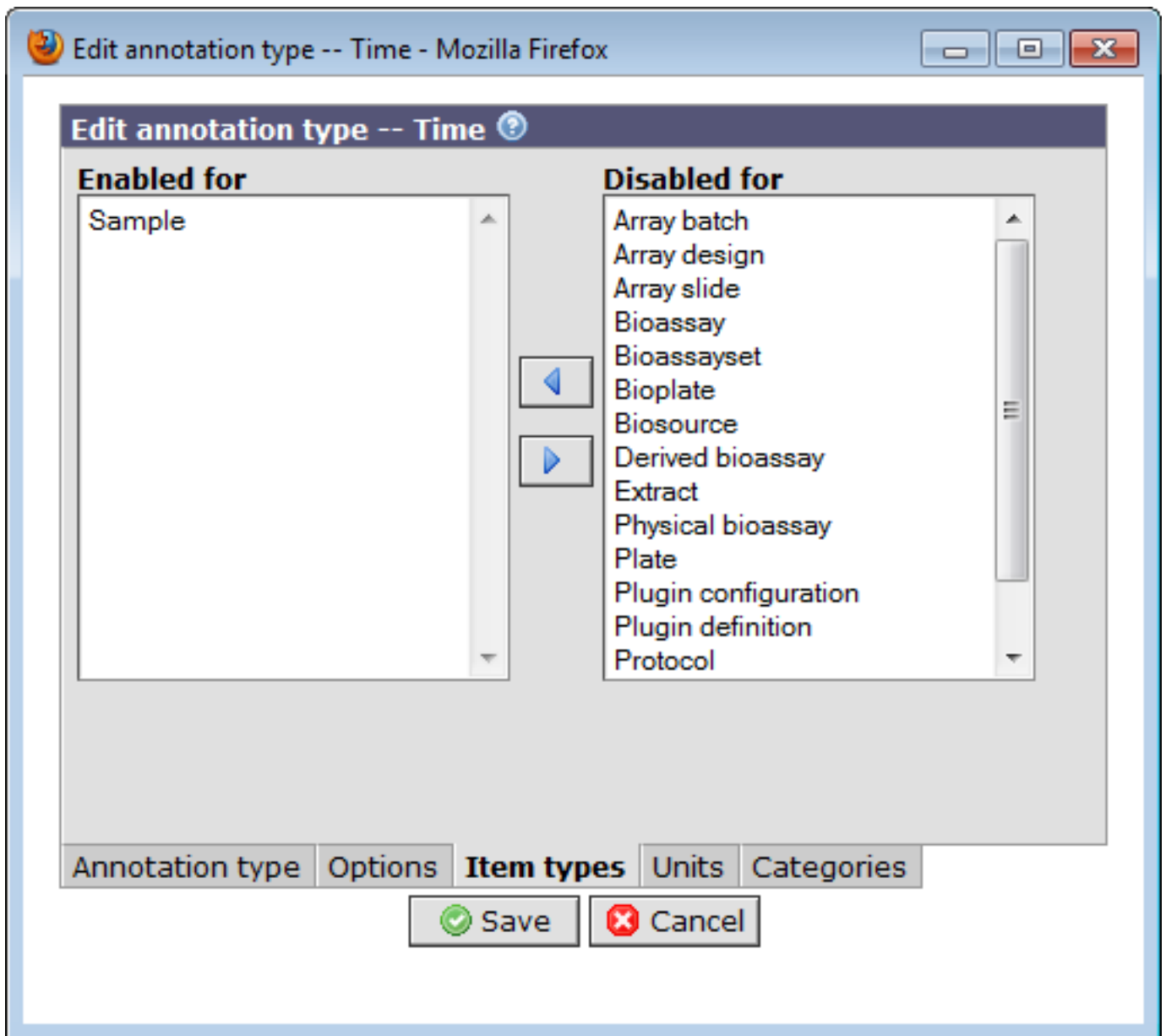
A suggested display width and height of the element used for input. These values are ignored in the current implementation.

Values

A list of predefined values that the user is allowed to select from. This option is only activated if the **Interface** option is set to **selection list** or **radiobuttons/checkboxes**. Actual values can be supplied using one line for each value (a return entry is used as separator).

10.1.3. Item types

Figure 10.3. Annotation type items



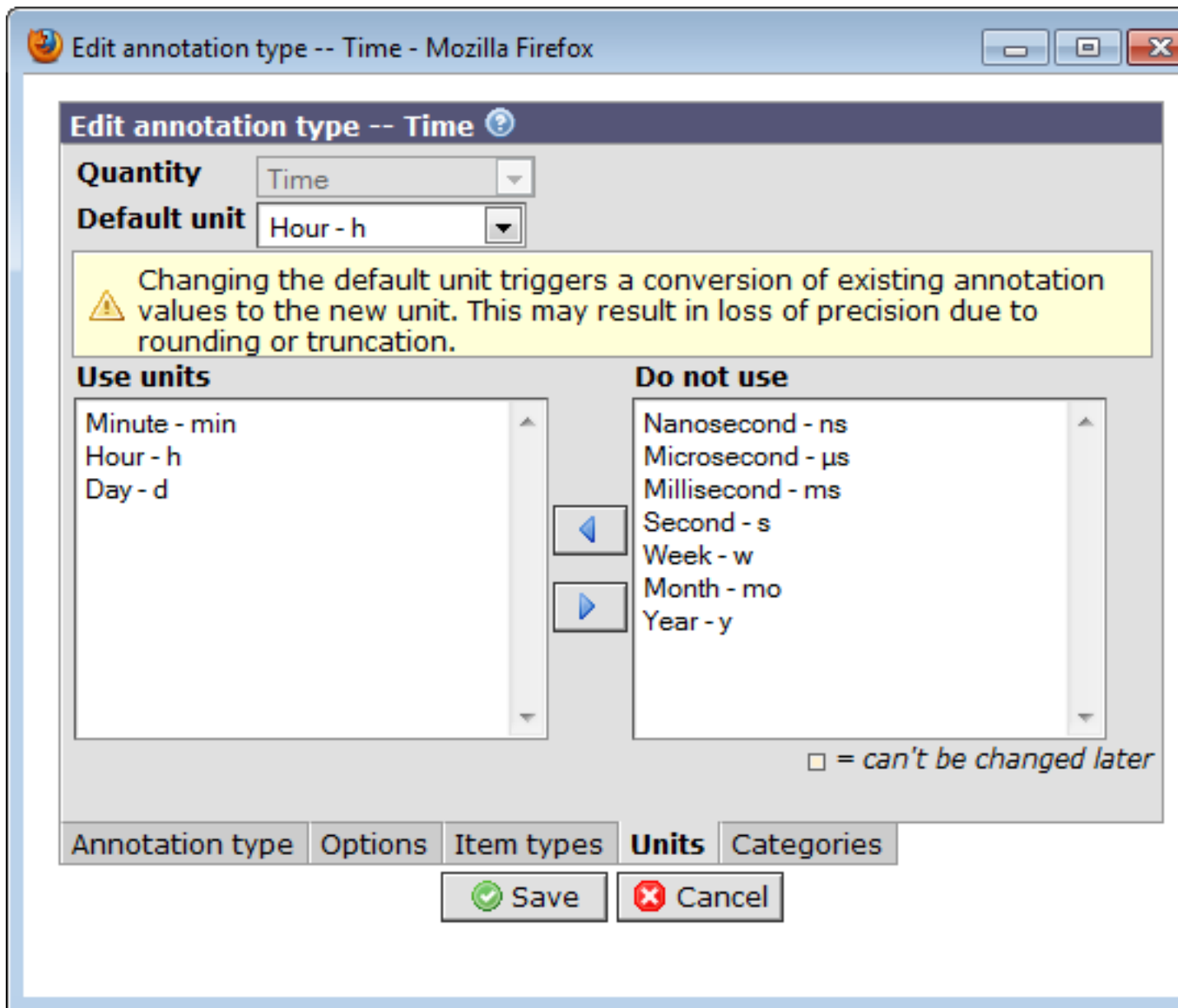
On this tab you select the item types that you wish to annotate with the annotation type. Simply use left and right buttons to move selection options between the **Enabled for** and **Disabled for** lists.

Note

If the annotation type has been marked as a **Protocol parameter**, these settings are ignored, with one exception. If you wish to view parameter values in the list view for a specific item type you must select the item type here. Otherwise the parameter will not be present as a displayable column.

10.1.4. Units

Figure 10.4. Annotation type units



Numerical annotation types can optionally be given a quantity and unit.

Quantity

Select which quantity to use for the annotation type. If you don't want to use units, select the *do not use units* option.

The quantity can't be changed later

Once a quantity has been selected and saved for an annotation type, it is not possible to change it to another quantity.

Default unit

This list will be populated with units from the selected quantity. You must select one default unit which is the unit that is used if a user leaves out the unit when annotating an item. The selected unit is also the unit that is used internally when storing the values in the database.

Do not change the default unit

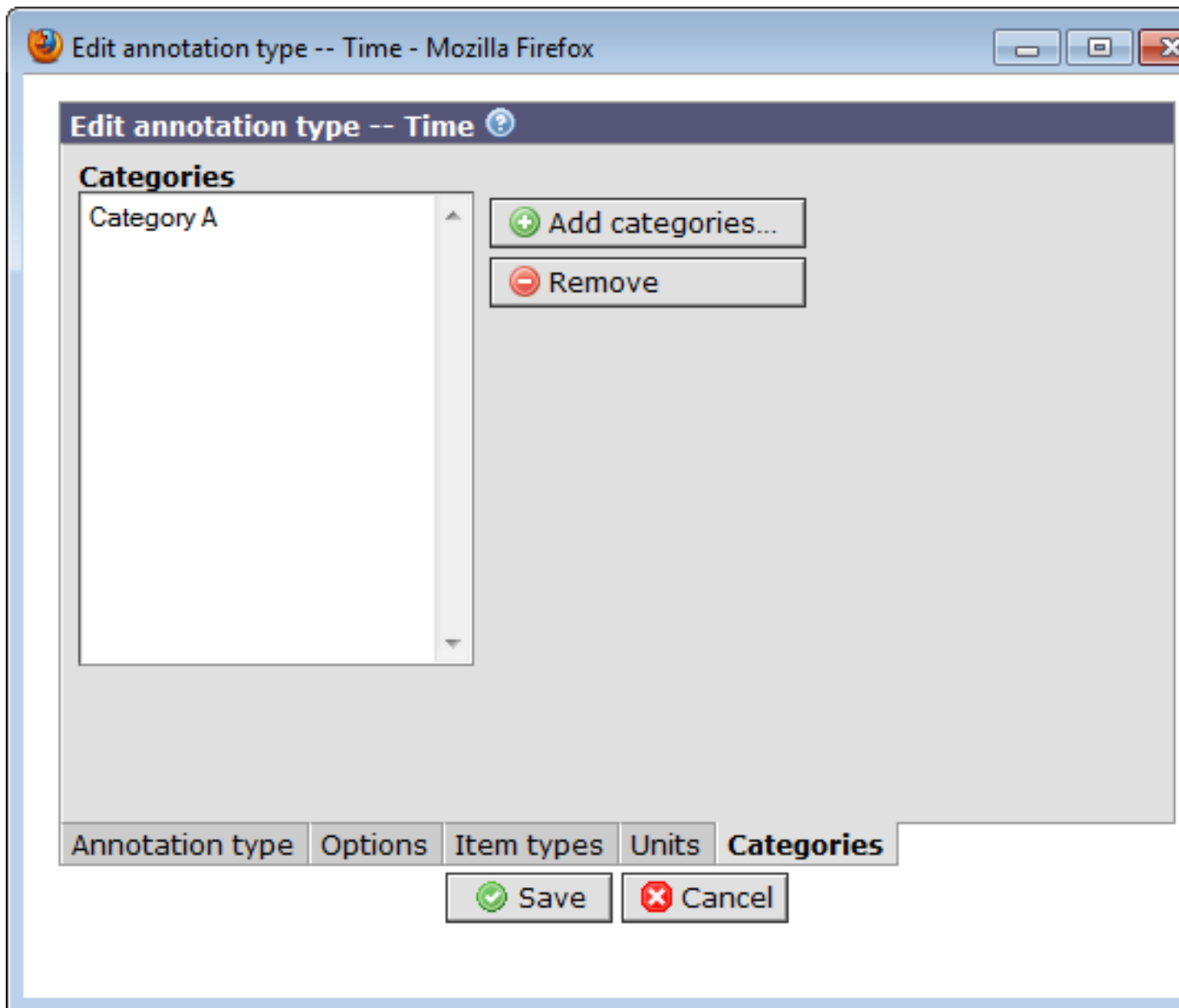
If you change the default unit for an existing annotation type, all annotation values that exists for it, must be converted to the new unit. This may result in loss of precision due to rounding/truncation errors.

Use units

By default, all units of the selected quantity can be used when annotating items. If you want, you may force the users to use some specific units by moving units into the *Use units* list. This is recommended since the range of available units is usually quite large. For example, if the weight of something is normally measured in milligrams, it may make sense to leave out kilograms, and only use microgram, milligram and gram.

10.1.5. Categories

Figure 10.5. Annotation type categories



Annotation type can be grouped together by placing them in one or more categories. This enhances display by avoid overcrowding the list of annotation types presented to users. It also allows to improve the display of information.

The **Categories** list displays the currently associated categories. Use the **Add categories** button to add more categories, or the **Remove** button to remove the selected categories.

Create categories with the same name as item subtypes

If you, for example, have defined multiple subtypes of extracts (see Chapter 12, *Item subtypes* (page 92)), it usually so that some annotation types are only intended for one subtype while some other annotation types are only intended for the other subtype. If you create categories with the same name as the item subtypes, BASE will automatically select the corresponding category when annotating an extract with a subtype. This makes the interface cleaner and easier to use since irrelevant annotation types are hidden. Note that it is possible for annotations to be part of more than one category so it is also possible to define annotation types that are intended for all types of extracts by including them in both categories.

10.2. Annotating items

Entering annotation values follow the same pattern for all items that can have annotations. They all have a **Annotations & parameters** tab in their edit view. On this tab you can specify values for all annotation types assigned to the type of item, and all parameters that are attached to the protocol used to create the item. Some items, for example *biosources* and *array designs* cannot have a protocol. In their case the tab is labelled **Annotations**.

Figure 10.6. Annotating a sample

The screenshot shows a web browser window titled "Edit sample -- Sample A.00h - Mozilla Firefox". Inside, there's a dialog box titled "Edit sample -- Sample A.00h". The dialog has a "Categories" dropdown menu currently set to "- all -". Below it is a list of annotation types: "Temperature" (marked with an 'x' and blue angle brackets) and "Time" (marked with an 'x'). To the right of this list is a form for "Temperature (Float)" with a text input field containing "25.3" and a unit dropdown menu set to "°C". Below the list, a legend states: "x = Has value(s) <> = Protocol parameter". At the bottom of the dialog, there are four tabs: "Sample", "Parents", "Annotations & parameters" (which is the active tab), and "Inherited annotations". Below the tabs are two buttons: "Save" (with a green checkmark icon) and "Cancel" (with a red X icon).

Click on an entry in the list of annotation types to show a form for entering a value for it to the right. Depending on the options set on the annotation type the form may be a simple free text field, a list of checkboxes or radiobuttons, or something else.

Annotation types with an **X** in front of their names already have a value.

Annotation types marked with angle brackets (**<>**) are protocol parameters.

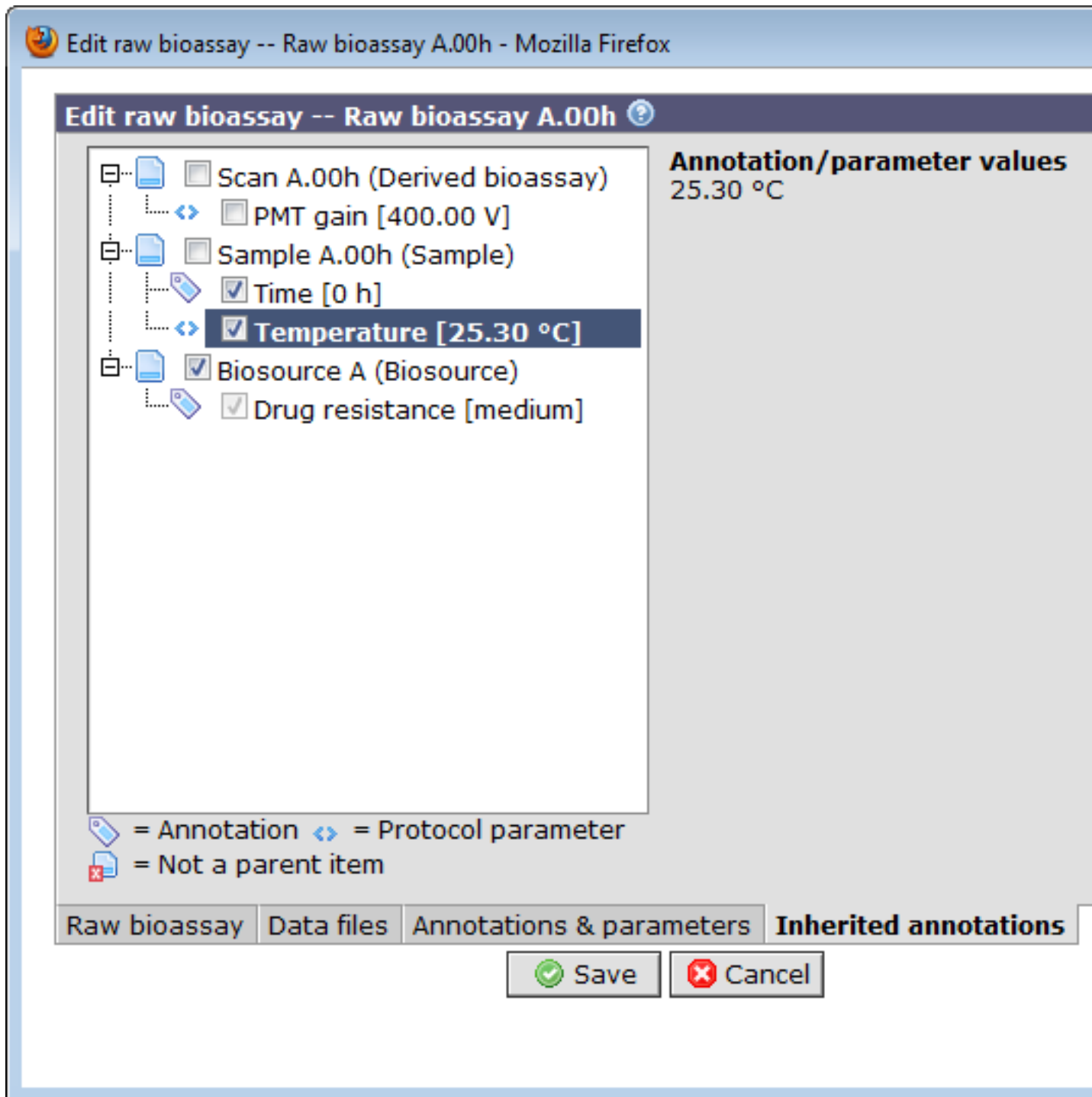
Select an option in the **Categories** list to filter the annotation types based on the categories they belong to. This list contains all available categories, and three special ones:

- **all**: Display all annotation types

- **protocol parameters:** Display only those annotation types that are parameters to the current protocol.
- **uncategorized:** Display only annotation types that has not been put into a category.

10.2.1. Inheriting annotations from other items

An item may inherit annotations from any of its parent items. E.g. an extract can inherit annotations from the sample or biosource it was created from. This is an important feature to make the experimental factors work. Annotations that should be used as experimental factors must be inherited to the raw bioassay level. See Section 17.3.2, “Experimental factors” (page 147) for more information about experimental factors.

Figure 10.7. Inheriting annotations from a parent item

On this screen is a tree-like structure in two levels. The first level lists all parent items which has at least one annotations. The second level lists the annotations and protocol parameters for the item. Selecting an item in the first level will inherit all annotations from that item, including those that you maybe add later. Selecting an annotation or protocol parameter at the second level will inherit only the selected one.

Note

- The inheritance is implemented by reference. This means that if you change the value of an annotation the new value is automatically picked up by those inheriting it.
- You cannot inherit annotations from an item which does not have annotations.

- If you delete an annotation from a parent item, the inheritance will be lost, even if you later add a value again.

Warning

If you rearrange links to parent items after you have specified inheritance, it may happen that you are inheriting annotation from non-parent items. This will be flagged with a warning icon in the list, and must be fixed manually. The item overview tool is an excellent help for locating this kind of problems. See Section 5.6, “Item overview” (page 36).

10.2.2. Mass annotation import plug-in

BASE includes a plug-in for importing annotations to multiple items in one go. The plug-in read annotation values from a simple column-based text file. Usually, a tab is used as the delimiter between columns, but this is configurable. The first row should contain the column headers. One column should contain the name or the external ID of the item. The rest of the columns can each be mapped to an annotation type and contains the annotation values. If a column header exactly match the name of an annotation type, the plug-in will automatically create the mapping, otherwise you must do it manually. You don't have to map all columns if you don't want to.

Each column can only contain a single annotation value for each row. If you have annotation types that accept multiple values you can map two or more columns to the same annotation type, or you can add an extra row only giving the name and the extra annotation value. Here is a simple example of a valid file with comma as column separator:

```
# 'Time' and 'Age' are integer types
# 'Subtype' is a string enumeration
# 'Comment' is a text type that accept multiple values
Name,Time (hours),Age (years),Subtype,Comment
Sample #1,0,0,alfa,Very good
Sample #2,24,0,beta,Not so bad
Sample #2,,,Yet another comment
```

The plug-in can be used with or without a configuration. The configuration keeps the regular expressions and other settings used to parse the file. If you often import annotations from the same file format, we recommend that you use a configuration. The mapping from file columns to annotation types is not part of the configuration, it must be done each time the plug-in is used.

The plug-in can be used from the list view of all annotatable items. Using the plug-in is a three-step wizard:

1. Select a file to import from and the regular expressions and other settings used to parse the file. In this step you also select the column that contains the name or external ID the items. If a configuration is used all settings on this page, except the file to import from, already has values.
2. The plug-in will start parsing the file until it finds the column headers. You are asked to select an annotation type for each column.
3. Set error handling options and some other import options.

Chapter 11. Experimental platforms and data file types

11.1. Platforms

An experimental platform in BASE can be seen as an item representing the set of data file types that are produced or needed by a given experimental setup. For example, the Affymetrix platform (as defined in BASE) uses CEL files for raw data and CDF files for array design information. The concept of a platform is also tightly coupled to the ability to keep data in files instead of importing it to the database. When you have selected a platform for a raw bioassay or an array design, you also know which files you should provide.

BASE comes pre-installed with three platforms.

- A generic platform that can be used with almost any type of data that can be imported into the database from simple column-based text files.
- The Affymetrix platform which keep data in CEL and CDF files instead of importing into the database.
- A sequencing platform which uses GTF files to define array designs and FPKM counts as raw data.

Other platforms, such as Illumina, are available as non-core plug-in packages, see Section 3.2, “BASE plug-ins site” (page 10). An administrator may define additional platforms and file types.

You can manage platforms going to [Administrative > Platforms > Experimental platforms](#).

Figure 11.1. Platform properties

Create platform ?

Name

External ID

File-only ☒ no ☐ yes

Raw data type

Channels

Description

☐ = required information
☐ = can't be changed later

Platform **Data file types**

☒ Save ☐ Cancel

Name

The name of the platform

External ID

An ID that is used to identify the platform. The ID must be unique and can't be changed once the platform has been created.

File-only

If the platform is a file-only platform or not. File-only platforms can't have it's data imported into the database. This option can't be changed once the platform has been created.

Raw data type

If you have selected **file-only=no**, you may select a raw data type. This will lock this platform to the selected raw data type. If you select **- any -**, raw data of any raw data type can be used. This option can't be changed once the platform has been created.

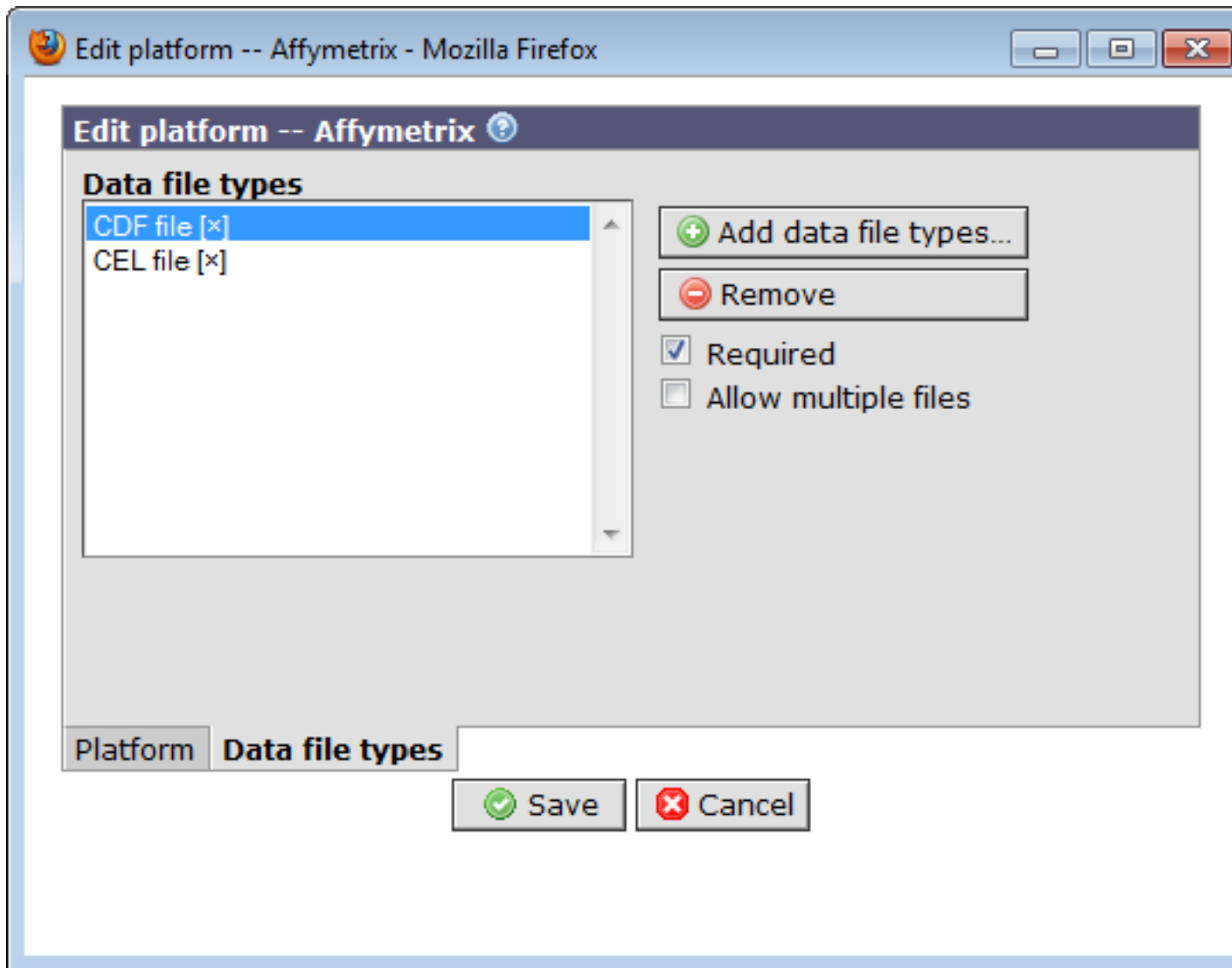
Channels

If you have selected **file-only=yes**, you must enter the number of channels the platform uses. This information is needed in the analysis module of BASE to create the proper database tables. This option can't be changed once the platform has been created.

Description

A description of the platform.

Figure 11.2. Select data file types



Data file types

This list contains the file types already associated with this platform. An [x] at the end of the name indicates a required file.

Required

Mark this checkbox to indicate that the file is required for the platform.

Note

The required flag is not enforced when creating items. It is used for generating warnings when validating an experiment and can be checked by plug-ins that may need the file in order to run.

Allow multiple files

Mark this checkbox if it is possible to have more than one file of the given type. In most cases, a single file is used, but some platforms (for example Illumina) may split data into multiple files.

Add data file types

Opens a popup window that allows you to add more file types to the platform.

Remove

Removes the selected file types from the platform.

11.2. Platform variants

It is possible for an administrator to define variants of a platform. The main purpose for this is to be able to select additional file types that are only used in some cases. The file types defined by the parent platform are always inherited by the variants.

You can create new variants from the single-item view of a platform. This view also has a **Variants** tab which lists all variants that has been defined for a platform.

11.3. Data file types

Each file type used by a platform must be registered as a **data file type**. For example, **CEL** and **CDF** files are file types used by the **Affymetrix** platform. There are several purposes of a data file type:

- Describe the file type and make it identifiable. Each file type must have a unique ID which makes it possible to find out if a specific file has been added to an item. For example, to find the CEL file of a raw bioassay.
- Connect a specific file type with a generic file type. For example, the CEL file is used to store raw data for an experiment. Another platform may use a different file type. Both file types are of the generic type **raw data**. This makes it possible for client applications or plug-ins to find the raw data for an experiment without actually knowing which file types that are used on various platforms.
- Make it possible to validate and extract metadata from attached files. This is done by extensions. Currently, BASE ships with extensions for CEL, CDF and GTF files, but the administrator may have installed extensions for other file types. See Section 26.8.8, “Fileset validators” (page 281) for more information about creating extensions.

You can manage data file types by going to [Administrative > Platforms > Data file types](#).

Figure 11.3. Data file type properties

Create data file type ?

Name CDF file

External ID affymetrix.cdf

Item type Array design ▼

File extension cdf

Generic file type Reporter map ▼

Description Affymetrix CDF file

□ = required information
□ = can't be changed later

Data file type

Save Cancel

Name

The name of the file type.

External ID

An ID that is used to identify the file type. The ID must be unique and can't be changed once the file type has been created.

Item type

The type of item files of this file type can be attached to. This option can't be changed once the file type has been created.

File extension

The commonly used file extension for files of this type. Optional.

Generic type

The generic type of data that files of this type contains. For example, CEL files contains raw data and CDF files contains a reporter map (in BASE terms).

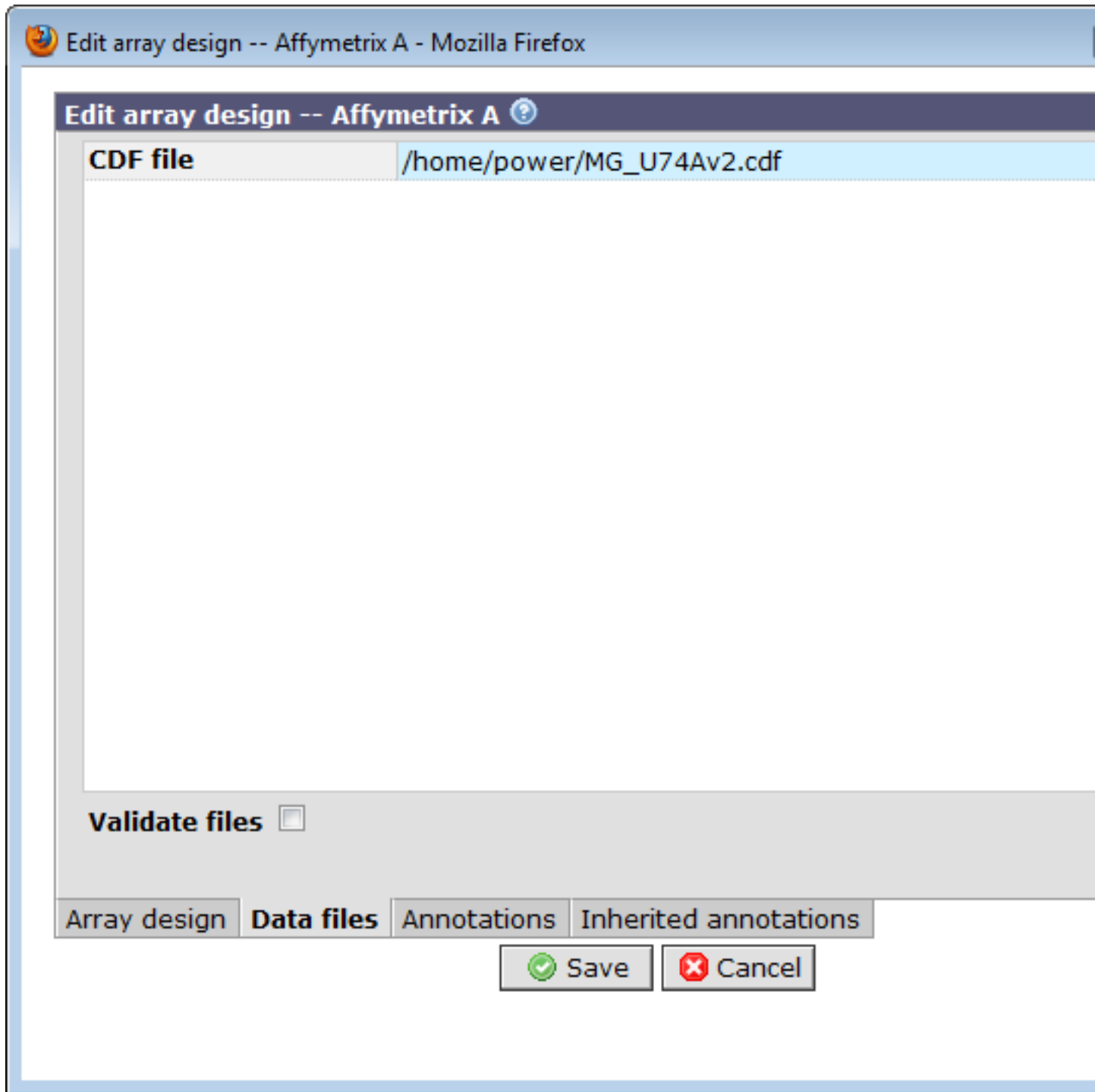
Description

A description of the file type.

11.4. Selecting files for an item

Selecting files for an item follows the same pattern for all items that supports it. They all have a **Data files** tab in their edit view. On this tab you can select files for all file types that are defined by the platform or subtype of the item.

Figure 11.4. Selecting files for an array design



The list contains all file types that are defined by by the platform or subtype that is selected on the **Properties** tab. Use the **Browse** or **Add** icons to select files from the file manager. Note that for some file types only a single file can be selected, but for other file types multiple files are allowed. The dropdown list contains recently used files as well as an option to clear the selected file.

Validate files

Mark this checkbox if you want to validate and extract metadata from the selected files. The checkbox is automatically checked if changes are made.

Note

Validation and metadata extraction is performed by extensions. The checkbox is only visible if there is at least one installed extension that supports validation of the current file types.

Chapter 12. Item subtypes

Several of the main item types in BASE can be subclassified by subtypes. This includes for example, *biosources*, *samples* and *extracts* as well as *protocols*, *hardware* and *software*. One of the main reasons for subtypes is to group items together and allowing users to filter out items that are or no interest in the current context. For example, the protocol selection list when creating an extract is filtered to only show *Extraction* protocols.

The filtering can be even more fine-grained by linking subtypes to each other. For example, in a standard BASE installation there are two extract subtypes: *Labeled extract* and *Library*. The labeled extract subtype is linked with the *Labeling* protocol subtype and the library subtype is linked with the *Library preparation* protocol subtype. So by assigning the correct subtype to a protocol, BASE knows which one that should be used when creating a labeled extract and which one should be used when creating a library.

Project default items

This feature can be a real time-saver when used together with project default items (see Section 6.2.4, “Default items” (page 46)). When creating new items (eg. a biomaterial) BASE will search among the default items in the active project for protocols, hardware, software, etc. and automatically select the best match based on the subtype of the new item.

Subtypes are typically added to a BASE server by an administrator and can be managed from Administrate Types Item subtypes.

12.1. Item subtype properties

Figure 12.1. Item subtype properties

Edit item subtype -- Scan

Name Scan

Main item type Derived bioassay

Description A hybridization that has been scanned to produce one or more images.

Related subtypes

Derived bioassay	- none -	Select...
Extract	Labeled extract	Select...
Hardware	Scanner	Select...
Physical bioassay	Hybridization	Select...
Protocol	Scanning	Select...
Software	- none -	Select...

☐ = required information

Item subtype File types

Save Cancel

Name

The name of the subtype.

Main item type

Select the main item type that the subtype can be used on. This can't be changed later.

Description

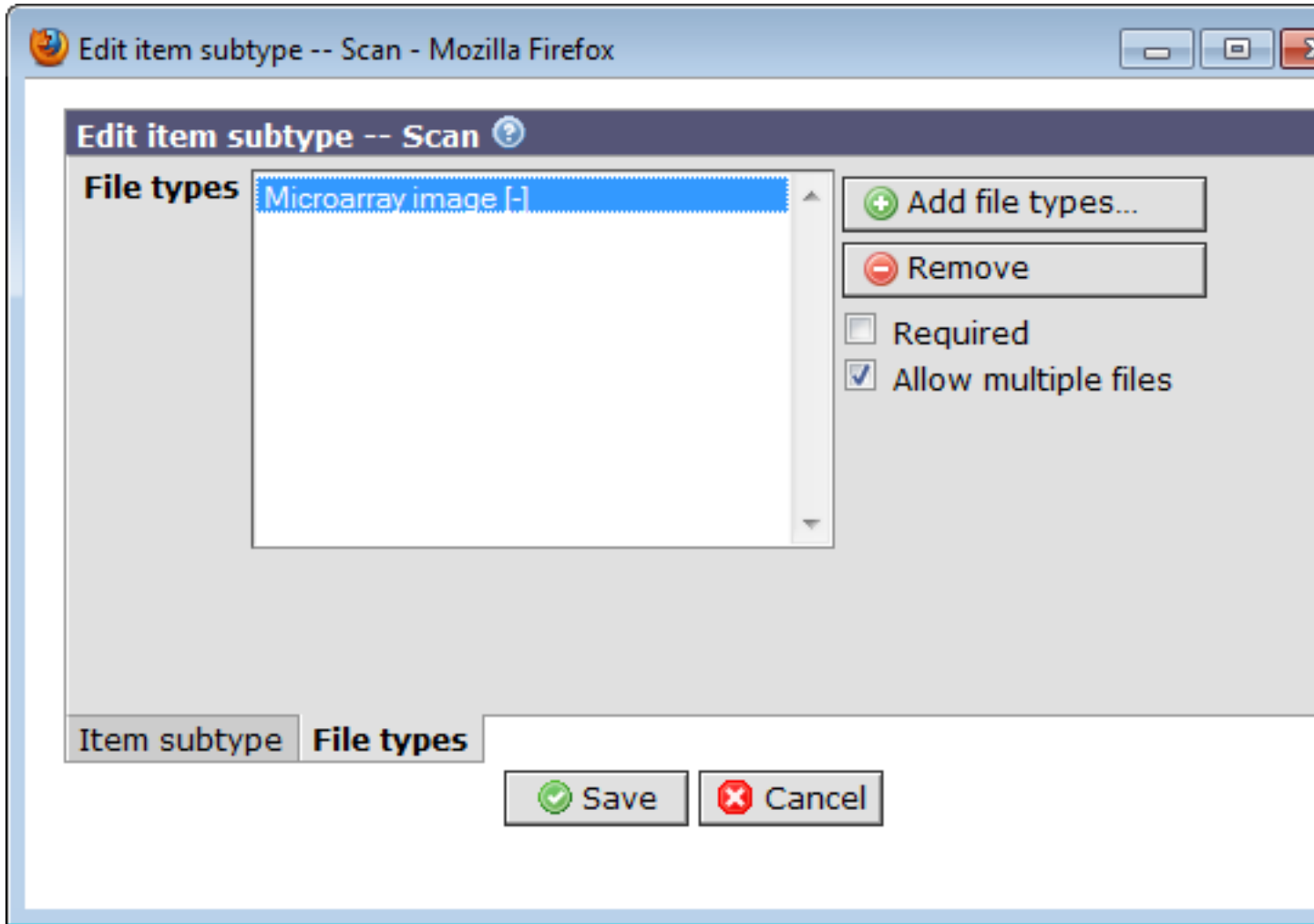
An optional description of the subtype.

Related subtypes

This section allows you to select other subtypes that are related to the current subtype. The possible options depend on which the main item type is. For example, for *biosource* there are no other related subtypes, but a *derived bioassay* can have a related physical bioassay and extract as well as protocol, hardware and software subtypes.

12.2. File types

Figure 12.2. Item subtype file types



If the main item type for a subtype is a *derived bioassay* (which has support for linking of data files) it is possible to select data file types (see Section 11.3, “Data file types” (page 88)) that can be used for items of the given subtype. This is very similar to how the selected platform of a raw bioassay or array design dictates which files types that can be used.

File types

The list contains the file types that are currently associated with the item subtype.

Required

Mark this checkbox to indicate that the file is required for all items of the given subtype.

Note

The required flag is not enforced when creating items. It is used for generating warnings when validating an experiment and can be checked by plug-ins that may need the file in order to run.

Allow multiple files

Mark this checkbox if it is possible to have more than one file of the given type. In most cases, a single file is used, but some subtypes, for example Scan, may generate multiple images.

Add file types

Click on this button to open a popup window for selecting more file types to associated with the item subtype.

Remove

Click on this button to remove the association with the selected file types.

Chapter 13. Protocols

A protocol is a document describing some kind of process that is typically performed in the lab to create an item of some type (for example, how to create an extract from a sample). It can also represent a procedure for running software programs to extract measured data (for example, from a scanned microarray image). A protocol in the simplest form is just a name with an optional link to a file. The file may for example be a PDF or some other document with a more detailed description.

Protocols are typically added to a BASE server by an administrator and can be managed from Administrate Protocols.

Figure 13.1. Protocol properties

The screenshot shows a web browser window titled "Edit protocol -- Extraction A - Mozilla Firefox". Inside the browser is a form titled "Edit protocol -- Extraction A". The form has several input fields: "Name" with the value "Extraction A", "Type" with a dropdown menu showing "Extraction", "External ID" (empty), "File" with a dropdown menu showing "- none -" and a "Select..." button, and "Description" (a large text area). At the bottom of the form are three tabs: "Protocol", "Parameters", and "Annotations". Below the tabs are "Save" and "Cancel" buttons. A legend at the bottom right indicates that a blue square icon means "required information".

This tab allows users to enter essential information about a protocol.

Name

The name of the protocol.

Type

The protocol type of the protocol. The list may evolve depending on additions by the server administrator. Selecting the proper protocol type is important and enables BASE to automatically guess the most likely protocol when creating new items. See Chapter 12, *Item subtypes* (page 92) for more information.

External ID

An ID identifying this protocol in an external database. The value does not have to be unique.

File

A document containing the protocol description, e.g. PDF documents from kit providers to the protocol. Use the **Select** button to select or upload a file.

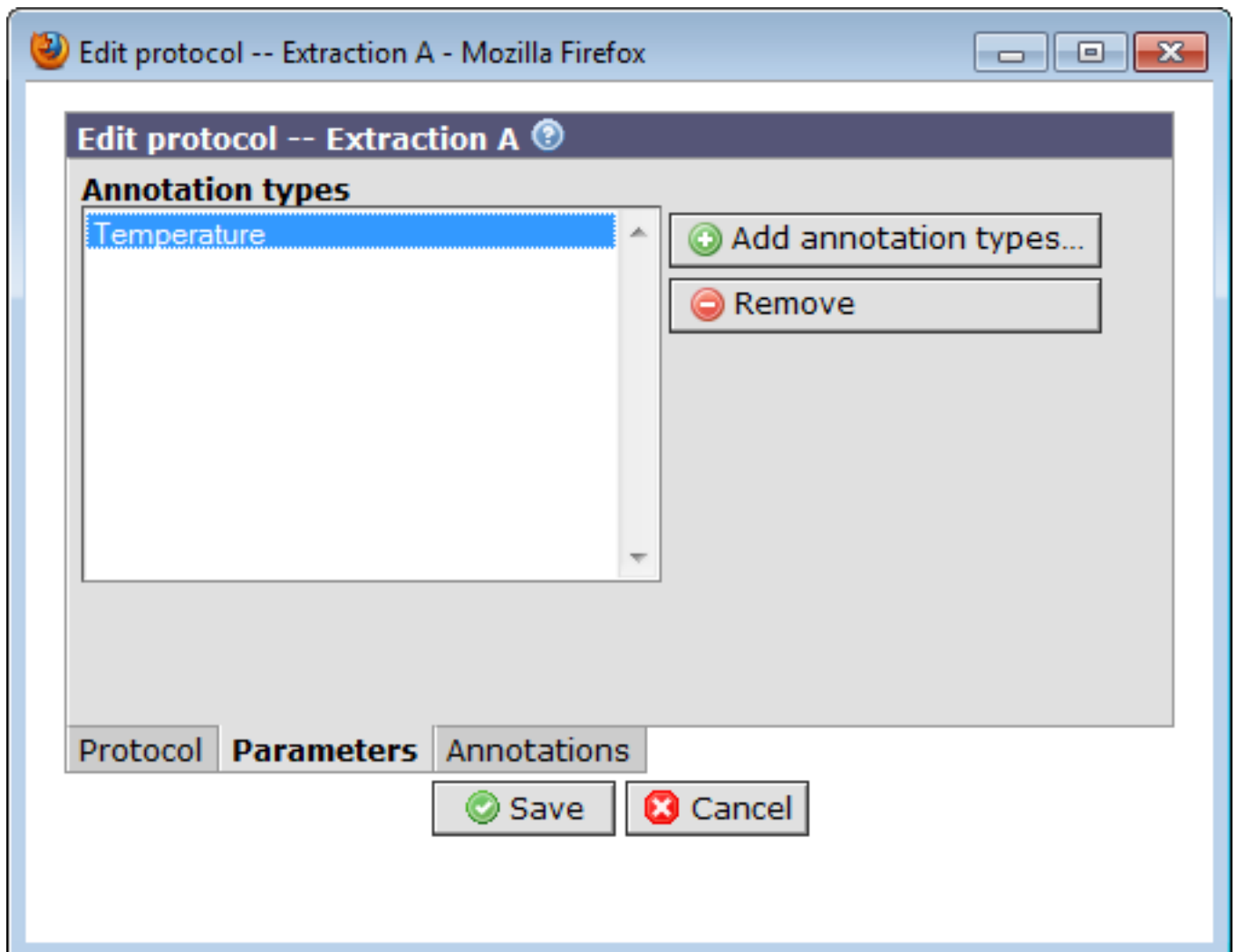
Description

A description of the protocol.

13.1. Protocol parameters

BASE users may declare parameters attached to a particular protocol. Parameters are selected from a list of annotation types which have been flagged as parameters. Annotation types which has been selected as parameters show up in the regular annotation dialog whenever the protocol is used for an item. For more information see Chapter 10, *Annotations* (page 73).

Figure 13.2. Protocol parameters



Annotation types

This list contains the annotation types selected as parameters for the protocol.

Add annotation types

Use this button to open a pop-up where you can select annotation type to use for parameters. The list only shows annotations types which has the **Protocol parameter** flag set.

Remove

Removes the selected annotation types from the list.

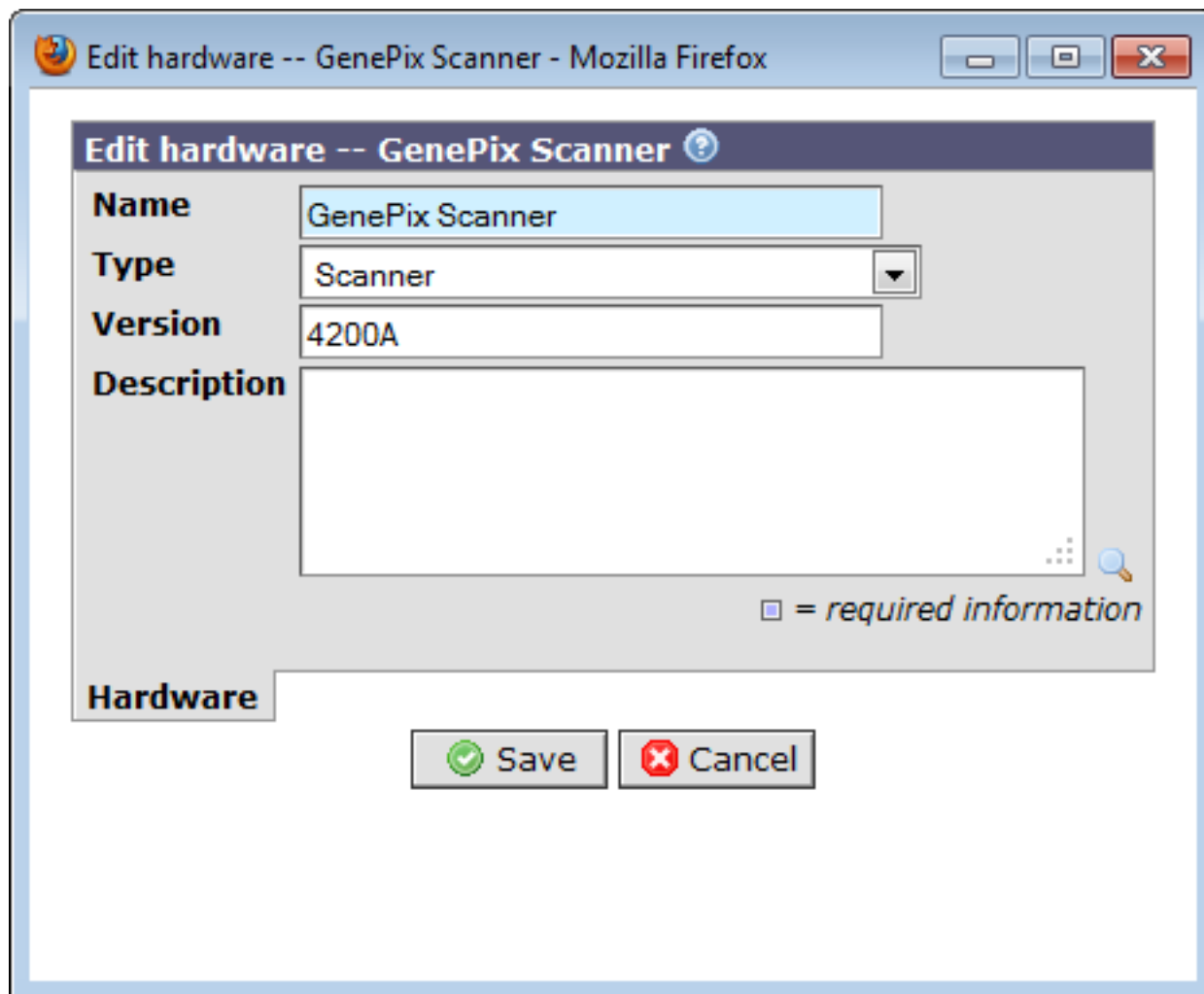
The **Annotations** tab allows BASE users to use annotation types to refine protocol description. More about annotating items can be read in Section 10.2, “Annotating items” (page 80).

Chapter 14. Hardware and software

14.1. Hardware

A hardware is used in BASE to represent a piece of equipment in the lab that is used in the experiments. Information about hardware is typically added to a BASE server by an administrator and can be managed from Administrative Hardware.

Figure 14.1. Hardware properties



The screenshot shows a web browser window titled "Edit hardware -- GenePix Scanner - Mozilla Firefox". Inside the browser is a form titled "Edit hardware -- GenePix Scanner". The form contains the following fields:

- Name:** A text input field containing "GenePix Scanner".
- Type:** A dropdown menu with "Scanner" selected.
- Version:** A text input field containing "4200A".
- Description:** A large, empty text area.

At the bottom right of the form, there is a legend: a blue square icon followed by the text "= required information". Below the form, there are two buttons: "Save" (with a green checkmark icon) and "Cancel" (with a red X icon).

Name

The name of the hardware.

Type

The hardware type of the hardware. The list may evolve depending on additions by the server administrator. Selecting the proper hardware type is important and enables BASE to automatically guess the most likely hardware when creating new items. See Chapter 12, *Item subtypes* (page 92) for more information.

Version

The version number or model of the hardware.

Description

A description of the hardware.

14.2. Software

A software item is used in BASE to represent the software that is used to process and analyze data outside of BASE. Analysis that is done inside BASE is usually represented as plug-ins. Information about software is typically added to a BASE server by an administrator and can be managed from Administrative Software.

Figure 14.2. Software properties

Figure 14.2 shows a screenshot of the 'Edit software -- GenePix Pro' dialog box, which is displayed within a Mozilla Firefox browser window. The dialog box has a title bar with the Firefox icon and the text 'Edit software -- GenePix Pro'. The main content area of the dialog is titled 'Edit software -- GenePix Pro' and contains four input fields: 'Name' (containing 'GenePix Pro'), 'Type' (a dropdown menu showing 'Feature extraction'), 'Version' (containing '6.0'), and 'Description' (a large empty text area). A legend at the bottom right of the form area indicates that a blue square icon represents 'required information'. At the bottom of the dialog are two buttons: 'Save' (with a green checkmark icon) and 'Cancel' (with a red X icon).

Name

The name of the software.

Type

The software type of the software. The list may evolve depending on additions by the server administrator. Selecting the proper software type is important and enables BASE to automatically guess the most likely software when creating new items. See Chapter 12, *Item subtypes* (page 92) for more information.

Version

The version number of the software.

Description

A description of the software.

Chapter 15. Array LIMS

Arrays are at the core of the BASE business and are essential elements to describe in order to be MIAME¹ compliant. It is also critical to track and manage information about microarray design as accurately as possible since mistakes could prove extremely costly in downstream analysis. As a good practice, all array related information should be entered into BASE prior to work on describing the sample processing and hybridizations or other events making up an experiment is begun.

15.1. Array designs

Array designs should be understood as a plan which can be realized during a printing process producing microarray slides. During the course of the printing process, reagents may run out leading to the interruption of this process. All slides created during this printing process belong to the same printing batch. It is the array slide that will eventually be used in a hybridization event. BASE allows user to track those 3 entities with great details. This is an important functionality for users producing their own arrays and for those caring for quality control and tracking of microarray slides in a printing facility. The following sections detail how to use BASE to help in these tasks.

Non-array platforms also need array designs

Array designs are needed also for non-array platforms (eg. sequencing). In this case a "virtual array design" is created which is simply a list of all features that is of interest in the experiment. Since there are no coordinates or positions to identify features a unique id need to be constructed in some other way. For example, in sequencing experiments we may use a GTF file to create a virtual array design using the *transcript_id* and *chromosome* as a unique identifier.

Use Array LIMS Array designs to get to the list page with array designs.

¹ <http://www.mged.org/Workgroups/MIAME/miame.html>

15.1.1. Properties

Figure 15.1. Array design properties

Figure 15.1 shows a screenshot of the "Edit array design -- Array design A" dialog box, which is displayed within a Mozilla Firefox browser window. The dialog box contains the following fields and controls:

- Name:** A text input field containing "Array design A".
- Platform:** A dropdown menu currently set to "Generic".
- Arrays / slide:** A text input field containing the value "1".
- Description:** A large, empty text area for providing additional information.
- Legend:** A small blue square icon followed by the text "= required information".
- Tabs:** Four tabs are visible at the bottom: "Array design" (the active tab), "Data files", "Annotations", and "Inherited annotations".
- Buttons:** Two buttons are located at the bottom center: a "Save" button with a green checkmark icon and a "Cancel" button with a red X icon.

Name

Provide an sensible name for the design (required).

Platform

Select the platform / variant used for the array design. The selected options affects which files that can be selected on the **Data files** tab.

Arrays/slide

The number of sub-arrays that can be placed on a single slide. The default value is 1, but some platforms, for example Illumina, has slides with 6 or 8 arrays. In sequencing platforms, this value is the number of lanes on a flow cell.

Description

Provide other useful information about the array design in this text area.

Click on the **Save** button to store the information in BASE or on the **Cancel** button to abort.

The **Data files** tab allows BASE users to attach files to the array design. The possible file types are defined by the array design's platform. See Section 11.4, "Selecting files for an item" (page 90) for more information.

The **Annotations** tab allows BASE users to use annotation types to refine array design description. More about annotating items can be read in Section 10.2, "Annotating items" (page 80).

This **Inherited annotations** tab contains a list of those annotations that are inherited from the array design's parents (eg. plates). Information about working with inherited annotations can be found in Section 10.2.1, "Inheriting annotations from other items" (page 82).

15.1.2. Importing features to an array design

Importing features is an important step in order to fully define an array design. It should be noted that BASE does not enforce the immediate feature import upon creation of array design. However, it is **STRONGLY** advised to do so when creating an array design. Performing the import enables use of the array design in downstream analysis with no further trouble. It also matters when importing raw bioassay data and matching those to the corresponding array design.

Depending on which platform and/or data files you selected when you created the array design the process to import features is different. For example, if you selected the Affymetrix platform, which is a file-only platform, the feature information has already been extracted from the CDF file (if you selected one). If the selected platform doesn't extract information from the selected data file automatically this may be done manually by executing an import plug-in.

From the array design item view, click on the **Import** button and use the reporter map importer and an appropriate plug-in configuration when following the instructions in Chapter 18, *Import of data* (page 150). If the import run is successful, go to the array design list view. The **Has features** column will show **Yes (db: x, file: y)** where x is the number of features actually imported into the database.

Note

The **Import** button only shows up if the logged in user has enough permissions.

Verify that probsets in a CDF file exist as reporters

File-only platforms, such as Affymetrix, require that all probesets must exist as reporters before data can be analysed. For other platforms this is usually checked when importing the features to the database. Since no import takes place for file-only platforms, another manual step takes it place. Use the **Import** button in the array design item view and select the **Affymetrix CDF probeset importer** plug-in. If you have enough permissions this function will also let you create missing reporters.

15.2. Array batches

Beside the common way of creating items in BASE, an array batch can also be created directly from an array design, both in list view and single item view.

In list view of array design

Click on the  icon available from the **Batches** column of the array design you want to use.

Tip

As default in BASE the **Batches** column is hidden and need therefore be made visible first, see Section 5.4.3, "Configuring which columns to show" (page 30)

New batch... is the corresponding button in single item view. The current array design will automatically be filled in the array design property for the new batch.

Figure 15.2. Array batch properties

Create array batch ?

Name

Array design Array design A ▼ Select...

Print robot - none - ▼ Select...

Protocol - none - ▼ Select...

Description

☐ = required information

Array batch | Annotations & parameters | Inherited annotations

Name

The name of the array batch (required).

Array design

Array design that is used for the batch.

Print robot

The print robot that is used.

Protocol

The printing protocol that was followed when producing the array batch

Description

Provide other useful information about the array batch in this text area.

Click on the **Save** button to store the information in BASE or on the **Cancel** button to abort.


15.3. Array slides

Use Array LIMS Array slides to get to the list page of array slides.

15.3.1. Creating array slides

In BASE, array slides can be created, except the common way, by 2 routes:

from the array batch list page

Clicking on the  icon in the **Slides** column for the batch you want to add a slide to. Corresponding button in the view page of a batch is **New slide...**

using a wizard to create multiple slides simultaneously

This can be started from three different places:


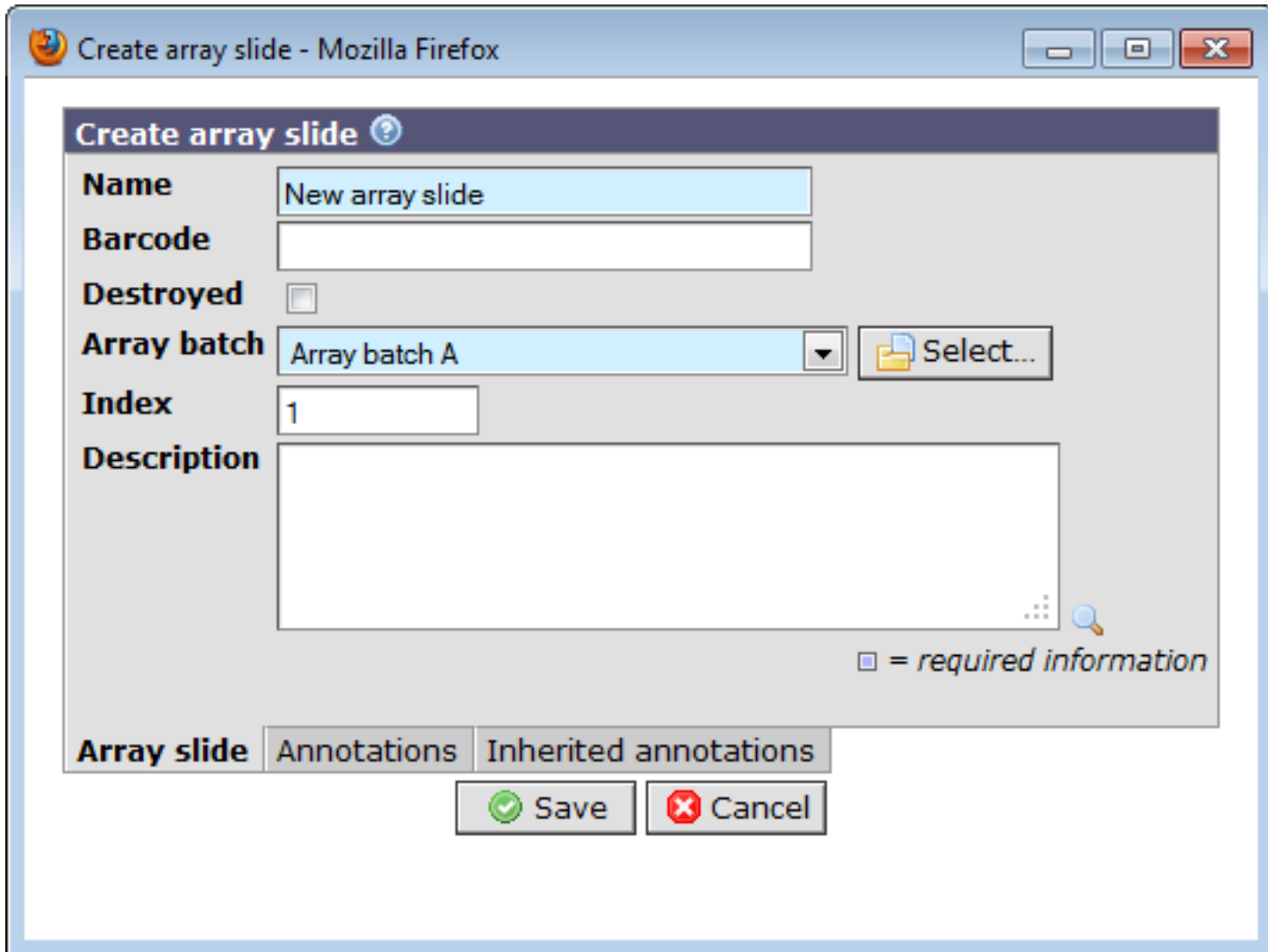
- array batch list view by clicking on  in the **Slides** column of the batch that should be used.
 - Using the **Create slides** in a single item view of an array batch.
 - In the list page of array slides, using the **Create multiple...** button
- The wizard is described further down in Section 15.3.2, “Multiple slides wizard” (page 106).

Figure 15.3. Create new array slide



Name

The name of the array slide (required).

Barcode

Does the array slide have a barcode, it can be put here.

Destroyed

This check-box can be ticked to mark the slide as destroyed, lost or damaged.

Array batch

Array batch that the slide belongs to (required).

Index

The index of the array slide in selected array batch.

Description

Any information useful information about the slide can be in this field.

Click on the **Save** button to store the information in BASE or on the **Cancel** button to abort.

15.3.2. Multiple slides wizard

As mentioned above there is an alternative to create one slide at a time if you have many to add. There is a wizard that can help you to create at the most 999 slides in one go. The wizard is in two steps:

Figure 15.4. Create multiple array slides - step 1

Create array slides - Mozilla Firefox

Create array slides ?

Name Array batch A.

Array batch Array batch A Select...

Quantity 10 (1-999)

Start at 1 **Pad size** 2

The index number will be padded with zeroes to this length (ie, 1 --> Slide.001; 10 --> Slide.010). Leave empty for automatic selection.

Description

☐ = required information

Array slides

Next Cancel

The first step reminds alot of the normal edit window of an array slide:

Name

The name prefix of the array slides. Each array slide will be numbered according to the *start at* and *pad size* settings below.

Array batch

Array batch that the slides belongs to (required).

Quantity

Number of slides to create with this wizard (required and must be between 1 and 999)

Start at

The index number to start from when indexing the name of the slides.





Pad size

The index will be filled out with zeros in front to always have this length.



Click on **Next** to move on to the second step of the wizard.

Figure 15.5. Create multiple array slides - step 2


The screenshot shows a web browser window titled 'Create array slides - Mozilla Firefox'. Inside the browser is a dialog box titled 'Create array slides'. The dialog has two main columns: 'Name' and 'Barcode'. The 'Name' column contains 10 rows of text, each starting with a number from 1 to 10, followed by 'Array batch A.' and a two-digit number from 01 to 10. The 'Barcode' column is empty. At the bottom of the dialog, there are two buttons: 'Save' (with a green checkmark icon) and 'Cancel' (with a red X icon). The dialog also has a 'Names and barcodes' tab selected at the bottom left.

	Name:  	Barcode:  
1	Array batch A.01	
2	Array batch A.02	
3	Array batch A.03	
4	Array batch A.04	
5	Array batch A.05	
6	Array batch A.06	
7	Array batch A.07	
8	Array batch A.08	
9	Array batch A.09	
10	Array batch A.10	

Names and barcodes

 Save  Cancel

This step allows the names of the array slides to be adjusted if needed. It is also possible to enter barcodes for each of the new slides. The information can either be filled in one-by-one or by using

the "scratchpad" icon . This opens up a larger text window where the names or barcodes can be entered one per line. This can be useful if the information can be pasted from an external source.

Click on the **Save** button to store the information in BASE or on the **Cancel** button to abort the wizard.

Chapter 16. Biomaterial LIMS

The generic term biomaterial refers to any biological material used in an experiment. Biomaterial is divided in three main components, *biosources*, *samples* and *extracts*. The biomaterials can then be subclassified further by the use of subtypes (see Chapter 12, *Item subtypes* (page 92)). BASE has, for example, defined two extracts subtypes: *Labeled extract* (used in microarray experiments) and *Library* (used in sequencing experiments). The order used in presenting those entities is not innocuous as it represents the sequence of transformation a source material undergoes until it is in a state compatible for further experimental processing. This progression is actually mimicked in the BASE Biomaterial LIMS menu again to insist on this natural progression.

- Biosources correspond to the native biological entity used in an experiment prior to any treatment.
- Samples are central to BASE to describe the sample processing. So samples can be created from other samples if user want to track sample processing event in a finely granular fashion.
- Extracts correspond to nucleic acid material extracted from a tissue sample or a cell culture sample.

BASE allows users to create any of the these entities fairly freely, however it is expected that users will follow the natural path of the laboratory workflow.

16.1. Biosources

Biosources correspond to the native biological entity used in an experiment prior to any treatment. Biosources can be added to BASE by most users and are managed from Biomaterial LIMS Biosources. Use the **New...** button to create a new biosource. This brings up the dialog below.

Figure 16.1. Biosource properties

Edit biosource -- Biosource A ?

Name Biosource A

Type -none-

External ID

Description

☐ = required information

Biosource **Annotations**

☒ Save ☐ Cancel

Name

This is the only mandatory field. BASE by default assigns *New biosource* as name but it is advised to provide unique sensible names.

Type

The subtype of the biosource. The list may evolve depending on additions by the server administrator. Selecting the proper subtype is recommended and enables BASE to automatically guess the most likely subtype when creating child biomaterial. See Chapter 12, *Item subtypes* (page 92) for more information.

External ID

An external reference identifiers (e.g. a patient identification code) can be supplied using this field.

Description

A free text description can be supplied using this field.

The **Annotations** tab allows BASE users to use annotation types to refine biosource description. More about annotating items can be read in Section 10.2, “Annotating items” (page 80)

16.2. Samples

Samples result from processing events applied to biosource material or other samples before they are turned into an extract. In other words, samples can be created from biosource items or from one or more sample items. When a sample is created from several other samples, a pooling event is performed.


For every step of transformation from biosource to sample, it is possible to provide information about the protocol used to perform this task. It is not enforced in BASE but it should serve as guidance when devising the granularity of the sample processing task. Also, it is good practice to provide protocol information to ensure MIAME compliance.

Use Biomaterial LIMS Samples to get to the list of samples.


16.2.1. Create sample

Beside the common way, using the **New...** button, a sample can be created in one of the following ways:

from either biosource list- or single view- page.

No matter how complex the sample processing phase is, at least one sample has to be anchored to a biosource. Therefore, a natural way to create an sample is to click on  in the **Samples** column of the biosource list view. There is also a corresponding button, **New sample...** in the toolbar when viewing a single biosource.

from the sample list page

Child samples can be created from a single parent by clicking on the  icon in the **Child samples** column. Pooled samples can be created by first selecting the parents from the list of samples and then click the **Pool...** button in the toolbar.

16.2.2. Sample properties

Figure 16.2. Sample properties

Edit sample -- Sample A.00h

Name

Type

External ID

Original quantity (µg)

Created

Registered

Protocol

Bioplate

Biowell

Description

☐ = required information

Sample **Parents** **Annotations & parameters** **Inherited annotations**

Name

The sample's name (required). BASE by default assigns names to samples (by suffixing *s#* when creating a sample from an existing biosource or *New Sample* otherwise) but it is possible to edit at will.

Type

The subtype of the sample. The list may evolve depending on additions by the server administrator. Selecting the proper subtype is recommended and enables BASE to automatically guess the most likely subtype when creating child biomaterial. See Chapter 12, *Item subtypes* (page 92) for more information.

External ID

An identification used to identify the sample outside BASE.

Original quantity

This is meant to report information about the actual mass of sample created.

Created

A date when the sample was created. The information can be important when running quality controls on data and account for potential confounding factor (e.g. day effect).

Registered

The date at which the sample was entered in BASE.

Protocol

The protocol used to produce this sample.

Bioplate

The bioplate where this sample is located.

Biowell

Biowell that holds this sample. **Bioplate** has to be defined before biowell can be selected.

Description

A text field to report any information that not can be captured otherwise.

16.2.3. Sample parents

Figure 16.3. Sample parents

Edit sample -- Sample A.00h

Parent type ☒ Biosource ☐ Sample

Biosource Biosource A

Samples

+ Select biosource...

+ Add samples...

- Remove

- used quantity (µg)

Sample Parents Annotations & parameters Inherited annotations

Save Cancel

This is meant to keep track of the sample origin. BASE distinguishes between two cases which are controlled by the **Parent type** radio-button in the edit pop-up window.

- If the parent is a biosource the radio-button is set to **Biosource**. Only a single biosource can be used as the parent. This option is automatically selected if the user selects a biosource with the **Select biosource** button.
- When the parent is one or several other samples the radio-button is set to **Sample**. This option is automatically selected if the user add samples with the **Add samples** button. For each parent sample, it is possible to specify the amount used in µg. This will automatically update the **remaining quantity** of the parent.

The **Annotations** tab allows BASE users to use annotation types to refine sample description. More about annotating items can be read in Section 10.2, "Annotating items" (page 80)

This **Inherited annotations** tab contains a list of those annotations that are inherited from the sample's parents. Information about working with inherited annotations can be found in Section 10.2.1, “Inheriting annotations from other items” (page 82).

16.3. Extracts

Extract items should be used to describe the events that transform a sample material into an extract material. An extract can be created from one sample item or from one or more extract items. When an extract is created from several other extracts, a pooling event is performed.


During the transformation from samples to extracts, it is possible to provide information about the protocol used to perform this task. It is not enforced in BASE but it should serve as guidance when devising the granularity of the sample processing task. Also, it is good practice to provide protocol information.

Use Biomaterial LIMS [Extracts](#) to get to the list of extracts.


16.3.1. Create extract

Beside the common way, using the **New...** button, an extract can be created in one of the following ways:

from either sample list- or single view- page.

No matter how complex the extract processing phase is, at least one extract has to be anchored to a sample. Therefore, a natural way to create an extract is to click on  in the **Child extracts** column for the sample that should be a parent of the extract. There is also a corresponding button, **New child extract...** in the toolbar when viewing a single sample.

from the extract list page

Child extracts can be created from a single parent by clicking on the  icon in the **Child extracts** column. Pooled extract can be created by first selecting the parents from the list of extracts and then press **Pool...** in the toolbar. The selected extracts will be put into the parent property.

16.3.2. Extract properties

Figure 16.4. Extract properties

Edit extract -- Extract A.00h

Name Extract A.00h

Type -none-

Tag - none - Select...

External ID

Original quantity 100.0 (µg)

Created 2011-10-11 Calendar...

Registered 2011-10-11

Protocol Extraction A Select...

Bioplate Bioplate A Select...

Biowell B1 Select...

Description

= required information

Extract Parents Annotations & parameters Inherited annotations

Save Cancel

Name

A mandatory field for providing the extract name. BASE by default assigns names to extract (by suffixing *e#* when creating an extract from an existing sample or *New extract* otherwise) but it is possible to edit it at will.

Type

The subtype of the extract. The list may evolve depending on additions by the server administrator. Selecting the proper subtype is recommended and enables BASE to automatically guess the most likely subtype when creating child biomaterial and bioassays. See Chapter 12, *Item subtypes* (page 92) for more information.

Tag

If the extract has been marked with a tag, select it here. Note that the subtype of the extract usually limits what kind of tag that can be used. For example, a *labeled extract* should be tagged with a *label*.

External ID

The extracts identification outside BASE

Original quantity

Holds information about the original mass of the created extract.

Created

The date when the extract was created. The information can be important when running quality controls on data and account for potential confounding factor (e.g. day effect)

Registered

This is automatically populated with a date at which the sample was entered in BASE system.

Protocol

The extraction protocol that was used to produce the extract.

Bioplate

The bioplate where this extract is located.

Biowell

Biowell that holds this extract. **Bioplate** has to be defined before biowell can be selected.

Description

A text field to report any information that not can be captured otherwise.

16.3.3. Extract parents

Figure 16.5. Extract parents

This is meant to keep track of the extract origin. BASE distinguishes between two cases which are controlled by the **Parent type** radio-button in the edit pop-up window.

- If the parent is a sample the radio-button is set to **Sample**. Only a single sample can be used as the parent. This option is automatically selected if the user selects a sample with the **Select sample** button.
- When the parent is one or several other extracts the radio-button is set to **Extract**. This option is automatically selected if the user add extracts with the **Add extracts** button.

For each parent item, it is possible to specify the amount used in micrograms. This will automatically update the **remaining quantity** of the parent.

The **Annotations** tab allows BASE users to use annotation types to refine extract description. More about annotating items can be read in Section 10.2, "Annotating items" (page 80)

This **Inherited annotations** tab contains a list of those annotations that are inherited from the extract's parents. Information about working with inherited annotations can be found in Section 10.2.1, “Inheriting annotations from other items” (page 82).

16.4. Tags

Before attempting to create tagged extracts, users should make sure that the appropriate tag object is present in BASE. To browse the list of tags, go to Biomaterial LIMS Tags

Figure 16.6. Tag properties

The tag item is very simple and does not need much explanation. There are only a few properties for a tag.

Name

The name of the tag (required).

Type

The subtype of the tag. The list may evolve depending on additions by the server administrator. Selecting the proper subtype is important and enables BASE to automatically guess the most likely tag when creating tagged extracts (eg. a *Label* for a *Labeled extract* or a *Barcode* for a *Library*). See Chapter 12, *Item subtypes* (page 92) for more information.

Description

An explaining text or other information associated with the tag.

16.5. Bioplates

With bioplates it is possible to organize biomaterial such as samples and extracts into wells. Each plate has a number of wells that is defined by the plate geometry.

Use Biomaterial LIMS Bioplates to get to the list of bioplates.

16.5.1. Bioplate properties

Figure 16.7. Bioplate properties

Create bioplate ?

Name

Bioplate type Storage plate ▼ Select...

Plate geometry 96-well (8 × 12) ▼ Select...

Freezer - none - ▼ Select...

Barcode

Destroyed ☐

Description

☒ = required information
☐ = can't be changed later

Plate **Annotations**

☒ Save ☐ Cancel

Name

The bioplate name. The name does not have to be unique but it is recommended to keep it unique. BASE by default assigns *New bioplate* as name. This field is mandatory.

Bioplate type

The type of the bioplate may be a generic storage plate that can store any type of biomaterial or a locked plate that can only store a single type of biomaterial. This field is mandatory and can only be set for new bioplates. See Section 16.5.3, “Bioplate types” (page 122) for more information.

Plate geometry

Information about the plate design defining the number of rows and columns on the bioplate. This field is mandatory and can only be set for new bioplates.

Freezer

The freezer where the bioplate is stored. Optional.

Barcode

Barcode of the bioplate. Optional.

Description

Other useful information about the bioplate. Optional.

The **Annotations** tab allows BASE users to use annotation types to refine bioplate description. More about annotating items can be read in Section 10.2, “Annotating items” (page 80)

16.5.2. Biowells

Biowells existence are managed through the bioplate they belong to. Creating a bioplate will automatically create the biowells (as given by the selected geometry) on the plate. The wells are initially empty. To add biomaterial to the plate go to the single-item view page for the bioplate. This page includes an overview of the layout of the plate. Clicking on an empty well will open a popup dialog that allows you to select a biomaterial. The same dialog can also be accessed from the **Wells** tab. Assigning a biomaterial to a biowell can also be done when editing a sample or extract, or by using the Place-on-plate wizard.

Figure 16.8. Biowell properties

The screenshot shows a web browser window titled "Edit biowell -- A1 on Bioplate A - Mozilla Firefox". Inside the browser is a dialog box titled "Edit biowell -- A1 on Bioplate A". The dialog has the following fields and controls:

- Bioplate:** A text field containing "Bioplate A".
- Well location:** A text field containing "A1".
- Biomaterial type:** A dropdown menu currently showing "Sample".
- Biomaterial:** A text field containing "Sample A.00h" and a dropdown arrow.
- Select...:** A button with a folder icon and the text "Select...".
- Biowell:** A tab-like label at the bottom left of the dialog.
- Save:** A button with a green checkmark icon.
- Cancel:** A button with a red X icon.

Bioplate

Shows which bioplate the biowell is located on. This property is read-only.

Well location

The biowell location on the bioplate in row+column format. This property is read-only.

Biomaterial type

The type of biomaterial stored in this biowell. This property must be selected before before a biomaterial can be selected. On some plates this is locked due to settings in the bioplate's type.

Biomaterial

Name of the biomaterial in this biowell. Before changing this you must select the appropriate **Biomaterial type**. A biomaterial can only be placed in a single well. If the selected biomaterial is already placed in another location it will be moved.

16.5.3. Bioplate types

Bioplate types are used to subclassify bioplates and may put restrictions on them. BASE ships with a few pre-defined bioplate types. The *Storage plate* type is a generic plate type that can be used for all types of biomaterial and doesn't have any other restrictions on it. The reaction plate types are locked to a single type of biomaterial and have a restriction that biomaterial can never be moved out from a well once it has been placed there.

Figure 16.9. Bioplate type properties

Create bioplate type ?

Name

Biomaterial type

Biomaterial subtype

Well lock mode

Description

☐ = required information
☐ = can't be changed later

Bioplate type

Name

The name of the bioplate type.

Biomaterial type

Select if bioplates using this type should be locked to specific biomaterial type or not. This property can only be set for new bioplate types.

Biomaterial subtype

If a specific biomaterial type has been selected it is also possible to further restrict the use of the bioplate to a certain biomaterial subtype. The restriction is not enforced by the core but is mainly used by the gui to provide smart filters in selection lists, in the bioplate event wizards, etc.

Well lock mode

This option controls the wells on bioplates using this type. There are four options:

- **Unlocked:** The wells are unlocked and biomaterial can be added and removed freely any number of times.
- **Locked after add+clear:** A biomaterial can be placed once in the well and then moved to another bioplate. After that the well becomes locked and it is not possible to add a different biomaterial to it.
- **Locked after add:** The wells are locked as soon as biomaterial is added to them. The biomaterial can't be moved to another place or be replaced with other biomaterial.
- **Locked at plate creation:** The wells are locked as soon as the bioplate has been saved to the database. This lock mode is primarily intended to be used when plug-ins are creating and populating the bioplate as a single event.

Description

Other useful information about the bioplate type. Optional.

16.5.4. Bioplate events

Certain actions can be applied collectively to the biomaterial on a bioplate, either as a whole or a subset that is picked by the user. A list of the available actions can be found on the list page [Biomaterial LIMS Bioplate event types](#). Although it is possible to create more event types here there is usually no meaning to do so, since each event needs a specialized GUI wizard to take care of it. The possibility to add more event types should be seen as an opportunity for extension development.

The place-on-plate event

Figure 16.10. Place on plate wizard

Place biomaterial on plate - Mozilla Firefox

Place biomaterial on plate ?

Event name

Event date

Protocol

Hardware

Items to place

A1	Sample A.00h
A2	Sample A.24h
	Sample A.ref

Destination plate: Bioplate A

	1	2	3	4	5	6	7
A							
B							
C							
D							
E							
F							
G							
H							

☒ Auto-select next unmapped item

This event is available on the sample and extract list pages and can be used to place multiple biomaterial on a bioplate in one go. Click on the **Place on plate** button to start the wizard. The wizard will automatically use the selected biomaterials or, if none are selected, all listed biomaterials that aren't already located on a plate.

Event name

Give a name to the event, or keep the suggested name.

Event date

The date of the event.

Protocol

The protocol used in the event, if any.

Hardware

The hardware item used in the event, if any.

Description

Other comments about the event.

Select plate...

You need to select an existing plate on which the biomaterial should be placed. It is only possible to use one plate in each event. If you want to place biomaterial on more than one plate, the wizard must be repeated for each destination plate.

Clear

Clear all current placement.

Place by row/column

Automatically place the remaining biomaterial by filling empty wells, starting with rows or columns.

Items to place

This column contains the biomaterial that should be placed on the plate. When a destination plate has been selected, it is displayed as a grid to the right. To place a biomaterial either use the **Place by row** or **Place by column** buttons, or select an item in this list. When an item has been selected, click on the destination well on the plate. The coordinate of the well is displayed in the gray area before the biomaterial name and a line is drawn between it and the destination well. The destination well is also marked with an icon. If the **Auto-select next unmapped item** is selected the selection is automatically moved to the next biomaterial which can then be placed by selecting another destination well. If a mistake is made it is easy to correct. Simply re-select the item and then click on the correct well.

When the biomaterial has been placed on the plate (it is not necessary to place all of them) click on **Save** to store everything. BASE will create a plate event for the selected destination plate and "other"-type events for each biomaterial that was placed on it.

The move biomaterials event

Figure 16.11. Move biomaterials wizard

Move biomaterial - Mozilla Firefox

Move biomaterial to plate ?

Event name

Event date

Protocol

Hardware

Description

Source plate: Bioplate A

	1	2	3	4	5	6	7	8	9	10	11	12
A												
B												
C												
D												
E												
F												
G												
H												

Destination plate

	1	2	3
A	A1		
B	A2		
C	A3		
D			
E			
F			
G			
H			

☒ Show source c...

This event is available on the single-item view page of a bioplate and can be used to move biomaterial from one plate to another. Click on the **Move biomaterial** button to start the wizard.

Event name

Give a name to the event, or keep the suggested name.

Event date

The date of the event.

Protocol

The protocol used in the event, if any.

Hardware

The hardware item used in the event, if any.

Description

Other comments about the event.

Select plate...

You need to select an existing plate to which the biomaterial should be moved. It is only possible to use one plate in each event. If you want to move biomaterial to more than one plate, the wizard must be repeated for each destination plate.

Clear

Clear all current mapping between the source and destination plates.

Place by row/column

Automatically move the remaining biomaterial by filling empty wells, starting with rows or columns.

Predefined mapping

Use this button to select a predefined mapping between source and destination wells. The biomaterial will be moved according to the mapping.

Source plate

This displays the source plate as a grid with icons that indicate filled and movable wells. When a destination plate has been selected, it is displayed as a similar grid to the right. To move a biomaterial either use the **Place by row**, **Place by column** or **Predefined mapping** buttons, or select a well on the source plate. When a source well has been selected, click on a well on the destination plate. A line is drawn between the source and destination wells and the icons are updated to show what is going on. The wells on the destination plate will also show the coordinate of the mapped source well unless the **Show source coordinates** checkbox is deselected. If a mistake is made it is easy to correct. Simply re-select the source well and then click on the correct destination well.

When the biomaterial has been mapped between the source and destination plates (it is not necessary to map all of them) click on **Save** to store everything. BASE will create a plate event for the selected plates and "other"-type events for each biomaterial that was moved.




The create child plate event

Figure 16.12. Create child plate wizard - step 1


Create child bioplate - step 1/2 - Mozilla Firefox

Create child bioplate - step 1/2 ?




Event



Event name	Create child plate		Description
Event date	2011-10-11	 Calendar...	
Protocol	Labeling A	 Select...	
Hardware	- none -	 Select...	

Child biomaterial

Type	Sample → Extract		Description
Subtype	Labeled extract		
Tag	cy3	 Select...	
Original quantity		(μg)	
Used from parent		(μg)	

Child plates

No. of plates	1		Description
Name prefix	Bioplate C		
Geometry	96-well (8 × 12)	 Select...	
Plate type	Extract reaction plate	 Select...	
Freezer	- none -	 Select...	

 Next  Cancel

This event is available on the single-item view page of a bioplate when the bioplate is limited to a single type of biomaterial (eg. only samples or only extracts). The event is used to create either a child bioplate with biomaterial that is derived from the biomaterial on the parent plate or to create one or more physical bioassays. Click on the **Create child bioplate** button to start the wizard.

The wizard has two steps. In the first step you set properties that are related to the event and to the creation of child plates and biomaterial. The first step is divided into three main sections.

Event

Event name

Give a name to the event, or keep the suggested name.

Event date

The date of the event.

Protocol

The protocol used in the event, if any.

Hardware

The hardware item used in the event, if any.

Description

Other comments about the event.

Child biomaterial

Type

The type of child items to create. If the source plate contains samples, you can select between sample and extract and if the source plate contains extract you can select between extract and physical bioassay.

Subtype

The subtype to assign to the newly created biomaterial (or physical bioassay). The list of options is automatically updated based on the selection in the **Type** list.

Tag

Visible when creating child extracts only. Select the tag to assign to the new extract. If no tag is selected and the source biomaterial is also extracts, the children will get the same tag as their parents.

Original quantity

The original quantity of the new biomaterial. Not visible when creating a physical bioassay.

Used quantity

The quantity that was used from the parent biomaterial in the process of creating child biomaterial.

Description

Other comments about the new biomaterial.

Child plates

No. of plates

The number of child plates to create. The default value is 1.

Name prefix

The child plates will be named using the prefix plus a running number starting with 0. Eg. New plate.0.

Geometry

The geometry of the child plates. The default is the same geometry as the parent plate. This option is replaced with **Size of bioassay** when creating a physical bioassay.

Plate type

The plate type of the child plates. Not used when creating a physical bioassay.

Freezer

The freezer in which the new child plates are located. Not used when creating a physical bioassay.

Description

Other comments about the new child plates.

Figure 16.13. Create child plate wizard - step 2

Create child bioplate - step 2/2 - Mozilla Firefox

Create child bioplate - step 2/2 ?

Source plate: Bioplate B

	1	2	3	4	5	6	7	8	9	10	11	12
A												
B												
C												
D												
E												
F												
G												
H												

Destination plate:

	1	2	3
A	A1		
B	B1		
C	C1		
D			
E			
F			
G			
H			

☒ Show source c

Child plate:

Name

Barcode

Child biomaterial:

Name

The second step display the source plate and new child plates as a grid. To create child biomaterial either use the **Place by row**, **Place by column** or **Predefined mapping** buttons, or select a well on the source plate. When a source well has been selected, click on a well on the destination plate. A line is drawn between the source and destination wells and the icons are updated to show what

is going on. The wells on the destination plate will also show the coordinate of the mapped source well unless the **Show source coordinates** checkbox is deselected. If a mistake is made it is easy to correct. Simply re-select the source well and then click on the correct destination well.

When a child biomaterial is selected you have the option to override the automatically generated name. It is also possible to change the name and barcode of the child plate.

Note

The principle is the same when creating physical bioassays, except that no new child biomaterial is created.

When the biomaterial has been mapped between the source and destination plates (it is not necessary to map all of them) click on **Save** to store everything. BASE will create a plate event for the selected plates and "create" or "bioassay"-type events for each biomaterial that was used.

16.6. Biomaterial lists

TODO

16.7. Physical bioassays

A physical bioassay represents the application of one or more extracts to an experimental setup designed to measure quantities that we are interested in. For example, a *Hybridization* event corresponds to the application of one or more *Labeled extracts* materials to a microarray slide under conditions detailed in hybridization protocols. Use View Physical bioassays to get to the bioassays.

16.7.1. Create physical bioassays

In BASE, there are two possible routes to create a physical bioassay except the common way with the **New...** button at the list page.

from the extract list view page

Select at least one extract, to create a bioassay from, by ticking the selection boxes before the name field. Click on the **New physical bioassay...** in the toolbar.

from the extract single-item page

When viewing an extract in single-item view, click on the **New physical bioassay...** button in the toolbar.

16.7.2. Bioassay properties

Figure 16.14. Physical bioassay properties

Edit physical bioassay -- Hybridization A.00h

Name Hybridization A.00h

Type Hybridization

Size 1

Created 2011-10-11 Calendar...

Registered 2011-10-11

Protocol Hybridization A Select...

Hardware Hybridization station A Select...

Array slide Array slide A.1 Select...

Description

= required in

Physical bioassay Extracts Annotations & parameters Inherited annotations

Save Cancel

Name

The bioassay's name (required). It is recommended that the default name is replaced with something that is unique.

Type

The subtype of the bioassay. The list may evolve depending on additions by the server administrator. Selecting the proper subtype is recommended and enables BASE to automatically guess the most likely subtype when assigning source biomaterials and when creating derived bioassays. See Chapter 12, *Item subtypes* (page 92) for more information.

Size

The size of the bioassay is the number of independent positions on the bioassay. Depending on the characteristics of the bioassay, multiple biomaterials may share the same position, but

are then usually required to have different tags. Two biomaterials in different positions can use the same tag. The default value is 1, but some platforms, for example Illumina BeadArrays, has slides with 6 or 8 positions and sequencing flow cells have 8 lanes.

Created

A date should be provided. The information can be important when running quality controls on data and account for potential confounding factor (e.g. to account for a day effect).

Registered

This field is automatically populated with a date at which the hybridization was entered in BASE system.

Protocol

The protocol that was used to create the bioassay.

Hardware

Information about the machine (if any) that was used when creating the bioassay.

Array slide

The array slide that was used for the bioassay.

Description

A free text field to report any information that can not be captured otherwise.

16.7.3. Parent extracts

Figure 16.15. Parent extracts

The screenshot shows a web browser window titled "Edit physical bioassay -- Hybridization A.00h - Mozilla Firefox". The main content area is titled "Edit physical bioassay -- Hybridization A.00h" and contains a tabbed interface. The "Extracts" tab is active, displaying a list of extracts. The list contains two entries: "1: Labeled extract... [50.0 µg]" and "1: Labeled extract... [50.0 µg]". To the right of the list are two buttons: "Add extracts..." (with a green plus icon) and "Remove" (with a red minus icon). Below the list, there are two input fields: "- used quantity" with a value of "50.0" and a unit of "(µg)", and "- position" with a value of "1" and a unit of "(1 -- size of bioassay)". At the bottom of the window, there are four tabs: "Physical bioassay", "Extracts" (which is selected), "Annotations & parameters", and "Inherited annotations". Below the tabs are two buttons: "Save" (with a green checkmark icon) and "Cancel" (with a red X icon).

This important tab allows users to select the extracts used by the bioassay, and specify the amount of material used, expressed in microgram.

Use the **Add extracts** button to add items and the **Remove** button to remove items. Select one or several extracts in the list and write the used mass and position number in the fields below the list.

The **Annotations** tab allows BASE users to use annotation types to refine bioassay description. More about annotating items can be read in Section 10.2, "Annotating items" (page 80)

This **Inherited annotations** tab contains a list of those annotations that are inherited from the bioassay's parents. Information about working with inherited annotations can be found in Section 10.2.1, "Inheriting annotations from other items" (page 82).

Chapter 17. Experiments and analysis

17.1. Derived bioassays

When you have processed your physical bioassay you can register any information that you have gathered as **Derived bioassay** items in BASE. The derived bioassay can represent both a physical process, such as scanning a microarray slide, or a software process such as aligning sequence data against a reference genome. It is even possible to register child derived bioassays if there are multiple steps involved that you want to register independently. Each derived bioassay can specify a protocol (with parameters if needed) and the hardware or software item that was used to create it.

It is also possible to link one or more files to the derived bioassay. This feature requires that a subtype is selected and that the subtype has been linked with the file types that are useful in that context. For example, the **Scan** subtype is linked with **Image** files. See Chapter 12, *Item subtypes* (page 92) for more information.


Note

A derived bioassay can only hold data in the form of files. When the data has been processed enough to make it a sensible option (performance-wise) to import into the database a **Raw bioassay** should be created.


17.1.1. Create derived bioassays

Beside the common way, using the **New...** button, a derived bioassay can be created in one of the following ways:

from either physical bioassay list- or single view- page.

A derived bioassay must always have a physical bioassay as it's parent. Therefore, a natural way to create a derived bioassay is to click on  in the **Derived bioassays** column in the list view. There is also a corresponding button, **New derived bioassay...** in the toolbar when viewing a single physical bioassay.

from either derived bioassays list- or single view- page

A child derived bioassay must always have another derived bioassay as it's parent. Click on the  icon in the **Child bioassays** column to create a new child derived bioassay. There is also a corresponding button, **New child bioassay...** in the toolbar when viewing a single derived bioassay.

17.1.2. Derived bioassay properties

Figure 17.1. Derived bioassay properties

New derived bioassay ?

Name Scan A.00h

Type Scan

Parent type ☒ Physical bioassay ☐ Derived bioassay

-physical bioassay Hybridization A.00h Select...

Extract - none -

Protocol Scanning A Select...

Hardware Scanner A Select...

Software - none - Select...

Description

☒ = required information
☐ = can't be changed later

Bioassay Data files Annotations Inherited annotations

Save Cancel

Name

The name of the derived bioassay.

Type

The subtype of the derived bioassay. The list may evolve depending on additions by the server administrator. Selecting the proper subtype is required to be able to attach data files to the bioassay. It will also help BASE to automatically guess the most likely subtype when creating child bioassays. See Chapter 12, *Item subtypes* (page 92) for more information.

Parent type

Select if the parent item is a physical bioassay or another derived bioassay. This option is only available when creating new derived bioassays. Once created it is not possible to change the parent.

Extract

Select the extract that this derived bioassay is linked with. The list contains all the extracts that are present on the physical bioassay. Do not select an extract if the derived bioassay represents more than one extract at this stage.

Protocol

The protocol used in the process that created this derived bioassay (optional). Parameters may be registered as part of the protocol.

Hardware

The machine used in the process that created this derived bioassay (optional).

Software

The software used in the process that created this derived bioassay (optional).

Description

A description of the derived bioassay (optional).

The **Data files** tab allows BASE users to select files that contains data for the derived bioassay. Read more about this in Section 11.4, “Selecting files for an item” (page 90).

The **Annotations** tab allows BASE users to use annotation types to refine bioassay description. More about annotating items can be read in Section 10.2, “Annotating items” (page 80)

This **Inherited annotations** tab contains a list of those annotations that are inherited from the bioassay's parents. Information about working with inherited annotations can be found in Section 10.2.1, “Inheriting annotations from other items” (page 82).

17.2. Raw bioassays

A **Raw bioassay** is the representation of the result of analyzing data from the physical bioassay down to the point where we have a file or a set of files containing measurements per feature (eg. spot, gene, etc.) for a single sample or extract. Further analysis is usually needed before we can say something about individual features or samples and how they relate to each other. This kind of analysis is done in **Experiments**. See Section 17.3, “Experiments” (page 143).

The term **Raw bioassay** is bit misleading since the real “raw data” is actually the images from a microarray scan or the output from a sequencer. For historical reasons we have chosen to keep the term raw bioassay since this represents the first possibility for a transition between file-base data and database-stored data. Typically, all pre-rawbioassay analysis is done outside of BASE, and although we now have the possibility to track this in detail, it will probably remain so for some time in the future. See Section 17.1, “Derived bioassays” (page 136).

17.2.1. Create raw bioassays

Creating a new raw bioassay is a two- or three-step process:

1. Create a new raw bioassay item with the **New...** button in the raw bioassays list view. It is also possible to create raw bioassays from the derived bioassays list- and single view- page.
2. Upload the file(s) with the raw data and attach them to the raw bioassay.
3. The used platform may require that data is imported to the database. See Chapter 18, *Import of data* (page 150). If the platform is a file-only platform, this step can be skipped.

Supported file formats

BASE has built-in support for most file formats where the data comes in a tab-separated (or similar) form. Data for one raw bioassay must be in a single file. Support for other file formats may be added through plug-ins.

17.2.2. Raw bioassay properties

Figure 17.2. Raw bioassay properties

Create raw bioassay - Mozilla Firefox

Create raw bioassay ?

Name Raw bioassay A.00h

Platform Generic

Raw data type GenePix

Parent bioassay Scan A.00h Select...

Parent extract - none - Select...

Array design Array design A Select...

Protocol 1. Feature extraction A Select...

Software 1. Software A Select...

Description

■ = required information

Raw bioassay Data files Annotations & parameters Inherited annotations

Save Cancel

Name

The name of the raw bioassay.

Platform

Select the platform / variant used for the raw bioassay. The selected options affects which files that can be selected on the **Data files** tab. If the platform supports importing data to the database you must also select a **Raw data type**.

Raw data type

The type of raw data. This option is disabled for file-only platforms and for platforms that are locked to a specific raw data type. This cannot be changed after raw data has been imported. See Section 17.2.4, "Raw data types" (page 141).

Parent bioassay

The derived bioassay that is the parent of this raw bioassay.

Parent extract

The extract which this raw bioassay has measured. This is normally selected among the extracts that are linked with the physical bioassay that this raw bioassay is coming from. Selecting the correct extract is important if the physical bioassay contains more than one extract, since otherwise it may affect how annotations are inherited and used in downstream analysis.

Array design

The array design used on the array slide (optional). If an array design is specified the import will verify that the raw data has the same reporter on the same position. This prevents mistakes but also speed up analysis since some optimizations can be used when assigning positions in bioassay sets. The array design can be changed after raw data has been imported, but this triggers a new validation. If the raw data is stored in the database, the features on the new array design must match the the raw data. The verification can use three different methods:

- Coordinates: Verify block, meta-grid, row and column coordinates.
- Position: Verify the position number.
- Feature ID: Verify the feature ID. This option can only be used if the raw bioassay is currently connected to an array design that has feature ID values already.

In all three cases it is also verified that the reporter of the raw data matches the reporter of the features.

For Affymetrix data, the CEL file is validated against the CDF file of the new array design. If the validation fails, the array design is not changed.

Software

The software used to generate the raw data (optional).

Protocol

The protocol used when generating the raw data (optional). Software parameters may be registered as part of the protocol.

Description

A description of the raw bioassay (optional).

The **Data files** tab allows BASE users to select files that contains data for the raw bioassay. Read more about this in Section 11.4, “Selecting files for an item” (page 90).

The **Annotations** tab allows BASE users to use annotation types to refine bioassay description. More about annotating items can be read in Section 10.2, “Annotating items” (page 80)

This **Inherited annotations** tab contains a list of those annotations that are inherited from the bioassay's parents. Information about working with inherited annotations can be found in Section 10.2.1, “Inheriting annotations from other items” (page 82).

17.2.3. Import raw data

Depending on the platform, raw data may have to be imported after you have created the raw bioassay item. This section doesn't apply to file-only platforms. The import is handled by plug-ins. To start the import click on the **Import...** button on the single-item view for the raw bioassay. If this button does not appear it may be because no file format has been specified for the raw data type used by the raw bioassay or that the logged in user does not have permission to use the import plug-in or file format. See Chapter 18, *Import of data* (page 150) for more information.

File-only platforms

File-only platforms, such as Affymetrix, is handled differently and data is not imported into the database.

17.2.4. Raw data types

A raw data type defines the types of measured values that can be stored for individual features in the database. Usually this includes some kind of foreground and background intensity values. The number and meaning of the values usually depends on the hardware and software used to analyze the data from the experiment. Many tools provide mean and median values, standard deviations, quality control information, etc. Since there are so many existing tools with many different data file formats BASE uses a separate database table for each raw data type to store data. The raw data tables have been optimized for the type of raw data they can hold and only has the columns that are needed to store the data. BASE ships with a large number of pre-defined raw data types. An administrator may also define additional raw data type. See Appendix D, *Platforms and raw-data-types.xml reference* (page 430) for more information.

File-only platforms

In some cases it doesn't make sense to import any data into the database. The main reason is that performance will suffer as the number of entries in the database gets higher. A typical Genepix file contains ~55K spots while an Affymetrix file may have millions.

The drawback of keeping the data in files is that none of the generic tools in BASE can read it. Special plug-ins must be developed for each type of data file that can be used to analyze and visualize the data. For the Affymetrix platform there are implementations of the RMAExpress and Plier normalizations available on the BASE plug-ins web site. BASE also ships with built-in plug-ins for extracting metadata from Affymetrix CEL and CDF files (ie. headers, number of spots, etc).

Users of other file-only platforms should check the BASE plug-ins website for plug-ins related to their platform. If they can't find any we recommend that they try to find other users of the same platform and try to cooperate in developing the required tools and plug-ins.

17.2.5. Spot images

This section only applies to microarray platforms where a coordinate system is used to identify spots on the array slides. The raw data must contain X and Y coordinates of each spot.

After raw data has been imported into the database you will find that a **Create spot images...** button appears in the toolbar on the single-item view for the raw bioassay. Click on this button to open a window that allows you to specify parameters for the spot image extraction.

Figure 17.3. Create spot images

Create spot images ?

X scale 0 **Y scale** 0

X offset 0 **Y offset** 0

Spot size 0 **Gamma correction** 1.8

Quality 0.8

Red image file - none - Select...

Green image file - none - Select...

Blue image file - none - Select...

Save as Browse...

☐ Overwrite existing file

= required information

Spot image parameters

Create Cancel

X/Y scale and offset

For the spot image creation process to be able to find the spots, the X and Y coordinates from the raw data must be converted into image pixel values. The formula used is: $\text{pixelX} = (\text{rawX} - \text{offsetX}) / \text{scaleX}$

Important

It is important that you get these values correct, or the spot image creation process may fail or generate incorrect spot images.

Spot size

The spot size is given in pixels and is the width and height around each spot that is large enough to contain the spot without having too much empty space or neighbouring spots around it.

Gamma correction

Gamma correction is needed to make the images look good on computer displays. A value between 1.8 and 2.2 is usually best. See http://en.wikipedia.org/wiki/Gamma_correction for more information.

Quality

The quality setting to use when saving the generated spot images as JPEG images. A value between 0 = poor and 1 = good can be used.

Red, green and blue image files

You must select which scanned image files to use for the red, green and blue component of the generated spot images. Use the **Select...** buttons to select existing images or upload new ones. The original image files must be 8- or 16-bit grey scale images. Some scanners, for example Genepix, can create TIFF files with more than one image in each file. BASE supports this and uses the images in the order they appear in the TIFF file.

Note

Avoid TIFF images which also contains previews of the full image. BASE may use the wrong image with an error as the result. If you have multi-image TIFF files these must only contain the full images.

Save as

Specify the path and filename where the generated spot images should be saved. The process will create a single zip file containing all the images.

Overwrite existing file

If a file with the same name already exists you must mark this checkbox to overwrite it.

Click on the **Create** button to add the spot image creation job to the job queue, or on **Cancel** to abort.

17.3. Experiments

Experiments are the starting point for analysis. When you have uploaded and imported your raw data, collected and registered all information and annotations about samples, bioassays, and other items, it is time to collect everything in an experiment.

To create a new experiment you can either mark one or more raw bioassays on the raw bioassays list view and use the **New experiment** button. You can also create a new experiment from the experiments list view.

17.3.1. Experiment properties

Figure 17.4. Experiment properties

Create experiment - Mozilla Firefox

Create experiment ?

Name Experiment A

Raw data type GenePix

Directory - none - Select...

Raw bioassays

- Raw bioassay A.00h
- Raw bioassay A.00h (dye-swap...
- Raw bioassay A.24h
- Raw bioassay A.24h (dye-swap...

Add raw bioassays...

Remove

Description

Experiment Publication Experimental factors

Save Cancel

☒ = required information
☐ = can't be changed

Name

The name of the experiment.

Raw data type

The raw data type to use in the experiment. All raw bioassays must have raw data with this type.

Directory

A directory in the BASE file system where plug-ins can save files that are generated during the analysis. This is optional and if not given the plug-ins must ask about a directory each time they need it. Use the **Select** button to browse the file system or create a new directory.

Raw bioassays

The raw bioassays you want to analyze in this experiment. If you created the experiment from the raw bioassays list the selected raw bioassays are already filled in. Use the **Add raw bioassays** button to add more raw bioassays or the **Remove** button to remove the selected raw bioassays from the list.

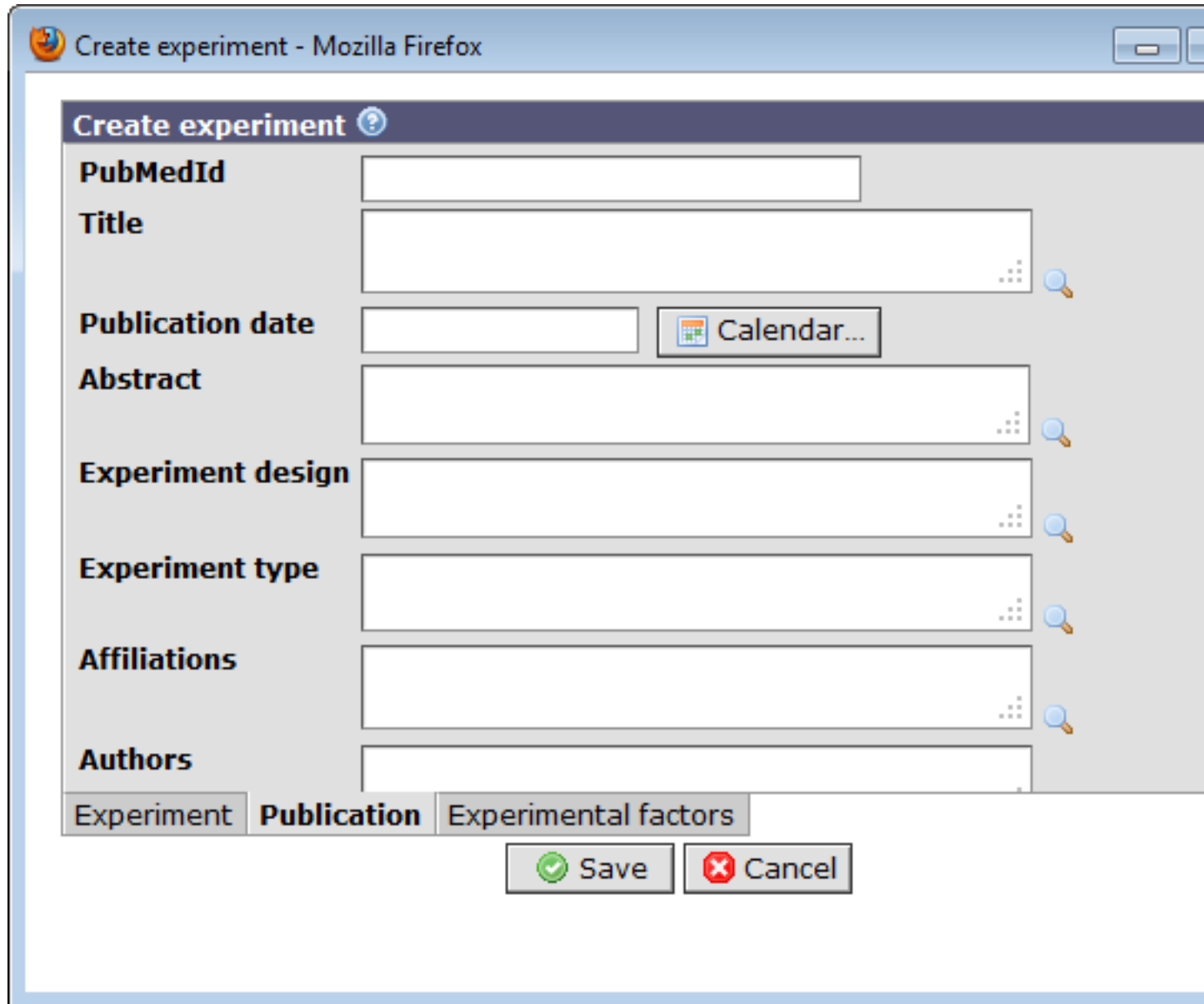
Description

A description of the experiment.

Click on the **Save** button to save the changes or on **Cancel** to abort.

The publication tab

Figure 17.5. Experiment publication



On this tab you can enter information about a publication that is the result of the experiment. All of this information is optional.

PubMedId

The ID of the publication in the PubMed¹ database.

Title

The title of the publication.

¹ <http://www.ncbi.nlm.nih.gov/pubmed/>

Publication date

The date the article was published. Use the **Calendar** button to select a date from a pop-up window.

Abstract

The article abstract.

Experiment design

An explanation of the experiment design.

Experiment type

A description of the experiment type.

Affiliations

Partners and other related organisations that have helped with the experiment.

Authors

The list of authors of the publication.

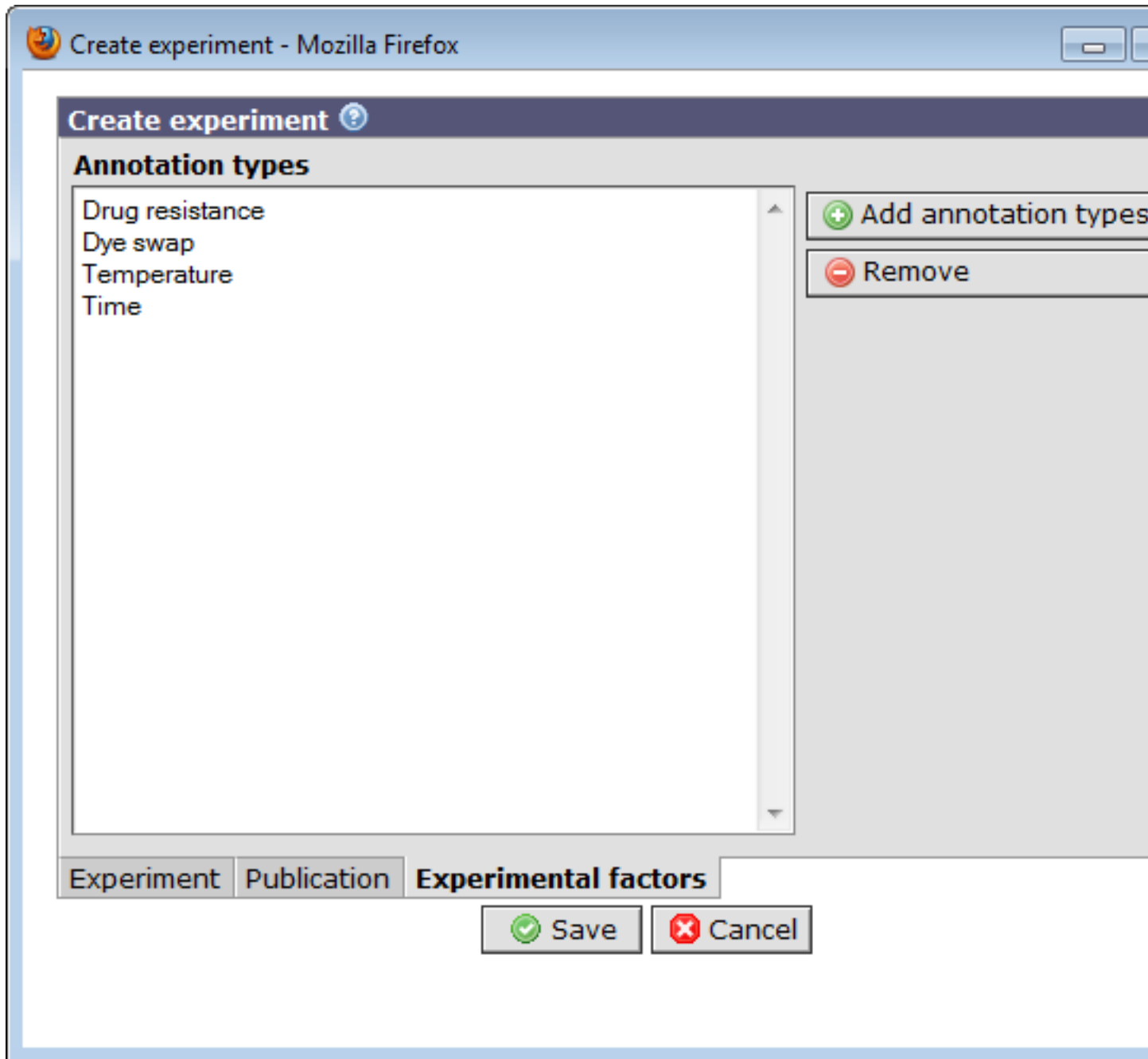
Publication

The body text of the publication.

Click on the **Save** button to save the changes or on **Cancel** to abort.

17.3.2. Experimental factors

Figure 17.6. Experimental factors



The experimental factors of an experiment are the variables you are studying in the experiment. Typically the value of an experimental factor is varied between samples or group of samples. Different treatment methods is an example of an experimental factor.

In the BASE world an experimental factor is the same as an annotation type. Since you probably have lots of annotations on your items that are not relevant for the experiment you must select the annotations types that should make up the experimental factors of the experiment.

Use the **Add annotation types** button to select the annotation types that should be used as experimental factors. The **Remove** button removes the selected annotation types.

Click on the **Save** button to save the changes or on **Cancel** to abort.

To be able to use the values of the experimental factors in the analysis of your data the values must be accessible from the raw bioassays. Since most of your annotations are probably made at the sample or biosource level the raw bioassays must inherit those annotations. Read Section 10.2.1, “Inheriting annotations from other items” (page 82) for more information about this.

Tip

Use the **Item overview** function to verify that all your raw bioassays has been annotated or inherited values for all experimental factors. If not, you should do that before starting with the analysis.

17.4. Analysing data within BASE

TODO

17.4.1. Transformations and bioassay sets

TODO

The root bioassay set

TODO

Overview plots

TODO

17.4.2. Filtering data

TODO

Formulas

TODO

17.4.3. Normalizing data

TODO

17.4.4. Other analysis plug-ins

TODO

17.4.5. The plot tool

TODO

Scatter plots

TODO

Histogram plots

TODO

Filtering plots

TODO

Save plots

TODO

17.4.6. Experiment explorer

TODO

Reporter view

TODO

Reporter search

TODO

Chapter 18. Import of data

In some places the only way to get data into BASE is to import it from a file. This typically includes *raw data*, *array design features*, *reporters* and other things, which would be inconvenient to enter by hand due to the large number of data items. There is also convenience batch importers for importing other items such as *biosources*, *samples*, and *annotations*. The batch importers are described later in this chapter after the general import description.

Normally, a plug-in handles one type of items and may require a configuration. For example, most import plug-ins need some information about how to find headers and data lines in files. BASE ships with a number of import plug-ins as a part of the core plug-ins package, cf. Section A.3, “Core import plug-ins” (page 415). The core plug-in section links to configuration examples for some of the plug-ins. Go to [Administrate Plug-ins & extensions Plug-in definitions](#) to check which plug-ins are installed on your BASE server. When BASE finds a plug-in that supports import of a certain type of item an **Import** button is displayed in the toolbar on either the list view or the single-item view.

No "Import" button?

If the import button is missing from a page where you would expect to find them this usually means that:

- The logged in user does not have permission to use the plug-in.
- The plug-in requires a configuration, but no one has been created or the logged in user does not have permission to use any of the existing configurations.

Contact the server administrator or a similar user that has permission to administrate the plug-ins.

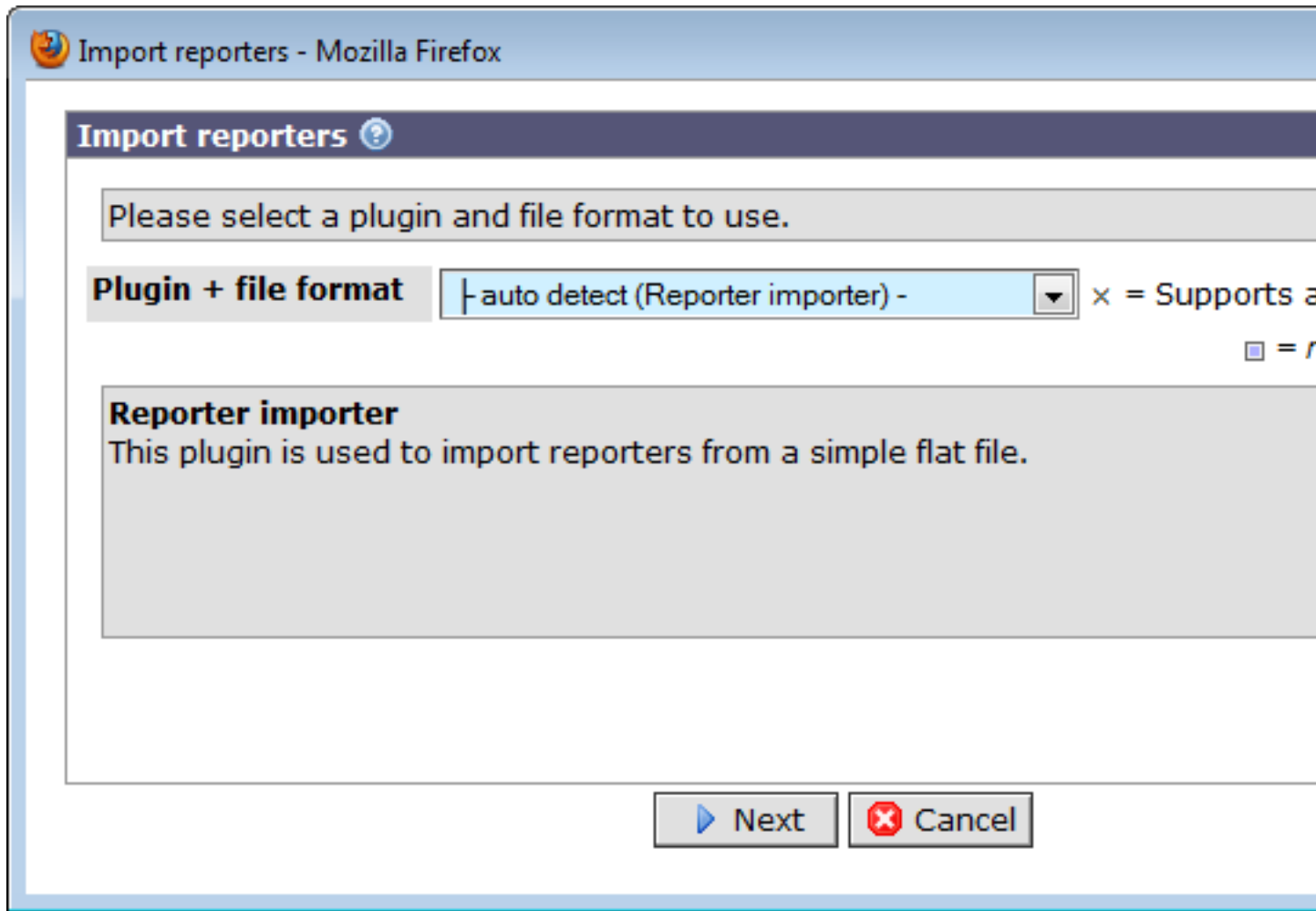
18.1. General import procedure

Starting a data import is done by a wizard-like interface. There are a number of steps you have to go through:

1. Select a plug-in and file format to use, or use the *auto detect* option.
2. If you selected the auto detection function, you must select a file to use.
3. Specify plug-in parameters.
4. Add the import job to the job queue.
5. Wait for the job to finish.

18.1.1. Select plug-in and file format

Click on the **Import** button in the toolbar to start the import wizard. The first step is to select which plug-in and, if supported, which file format to use. There is also an **auto detect** option that lets you select a file and have BASE try to find a suitable plug-in/file format to use.

Figure 18.1. Select plug-in and file format**Plugin + file format**

This is a combined list of plug-ins and their respective file format configurations. The list only includes combinations that the logged in user has permission to use. If you select an entry a short description about the plug-in and configuration is displayed below the lists. More information about the plug-ins can be found under the menu choices **Administrate Plug-ins & extensions Plug-in definitions and Administrate Plug-ins & extensions Plug-in configuration**

File format vs. Configuration

A file format is the same thing as a plug-in configuration. It may be confusing that the interface sometimes use *file format* and sometimes use *configuration*, but for now, we'll have to live with it.

Proceed to the next step by clicking on the **Next** button.

The auto detect function

The auto detect function lets you select a file and have BASE try to find a suitable plug-in and file format. This option is selected by default in the combined plug-in and file format list when there is at least one plug-in that supports auto detection.

Support of auto detect

Not all plug-ins support auto detection. The ones that do are marked in the list with **x**.

Select the **auto detect (all)** option to search for a file format in all plug-ins that supports the feature, or select the **auto detect (plugin)** option to only search the file formats for a specific plug-in. Continue to the next step by clicking on the **Next** button.

You must now select a file to import from.

Figure 18.2. Select file for auto detection

Plugin

Displays the selected plug-in or **all** if the auto-detection is used on all supporting plug-ins.

File

Enter the path and file name for the file you want to use. Use the **Browse...** button to browse after the file in BASE's file system. If the file does not exist in the file system you have the option to upload it. Read more about this in Chapter 7, *File management* (page 50).

Character set

The character set used in text files. If the selected file has been configured with a character set the correct option is automatically selected. In all cases, you have the option to override the default selection. Most files, typically use either the UTF-8 or ISO-8859-1 character set.

Recently used

A list of files you have recently used for auto detection.

Click on the **Next** button to start the auto detection. There are three possible outcomes:

- Exactly one matching plug-in and file format is found. The next step is to configure any additional parameters needed by the plug-in. This is the same step as if you had selected the same plug-in and file format in the first step.
- If no matching plug-in and file format is found an error message is displayed. If logged in with enough permissions to do so there is an option to create a new file format/configuration.
- If multiple matching plug-ins and file formats are found you will be taken back to the first step. This time the lists will only include the matching plug-ins/file formats and the auto detect option is not present.

18.1.2. Specify plug-in parameters

When you have selected a plug-in and file format or used the auto detect function to find one, a form where you can enter additional parameters for the plug-in is displayed.

Figure 18.3. Specify plug-in parameters

Select a file to import reporters from - Mozilla Firefox

Select a file to import reporters from ?

Plugin Reporter importer Configuration Reporters for project A

Here you select which file to import the reporters from, and if existing reporters should be updated or not.

☒ File
☒ Mode

File parser regular expressions...

Header
☒ Data header
☒ Data splitter
☒ Remove quotes
☐ Ignore
☐ Data footer
☒ Min data columns
☐ Max data columns
☒ Character set
☒ Decimal separator

Column mapping expressions

☒ Complex column mappings
☒ Name
☒ External ID
☐ Description

x = has value(s), = required

File
 /plates_and_reporters.mouse.v4.37k.txt Browse...

The file to import the data from

Test with file... Next Cancel

The top of the window displays the names of the selected plug-in and configuration, a list with parameters to the left, an area for input fields to the right and buttons to proceed with at the bottom. Click on a parameter in the parameter list to show the form fields for entering values for the parameter to the right. Parameters with an **X** in front of their names already have a value. Parameters marked with a blue rectangle are required and must be given a value before it is possible to proceed.

The parameter list is very different from plug-in to plug-in. Common parameters for import plug-ins are:

File

The file to import data from. A value is already set if you used the auto detect function.

File parser regular expressions

Various regular expressions that are used when parsing the file to ensure that the data is found. In most cases, all values are taken from the matched configuration and can be left as is.

Error handling

A section which contains different options how to handle errors when parsing the file. Normally you can select if the import should fail as a whole or if only the line with the error should be skipped.

Continue to the next step by clicking the **Next** button.

18.1.3. Add the import job to the job queue

Figure 18.4. Job name and options

Set job name and options - Mozilla Firefox

Set job name and options ?

Plugin	Reporter importer
Configuration	Reporters for project A
Job name	Import reporters from 'plates_and_reporters.m'
Use job agent	- automatic -
Job description	
Send message	<input checked="" type="checkbox"/> Send a message when the job is completed
Remove job	<input type="checkbox"/> Remove job when finished

☐ = required information

Finish Cancel

In this window should information about the job be filled in, like name and description. Where name is required and need to have valid string as a value. There are also two check boxes in this page.

Name

Most plug-ins should suggest a name for the job, but you can change it if you want to.

Use job agent

This option is only available if the BASE system has been configured with job agents and the logged in user has `SELECT_JOBAGENT` permission. Select the **automatic** option to let BASE automatically select a job agent or select a specific option to force the use of that particular job agent.

Send message

Tick this check box if the job should send you a message when it is finished, otherwise untick it

Remove job

If this check box is ticked, the job will be marked as removed when it is finished, on condition that it was finished successfully. This is only available for import- and export- plugins.

Clicking on **Finish** when everything is set will end the job configuration and place the job in the job queue. A self-refreshing window appears with information about the job's status and execution time. How long time it takes before the job starts to run depends on which priority it and the other

jobs in the queue have. The job does not depend on the status window to be able to run and the window can be closed without interrupting the execution.

View job status

A job's status can be viewed at any time by opening it from the job list page, View Jobs.

18.2. Batch import of data

There are in general several possibilities to import data into BASE. Bulk data such as reporter information and raw data imports are handled by plug-ins created for these tasks. For item types that are imported in more moderate quantities a suite of batch item importers are available (Section A.3.1, “Core batch import plug-ins” (page 417)). These importers allow the user to create new items in BASE and define item properties and associations between items using tab-separated (or equivalent) files.

The batch importers are available for most users and they may have been pre-configured but there is no requirement to configure the batch importer plug-ins. Here we assume that no plug-in configuration exists for the batch importers. Pre-configuration of the importers is really only needed for facilities that perform the same imports regularly whereas for occasional use the provided wizard is sufficient. Configuring the importers follows the route described in Section 21.2, “Plug-in configurations” (page 186).

The batch importers either create new items or update already existing items. In either mode the plugin can set values for

- Simple properties, *eg.*, string values, numeric values, dates, etc.
- Single-item references, *eg.*, protocol, label, software, owner, etc.
- Multi-item references are references to several other items of the same type. The extracts of a physical bioassay or pooled samples are two examples of items that refer to several other items; a physical bioassay may contain several extracts and a sample may be a pool of several samples. In some cases a multi-item reference is bundled with simple values, *eg.*, used quantity of a source biomaterial, the position an extract is used on, etc. Multi-item references are never removed by the importer, only added or updated. Removing an item from a multi-item reference is a manual procedure to be done using the web interface.

The batch importers do not set values for annotations since this is handled by the annotation importer plug-in (Section 10.2.2, “Mass annotation import plug-in” (page 84)). However, the annotation importer and batch item importers have similar behaviour and functionality to minimize the learning cost for users.

The importer only works with one type of items at each use and can be used in a *dry-run* mode where everything is performed as if a real import is taking place, but the work (transaction) is not committed to the database. The result of the test can be stored to a log file and the user can examine the output to see how an actual import would perform. Summary results such as the number of items imported and the number of failed items are reported after the import is finished, and in the case of non-recoverable failure the reason is reported.

18.2.1. File format

For proper and efficient use of the batch importers users need to understand how the files to be imported should be formatted. The input file must be organised into columns separated by a specified character such as a tab or comma character. The data header line contains the column headers which defines the contents of each column and defines the beginning of item data in the file. The item data block continues until the end of the file or to an optional data footer line defining the end of the data block.

When reading data for an item the plug-in must use some information for identifying items. Depending on item type there are two or three options to select the item identifier

- Using the internal id. This is always unique for a specific BASE server.
- Using the name. This may or may not be unique.
- Some items have an externalId. This may or may not be unique.
- Array slides may have a barcode which is similar to the externalId.

It is important that the identifier selected is *unique* in the file used, or if the file is used to update items already existing in BASE the identifier should also be unique in BASE for the user performing the update. The plug-in will check uniqueness when default parameters are used but the user may change the default behaviour.

Data for a single item may be split into multiple lines. The first line contains simple properties and single-item references, and the first multi-item reference. If there are more multi-item references they should be on the following lines with empty values in all other columns, except for the column holding the item identifier. The item identifier must have the same value on all lines associated with the item. Lines containing other data than multi-item references will be ignored or may be considered as an error depending on plug-in parameter settings. The reason for treating copied data entries as an error is to catch situations where two items is given the same item identifier by accident.

18.2.2. Running the item batch importer

This section discuss specific parameters and features of the batch importers. The general use of the batch importers follow the description outlined in Section 18.1, “General import procedure” (page 150) and the setting of column mapping parameters is assisted with the **Test with file** function described in Section 21.2.3, “The **Test with file** function” (page 189). The column headers are mapped to item properties at each use of the plug-in but, as pointed out above, they can also be predefined by saving settings as a plug-in configuration. The configuration also includes separator character and other information that is needed to parse files. The ability to save configurations depends on user credential and is by default only granted to administrators.

The plug-in parameter follows the standard BASE plug-in layout and shows help information for selected parameters. The list below comments on some of the parameters available.

Mode

Select the mode of the plug-in. The plug-in can create new items and/or update items already existing in BASE. This setting is available to allow the user to make a conscious choice of how to treat missing or already existing items. For example, if the user selects to only update items already existing the plug-in will complain if an item in the file does not exist in BASE (using default error condition treatment). This adds an extra layer of security and diagnostics for the user during import.

Data directory

This option is only available for items that has support for attaching files (eg. array design, derived bioassay, etc.). This setting is used to resolve file references that doesn't include a complete absolute path.

Identification method

This parameter defines the method to use to find already existing items. The parameter can only be set to a set of item properties listed in the plug-in parameter dialog. The property selected by the user must be mapped to a column in the file. If it is not set there is obviously no way for the plug-in to identify if an item already exists.

Item subtypes

Only look for existing items among the selected subtypes. If no subtype is selected all items are searched. If exactly one subtype is selected new items are automatically created with this subtype (unless it is overridden by specific subtype values in the import file).

Owned by me, Shared to me, In current project, and Owned by others

Defines the set of items the plug-in should look in when it checks whether an item already exists. The options are the same that are available in list views and the actual set of parameters depends in user credentials.

When id is used as the **Identification method**, the plug-in looks for the item irrespective the setting of these parameters. Of course, the user still must have proper access to the item referenced.

Column mapping expressions

Use the **Test with file** function described in Section 21.2.3, “The **Test with file** function” (page 189) to set the column mapping parameters.

When working with biomaterial items, the **Parent type** property is used to tell the plug-in how to find parent items. This only has to be set if the parent item is of the same type as the biomaterial being imported since the default is to look for the nearest parent type in the predefined hierarchy. In ascending order the BASE ordering of *parent - child - grandchild - ...* item relation is *biosource - sample - extract*.

The values accepted for **Parent type** are BIOSOURCE, SAMPLE or EXTRACT. Sometimes all items in a file to be imported have the same parent type but there is no column with this information. This can be resolved by setting the **Parent type** mapping to a constant string (eg. no backslash '\' character).

Permissions

This is a column mapping that can be used to update the permissions set on items. Normally, new items are only shared to the active project (if any). By naming a permission template, new items are shared using the permissions from that template instead. Permissions on already existing items are merged with the permission from the template.

After setting the parameters, select **Next**. Another parameter dialog will appear where error handling options can be set among with

Log file

Setting this parameter will turn on logging. The plug-in will give detailed information about how the file is parsed. This is useful for resolving file parsing issues.

Dry run

Enable or disable test run of the plug-in. If enabled the plug-in will parse and simulate an import. When enabling this option you should set the **Log file** also. The dry run mode allows testing of large imports and updates by creating a log file that can be examined for inconsistencies before actually performing the action without a safety net.

During file parsing the plug-in will look for items referenced on each line. There are three outcomes of this item search

- No item is found. Depending on parameter settings this may abort the plug-in, the plug-in may ignore the line, or a new item is created.
- One item is found. This is the item that is going to be updated.
- More than one item is found. Depending on parameter settings this may abort the plug-in or the plug-in may ignore the line.

18.2.3. Comments on the item batch importers

The item batch importers are not designed to change or create annotations. There is another plug-in for this, see Section 10.2.2, “Mass annotation import plug-in” (page 84) for an introduction to the annotation importer.

There is no need to map all columns when running the importer. When new items are created usually the only mandatory entry is Name, and when running the plug-in in update mode only the column defining the item identification property needs to be defined. This can be utilized when only one or a few properties needs to be updated; map only columns that should be changed and the plug-in will ignore the other properties and leave them as they are already stored in BASE. This also means that if one property should be deleted then that property must be mapped and the value must be empty in the file. Note, multi-item reference cannot be deleted with the batch importer, and deletion of multi-item references must be done using the web interface.

When parent and other relations are created using the plug-in the referenced items are properly linked and updated. This means that when a quantity that decreases a referenced item is used, the referenced item is updated accordingly. In consequence, if the relation is removed in a later update - maybe wrong parent was referenced - the referenced item is restored and any decrease of quantities are also reset.

A common mistake is to forget to make sure that some of the referenced items already exists in BASE, or at least are accessible for the user performing the import. Items such as protocols and labels must be added before referencing them. This is of course also true for other items but during batch import one usually follows the natural order of first importing biosources, samples, extracts, and so on. In this way the parents are always present and may be referenced without any issues.

Chapter 19. Export of data

Before data stored in BASE can be used outside of BASE, the data must first be exported to a file. When the export job finishes the file can be downloaded from the BASE file system or optionally downloaded immediately. Exporting data is possible for almost all kind of data and the export is done by a job that runs an export plug-in for the current context. An export job is started by clicking on **Export...** in the toolbar, the click action will open a pop-up window allowing you to select plug-in and specify parameters for it.

Normally, specific plug-ins handles different type of items, but some plug-ins, for example the table exporter plug-in, can work with several types of items. BASE ships with a number of export plug-ins as a part of the core plug-ins package, cf. Section A.2, “Core export plug-ins” (page 415). Go to [Administrate Plug-ins & extensions Plug-in definitions](#) to check which plug-ins are installed on your BASE server. When BASE finds a plug-in that supports export of a certain type of item an **Export** button is displayed in the toolbar on either the list view or the single-item view.

No "Export" button?

If the export button is missing from a page were you would expect to find them this usually means that:

- The logged in user does not have permission to use the plug-in.
- The plug-in requires a configuration, but no one has been created or the logged in user does not have permission to use any of the existing configurations.

Contact the server administrator or a similar user that has permission to administrate the plug-ins.

19.1. Select plug-in and configuration

This dialog is very similar to the dialog for selecting an import plug-in. See Figure 18.1, “Select plug-in and file format” (page 151) for a screenshot example.

The first thing in the configuration process is to choose which plug-in to use and a configuration for those plug-ins that require it. More information about the plug-ins can be found in each plug-in's documentation.

Note

If there is only one plug-in and configuration available, this step is skipped and you are taken directly to next step.

Plugin + configuration

Select the plug-in and configuration to use. The list only shows combinations that the logged in user has permission to use.

Click on **Next** to show the configuration of parameters for the job.

19.2. Specify plug-in parameters

The top of the window displays the names of the selected plug-in and configuration, a list with parameters to the left, an area for input fields to the right and buttons to proceed with at the bottom. Click on a parameter in the parameter list to show the form fields for entering values for the parameter to the right. Parameters with an **X** in front of their names already have a value. Parameters marked with a blue rectangle are required and must be given a value before it is possible to proceed.

The parameters list is very different from plug-in to plug-in. Common parameters for export plug-ins are:

Save as

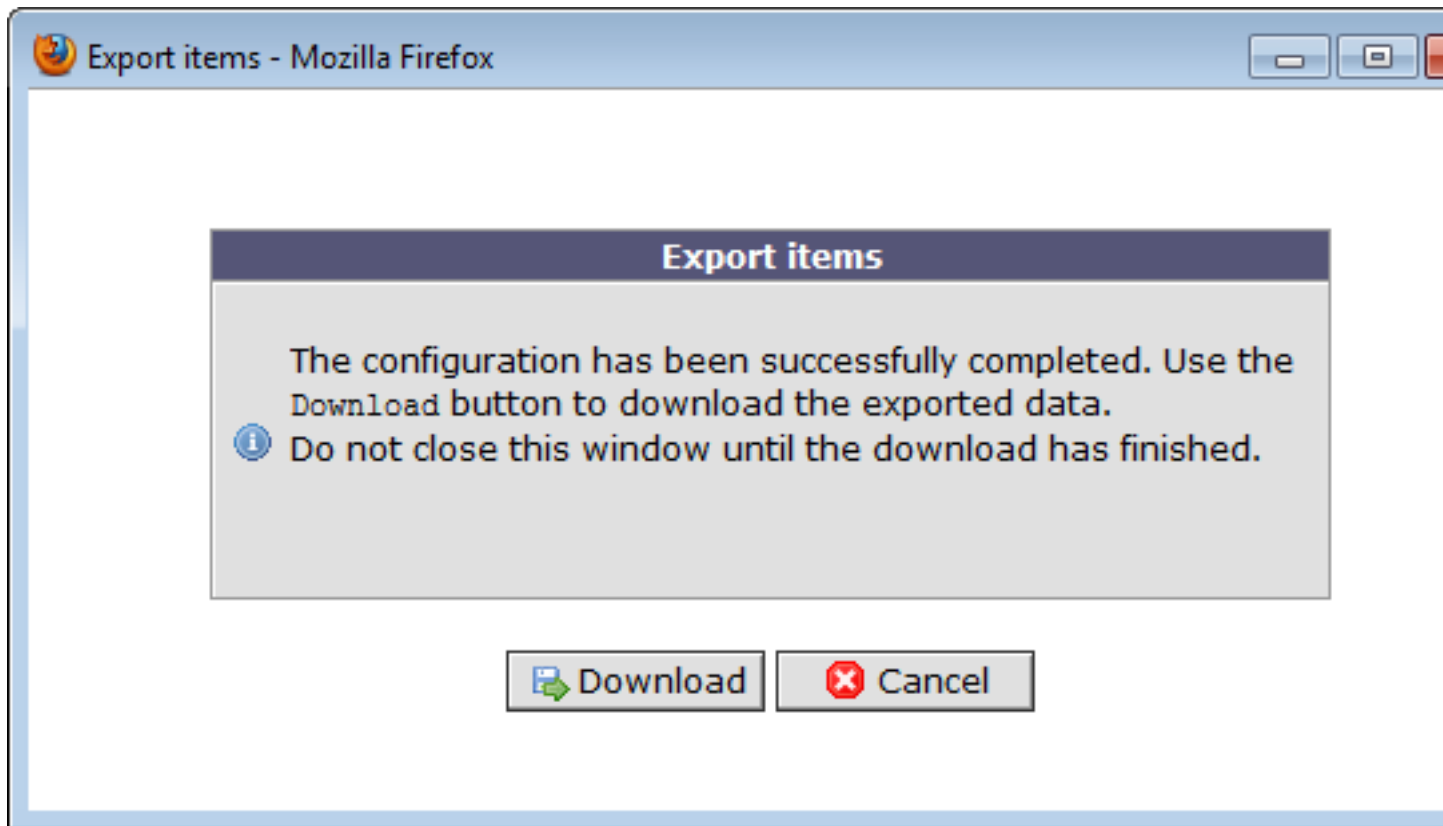
The path and file name in the BASE file system where the exported data should be saved. Some plug-ins support immediate download to the local file system if you leave the file parameter empty. For saving the exported data within the BASE file system, it's recommended to use the **Browse...** button to get the right path and then complement it with the file's name.

Click on **Next** to proceed to next configuration window.

Immediate download of the exported data

If the selected plug-in supports immediate download and the file parameter were left empty a new window with a **Download** button is displayed. Click on this button to start the plug-in execution. Do not close the window until a message saying that the export was successful (or failed) is displayed. Your browser should open a dialog asking you were to save the file on your local computer.

Figure 19.1. Download immediately

**Saving the exported data in the BASE file system**

If you choose to save the file within the BASE file system, there will be a window where the job should get a name and optionally a description. There are also two check boxes in this window.

Send message

Tick this check box if the job should send you a message when it is finished, otherwise untick it

Remove job

If this is ticked, the job will be marked as removed when it is finished, on condition that it was finished successfully. This is only available for import- and export- plugins.

By then clicking on **Finish** the configuration process will end and the job will be put in the job queue. A self-refreshing window appears with information about the job's status and execution time.

The job is not dependent on the status window to run and it therefore be closed without interrupting the execution of the job.

View job status

A job's status can be viewed at any time by opening it from the job list page, View Jobs.

19.3. The table exporter plug-in

The table exporter is a generic export plug-in that works with almost all list views in BASE. It can export the lists as an XML-file or a tab-separated text file. The table exporter is started, like the other export plug-ins, by clicking on **Export...** in the toolbar.

Then select the **Table exporter** and click on the **Next** button. The plug-in selection step is only displayed if there is more than one export plug-in that can be used in the current context. Usually, the table exporter is the only plug-in and you will be take directly to the configuration dialog.

Unlike other plug-ins, the table exporter does not use the generic parameter input dialog. It has a customized dialog that should be easier to use.

Figure 19.2. The table exporter configuration dialog

Export options ?

Format

☒ Tab-separated text file

☐ XML

Which items?

☐ Selected items

☐ Current page

☒ All pages

Which columns?

Current ▼

▲ ▼

Exported columns

Name

External ID

Gene symbol

Description

Not exported

ID

Registered

Last update

Last source

Type

Species

Cluster ID

Length

Sequence

Vector

Units

☒ Use same unit for all annotations in a column

Column prefix

Save as

Leave empty to download immediately

☐ Overwrite existing file

Ok Cancel

Format

The generated file can be either a **tab-separated text file** or an **XML** file. The XML-file option will generate a tag for each item and these will contain a child tag with property value for each selected column.

Which items?

This option decide which items that should be included in the exported data.

- **Selected items:** Only selected items will be exported. This options is not available if no items have been selected on the list page.
- **Current page:** Exports all items viewed on the current list page.

- **All pages:** Items on all pages will be exported.

Which columns?

Names of those columns that should be included in the export should be listed in the **Exported columns** to the left. A column name is moved to the other list-box by first marking it and then clicking on one of the buttons located between the list-boxes.

The order in which the columns should be exported in can be changed with the buttons to the left of the list. Simply mark a name of a column and click on the buttons to move the name either up or down in the list.

Using presets

Use the drop down list of presets, located under the option name, to easily get predefined or own presets of column settings.

Units

If the export includes annotation with units, the exporter normally uses the same unit that was used when annotating each item. This can be different from item to item. Select this option to force all annotation values to use the same unit (=the default unit for the annotation type).

Column prefix

Adds a prefix to each column header. This is useful for keeping column names unique when combining multiple files outside of BASE.

Save as

The path and file name where the exported data should be saved. Leave the text field empty if the file is to be downloaded immediately or enter a path within the BASE file system to store the file on the server. Check **Overwrite existing file** if an already existing file with the same name should be overwritten.

Click on **Ok** to proceed when all options have been set for the export.

Part III. Admin documentation

Chapter 20. Installation and upgrade instructions

Note

These instructions apply only to the BASE release which this document is a part of. The instructions here assume that Apache Tomcat 6¹ is used on the server side. Other servlet engines may work but we only test with Tomcat.

20.1. Upgrade instructions

Important information for upgrading from BASE 2.17 to BASE 3.0

Upgrading is supported from BASE 2.17 only

If your BASE is an older 2.x version you'll need to upgrade to BASE 2.17 before an upgrade to BASE 3.0 is possible.

Make sure that you have a recent backup of the BASE 2.17 database

Before starting the upgrade from BASE 2.17 to BASE 3 ensure that you have a recent backup. If the upgrade fails you must restore the 2.17 database before you can try again. The upgrade only changes the 'static' part of the database, so you do not have to restore the 'dynamic' part or the uploaded files.

Old plug-ins and extensions may not work

The BASE API has changed in several places and it is not certain that plug-ins and extensions developed for BASE 2 works with BASE 3. The upgrade will disable all plug-ins and extensions that are currently installed. Before you upgrade we recommend that you go through all (external) plug-ins and check if there is an updated version. The recommended approach is to first upgrade BASE and then install updated versions of plug-ins and extensions following the instructions in Section 21.1, "Managing plug-ins and extensions" (page 178).

If there is no updated version of a specific plug-in you may try a manual re-installation of the old plug-ins. Follow the instructions in Section 21.1.2, "Manual plug-in registration" (page 181).

If there is no updated version and the old plug-in doesn't work with BASE 3, you'll need to decide if you really need the plug-in or if the upgrade should wait until a new version of the plug-in has been released.

Batch item importer changes

There are several changes to batch item importers that may affect current workflows and file templates used for importing data.

- Sample and extract importers: The 'pooled' column is no longer used. Instead a 'parent type' column should be used with the parent type as a string value (BIOSOURCE, SAMPLE or EXTRACT). Existing importer configurations and file templates may have to be updated. If no parent type is specified the sample importer assumes a biosource and the extract importer assumes a sample.
- Labeled extract importer: This has been deprecated and it is recommended that the *Extract importer* is used instead. We recommend that existing labeled extract importer configurations are re-created as extract importer configurations. The old labeled extract importer can

¹ <http://tomcat.apache.org/>

be re-enabled, but note that the existing configurations still need to be changed due to the 'pooled' column is no longer used.

- Hybridization importer: This has been deprecated and we recommend that the *Physical bioassay importer* is used instead. Existing hybridization importer configurations should be re-created as physical bioassay importer configurations.
- Scan importer: This has been deprecated and it is recommended that the *Derived bioassay importer* is used instead. Existing scan importer configurations should be re-created as derived bioassay importer configurations.

Note

The deprecated importers can be re-enabled by an administrator from the Administrative Plug-ins & extensions Overview page, but they are lacking features that are available in the new importers so this is not something that we recommend.

MySQL and PostgreSQL versions

We have only tested BASE 3 with PostgreSQL 9.1. If anyone experiences any issues with earlier PostgreSQL versions, we recommend an upgrade to PostgreSQL 9.1. This is a change since BASE 2 which was tested with PostgreSQL 8.4. Even though BASE 3 may work with older PostgreSQL versions, we don't have the resources needed to test and provide support for it.

We have only tested BASE 3 with MySQL 5.1 (no change since BASE 2). If anyone experiences any issues with earlier (or later) MySQL versions, we recommend an upgrade/downgrade to MySQL 5.1.

As always, backup your database before attempting an upgrade. The BASE team performs extensive testing before releasing a new version of BASE but there are always a possibility for unexpected events during upgrades. In upgrades requiring a change in the underlying database there is no (supported) way to revert to a previous version of BASE using BASE tools, you need to use your backup for this use case.

The strategy here is to install the new BASE release to another directory than the one in use. This requires transfer of configuration settings to the new install but more on that below.

Shut down the Tomcat server

If the BASE application is not shut down already, it is time to do it now. Do something like **sudo /etc/init.d/tomcat6.0 stop**

Notify logged in users!

If there are users logged in to your BASE server, it may be nice of you to notify them a few minutes prior to shutting down the BASE server. See Section 20.4.1, "Sending a broadcast message to logged in users" (page 176).

Rename your current server

Rename your current BASE installation **mv /path/to/base /path/to/base_old**.

Download and unpack BASE

There are several ways to download BASE. Please refer to section Section 3.1.1, "Download" (page 9) for information on downloading BASE, and select the item matching your download option:

Pre-compiled package

If you selected to download a pre-compiled package, unpack the downloaded file with **tar xzpf base-...tar.gz**.

Source package

If you selected to download a source package, unpack the downloaded file with **tar xzpf base-...src.tar.gz**. Change to the new directory, and issue **ant package.bin**. This will create a binary package in the current directory. Unpack this new package (outside of the source file hierarchy).

Subversion checkout

This option is for advanced users only and is not covered here. Please refer to <http://base.thep.lu.se/wiki/BuildingBase> for information on this download option.

Transfer files and settings

Settings from the previous installation must be transferred to the new installation. This is most easily done by comparing the configuration files from the previous install with the new files. Do not just copy the old files to the new install since new options may have appeared.

In the main BASE configuration file, `<base-dir>/www/WEB-INF/classes/base.config`, fields that needs to be transferred are usually `db.username`, `db.password`, and `userfiles`.

Local settings in the raw data tables, `<base-dir>/www/WEB-INF/classes/raw-data-types.xml`, may need to be transferred. This also includes all files in the `<base-dir>/www/WEB-INF/classes/raw-data-types` and `<base-dir>/www/WEB-INF/classes/extended-properties` directories.

Updating database schema

It is recommended that you also perform an update of your database schema. Running the update scripts are not always necessary when upgrading BASE, but the running the update scripts are safe even in cases when there is no need to run them. Change directory to `<base-dir>/bin/` and issue

```
sh ./updatedb.sh [base_root_login] base_root_password
sh ./updateindexes.sh
```

where `base_root_login` is the login for the root user and `base_root_password` is the password. The login is optional. If not specified, `root` is used as the login.

Start Tomcat

Start the Tomcat server: **`sudo /etc/init.d/tomcat6.0 start`**

Done! Upgrade of BASE is finished.

20.2. Installation instructions

Java

Download and install Java SE 6, available from <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. You can select either the JDK or the JRE version. We have only made a few tests with BASE and Java SE 7. While it seems to be working just fine we can't make any promises or provide any support for it.

Tomcat

Download and install Apache Tomcat 6.0.20 or any later 6.x release, available from <http://tomcat.apache.org>. We have only made a few tests with BASE and Tomcat 7. While it seems to be working fine (except for the XJSP compiler) we can't make any promises or provide support for it. There are a few configuration options that are needed to make Tomcat work properly with BASE. The options are set in the `CATALINA_OPTS` environment variable.

- Increase the amount of memory that Tomcat is allowed to use. The default setting is usually not enough. To give Tomcat 1 gigabyte, use `-Xmx1G`.
- Disable strict parsing of JSP files. `-Dorg.apache.jasper.compiler.Parser.STRICT_QUOTE_ESCAPING=false`
- Unless you have manually downloaded and installed JAI (Java Advanced Imaging) native acceleration libraries (see <http://java.sun.com/javase/technologies/desk->

top/media/jai/) it is a good idea to disable the native acceleration of JAI. -
Dcom.sun.media.jai.disableMediaLib=true

- Enable headless mode if your are setting up a server which doesn't have a display device connected to it. -Djava.awt.headless=true.

Depending on your system there are probably several ways to set the the CATALINA_OPTS variable. One suggestion is to add the following line (as a single line) close to the top of the catalina.sh script that comes with Tomcat (directory bin):

```
CATALINA_OPTS="-Xmx1G  
-Dorg.apache.jasper.compiler.Parser.STRICT_QUOTE_ESCAPING=false  
-Dcom.sun.media.jai.disableMediaLib=true  
-Djava.awt.headless=true"
```

For more information about Tomcat options see <http://tomcat.apache.org/tomcat-6.0-doc/index.html>.

Set up an SQL database

BASE utilize Hibernate² for object persistence to a relational database. Hibernate supports many database engines, but so far we only work with MySQL³ and PostgreSQL⁴.

MySQL

Download and install MySQL (tested with version 5.1), available from <http://www.mysql.com/>. You need to be able to connect to the server over TCP, so the *skip-networking* option must **not** be used. The InnoDB table engine is also needed, so do not disable them (not that you would) but you may want to tune the InnoDB behaviour before creating BASE databases. BASE comes pre-configured for MySQL so there is no need to change database settings in the BASE configuration files.

PostgreSQL

PostgreSQL 9.1 seems to be working very well with BASE and Hibernate. Download and install PostgreSQL, available from <http://www.postgresql.org/>. You must edit your <base-dir>/www/WEB-INF/classes/base.config file. Uncomment the settings for PostgreSQL and comment out the settings for MySQL.

BASE (download and unpacking)

Download BASE⁵ and unpack the downloaded file, i.e. **tar xzpf base-...tar.gz**. If you prefer to have the bleeding edge version of BASE, perform a checkout of the source from the subversion repository (subversion checkout instructions at BASE trac site⁶).

If you choose to download the binary package, skip to the next item. The rest of us, read on and compile BASE. If you downloaded a source distribution, unpack the downloaded file **tar xzpf base-...src.tar.gz**, or you may have performed a subversion checkout. Change to the 'root' base2 directory, and issue **ant package.bin**. This will create a binary package in the base2 'root' directory. Unpack this new package (outside of the source file hierarchy), and from now on the instructions are the same irrespective where you got the binary package.

This section is intended for advanced users and programmers only. In cases when you want to change the BASE code and try out personalized features it may be advantageous to run the tweaked BASE server against the development tree. Instructions on how to accomplish this is available in the building BASE document⁷. When you return back

² <http://www.hibernate.org/>

³ <http://www.mysql.com>

⁴ <http://www.postgresql.org/>

⁵ <http://base.thep.lu.se/wiki/DownloadPage>

⁶ <http://base.thep.lu.se/wiki/DownloadPage>

after compiling in the subversion tree you can follow the instruction here (with obvious changes to paths).

BASE (database engine)

Instructions for MySQL and PostgreSQL are available below. The database names (base2 and base2dynamic is used here), the *db_user*, and the *db_password* can be changed during the creation of the databases. It is recommended to change the *db_password*, the other changes can be made if desired. The database names, the *db_user*, and the *db_password* are needed in a later step below when configuring BASE.

Note

Note that the *db_user* name and *db_password* set here is used internally by BASE in communication with the database and is never used to log on to the BASE application.

The database must use the UTF-8 character set

Otherwise there will be a problem with storing values that uses characters outside the normal Latin1 range, for example unit-related such as μ (micro) and Ω (ohm).

MySQL

Create a new database for BASE, and add a *db_user* with at least *SELECT*, *INSERT*, *UPDATE*, *DELETE*, *CREATE*, *DROP*, *INDEX*, and *ALTER* permission for the new database. To do this, connect to your MySQL server and issue the next lines:

```
CREATE DATABASE base2 DEFAULT CHARACTER SET utf8;
CREATE DATABASE base2dynamic DEFAULT CHARACTER SET utf8;
GRANT ALL ON base2.* TO db_user@localhost IDENTIFIED BY 'db_password';
GRANT ALL ON base2dynamic.* TO db_user@localhost;
```

The `<base-dir>/misc/sql/createdb.mysql.sql` file contains the above statements and can be used by the `mysql` command-line tool (remember to edit the *db_user*, *db_password*, and the database names in the script file before executing the command): **`mysql -uroot -p < ./misc/sql/createdb.mysql.sql`**. The header in the script file contains further information about the script.

PostgreSQL

Create a new database for BASE, and add a *db_user* with the proper privileges. To do this, log in as your PostgreSQL user and issue these lines (omit comments):

```
createuser db_user -P
# this will prompt for an password for the new user, and issue two
# more question that should be answered with character 'n' for no.
createdb --owner db_user --encoding UTF8 base2
psql base2
# this will start the psql command line tool. Issue the next line
# within the tool and quit with a '\q'.
CREATE SCHEMA "dynamic" AUTHORIZATION "db_user";
```

The `<base-dir>/misc/sql/createdb.postgresql.sql` file contains the above statements and can be used by the `psql` command-line tool: **`psql -f ./misc/sql/createdb.postgres.sql`** **template1** The header in the script file contains further information about the script.

BASE (file storage setup)

An area for file storage must be setup. Create an empty directory in a proper location in your file system, and set the owner to be the same as the one that the Tomcat server will be running as. Remember this location for later use. The default location is `/usr/local/base2/files`.

BASE (plug-in setup)

An area for plug-in and extensions installation must be setup. Create an empty directory in a proper location in your file system, and make sure that the user that the Tomcat server will be running as has read permission. Remember this location for later use. The default location is `/usr/local/base2/plugins`.

BASE (configuration)

Basic BASE configuration is done in `<base-dir>/www/WEB-INF/classes/base.config`:

- Uncomment the database engine section that match your setup.
- Modify the `db.url`, `db.dynamic.catalog`, `db.username`, and `db.password` settings to match your choice above. (*database host and database name (e.g. base2), e.g. base2dynamic, db_user, and db_password, respectively.*)
- Modify the `userfiles` setting to match your choice in file storage setup above.
- Modify the `plugins.dir` setting to match your choice in plug-in setup above.

See the Appendix B, *base.config reference* (page 418) for more information about the settings in the `base.config` file.

Optional but recommended. You may want to modify extended properties to fit your needs. Extended properties are defined in `<base-dir>/www/WEB-INF/classes/extended-properties.xml`. There is an administrator document discussing extended properties⁸ available. If you plan to perform a migration of a BASE version 1.2 database you should probably not remove any extended properties columns (this is not tested so the outcome is currently undefined). However, adding columns does not affect migration.

BASE (database initialization)

Change directory to `<base-dir>/bin` and execute the following commands:

```
./initdb.sh [base_root_login] base_root_password
./updateindexes.sh
```

The second command is important for PostgreSQL users since the Hibernate database initialisation utility is not able to create all indexes that are required. BASE will still work without the indexes but performance may suffer.

Important

The `base_root_login` and `base_root_password` you use here is given to the BASE web application root user account. The `base_root_login` is optional. If not specified, `root` is used for the login.

If the initialisation script fail, it is most probably a problem related to the underlying database. Make sure that the database accepts network connection and make sure that `db_user` has proper credentials.

BASE and Tomcat

Either move the `<base-dir>/www` directory to the Tomcat `webapps` directory or create a symbolic link from the Tomcat `webapps` directory to the `<base-dir>/www` directory

```
cd /path/to/tomcat/webapps
ln -s /path_to_base/www base2
```

If you plan to install extensions you should make sure that the `<base-dir>/www/extensions` directory is writable by the user account Tomcat is running as.

Start/restart Tomcat, and try `http://hostname:8080/base2` (change `hostname` to your host-name) in your favourite browser. The BASE log-in page should appear after a few seconds.

BASE, Apache, and Apache/Tomcat connector

This step is optional.

⁸ <http://base.thep.lu.se/chrome/site/doc/historical/admin/extended-properties.html>

If you want run the Tomcat server through the Apache web server, you need to install the Apache version 2 web server, available from <http://httpd.apache.org/>, and a apache-tomcat connector, available from <http://tomcat.apache.org/connectors-doc/index.html>

Setup done!

Happy BASEing. Now you can log on to your BASE server as user *root* (use the *base_root_password* from the database initialization step above). You should begin with creating a couple user accounts, for more information on how to create user accounts please refer to Chapter 22, *Account administration* (page 194).

20.3. Installing job agents

It is important to understand that the BASE application can be spread on to several computers. The main BASE application is serving HTTP requests, the underlying database engine is providing storage and persistence of data, and job agents can be installed on computers that will serve the BASE installation with computing power and perform analysis and run plug-ins. In a straight forward setup one computer provides all services needed for running BASE. From this starting point it is easy to add computers to shares load from the BASE server by installing job agents on these additional computers.

A job agent is a program running on a computer regularly checking the BASE job queue for jobs awaiting execution. When the job agent finds a job that it is enabled to execute, it loads the plug-in and executes it. Job agents will in this way free up resources on the BASE application server, and thus allow the BASE server to concentrate on serving web pages. Job agents are optional and must be installed and setup separately. However, BASE is prepared for job agent setup and to utilize the agents, but the agent are not required.

A job agent supports many configuration options that are not supported by the internal job queue. For example, you can

- Specify exactly which plug-ins each job agent should be able to execute.
- Give some plug-ins higher priority than other plug-ins.
- Specify which users/groups/projects should be able to use a specific job agent.
- Override memory settings and more for each plug-in.
- Execute plug-ins in separate processes. Thus, a misbehaving plug-in cannot bring the main application server down.
- Add more computers with job agents as needed.

All these options make it possible to create a very flexible setup. For example one job agent can be assigned for importing data only, another job agent can be assigned for running analysis plug-ins for specific project only, and a third may be a catch-all job agent that performs all low-priority jobs.

20.3.1. BASE application server side setup

Make sure the internal job queue doesn't execute all plug-ins

The setting **jobqueue.internal.runallplugins** should be set to **false** for the BASE server. This setting is found in the `<base-dir>/www/WEB-INF/classes/base.config` file. The changes will not take effect until the application server is restarted.

Enable the job agent user account

During installation of BASE a user account is created for the job agent. This account is used by the job agents to log on to BASE. The account is disabled by default and must be enabled. Enable the account and set a password using the BASE web interface. The same password must

also be set in the `jobagent.properties` file, see item [Edit the `jobagent.properties` file](#) (page 172) below.

20.3.2. Database server setup

Create a user account on the database

This is similar to granting database access for the BASE server user in the regular BASE installation, cf. [BASE \(database engine\)](#) (page 169). You must create an account in the database that is allowed to connect from the job agent server. MySQL example:

```
GRANT ALL ON base2.* TO db_user@job.agent.host IDENTIFIED BY 'db_password';
GRANT ALL ON base2dynamic.* TO db_user@job.agent.host;
```

Replace **job.agent.host** with the host name of the server that is going to run the job agent. You should also set password. This password is used in item [Edit the `base.config` file](#) (page 172) below in job agent server setup. You can use the same database user and password as in the regular database setup.

20.3.3. Job agent client setup

Download and unpack a regular BASE distribution

You *must* use the same version on the web server and all job agents. You find the downloads at <http://base.thep.lu.se/wiki/DownloadPage>

Edit the `base.config` file

The `<base-dir>/www/WEB-INF/classes/base.config` file must be configured as in regular BASE installation, cf. [BASE \(configuration\)](#) (page 170), to use the same database as the web server application. The most important settings are

- **db.username:** The database user you created in item [Create a user account on the database](#) (page 172) above.
- **db.password:** The password for the user.
- **db.url:** The connection url to the database.
- **userfiles:** The path to the directory where user files are located. This directory must be accessible from all job agents, i.e., by nfs or other file system sharing method.
- **plugins.dir:** The path to the directory where plug-ins are located. This directory must be accessible from all job agents, i.e., by nfs or other file system sharing method.

See the Appendix B, *base.config reference* (page 418) for more information about the settings in the `base.config` file.

Edit the `jobagent.properties` file

The `<base-dir>/www/WEB-INF/classes/jobagent.properties` file contains settings for the job agent. The most important ones to specify value for are

- **agent.password:** The password you set for the job agent user account in item [Enable the job agent user account](#) (page 171) above.
- **agent.id:** An ID that must be unique for each job agent accessing the BASE application.
- **agent.remotecontrol:** The name or ip address of the web server if you want it to be able to display info about running jobs. The job agent will only allow connections from computers specified in this setting.

The `jobagent.properties` file contains many more configuration options. See the Appendix F, *jobagent.properties reference* (page 436) for more information.

Register the job agent

From the `bin` directory, register the job agent with

```
./jobagent.sh register
```

Start the job agent

From the `bin` directory, start the job agent with

```
./jobagent.sh start &
```

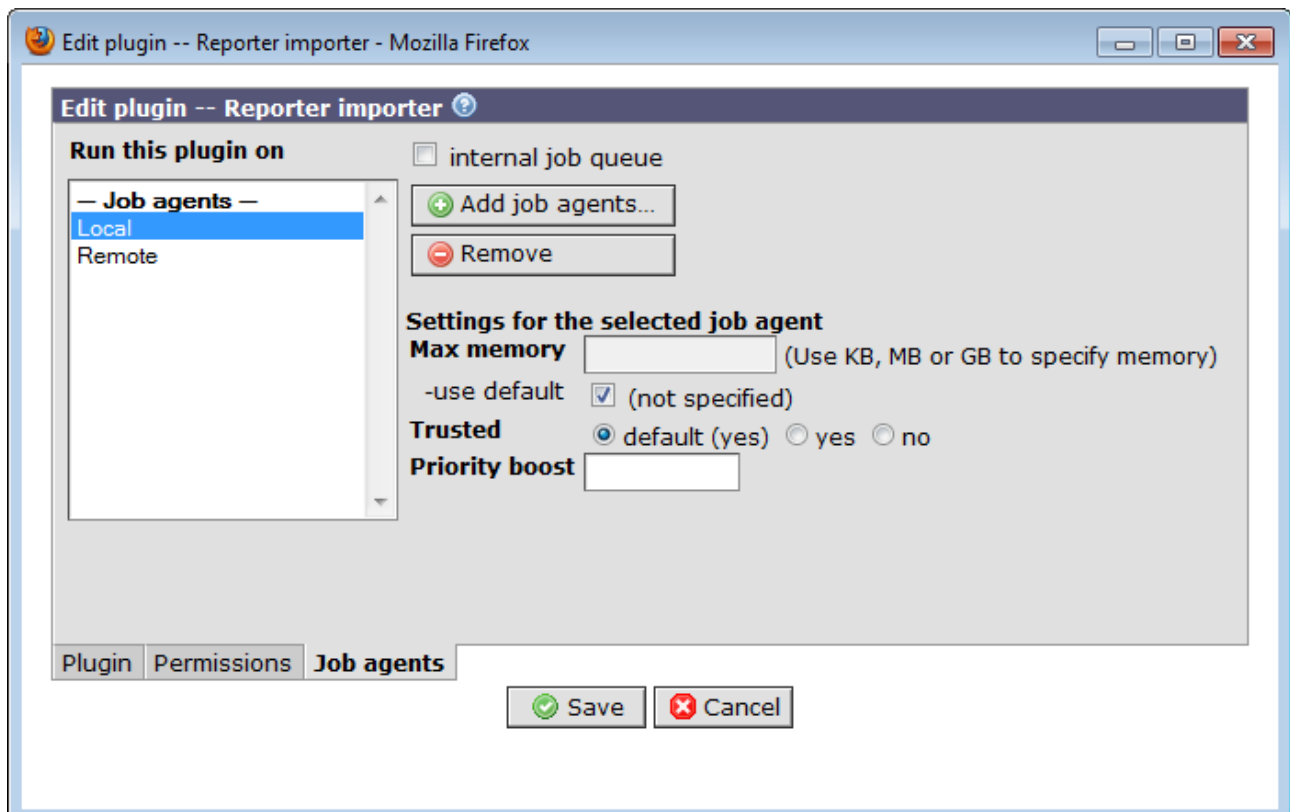
See the Appendix G, *jobagent.sh reference* (page 440) for more information about what you can do with the job agent command line interface.

20.3.4. Configuring the job agent

A job agent will not execute a plug-in unless the administrator has configured the job agent to do so. There are two things that must be done:

- Share the job agent to the users, groups and project that should be able to use it. If the job agent is not shared, only the owner of job agent is allowed to use it. Use the regular **Share** functionality to specify which users/groups/projects should be able to use the job agent. You must give them at least **USE** permission. To give all users permission to the job agent share it to the **EVERYONE** group.
- Selecting plug-ins that the job agent should handle. This can be done either from the plug-in pages or from the job agent pages. To register a plug-in with one or more job agents from the plug-in pages, go to the edit view of the plug-in and select the **Job agents** tab. To do the same from the job agent pages, go to the edit view of the job agent and select the **Plugins** tab. The registration dialogs are very similar but only the plug-in side of registration is described here. The major difference is that it is not possible to enable/disable the internal job queue for plug-in when using the jobagent side of the registration.

Figure 20.1. Select job agents for a plug-in



Use this tab to specify which job agents the plug-in is installed and allowed to be executed on.

Run this plugin on

You may select if the internal job queue should execute the plug-in or not.

Job agents

A list with the job agents where the plug-in is installed and allowed to be executed. Select a job agent in this list to display more configuration options for the plug-in.

Add job agents

Use this button to open a pop-up window for selecting job agents.

Remove

Remove the selected plug-in from the list.

The following properties are only displayed when a job agent has been selected in the list. Each job agent may have it's own settings of these properties. If you leave the values unspecified the job agent will use the default values specified on the **Plugin** tab.

Max memory

The maximum amount of memory the plug-in is allowed to use. Add around 40MB for the Java run-time environment and BASE. If not specified Java will choose it's default value which is 64MB.

Trusted

If the plug-in should be executed in a protected or unprotected environment. Currently, BASE only supports running plug-ins in an unprotected environment.

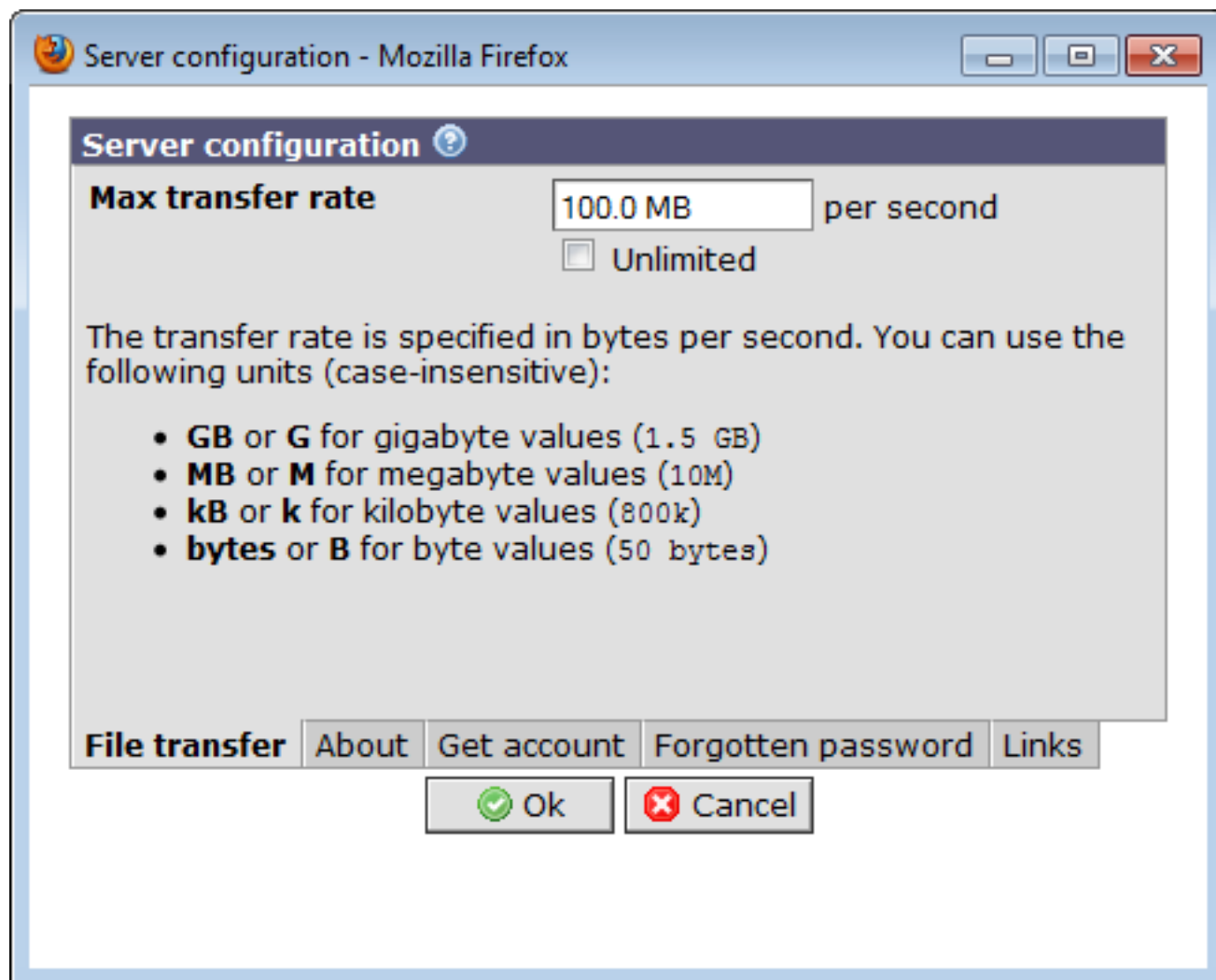
Priority boost

Used to give a plug-in higher priority in the job queue. Values between 0 and 10 are allowed. A higher value will give the plug-in higher priority. The priority boost is useful if we, for example, want to use one server mainly for importing data. By giving all import plugins a priority boost they will be executed before all other jobs, which will have to wait until there are no more waiting imports.

20.4. Server configurations

Some server configurations can be done when the installation process is finished and BASE is up and running. Log into BASE with administration rights and then open the configuration dialog from menu **Administrate** **Server settings**. Each tab in the configuration dialog-window is described below.

Figure 20.2. Server configuration



File transfer

Max transfer rate

This is a limit of how many bytes of data that should be transferred per second when uploading files to BASE. Prefixes like k, M or G can be used for larger values, just like described in the tab. The limit is per ongoing upload and the default value is 100MB/s.

Unlimited

Check this to not limit the transfer rate. In this case, the Internet connection of the server is the limit.

About

Administrator name

Name of the responsible administrator. The name is displayed at the bottom of each page in BASE and in the about-dialog.

Administrator email

An email which the administrator can be contacted on. The administrator name, visible at the bottom of each page, will be linked to this email address.

About

Text written in this field is displayed in the **About this server** section on the login page and in the about-dialog window. We recommend changing the default Latin text to something meaningful, or remove it to hide the section completely.

Get account

A description what a user should do to get an account on the particular BASE server. This text is linked to the **Get an account!** link on the login page. We recommend that the Latin text is replaced with some useful information, or that it is removed to hide the link.

Forgotten password

A description what a user should do if the password is forgotten. This text is linked to the **Forgot your password?** link on the login page. We recommend that the Latin text is replaced with some useful information, or that it is removed to hide the link.

Links

External configurable link-types inside BASE.

Note

Only link-types that have been set will be visible in the web client.

Help

Links to where the help text is located. By default this is set to the documentation for the latest released BASE version on the BASE web site, <http://base.thep.lu.se/chrome/site/doc/html/index.html>⁹. If you want the documentation for a specific version you will have to setup a site for that yourself and then change the link to that site. The documentation is included in the downloaded package in the directory `<basedir>/doc/html`.

FAQ

Where frequently asked questions can be found. Empty by default.

Report a bug

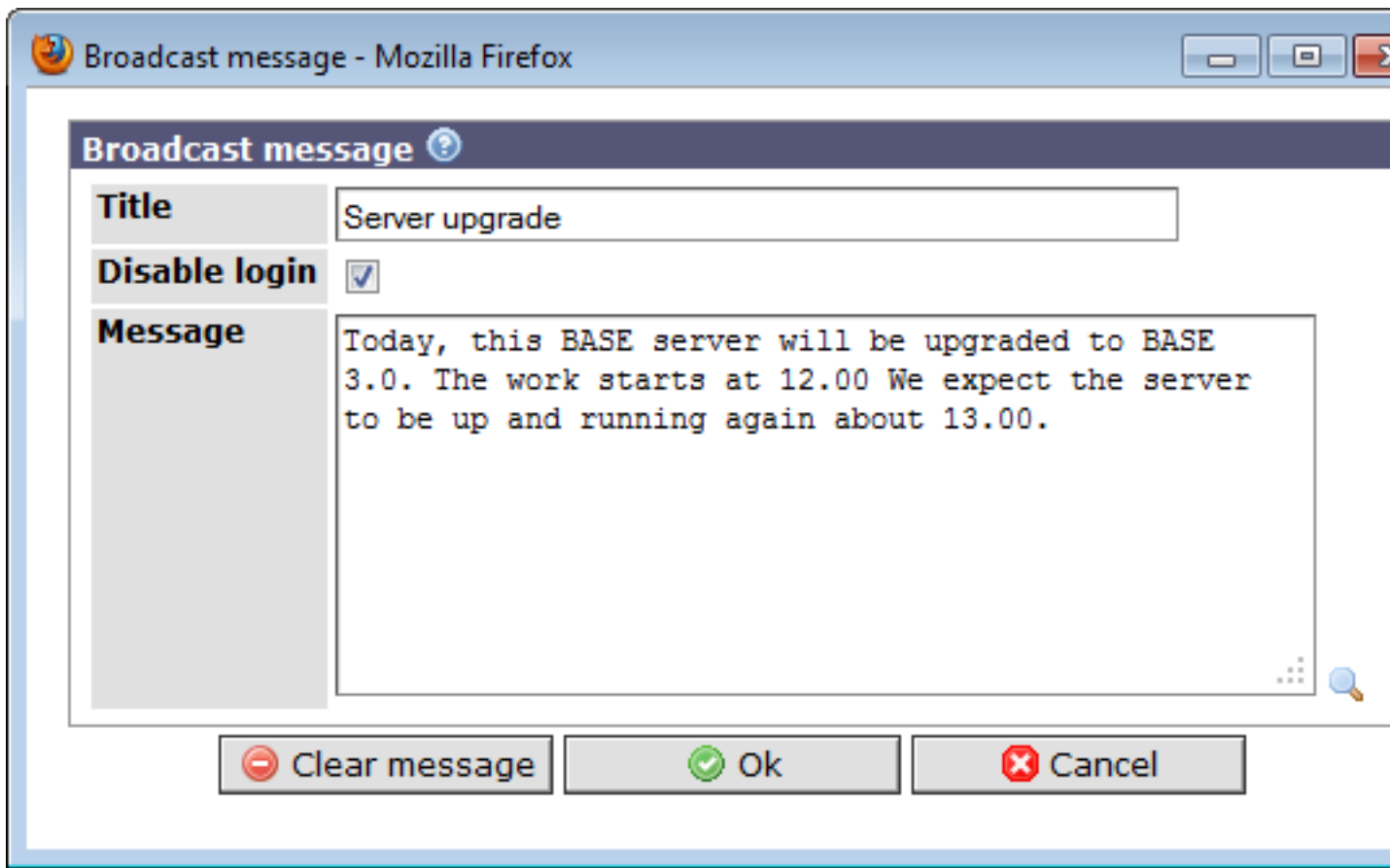
Where the user could report bugs, feature request or perhaps other feedback that concerns the program. As default this is set to the feedback section on BASE web site, <http://base.thep.lu.se/#Feedback>. Note that users must login in order to submit information.

20.4.1. Sending a broadcast message to logged in users

It is possible to send a message to all logged in user. Open the Administrative Broadcast message dialog box.

⁹ <http://base.thep.lu.se/chrome/site/doc/html/index.html>

Figure 20.3. Broadcast message



This dialog allows you to specify a message that is sent to all logged in users as well as on the login form. It is also possible to "disable" login.

Title

The title of the message. It should be a short and concise to avoid confusion. The title will be displayed on a lot of places and a user may have to click on it to read the more detailed message.

Disable login

Mark this check-box to try to prevent new users from logging in. To avoid problems that can be caused by blocking the server admin out, the login is not completely disabled. Any user can still login but only after by-passing several warnings.

Message

If needed, a longer message giving more information. Users may have to click on a link to be able to see the complete message.

Note

The message will be enabled until it is manually removed by saving an empty form, or until the Tomcat server is restarted. Since the message is only kept in memory, a restart will always remove it.

Chapter 21. Plug-ins and extensions

BASE can get extended functionality by the use of plug-ins and extensions. Much of the hard work, such as data import/export and analysis is done with plug-ins. BASE ships with a number of standard plug-ins, the core plug-ins, which gives basic import/export and analysis functionality. Typically a plug-in interacts with a user by asking for parameters that it need to be able to do it's work. For example, which file to import data from, and maybe some regular expressions that should be used when parsing the file and then some information about how the data in the file should be mapped to items and properties in BASE. When the plug-in has all parameters it needs a **Job** is added to a job queue. A job agent or similar is then responsible for scheduling and executing (possibly on a different machine) the plug-in code.

Extensions are historically more targeted at additions to the user interface, such as additional menu items, toolbar buttons, etc. As a result, extensions have a lot more flexibility when it comes to the visual appearance. On the other hand they are executed immediately as a result of user interaction and are expected to perform quickly and without delay.

Starting with BASE 3 the extension mechanism has been somewhat extended to cover other things that are not directly related to the web interface. For example, extensions can be used to add support for other protocols than HTTP when using external files. The main difference between a plug-in and extension is that an extension must execute immediately it's service is requested, but a plug-in can be scheduled for later execution.

21.1. Managing plug-ins and extensions

Changes since BASE 2

The plug-in and extensions installation has changed since BASE 2. The major changes are:

- The main JAR file must be installed in the directory specified by the `plugins.dir` setting in `base.config`. Subdirectories are not allowed. This applies to both plug-ins and extensions. The `WEB-INF/extensions` is not used for extensions anymore.
- A package must be installed as a whole. It is no longer possible to only select some of the plug-ins to install. If necessary, the administrator can always disable plug-ins that is not wanted.

The first step is to install the actual code on the web server. The recommendation to developers is to ship the entire package as a single JAR file. If everything is JAVA based this should not be a problem. A common exception is that configuration files should be installed (and configured) separately. Always read the installation instructions for the package you are installing. The rest of the instructions in this section assume that the plug-in/extensions comes as a single JAR file.

Make sure the extensions folder is writable by Tomcat

The package you are installing may include resources such as HTML files, JSP scripts, images, etc. that needs to be extracted to the web application path before they can be used. This extraction is automatically done by the installation wizard, but you have to make sure that the user account Tomcat is running as has permission to create (and delete) new files in the `<base-dir>/www/extensions` directory.

So, the first step should be simple. Just put the JAR file in the dedicated plug-ins directory. This is the directory that is specified in the `plugins.dir` setting in `base.config`.

21.1.1. Automatic installation wizard

When the plug-in/extensions package is installed on the server you must register it with BASE. Go to [Administrate Plug-ins & extensions Overview](#).

Figure 21.1. Installed extensions & plug-ins

Firefox ▾

BASE 3.0.0 @ base2.thep.lu.se --

base2.thep.lu.se/demo/

BASE View Biomaterial LIMS Array LIMS Administrate Help - no active extensions

Installed extensions & plug-ins

By extension point

- Toolbars
- Edit dialogs
- Bioassay set: Overview plots
- MA/Correction factor plots [web-ex]
- Bioassay set: Tools
- Connection manager
- File set validators
- Menu: extensions
- Services

By file

- core-extensions.xml
- Connection manager
- File set validators
- CDF file validator [File set validators]
- CEL file validator [File set validators]
- GTF file validator [File set validators]
- HTTP(s) connection manager [Con]
- core-plugins.xml
- web-extensions.xml

Install/uninstall... Help...

Last installation Successful

- ended 2011-10-12 11:11

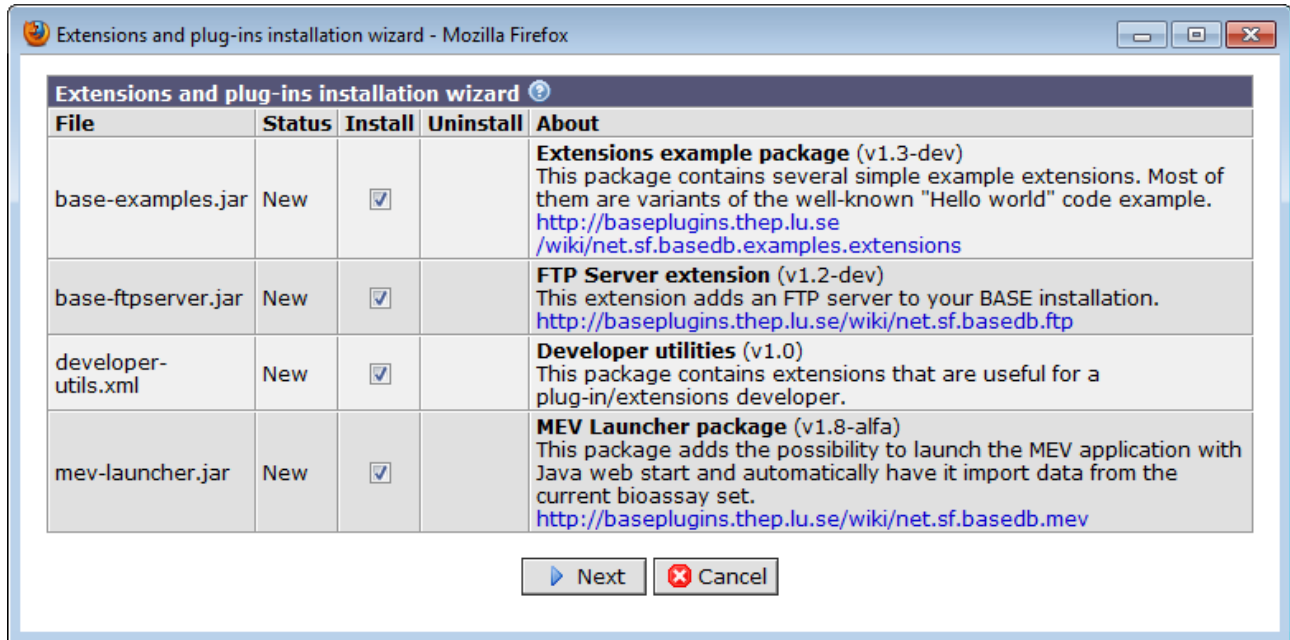
- summary 3 installed/upd
5 extension(s)

[More details...](#)

The development of BASE is currently supported by Lund University through SCIBLU. P and Alice Wallenberg Foundation and the Swedish Cancer Society. This server administ

The left-hand side of the screen shows a tree with all installed plug-ins and extensions, sorted by extension point and by file. Use the + and - icons to expand and collapse parts of the tree. Click on an item in the tree to display detailed information about it on the right-hand side of the screen. Click on the **Install/uninstall** button to start the installation wizard (which can also be used for uninstallation).

Figure 21.2. Extensions and plug-ins installation wizard

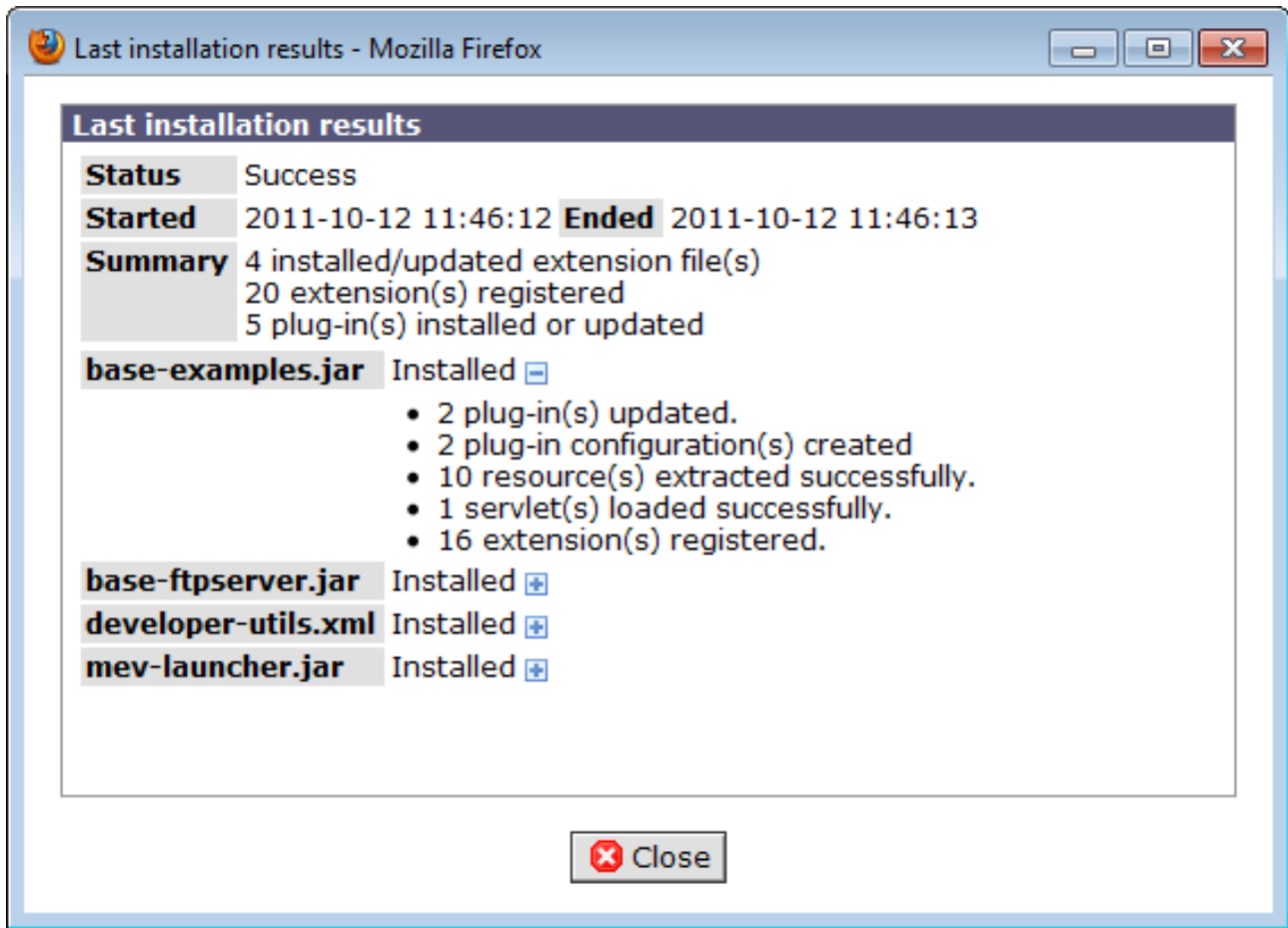


This wizard can be used for both installing, re-installing and un-installing plug-in and extension packages. The first dialog shows a list with all currently installed packages as well as any new packages that has been found on the server. The list includes the name of the JAR file, the current status and some information provided by the author of the package. There are also two columns **Install** and **Uninstall** which may or may not have a checkbox in them. For new packages there should be a checkbox in the install column that is already checked. Already installed packages can either be re-installed or uninstalled by checking the appropriate checkbox. If there is a problem with a package an error message is displayed and neither installation or uninstallation is possible.

Click on **Next** to perform the selected actions. The next dialog should display a summary with the installation results. Hopefully everything was successful. Close the dialog and refresh the overview tree to see the changes.

Note

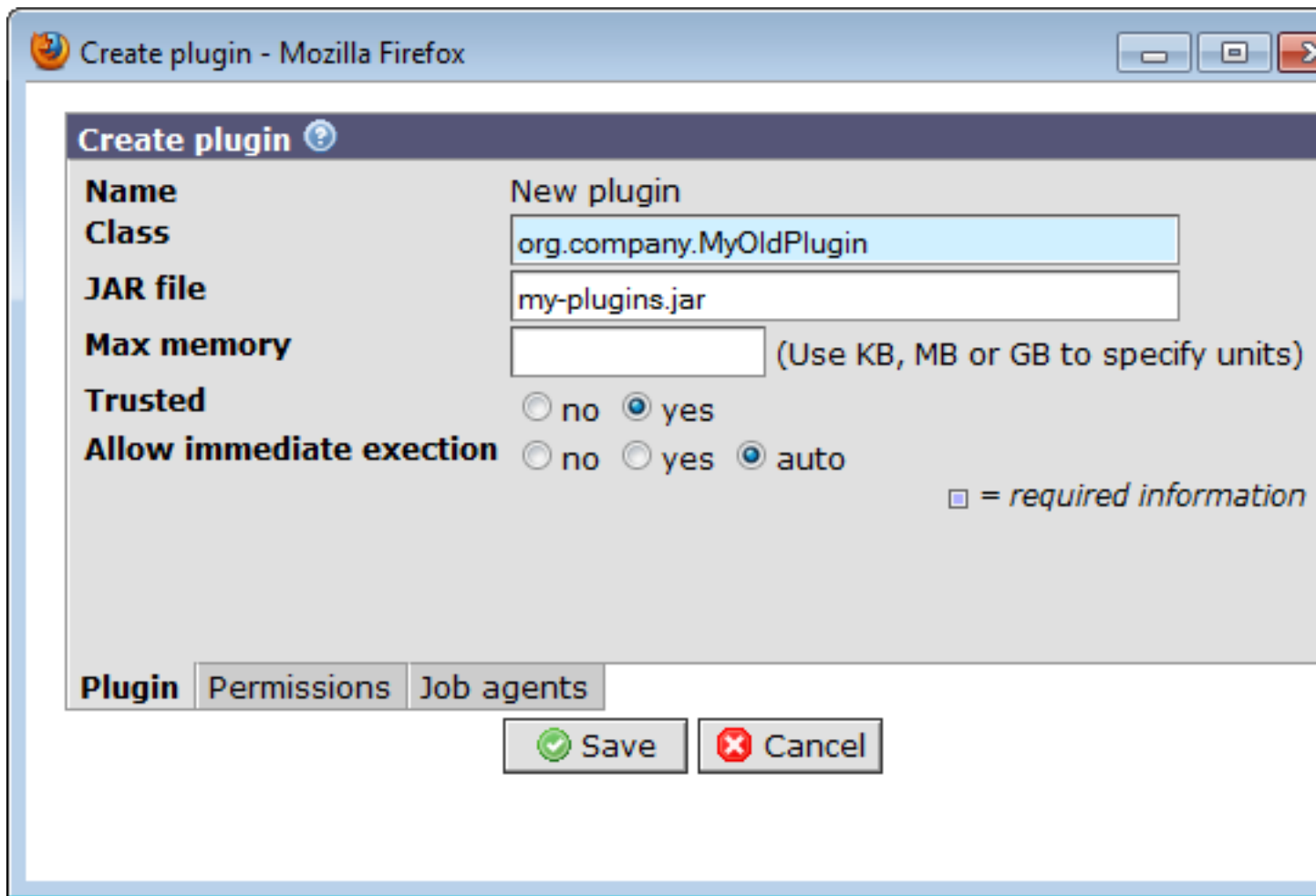
Uninstalling a package doesn't remove the plug-in definitions from BASE nor does it remove the JAR file from the server. This is because there may be jobs and other items referencing the plug-ins. The plug-ins are only marked as disabled and it is up to the administrator to actually delete them if it is possible.

Figure 21.3. Extensions and plug-ins installation results

21.1.2. Manual plug-in registration

Manual installation of a plug-in should almost never be necessary. One exception is that developers may want to do this as a first step before everything has been properly packaged. Another exception is plug-ins developed for BASE 2 that doesn't support the automatic installation procedure. If the old plug-in still works API-wise in BASE 3, manual installation can be used to install it. Repackaging such a plug-in is however not difficult so we recommend that the plug-in author is asked to provide an updated version.

To perform a manual installation the plug-in's JAR file must be located in the the directory specified by the `plugins.dir` setting in `base.config`. Subdirectories are not allowed. This is a change since BASE 2 where plug-ins could be installed almost anywhere. When the JAR file is in place, go to **Administrate** **Plug-ins & extensions** **Plug-in definitions** and click on the **New** button.

Figure 21.4. Manually installing a plug-in**Name**

The name of the plug-in. This name is set automatically by the plug-in and cannot be changed.

Class

The full Java class name of the plug-in.

JAR file

The name of the JAR file on the web server. The JAR file must be installed in the directory specified by the `plugins.dir` setting in `base.config`. If left empty the plug-in must be on the web server's class path (not recommended).

Max memory

The maximum amount of memory the plug-in may use. This setting only applies when the plug-in is executed with a job agent. If the internal job queue is used this setting has no effect and the plug-in may use as much memory as it likes. See Section 20.3.4, "Configuring the job agent" (page 173) for more information.

Trusted

If the plug-in is trusted enough to be executed in an unprotected environment. This setting has currently no effect since BASE cannot run in a protected environment. When this becomes implemented in the future a **no** value will apply security restrictions to plug-ins similar to those a web browser put on applets. For example, the plug-in is not allowed to access the file system, open ports, shut down the server, and a lot of other nasty things.

Allow immediate execution

If the plug-in is allowed to bypass the job queue and be executed immediately.

- **No:** The plug-in must always use the job queue.
- **Yes:** The plug-in is allowed to bypass the job queue. This also means that the plug-in always executes on the web server, job agents are not used. This setting is mainly useful for export plug-ins that needs to support immediate download of the exported data. See the section called “Immediate download of the exported data” (page 160).

Note

If a plug-in should be executed immediately or not is always decided by the plug-in. BASE will never give the users a choice.

- **Auto:** BASE will allow export plug-ins to execute immediately, and deny all other types of plug-ins. This alternative is only available when registering a new plug-in.

Click on **Save** to finish the registration or on **Cancel** to abort.

21.1.3. BASE version 1 plug-ins

BASE version 1 plug-ins are supported through the use of the *Base1PluginExecuter* plug-in. This is itself a plug-in and BASE version 1 plug-ins are added as configurations to this plug-in (cf. Section 21.2, “Plug-in configurations” (page 186)). To install a BASE version 1 plug-in follow these instructions:

1. Install the BASE version 1 plug-in executable and any other files needed by it on the BASE server. Check the documentation for the plug-in for information about what is needed.
2. Upload the *.base file for the BASE version 1 plug-in. If you cannot find the file, you can let your BASE version 1 server create one for you. In your BASE version 1 installation go to Analyze data Plug-ins and use the **Export** function. This will create a configuration file for your BASE version 1 plug-in that you can upload to your new BASE server.
3. Create a new plug-in configuration using, for example, the **New configuration** button in single-item view for the *Base1PluginExecuter* plug-in.
4. Start the configuration wizard and select parameters:
 - **File:** The *.base file describing the BASE version 1 plug-in. This can be left empty for manual configuration, but in reality it is only usable for tweaking an existing configuration that has been created from a file in the first place.
 - **Plugin executables path:** The path to the executable program that was installed in the first step.
 - **Source intensities:** Select if the plug-in can work with regular or logged data (or both).
 - **Resulting intensities:** Select if the plug-in generates regular or logged data.

Click **Next** to finish the wizard.

5. To check that the new plug-in works correctly, you need to have an experiment with some data. Go to the single-item view for a bioassay set and click on the **Run analysis** button. Select the *Base1PluginExecuter* plug-in. The list of configurations should include the newly installed plug-in. Select it and click on **Next**.
6. This will enter regular plug-in execution wizard and you will have to enter parameters needed by the plug-in.

21.1.4. Installing the X-JSP compiler

Some extensions may want to use custom JSP files that also uses classes that are stored in the extension's JAR file. The problem with this is that Tomcat usually doesn't know to look for classes

in the `plugins.dir` directory. To solve this problem BASE ships with a X-JSP compiler that can do this. This compiler has been mapped to files with a `.xjsp` extension, which are just regular JSP files with a different extension.

The X-JSP compiler must be installed into Tomcat's internal library folder (`$CATALINA_HOME/lib`) since this is the only place where Tomcat look for compilers. The installation is easy. Simply copy `<base-dir>/bin/jar/base-xjsp-compiler-3.x.jar` to `$CATALINA_HOME/lib` and restart Tomcat.

X-JSP is experimental

This is an experimental feature that depends on internal functionality in Tomcat. It may or may not work with future versions of Tomcat. The compiler will most likely not work with other servlet containers. It is known that the compiler shipped with BASE doesn't work with Tomcat 7. To get a working version, BASE must be compiled against the `jasper.jar` that ships with Tomcat 7. Eg. download the BASE source code and replace the `lib/servlet/jasper.jar` file with the one from Tomcat 7 and compile BASE. The resulting XJSP compiler should work with Tomcat 7 (but not with Tomcat 6).

21.1.5. Disable/enable plug-ins and extensions

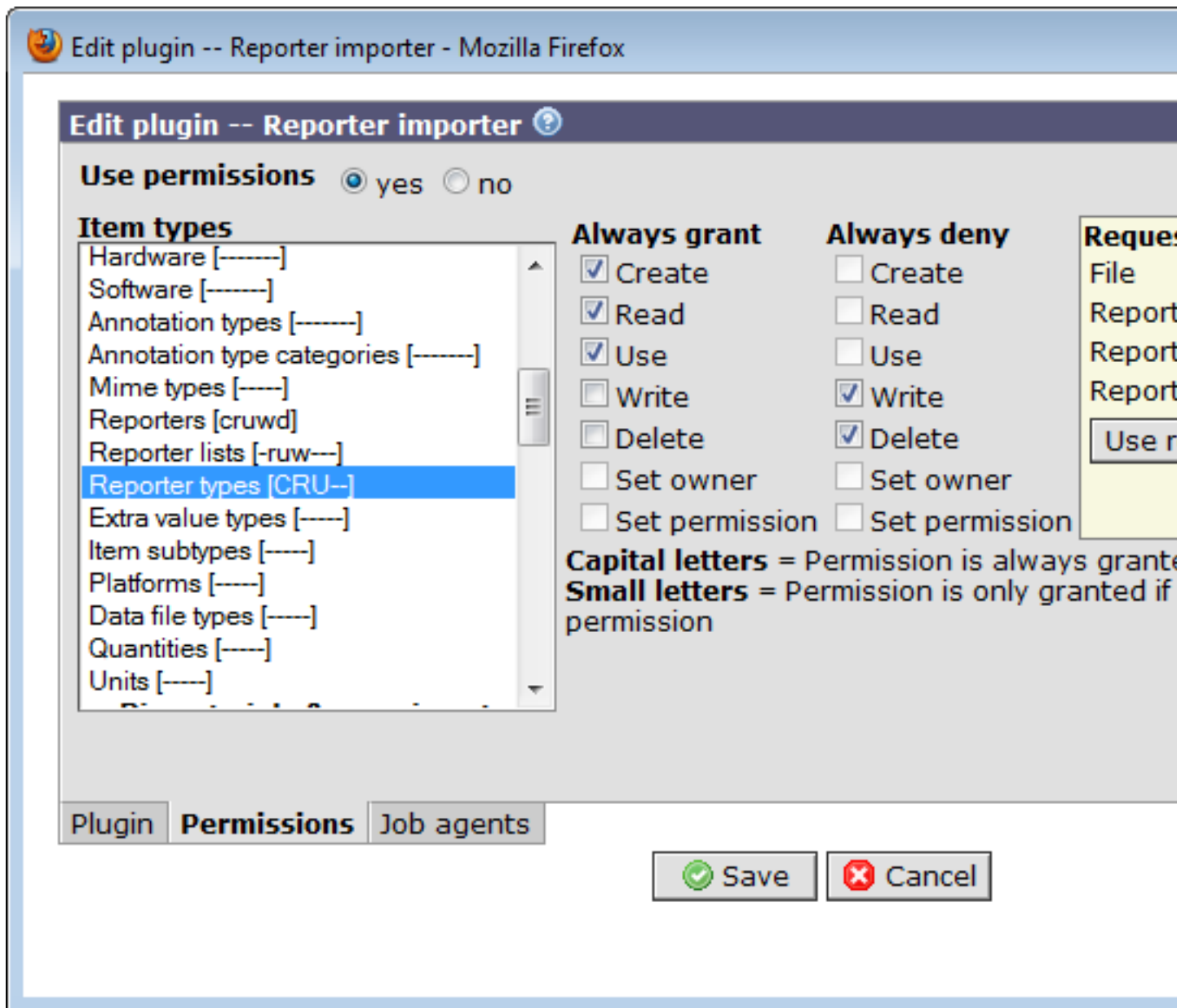
It is possible to disable specific extensions, extension point and or a plug-in without uninstalling the XML or JAR file. When you click on an item in the tree on the left-hand side of the screen a lot of detailed information about it will show up on the right-hand side. The right-hand side usually has a **Disable** or **Disable all** button in the toolbar. Click on that button to disable the plug-in, extension or all extensions for an extension point. The button will change to **Enable** or **Enable all** which lets you enable the extensions and plug-ins again.

21.1.6. Plug-in permissions

When a plug-in is executed the default is to give it the same permissions as the user that started it. This can be seen as a security risk if the plug-in is not trusted, or if someone manages to replace the plug-in code with their own code. A malicious plug-in can, for example, delete the entire database if invoked by the root user.

To limit this problem it is possible to tune the permissions for a plug-in so that it only has permission to do things that it is supposed to do. For example, a plug-in that import reporters may only need permission to update and create new reporters and nothing else.

To enable the permission system for a plug-in go the edit view of the plug-in and select the **Permissions** tab.

Figure 21.5. Setting permissions on a plug-in**Use permissions**

Select if the plug-in should use the permission system or not. If **no** is selected, the rest of the form is disabled.

Item types

The list contains all item types found in BASE that can have permissions set on them. The list is more or less the same as the permission list for roles. See the section called “Permissions” (page 206).

Always grant

The selected permissions will always be granted to the plug-in no matter if the logged in user had the permission to begin with or not. This makes it possible to develop a plug-in that allows users to do things that they are normally not allowed to do. The reporter importer is for example allowed to create and use reporter types.

Always deny

The selected permissions will always be denied to the plug-in no matter if the logged in user had the permission to begin with or not. The default is to always deny all permissions. Permissions that are not always denied and not always granted uses permissions from the logged in user.

Requested by plug-in

To make it easier for the server administrator to assign permissions, the plug-in developer can let the plug-in include a list of permissions that are needed. Plug-in developers are advised to only include the minimal set of permissions that are required for the plug-in to function. Click on the **Use requested permissions** button to give the plug-in the requested permissions.

21.2. Plug-in configurations

While some plug-ins work right out of the box, some may require configuration before they can be used. For example, most of the core import plug-ins need configurations in the form of regular expressions to be able to find headers and data in the data files and the Base1PluginExecuter uses configurations to store information about the BASE version 1 plug-ins.

Configurations are managed from a plug-in's single-item view page or from the Administrate Plug-ins & extensions Plug-in configurations page or from the single-item view page of each plug-in.

Click on the **New...** button to create a new configuration.

Figure 21.6. Create plug-in configuration

Plugin

The plug-in this configuration belongs to. This cannot be changed for existing configurations. Use the **Select...** button to open a pop-up window where you can select a plug-in.

Name

The name of the configuration.

Description

A description of the configuration (optional).

Note

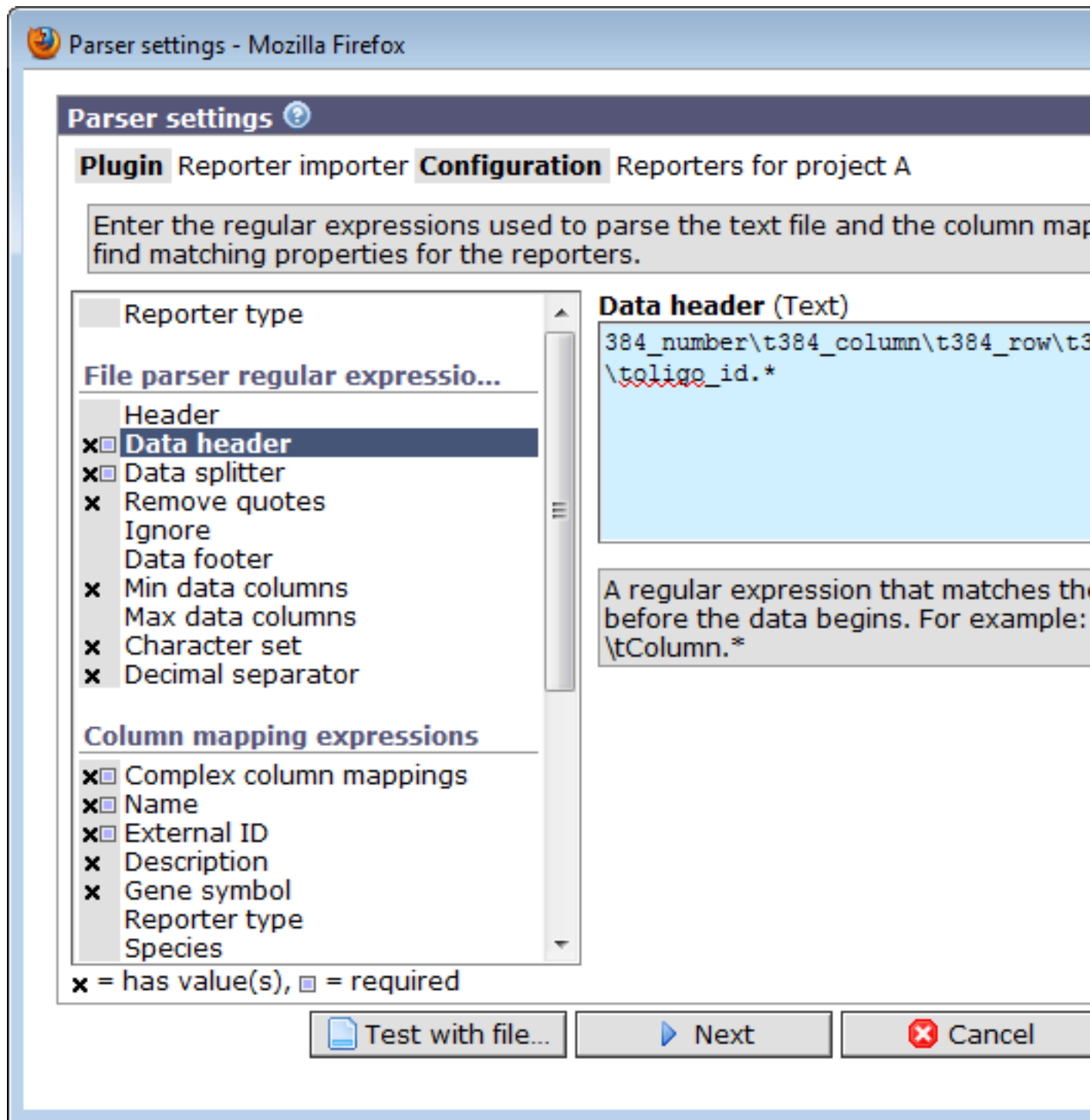
You cannot create configurations for plug-ins that does not support being configured.

Use the **Save** button to save the configuration or the **Save and configure** button to save and then start the configuration wizard.

21.2.1. Configuring plug-in configurations

Configuring a plug-in is done with a wizard-like interface. Since the configuration parameters may vary from plug-in to plug-in BASE uses a generic interface to enter parameter values. In short, it works like this:

1. BASE asks the plug-in for information about the parameters the plug-in needs. For example, if the value is a string or number or should be selected among a list of predefined values.
2. BASE uses this information to create a generic form for entering the values. The form consists of three parts:

Figure 21.7. The plug-in configuration wizard

- *The top part:* Displays the name of the selected plug-in and configuration.
- *The left part:* Displays a list of all parameters supported by the plug-in. Parameters with an **X** in front of their names already have a value. Parameters marked with a blue rectangle are required and must be given a value before it is possible to proceed.
- *The right part:* Click on a parameter in the list to display a form for entering values for that parameter. The form may be a simple free text field, a list of checkboxes or radiobuttons, or something else depending on the kind of values supported by that parameter.

3. When the user clicks **Next** the entered values are sent to the plug-in which validate the correctness. The plug-in may return three different replies:

- **ERROR:** There is an error in the input. BASE will redisplay the same form with any additional error information that the plug-in sends back.
- **DONE:** All parameter values are okay and no more values are needed. BASE will save the values to the database and finish the configuration wizard.
- **CONTINUE:** All parameter values are okay, but the plug-in wants more parameters. The procedure is repeated from the first step.

Do not go back

It is not possible to go backwards in the wizard. If you try it will most likely result in an unexpected error and the configuration must be restarted from the beginning.

21.2.2. Importing and exporting plug-in configurations

BASE ships with one importer and one exporter that allows you to import and export plug-in configurations. This makes it easy to copy configurations between servers.

Both the import and the export is started from the plug-in configuration list view: [Administrate Plug-ins & extensions](#) [Plug-in configurations](#)

The importer supports auto detection. Simply upload and select the XML file with the configurations. No more parameters are needed.

If you don't want to import all configurations that exist in the XML-file, there is an option that lets you select each configuration individually. When the option to import all configurations is set to **FALSE** in the first step of job-configuration, the following step after pressing **Next** will be to select those configurations that should be imported, otherwise this step is skipped.

To use the exporter you must first select the configurations that should be exported in the list. Then, enter a path and file name if you wish to leave the XML file on the BASE server or leave it empty to download it immediately.

Note

The import and export only supports simple values, such as strings, numbers, etc. It does not support configuration values that reference other items. If the plug-in has such values they must be fixed manually after the import.

21.2.3. The Test with file function

The **Test with file** function is a very useful function for specifying import file formats. It is supported by many of the import plug-ins that read data from a simple text file. This includes the raw data importer, the reporter importer, plate reporter, etc.

Note

The **Test with file** function can only be used with simple (tab- or comma-separated) text files. It does not work with XML files or binary files. The text file may have headers in the beginning.

As you can see in figure [Figure 21.7](#), “The plug-in configuration wizard”(page 188) there is a **Test with file** button. This will appear in the file format setup step for all plug-ins that support the test with file function. For detailed technical information about this see [Section 25.3](#), “Import plug-ins”(page 240) in [Chapter 25](#), *Plug-in developer*(page 222) Clicking on the **Test with file** button opens the following dialog:

Figure 21.8. The test with file function

Test with file

File to test

Lines to parse

Header regexp

Data splitter regexp

Ignore regexp

Data header regexp

Data footer regexp

Character set

Min data columns

Max data columns

Remove quotes ☒

■ = required information

File data **Column mappings**

Line	Columns	Type	Use as	File data
1	15	Data header		384_number 384_column 384_row 384_position oligo_id oligo_type
2	15	Data		1 01 A A01
3	15	Data		1 03 A A03 M200000001 I
4	15	Data		1 05 A A05 M200000002 I
5	15	Data		1 07 A A07 M200000004 I
6	15	Data		1 09 A A09 M200000005 I

The window consists of two parts, the upper part where the file to parse and the parameters used to parse it are entered, and the lower part that displays information about the parsing.

File to test

The path and file name of the file to use for testing. Use the **Browse** button to select a file from the BASE file system or upload a new file. Click on the **Parse the file** button to start parsing. The lower part will update itself with information about the parsed file. The file must follow a few simple rules:

- Data must be organised into columns, with one record per line.
- Each data column must be separated by some special character or character sequence not occurring in the data, for example a tab or a comma. Data in fixed-size columns cannot be parsed.
- Data may optionally be preceded by a data header, for example, the names of the columns.
- The data header may optionally be preceded by file headers. A file header is something that can be split into a name-value pair.
- The file may contain comments, which are ignored by the parser.

Lines to parse

The number of lines to parse. The default is 100 and rarely needs to be changed. One reason to increase the number is when the data header line is beyond the default value.

Character set

The character set used in the file. The default is ISO-8859-1 (same as Latin-1). This list contains all character sets supported by the underlying Java run-time and can be quite long.

Header regexp

A regular expression matching a header line. A header is a key-value pair with information about the data in the file. The regular expression must contain two capturing groups, the first should capture the name and the second the value of the header. For example, the file contains headers like:

```
"Type=GenePix Results 3"
"DateTime=2006/05/16 13:17:59"
```

To match this we can use the following regular expression: `"(.*)=(.*)"`.

Use the **Predefined** button to select from a list of common regular expressions.

Data splitter regexp

A regular expression used to split a data line into columns. For example, `\t` to split on tabs. Use **Predefined** button to select from a list of common regular expressions.

Ignore regexp

A regular expression that matches all lines that should be ignored. For example, `\#.*` to ignore all lines starting with a `#`. Use **Predefined** button to select from a list of common regular expressions.

Data header regexp

A regular expression that matches the line containing the data header. Usually the data header contains the column names separated with the same separator as the data. For example, the file contains a header like:

```
"Block"{tab}"Column"{tab}"Row"{tab}"Name"{tab}"ID" ...and so on
```

To match this we can use the following regular expression:
`"Block"\t"Column"\t"Row"\t"Name"\t"ID".*`

The easiest way to set this regular expression is to leave it empty to start with, click on the **Parse the file** button. Then, in the **File data** tab, use the drop-down lists in the **Use as** column to select the line containing the data header. BASE will automatically generate a regular expression matching the line.

Date footer regexp

A regular expression that matches the first line of non-data after all data lines. In most cases you can leave this empty.

Min and max data columns

If you specify values a data line is ignored if the number of columns does not fall within the range. If your data file does not have a data header with column names you can use these settings to find the start of data.

Remove quotes

If enabled, the parser will remove quotes around data entries.

File data

Press the **Parse the file** button to start parsing the file. This tab will be updated with the data from the file, organised as a table. For each line the following information is displayed:

- *Line*: The line number in the file
- *Columns*: The number of columns the line could be split into with the data splitter regular expression.

- *Type*: The type of line as detected by the parser. It should be one of the following: *Unknown*, *Header*, *Data header*, *Data* or *Data footer*.
- *Use as*: Use the drop-down lists to use a line as either the data header or data footer. BASE will automatically generate a regular expression.
- *File data*: The contents of the file after splitting and, optionally, removal of quotes.

Column mappings

After defining the data header you may need to press the **Parse the file** button to make this tab visible because this tab is only displayed when data has been found in the file and a data header was recognized. It allows you to easily select the mapping between columns in the file and the properties in the database.

Figure 21.9. Mapping columns from a file

Property	Mapping expression	File columns
Name	\oligo_id\	
External ID	=col('oligo_id')	
Description		
Gene symbol		
Reporter type		
Species		
Cluster ID		
Length		
Sequence		

- *Mapping style*: The type of mapping to use when you pick a column from the *File columns* list boxes.
- *Property*: The database property.
- *Mapping expression*: An expression that maps the data in the file columns to the property in the database. There are two types of mappings, simple and expressions. A simple mapping is a string template with placeholders for data from the file. An expression mapping starts with an equal sign and is evaluated dynamically for each line of data. The simple mapping has better performance and we recommend that you use it unless you have to recalculate any of the numerical values. In both cases, if no column matching the placeholder exactly is found the placeholder is interpreted as a regular expression that is matched against each column. The first one found is used. A few mapping examples are listed in Table 21.1, “Mapping expression examples” (page 192).

Table 21.1. Mapping expression examples

Expression	Explanation
\Name\	Exact match is required.
\1\	Column with index 1 (the second column).
[\row\, \column\]	Combining row and column to a single coordinate.
=2 * col('radius')	Calculate the diameter dynamically.
\F63(3 5) Median\	Use regular expression to match either F633 or F635.
constant_string	Use constant_string as value for this column for each line.

Note

Column numbers are 0-based. We recommend that you use column names at all times if they are present in the file.

- *Auto generate*: Click on this button to let BASE try to automatically generate mappings based on fuzzy string matching between the property names and file column headers. Each match get a score between 0 and 1 where 1 indicates a better match. Use the *similarity score* to limit the automatically generated mappings to matches with at least the given score. A value between 0.7 and 0.9 is usually a good choice.
- *File columns*: Lists of column found in the file. Select a value from this list to let BASE automatically generate a mapping that picks the selected column.

Chapter 22. Account administration

Read Chapter 6, *Projects and the permission system* (page 40)

This chapter contains important information about the permission system BASE uses. It is essential that an administrator knows how this works to be able to set up user, groups and roles smoothly.

22.1. Users administration

The user list is accessed with `Administrate Users` and from here are the users' account and contact information managed.

22.1.1. Edit user

The pop-up window where information and settings for a user can be edited has three tabs, one for the account related, one with information about the user and one that shows the user's memberships.

Properties

Figure 22.1. User properties

The screenshot shows a Mozilla Firefox window titled "Edit user -- User - Mozilla Firefox". Inside is a dialog box titled "Edit user -- User". The dialog contains the following fields and controls:

- Name:** Text field with "User" entered.
- Login:** Text field with "user" entered.
- External ID:** Empty text field.
- New password:** Empty text field.
- Retype password:** Empty text field.
- Quota:** Dropdown menu showing "1 GB total (1.0 GB total)".
- Quota group:** Dropdown menu showing "Group A (1.0 GB total)".
- Home directory:** Dropdown menu showing "user".
- Expiration date:** Empty text field with a "Calendar..." button.
- Multi-user account:** Unchecked checkbox.
- Disabled:** Unchecked checkbox.

At the bottom right, there is a legend: ☐ = required information.

At the bottom, there are four tabs: "User", "Contact information", "Additional info", and "Membership". Below the tabs are two buttons: "Save" (with a green checkmark icon) and "Cancel" (with a red X icon).

These are the properties for a user account.

Name

The full name of the user that is associated with the account.

Login

A login name to use when logging in to the account. The login must be unique among all users.

External ID

An id that is used to identify the user outside BASE (optional). If a value is given it must be unique among all users.

New password

This is used together with the login name to log in to the account. This is a required field for a new user or if the password should be changed. If the field is left empty the password will be unchanged

Retype password

Retype the password that is written in **New password**.

Quota

Set disk quota for the account.

Quota group

Set this if the account should belong to a group with specified quota (optional). With this set the user's possibilities to save items to disk will also depend on how much the rest of the group has saved.

Home directory

Set the account's home directory (optional). A new directory, either empty or from a template, can be created if editing a new user. Select - **none** - if there should not be any home directory associated with the account.

Expiration date

Define a date in this field if the account should expire on a certain day (optional). The account will be disabled after this date. Leave this empty if the account never should expire.

Tip

Use the **Calendar...** button to pick a date from a calendar in a pop-up window.

Multi-user account

This checkbox should be checked if the account should be used by more one user. This will prevent the users from changing the password, contact information and other settings. It will also reset all list filters, column configurations, etc. when the user logs out. Normally, these settings are remembered between log ins.

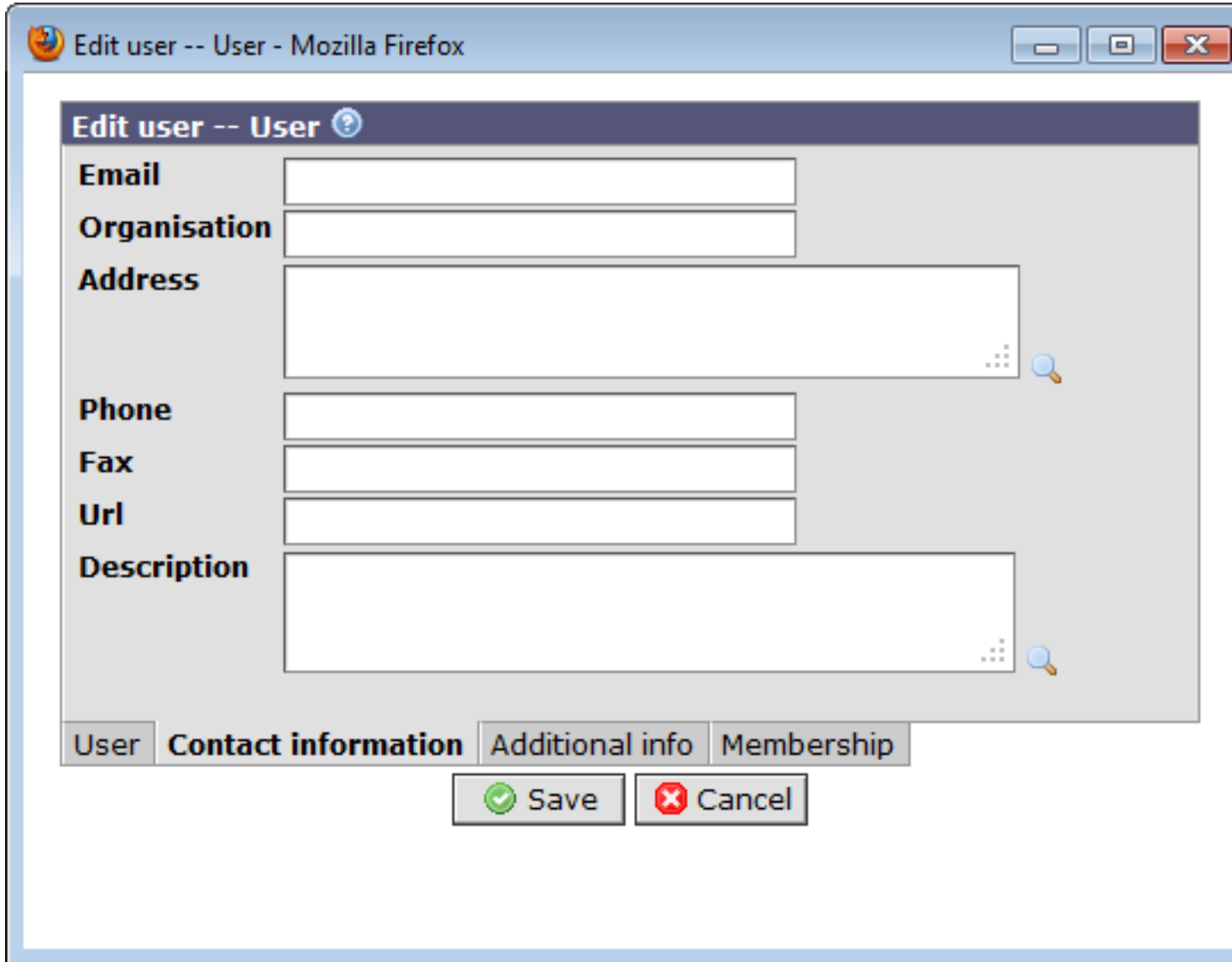
Disabled

Disable the account.

Go to the other tabs if there are any changes to do otherwise press **Save** to save the values or **Cancel** to abort.

Contact information

Figure 22.2. Contact information



Information about how to get in contact with the user that is associated with the account. All fields on this tab are optional and do not necessarily need to have a value but some can be good to set, like email or phone number.

Email

User's email address. There is some verification of the value but there is no check if the email really exists.

Organization

The company or organization that the user works for.

Address

User's mail address. Use the magnifying glass down to the right, to edit this property in a larger window.

Phone

User's phone number(s)

Note

There is no special field for mobile phone, but it works fine to put more than one number in this field.

Fax

User's fax number.

Url

A URL that is associated with the user.

Description

Other useful contact information or description about the user can be written in this field. Use the magnifying glass to edit the information in a pop-up window with a larger text-area.

Go to the other tabs if there are any changes to do otherwise press **Save** to save the values or **Cancel** to abort.

Additional information

Figure 22.3. Additional user information

The screenshot shows a web browser window titled 'Edit user -- User - Mozilla Firefox'. Inside the browser, there is a web application window titled 'Edit user -- User' with a help icon. The application has four tabs: 'User', 'Contact information', 'Additional info' (which is selected), and 'Membership'. In the 'Additional info' tab, there are two text input fields labeled 'Mobile' and 'Skype'. At the bottom of the application window, there are two buttons: 'Save' (with a green checkmark icon) and 'Cancel' (with a red X icon).

This tab contains fields that hold various information about the user. There are by default two fields in BASE but this could easily be changed by the server administrator. How this configuration is done can be read in Appendix C, *extended-properties.xml* reference (page 426).

Note

The **Additional info** tab is only visible if there is one or more property defined for `UserData` in the configuration file for extended properties.

These are the fields that are installed with BASE

Mobile

The user's mobile number could be put in this field. This field could be left empty.

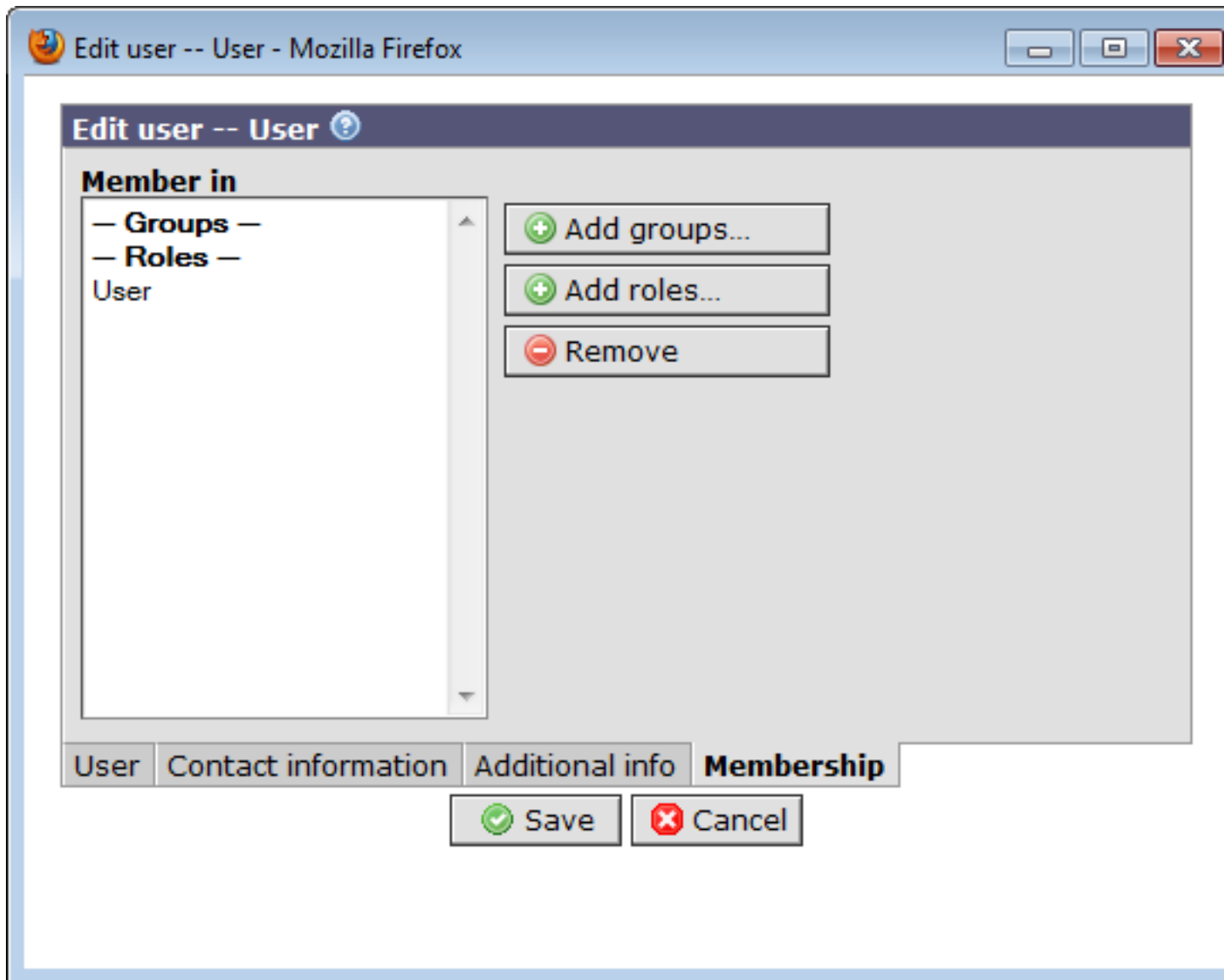
Skype

Skype contact information, if the user has a registered Skype account. This field could be left empty.

Go to the other tabs if there are any changes to do otherwise press **Save** to save the values or **Cancel** to abort.

Group and role membership

Figure 22.4. Group and role membership



On this tab, the group and role membership of a user can be specified. The membership can also be changed by editing the group and/or role.

Note

When adding a new user, the user is automatically added as a member to all groups and roles that has been marked as *default*. In the standard BASE distribution the *User* role is marked as a default role.

Member in

Lists the groups and roles the user already is a member of.

Add groups...

Opens a pop-up window that allows you to select groups. In the pop-up window, mark one or more groups and click on the **Ok** button. The pop-up window will not list groups that the user already is a member of.

Add roles...

Opens a pop-up window that allows you to select roles. In the pop-up window, mark one or more roles and click on the **Ok** button. The pop-up window will not list roles that the user already is a member of.

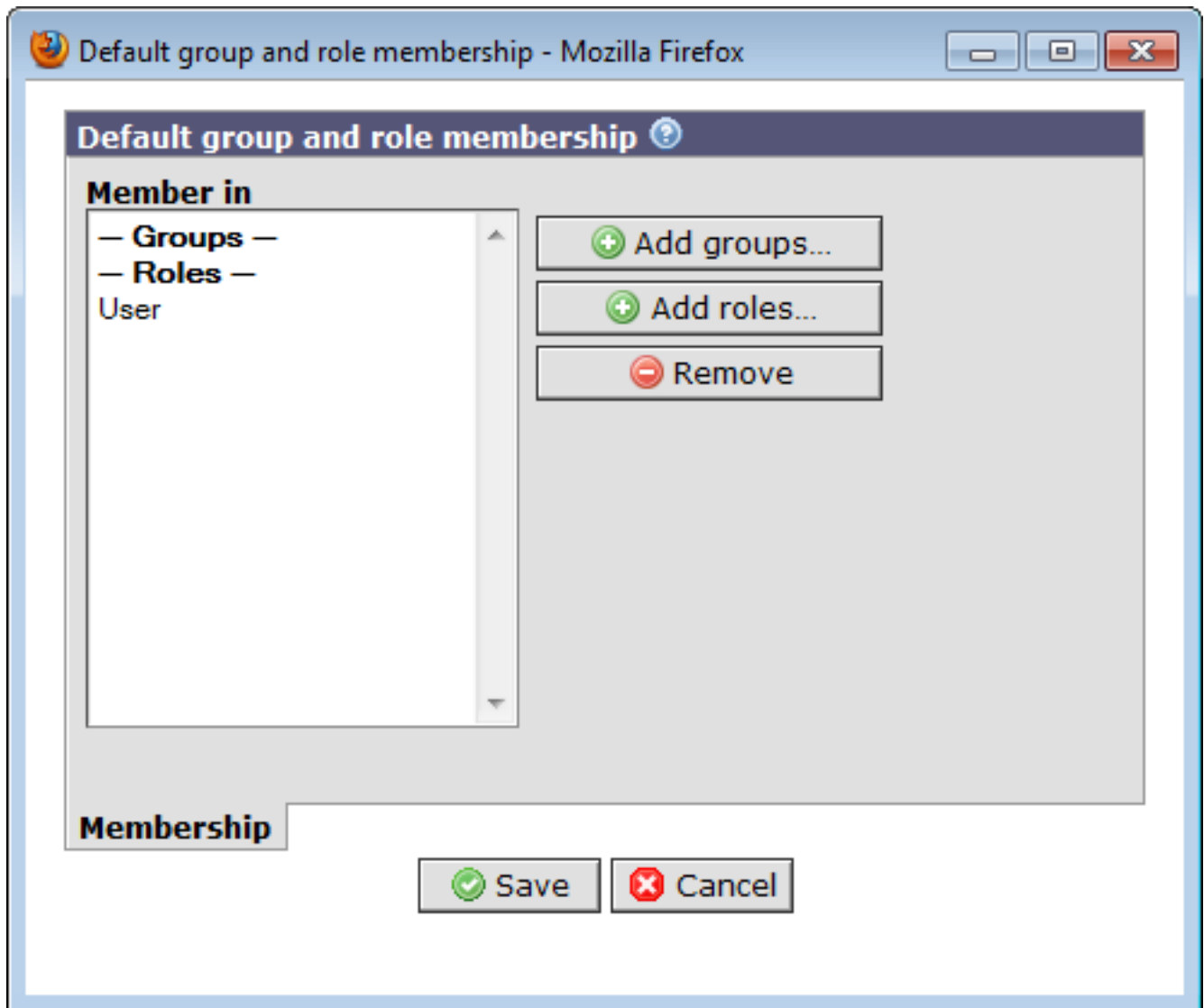
Remove

Use this button to remove the user from the selected groups and/or roles. The selected items will then disappear from the list of memberships.

Go to the other tabs if there are any changes to do otherwise press **Save** to save the values or **Cancel** to abort.

22.1.2. Default group and role membership

Figure 22.5. Default group and role membership



It is possible to automatically let BASE add new users as a member of a pre-defined list of groups and/or roles. This is done by marking those groups and roles as *default* groups and roles. There are two ways to do this.

1. Change the flag in the edit-dialog for each of the groups/roles that you want to assign as default.
2. Use the **Default membership** button on the Administrate Users page and select groups and roles in a pop-up dialog. The dialog lists all groups and roles that are currently assigned as default. Use the **Add groups** and **Add roles** buttons to select more groups and roles. Use the **Remove** button to remove the selected groups/roles.

Note

Changing which groups and roles that are the default does not affect existing user accounts. They are only used to assign membership to new users.

22.2. Groups administration

Groups in BASE are meant to represent the organizational structure of a company or institution. For example, there can be one group for each department and subgroups for the teams in the

departments. The group-membership is normally set when the user is added to BASE and should not have to be changed later, except when the company is re-organizing.

There is one pre-installed group in BASE, a system group, called Everyone. It is, like the name says, a group in which everyone (all users) are members. The users that are allowed to share to everyone can easily share items to all users by sharing the item to this group.

22.2.1. Edit group

The pop-up window where a group can be edited has two tabs, **Group** and **Members**.

Properties

Figure 22.6. Group properties

Edit group -- Group A ?

Name Group A

Default ☒ no ☐ yes

Hidden members ☒ no ☐ yes

Quota 1 GB total (1.0 GB total) ▼

Description

☐ = required information

Group **Members**

☒ Save ☐ Cancel

Name

The name of the group.

Default

Mark this checkbox to let BASE automatically add new users as members to this group.

Hidden members

Mark this checkbox to create a group with hidden members. This means that a user will not be able to see information about other members in the group, but it is still possible to share items to the group as a whole.

Description

Description about the group. The magnifying glass, down to the right, can be used to open and edit the text in a larger text area.

Quota

With this property it's possible to limit the quota of total disk space for the group members. Select **-none-** from the drop-down list if the group should not have any quota. There are some presets of quotas that comes with the BASE installation, besides a couple with different size of total disk space there are one called **No quota** and one with **Unlimited quota**. Their names speak for them self.

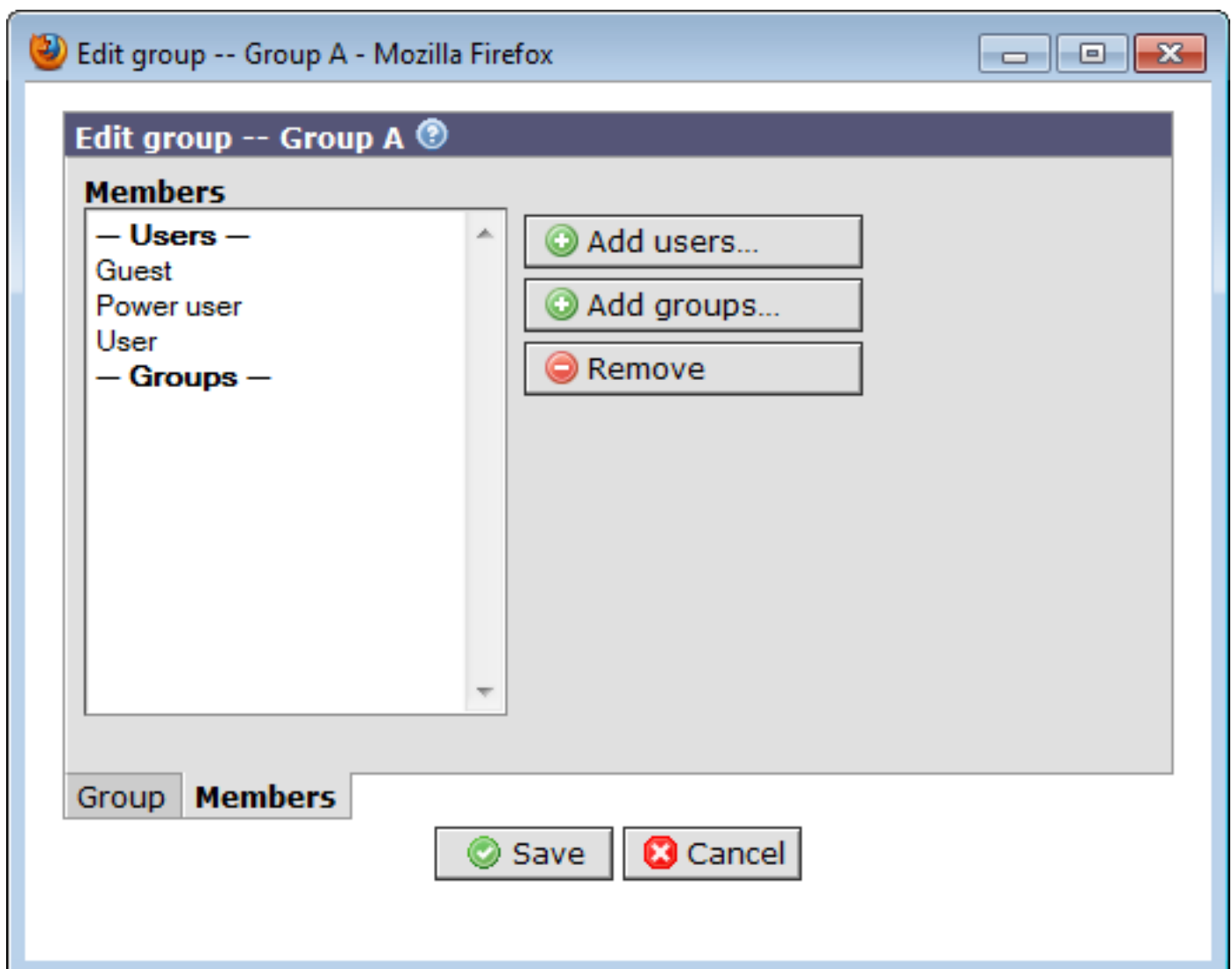
Note

A user can only take quota from one group, which has to be specified as the **Quota group** of the user.

Go to the other tab, **Members**, if there are any changes to do otherwise use **Save** to save the settings or **Cancel** to abort.

Group members

Figure 22.7. Group members



A group can have both single users and other groups as members. Group members have access to those items that are shared to the group. Each user in the group has the possibility to share their own items to one or more of the other members or to the whole group.

Members

Lists the user and groups that are already members of this group.

Add users...

Opens a pop-up window that allows you to add users to the group. In the pop-up window, mark one or more users and click on the **Ok** button. The pop-up window will not list users that are already members of the group.

Add groups...

Opens a pop-up window that allows you to add other groups to the group. In the pop-up window, mark one or more groups and click on the **Ok** button. The pop-up window will not list groups that are already members of the group.

Remove

Use this button to remove the selected users and/or groups from this group. The selected items will disappear from the list of memberships.

Go to the other tab if there are any changes to do, otherwise use **Save** to save the values or **Cancel** to abort.

22.3. Roles administration

Roles are meant to represent different kinds of working positions that users can have, like server administrator or regular user just to mention two. Users are normally assigned a role, perhaps more than one, when they are created and registered in BASE.

22.3.1. Pre-defined system roles

BASE comes with some pre-defined roles. These are configured to cover the normal user roles that can appear. A more detailed description of the different roles and when to use them follows here.

Administrator

This role gives the user full permission to do everything in BASE and also possibility to share items with the system-group 'Everyone'. Users that are supposed to administrate the server, user accounts, groups etc. should have this role.

Supervisor

Users that are members of this role has permission to read everything in BASE. This role does not let the members to actually do anything in BASE except read and supervise.

Power user

This role allows it's members to do some things that an ordinary user not is allowed to. Most things are related to global resources like reporters, the array lims and plug-ins. This role can be proper for those users that are in some kind of leading position over work groups or projects.

User

A role that is suitable for all ordinary users. This allows the members to do common things in BASE such as creating biomaterials and experiments, uploading raw data and analyse it.

Guest

This is a role with limited access to create new things. It is useful for those who wants to have peek at the program. It can also be used for someone that is helping out with the analysis of an experiment.

Job agent

This role is given to the job agents and allows them to read and execute jobs. Job agents always runs the jobs as the user who created the job and therefore it have to be able to act as another user.

22.3.2. Edit role

Creating a new role or editing the system-roles are something that do not needs to be done very often. The existing roles will normally be enough but there can be some cases when they need to be complemented, either with a new role or with different permissions.

Properties

Figure 22.8. Role properties

Name

The name of the role.

Share to Everyone

Allows the user to share items to the system-group 'Everyone'.

Act as another user

Allows the user to login as another user without knowing the password. This is used by job agents to make it possible for them to execute a plug-in as the user that created the job. This

permission will also make it possible to switch user in the web interface. It can be useful for an administrator who needs to check out a problem, but use this permission with care.

Select job agent for jobs

Allows the user to select a specific job agent when running jobs. Users without this permission will always have a randomly selected job agent.

Default

Mark this checkbox to let BASE automatically add new users as members to the role.

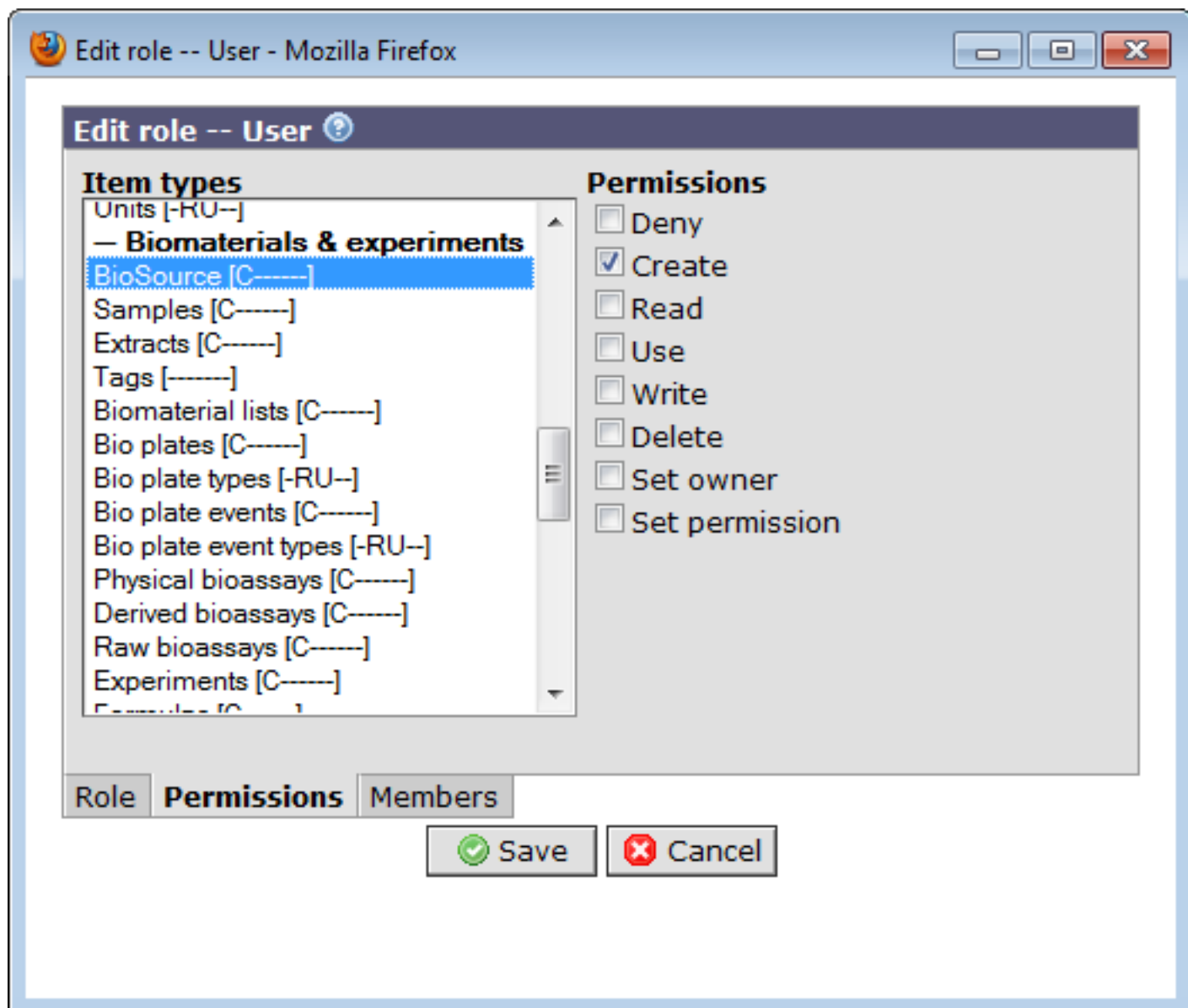
Description

Description and information about the role.

Set the properties and proceed then to either one of the other tabs or by clicking on one of the buttons: **Save** to save the changes or **Cancel** to abort.

Permissions

Figure 22.9. Role permissions



A role's permissions are defined for each item type within BASE. Set the role's permission on an item type by first selecting the item(s) in the list and then tick those permissions that should be applied. Not all permissions can be applied to every item type, that's why permission check-boxes becomes disabled when selecting some of the item types

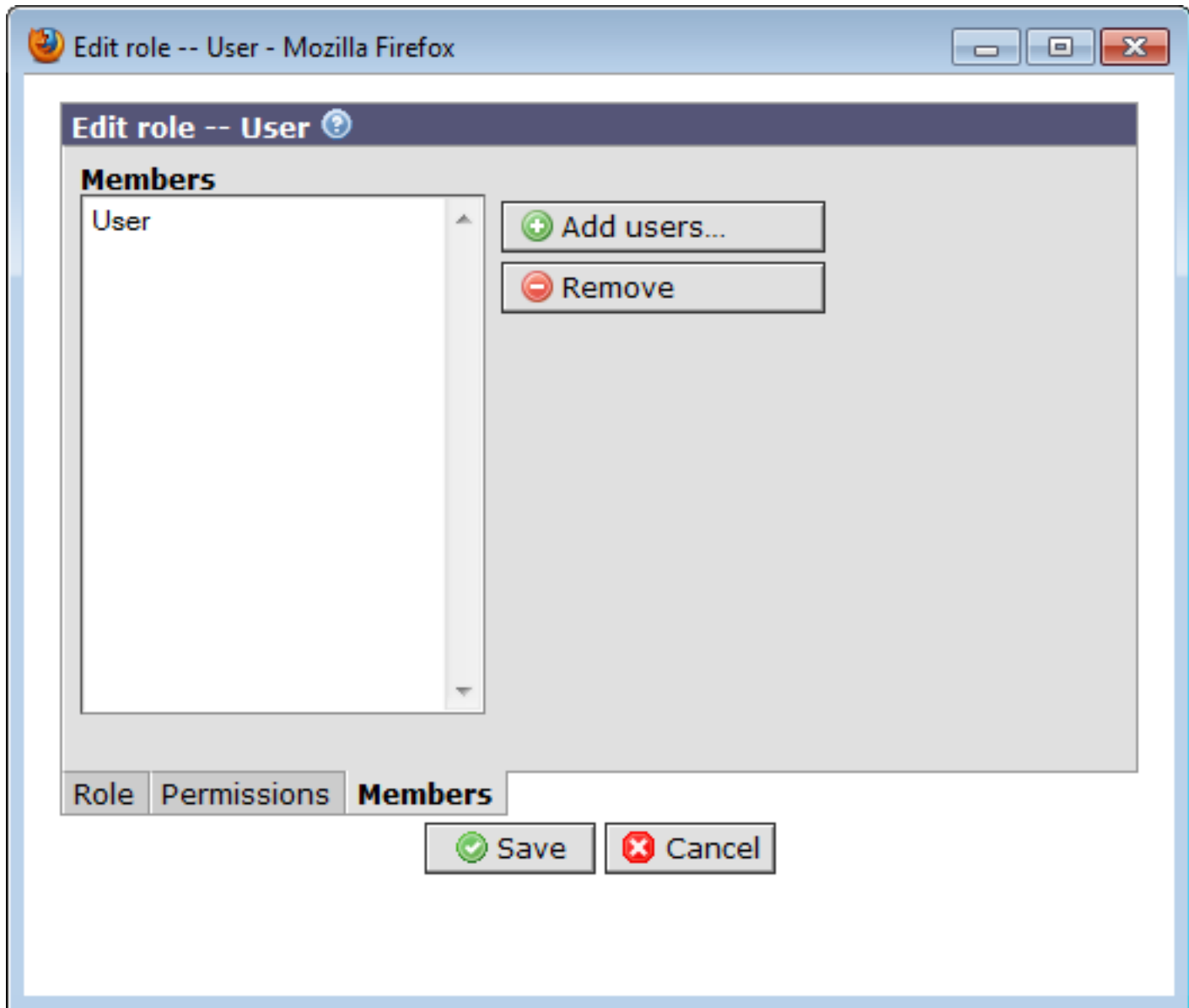
After each item type in the list is a string inside square brackets that shows what kind of permissions the current role has on that particular item type. The permissions that do not have been set are represented with '-' inside the square brackets and those which have been set are represented with characters that are listed below.

- **DENIED** = Deny access to the selected item type. This exclude all the other permissions by unchecking the other check boxes.
- **C** = Create
- **R** = Read
- **U** = Use
- **W** = Write
- **D** = Delete
- **O** = Set owner
- **P** = Set permission

Set the role's permission on each one of the item types and proceed then to one of the other tabs or click on **Save** to save the changes or **Cancel** to abort.

Members

Figure 22.10. Role members



Members

Users that are members of a role are listed in the list-box located on this tab.

Add users

Select the users that should be added from the list in the pop-up window. Click on the **Ok** button to close the pop-up window and add the selected users.

Remove

Removes the selected users from the role.

Press **Save** to save the role or go to one of the other tabs if there are more that needs to be set. Use **Close** to abort and close the window without saving the changes.

22.4. Disk space/quota

The administrator can control the maximum size of disk space for users and groups. A user must be assigned a quota of their own and may optionally have a group quota as well. If so, the most

restrictive quota is checked whenever the user tries to do something that counts as disk-consuming, for example uploading a file.

Note

The quota is checked before an operation, which is allowed to continue if there is space left. For example, even if you have only one byte left of disk space you are allowed to upload a 10MB file.

Read Section 22.1.1, “Edit user” (page 194) and Section 22.2.1, “Edit group” (page 202) for information about how to set a quota for a user and group.

The list of quotas in BASE can be found by using the menu **Administrate Quota**.

22.4.1. Edit quota

The edit window has two tabs, one with information about the quota and one where the limits are defined.

Properties

Figure 22.11. Quota properties

The screenshot shows a web browser window titled "Edit quota -- 1 GB total - Mozilla Firefox". The main content area has a header "Edit quota -- 1 GB total ?". Below the header, there are two form fields: "Name" with the value "1 GB total" and "Description" with the value "1 GB total quota.". A legend at the bottom right indicates that a blue square icon means "required information". At the bottom of the window, there are two tabs: "Quota" (selected) and "Quota values". Below the tabs are "Save" and "Cancel" buttons.

Name

Name of the quota.

Description

Description of the quota. It could be a good idea to describe the quota's details here. Use the magnifying glass to edit the text in a larger text area.

Go to the other tab if there are values that have not been set. Otherwise use **Save** to save the settings or **Cancel** to abort.

Values

Figure 22.12. Quota values

Edit quota -- 1 GB total ?

	Primary location	Secondary location
Total quota	1.0 GB <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>
Files	<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>
Raw data	<input type="text"/> <input type="checkbox"/>	N/A
Experiment	<input type="text"/> <input type="checkbox"/>	N/A

☐ = *required information*

The individual quota values doesn't have to sum up to the total value. The most restrictive value is always used. You can use the following units (case-insensitive):

- **Gb** or **G** for gigabyte values (1.5 Gb)
- **Mb** or **M** for megabyte values (10M)
- **Kb** or **K** for kilobyte values (800k)
- **bytes** or **b** for byte values (50 bytes)

Quota **Quota values**

The quota values are defined here, both for the primary location and the secondary location. Use the check box to the right of the input fields to set unlimited quota. You can use the abbreviations kb, Mb and Gb to specify the quota values.

Total

Limit of total quota. The sum of the other three quotas does not have to be the same as this, it is always the most restricted value that is used.

Files

Limit of disk space to save files in.

Raw data

Limit of disk space to save raw data in.

Experiments

Limit of disk space that can be used by experiments.

When everything have been set the quota is saved by using **Save**. To discard changes use **Cancel**.

22.4.2. Disk usage

Go to [Administrate Disk usage](#) if you want to get statistics about how the disk is used. There are three tabs:

Overview

Gives an overview of the total disk usage. It is divided per location and quota type.

Per user

Gives an overview of the disk usage per user. For each user you can get a summary displaying the total disk usage and divided per location and quota type. Use the **View details** link to list all items that uses up disk space. The list displays the name and type of each item and the amount of disk space it uses.

Per group

Gives an overview of the disk usage per group, with the same functionality as the per user overview.

Part IV. Developer documentation

Chapter 23. Migrating code from BASE 2 to BASE 3

This section gives a brief overview what has changed between BASE 2 and BASE 3 and how to migrate plug-in or extension code to BASE 3. Do not expect that your code can be installed and function as intended right out of the box.

23.1. Compiling the code against BASE 3

All deprecated methods and classes have been removed

Before trying to compile your code against the BASE 3 API we recommend that you make sure that you are not using any deprecated methods or classes from the BASE 2.17 API. So the first step should always be to compile your code against the latest BASE 2.17 release. Fix any warnings according to the instructions in the javadoc. In most cases, there is a simple replacement API that can be used instead. Since the deprecated API has been removed in BASE 3 so has the instructions. You'll need to check with the BASE 2.17 documentation which is located at:

- <http://base.thep.lu.se/chrome/site/2.17/html/index.html>: User manual.
- <http://base.thep.lu.se/chrome/site/2.17/api/index.html>: API documentation.

Do not proceed until you are sure that your code is not using any deprecated API.

BASE JAR files have new names

All JAR files with the BASE API has been renamed to better follow the scheme used by many other projects.

Table 23.1. JAR filename changes

Old filename	BASE 3 filename
BASE2Core.jar	base-core-3.0.0.jar
BASE2CorePlugins.jar	base-coreplugins-3.0.0.jar
BASE2Webclient.jar	base-webclient-3.0.0.jar
BASE2WSCClient.jar	base-webservices-client-3.0.0.jar

Note

Also note that the version number is included in the file name, so the files will change when new versions are released. You'll need to make sure that your build system can handle this. The BASE JAR files can be downloaded from <http://base2.thep.lu.se/base/jars/>.

23.2. Core API changes

There are lot's of other changes to the API between BASE 2 and BASE 3. If your code is affected by those changes, you will have to update your code. Since there are many changes, both big and small, it is not possible to list everything here. One good way to find out more about the changes is to use the BASE Trac or ask on the developers mailing list. Some of the major changes are:

- Removed the `Plugin.getAbout()` method from the `Plugin` interface. The information should instead be placed in the `META-INF/extensions.xml` file inside the JAR file that the plug-in is shipped in.

- `HardwareType`, `SoftwareType`, `ProtocolType` and `FileType` has been replaced with `ItemSubtype`. Additionally, several other items implement the new `Subtypable` interface. See ticket #1597 (Subtypes of items)¹ for more information.
- `Label` has been replaced with `Tag`. `LabeledExtract` has been merged with `Extract`. `Hybridization` has been replaced with `PhysicalBioAssay`. `Scan` and `Image` has been replaced with `DerivedBioAssay`. The API for linking parent/child biomaterial has been changed. The pooled property of biomaterials has been removed. The changes are driven by the support for sequencing experiments. The new items are using the new subtype feature. For example, a hybridization in BASE 2 has been converted to a physical bioassay in BASE 3 with the subtype 'Hybridization'. The server admin can define additional subtypes. See ticket #1153 (Handling short read transcript sequence data)² for more information.
- `RawBioAssay` has a direct link to `Extract` instead of the `arrayIndex` property which has been removed.
- Removed validator and metadata reader properties from `DataFileType`. This feature is now implemented as extensions. The API is slightly different and classes have been moved around a bit. See Section 26.8.8, “Fileset validators” (page 281) and ticket #1598 (Use the extensions system for data file validators and metadata readers)³ for more information.
- A `FileSet` may store more than one file for each `DataFileType` which was not possible before. The API for adding files have changed. See ticket #1604 (Support for multiple files of the same type in a `FileSet`)⁴ for more information.
- Changes to the `FileUnpacker` interface. The `unpack()` method signature has changed and any plug-ins that implement this interface need to be updated. The changes make it possible to get information about the main zip/tar file that is being unpacked. See ticket #978 (Unzipped files never inherit file type specified during upload)⁵ for more information.
- Encrypting passwords before logging in is no longer supported. The `SessionControl.login()` has been changed to reflect this. While this may seem like a reduction in security it is not. The previously used scheme with MD5 hashes can be cracked by brute-force on a moderate computer today. If additional security is needed we recommend that BASE is installed with HTTPS access only. See ticket #1641 (Use `bcrypt` for storing passwords instead of MD5)⁶ for more information.

23.3. Packaging your plug-in so that it installs in BASE 3

The installation system for plug-ins and extensions has been reworked. We hope that it is easier to install things now. Basically, the plug-in installation wizard has been merged with the extensions installation wizard. If your package contain only extensions it will probably install without changes. Packages with plug-ins need some changes to the XML files.

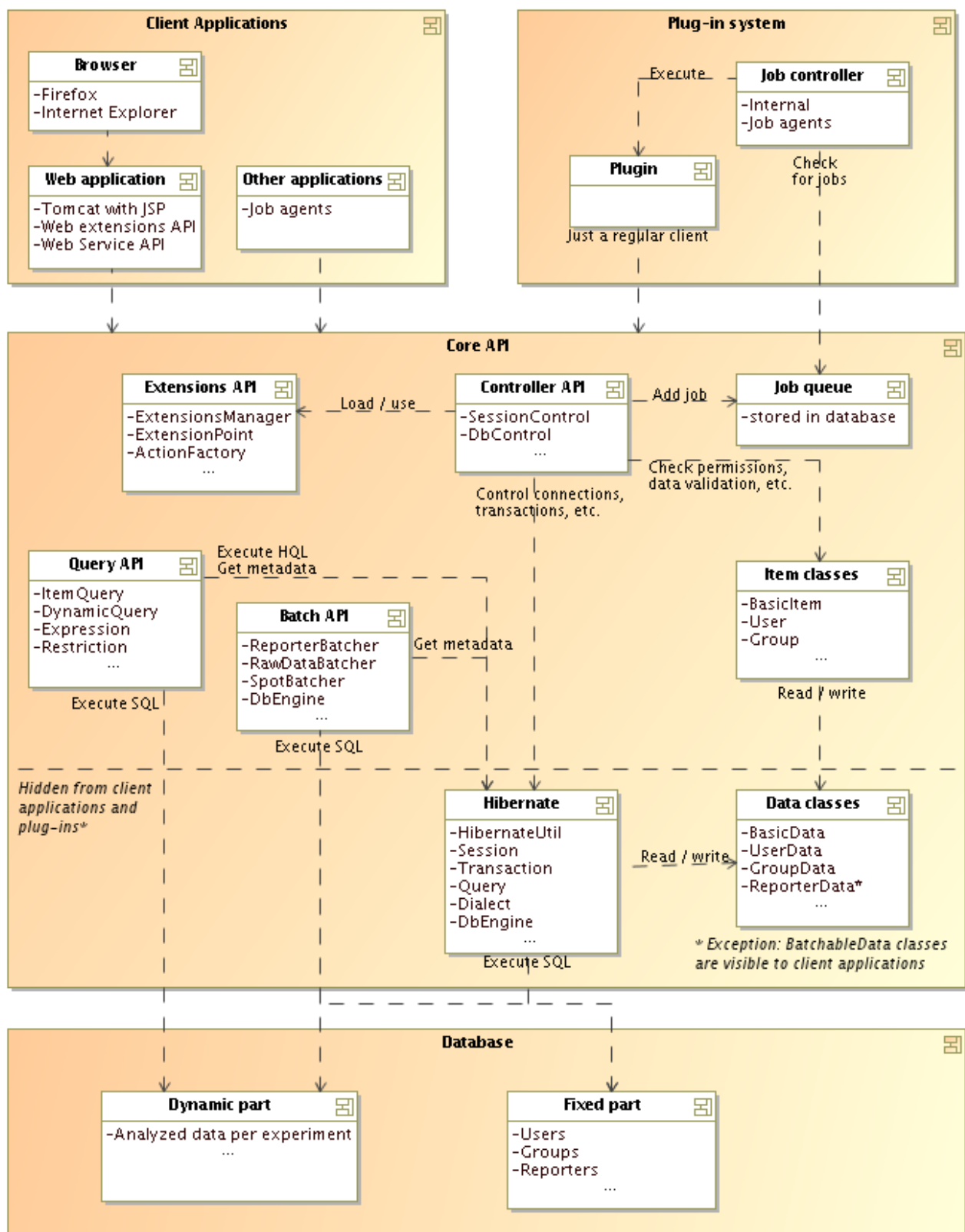
- The information that was in `META-INF/base-plugins.xml`, should be moved to `META-INF/extensions.xml`. The XML syntax in the new file is different from the old file. You'll also need to put the information was returned by the `Plugin.getAbout()` method in this file. See Section 25.1.2, “Make the plug-in compatible with the auto-installation wizard”(page 224) for more information.
- The information that was in `META-INF/base-configurations.xml`, should be moved to `META-INF/plugin-configurations.xml`. The XML syntax in the new and old file is the same.
- All plug-ins and extensions are required to be installed in the directory specified by `plugins.dir` setting in the `base.config` file. Sub-directories are *not* searched any longer. This is usually not something that you need to worry about as a developer except that you should make sure the

installation instruction are up to date. See Section 21.1, “Managing plug-ins and extensions” (page 178) and ticket #1592 (Unified installation procedure for plug-ins, extensions and more...) ⁷ for more information.

- Plug-ins/extensions that depend on 3-rd party JAR files are now recommended to include those JAR file inside the `META-INF/lib` directory in the plug-in JAR file. The `Class-Path` attribute in `META-INF/MANIFEST.MF` must still be set. See Section 25.1, “How to organize your plug-in project” (page 222) and ticket #1594 (JarClassLoader support for JARs within JARs) ⁸.

Chapter 24. Developer overview of BASE

This section gives a brief overview of the architecture used in BASE. This is a good starting point if you need to know how various parts of BASE are glued together. The figure below should display most of the important parts in BASE. The following sections will briefly describe some parts of the figure and give you pointers for further reading if you are interested in the details.

Figure 24.1. Overview of the BASE application

24.1. Fixed vs. dynamic database

BASE stores most of its data in a database. The database is divided into two parts, one fixed and one dynamic part.

The fixed part contains tables that corresponds to the various items found in BASE. There is, for example, one table for users, one table for groups and one table for reporters. Some items share the same table. Biosources, samples and extracts are all biomaterials and share the `BioMaterials` table. The access to the fixed part of the database goes through Hibernate in most cases or through the Batch API in some cases (for example, access to reporters).

The dynamic part of the database contains tables for storing analyzed data. Each experiment has it's own set of tables and it is not possible to mix data from two experiments. The dynamic part of the database can only be accessed by the Batch API and the Query API using SQL and JDBC.

Note

The actual location of the two parts depends on the database that is used. MySQL uses two separate databases while PostgreSQL uses one database with two schemas.

More information

- Section 28.5, “The Dynamic API” (page 338)

24.2. Hibernate and the DbEngine

Hibernate (www.hibernate.org¹) is an object/relational mapping software package. It takes plain Java objects and stores them in a database. All we have to do is to set the properties on the objects (for example: `user.setName("A name")`). Hibernate will take care of the SQL generation and database communication for us. This is not a magic or automatic process. We have to provide mapping information about what objects goes into which tables and what properties goes into which columns, and other stuff like caching and proxy settings, etc. This is done by annotating the code with Javadoc comments. The classes that are mapped to the database are found in the `net.sf.basedb.core.data` package, which is shown as the **Data classes** box in the image above. The `HibernateUtil` class contains a lot of functionality for interacting with Hibernate.

Hibernate supports many different database systems. In theory, this means that BASE should work with all those databases. However, in practice we have found that this is not the case. For example, Oracle converts empty strings to null values, which breaks some parts of our code that expects non-null values. Another difficulty is that our Batch API and some parts of the Query API:s generates native SQL as well. We try to use database dialect information from Hibernate, but it is not always possible. The `DbEngine` contains code for generating the SQL that Hibernate can't help us with. We have implemented a generic `DefaultDbEngine` which follows ANSI specifications and special drivers for MySQL (`MySQLEngine`) and PostgreSQL (`PostgresDbEngine`). We don't expect BASE to work with other databases without modifications.

More information

- Section 30.3.4, “Data-layer rules” (page 375)
- www.hibernate.org²

24.3. The Batch API

Hibernate comes with a price. It affects performance and uses a lot of memory. This means that those parts of BASE that often handles lots of items at the same time doesn't work well with Hibernate. This is for example reporters, array design features and raw data. We have created the Batch API to solve these problems.

The Batch API uses JDBC and SQL directly against the database. However, we still use metadata and database dialect information available from Hibernate to generate most of the SQL we need. In

¹ <http://www.hibernate.org>

theory, this should make the Batch API just as database-independent as Hibernate is. In practice there is some information that we can't extract from Hibernate so we have implemented a simple `DbEngine` to account for missing pieces. The Batch API can be used for any `BatchableData` class in the fixed part of the database and is the only way for adding data to the dynamic part.

Note

The main reason for the Batch API is to avoid the internal caching of Hibernate which eats lots of memory when handling thousands of items. Hibernate 3.1 introduced a new stateless API which among other things doesn't do any caching. This version was released after we had created the Batch API. We made a few tests to check if it would be better for us to switch back to Hibernate but found that it didn't perform as well as our own Batch API (it was about 2 times slower). In any case, we can never get Hibernate to work with the dynamic database, so the Batch API is needed.

More information

- Section 28.3.6, “Batch operations” (page 330)
- Section 28.5, “The Dynamic API” (page 338)
- Section 30.3.6, “Batch-class rules” (page 407)

24.4. Data classes vs. item classes

The data classes are, with few exceptions, for internal use. These are the classes that are mapped to the database with Hibernate mapping files. They are very simple and contains no logic at all. They don't do any permission checks or any data validation.

Most of the data classes has a corresponding item class. For example: `UserData` and `User`, `GroupData` and `Group`. The item classes are what the client applications can see and use. They contain logic for permission checking (for example if the logged in user has `WRITE` permission) and data validation (for example setting a required property to null).

The exception to the above scheme are the batchable classes, which are all subclasses of the `BatchableData` class. For example, there is a `ReporterData` class but no corresponding item class. Instead there is a batcher implementation, `ReporterBatcher`, which takes care of the more or less the same things that an item class does, but it also takes care of it's own SQL generation and JDBC calls that bypasses Hibernate and the caching system.

More information

- Section 30.3.4, “Data-layer rules” (page 375)
- Section 30.3.5, “Item-class rules” (page 389)
- Section 30.3.6, “Batch-class rules” (page 407)
- Section 28.3.2, “Access permissions” (page 330)
- Section 28.3.3, “Data validation” (page 330)
- Section 28.3.6, “Batch operations” (page 330)

24.5. The Query API

The Query API is used to build and execute queries against the data in the database. It builds a query by using objects that represents certain operations. For example, there is an `EqRestriction` object which tests if two expressions are equal and there is an `AddExpression` object which adds two expressions. In this way it is possible to build very complex queries without using SQL or HQL.

The Query API knows how to work both via Hibernate and via SQL. In the first case it generates HQL (Hibernate Query Language) statements which Hibernate then translates into SQL. In the second case SQL is generated directly. In most cases HQL and SQL are identical, but not always. Some situations are solved by having the Query API generate slightly different query strings (with the help of information from Hibernate and the DbEngine). Some query elements can only be used with one of the query types.

Note

The object-based approach makes it a bit difficult to store a query for later reuse. The `net.sf.basedb.util.jep` package contains an expression parser that can be used to convert a string to `Restriction:s` and `Expression:s` for the Query API. While it doesn't cover 100% of the cases it should be useful for the `WHERE` part of a query.

More information

- Section 28.4, “The Query API” (page 338)

24.6. The Controller API

The Controller API is the very heart of the BASE system. This part of the core is used for boring but essential details, such as user authentication, database connection management, transaction management, data validation, and more. We don't write more about this part here, but recommends reading the documents below.

More information

- Section 28.3, “The Core API” (page 330)

24.7. The Extensions API

An extensions mechanism makes it possible to add functionality to BASE by external parties without having to modify the BASE code. This is not something that can be done at random, but BASE define a number of *extension points* which can be seen as a contract that must be fulfilled by the external code. Extension points are defined both by the BASE core (for example, file validators) and by the BASE web client (for example, menu and toolbar entries). In many cases, the distinction between an extension and a plug-in is fine. One major difference is that extensions are invoked and used immediately and are never queued for later execution. The installation mechanism is the same for both extensions and plug-ins and many packages use both types to provide a better user experience.

More information

- Chapter 26, *Extensions developer* (page 264)
- Section 28.6, “The Extensions API” (page 338)

24.8. Plug-ins

From the core code's point of view a plug-in is just another client application. A plug-in doesn't have more powers and doesn't have access to some special API that allows it to do cool stuff that other clients can't.

However, the core must be able to control when and where a plug-in is executed. Some plug-ins may take a long time doing their calculations and may use a lot of memory. It would be bad if a several users started to execute a resource-demanding plug-in at the same time. This problem is solved by adding a job queue. Each plug-in that should be executed is registered as `Job` in the database. A

job controller is checking the job queue at regular intervals. The job controller can then choose if it should execute the plug-in or wait depending on the current load on the server.

Note

BASE ships with two types of job controllers. One internal that runs inside the web application, and one external that is designed to run on separate servers, so called job agents. The internal job controller should work fine in most cases. The drawback with this controller is that a badly written plug-in may crash the entire web server. For example, a call to `System.exit()` in the plug-in code shuts down Tomcat as well.

More information

- Chapter 25, *Plug-in developer* (page 222)
- Section 28.3.8, “Plugin execution / job queue” (page 330)

24.9. Client applications

Client applications are application that use the BASE Core API. The current web application is built with Java Server Pages (JSP). JSP is supported by several application servers but we have only tested it with Tomcat. Another client application is the external job agents that executes plug-ins on separate servers.

Although it is possible to develop a completely new client application from scratch we don't see this as a likely thing to happen. Instead, there are some other possibilities to access data in BASE and to extend the functionality in BASE.

The first possibility is to use the Web Service API. This allows you to access some of the data in the BASE database and download it for further use. The Web Service API is currently very limited but it is not hard to extend it to cover more use cases.

A second possibility is to use the Extension API. This allows a developer to add functionality that appears directly in the web interface. For example, additional menu items and toolbar buttons. This API is also easy to extend to cover more use cases.

More information

- Chapter 27, *Web services* (page 282)
- Chapter 26, *Extensions developer* (page 264)
- The BASE plug-ins site³ also has examples of extensions and web services implementations.

Chapter 25. Plug-in developer

25.1. How to organize your plug-in project

25.1.1. Using Ant

Here is a simple example of how you might organize your project using ant (<http://ant.apache.org>) as the build tool. This is just a recommendation that we have found to be working well. You may choose to do it another way.

Directory and file layout

Create a directory on your computer where you want to store your plug-in project. This directory is the *pluginname/* directory in the listing below. You should also create some subdirectories:

```
pluginname/  
- /bin/  
- /lib/  
- /src/ org/company/  
- /META-INF/  
  - /META-INF/MANIFEST.MF  
  - /META-INF/extensions.xml  
  - /META-INF/lib/
```

The `bin/` directory is empty to start with. It will contain the compiled code. In the `lib/` directory you should put `base-core-3.x.x.jar` and other library files that is needed when compiling the source code, but doesn't have to be distributed with your plug-in (since they are already included in the BASE distribution). In the `META-INF/lib/` directory you should put other library files that are needed both when compiling and when distributing your plug-in.

The `src/` directory contains your source code. In this directory you should create subdirectories corresponding to the package name of your plug-in class(es). See http://en.wikipedia.org/wiki/Java_package for information about conventions for naming packages. The `META-INF/` directory contains metadata about the plug-in and are needed for easy installation. Typically you'll always need `MANIFEST.MF` and `extensions.xml` but, depending on what you are developing, other files are also needed.

The build file

In the root of your directory, create the build file: `build.xml`. Here is an example that will compile your plug-in and put it in a JAR file.

Example 25.1. A simple build file

```
<?xml version="1.0" encoding="UTF-8"?>
<project
  name="MyPlugin"
  default="build.plugin"
  basedir="."
>

  <!-- variables used -->
  <property name="plugin.name" value="MyPlugin" />
  <property name="src" value="src" />
  <property name="bin" value="bin" />

  <!-- set up classpath for compiling -->
  <path id="classpath">
    <fileset dir="lib">
      <include name="**/*.jar"/>
    </fileset>
    <fileset dir="META-INF/lib">
      <include name="**/*.jar"/>
    </fileset>
  </path>

  <!-- main target -->
  <target
    name="build.plugin"
    description="Compiles the plug-in and put in jar"
  >
    <javac
      encoding="UTF-8"
      srcdir="${src}"
      destdir="${bin}"
      classpathref="classpath">
    </javac>
    <jar
      jarfile="${plugin.name}.jar"
      manifest="META-INF/MANIFEST.MF"
    >

      <!-- compiled source code -->
      <fileset dir="${bin}" />

      <!-- metadata and extra JAR files -->
      <fileset dir="." includes="META-INF/**" />
    </jar>
  </target>
</project>
```

If your plug-in depends on JAR files not included with the standard BASE installation, you must create the `META-INF/MANIFEST.MF` file. List the other JAR files as in the following example. If your plug-in does not depend on other JAR files, you may remove the `manifest` attribute of the `<jar>` tag.

```
Manifest-Version: 1.0
Class-Path: lib/OtherJar.jar lib/ASecondJar.jar
```

See also Section 25.7, “How BASE load plug-in classes”(page 261) for more information regarding class loading when a plug-in depends on a external JAR files.

To make installation of your plug-in easier it is a good idea to include the `META-INF/extensions.xml`. This file should include information about your plug-in, such as the name, author, version, and the main Java class for the plug-in. See Section 25.1.2, “Make the plug-in compatible with the auto-installation wizard” (page 224) for get more information about this feature.

Building the plug-in

Compile the plug-in simply by typing **ant** in the console window. If all went well the `MyPlugin.jar` will be created in the same directory.

To install the plug-in copy the JAR file to the server's plug-in directory. For more information about the actual installation read Section 21.1, “Managing plug-ins and extensions” (page 178).

25.1.2. Make the plug-in compatible with the auto-installation wizard

BASE has support for automatically detecting new plug-ins with the auto-installation wizard. The wizard makes it very easy for a server administrator to install new plug-ins. See Section 21.1.1, “Automatic installation wizard” (page 178).

The auto-install feature requires that a plug-in provides some information about itself. The wizard looks in all JAR file for the file `META-INF/extensions.xml`. This file contains some information about the plug-in(s). Here is an example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<extensions xmlns="http://base.thep.lu.se/extensions.xsd">
  <!-- information about the complete package (which may contain many plug-in definitions) -->
  <about>
    <name>My plug-in package</name>
    <description>
      This package contains some very useful plug-ins...
    </description>
    <version>1.3</version>
    <min-base-version>3.0</min-base-version>
    <copyright>You</copyright>
    <url>http://www.company.org/with/more/info/about/plugins</url>
    <email>some.email.address@company.org</email>
  </about>

  <!-- Defines a single plug-in (can be repeated multiple times) -->
  <plugin-definition id="PluginX">
    <about>
      <name>Plug-in X</name>
      <description>
        Calculates the X transform of....
      </description>
    </about>
    <plugin-class>org.company.PluginClassX</plugin-class>
    <settings>
      <property name="everyone-use">1</property>
    </settings>
  </plugin-definition>
</extensions>
```

The first two lines should be the same in all `extensions.xml` files. The rest of the tags are described in following list.

`<about>`

The first `<about>` tag is information about the package as a whole. We recommend putting in as much information as possible here. Supported sub-tags are: `<name>`, `<description>`, `<version>`, `<min-base-version>`, `<max-base-version>`, `<contact>`, `<email>`, `<url>` and `<copyright>`

Each plug-in that is defined in this file may have it's `<about>` tag which override the global information. Typically, you'll only override the name and description tags and maybe also the min/max BASE version tags.

`<plugin-definition>`

This is the main element for defining a plug-in. It can override the global `<about>` information. We recommend that at least a `<name>` and `<description>` is provided.

`<plugin-class>`

The value is the full class name of the plug-in's main class.

`<min-base-version>` `<max-base-version>`

Optional sub-tags in the `<about>` section. Values are two- or three-digits separated by a dot. For example, 3.0.0 and 3.1. If values are given the plug-in will only be installed if the server's BASE version is within the bounds.

`<settings>`

Optional sub-tag that can contain one or more `<property>` tags. The settings can be used to configure some properties of the plug-in definition. So far, the following properties are recognized:

- `everyone-use`: If set, the plug-in is shared to the *EVERYONE* group with use permissions.
- `immediate-execution`: If set, the plug-in is given permissions to use the immediate execution feature. This is typically used for export plug-in that need to support immediate download.
- `max-memory`: The maximum memory (in bytes) to give to the plug-in when it is executed on a job agent. This setting is ignored when plug-ins are executed within the web client.
- `deprecated`: The plug-in has been deprecated. If it already exists on the BASE installation it will be disabled. If it doesn't exist, it will not be installed.

Installing plug-in configurations

The installation wizard has support for installing plug-in configurations. If some of your plug-ins has support for different configurations that you want to ship as part of the package, use the "Plug-in configuration exporter" in BASE and include the generated file in `META-INF/plugin-configurations.xml` inside your JAR file.

Note

The installation wizard can only install configurations that belong to plug-ins defined by the same package. It is currently not possible to, for example, include configuration for core plug-ins.

25.2. The Plug-in API

25.2.1. The main plug-in interfaces

The BASE API defines two interfaces and one abstract class that are vital for implementing plug-ins:

- `net.sf.basedb.core.plugin.Plugin`
- `net.sf.basedb.core.plugin.InteractivePlugin`
- `net.sf.basedb.core.plugin.AbstractPlugin`

A plug-in must always implement the `Plugin` interface. The `InteractivePlugin` interface is optional, and is only needed if you want user interaction. The `AbstractPlugin` is a useful base class that your plug-in can use as a superclass. It provides default implementations for some of the interface methods and also has utility methods for validating and storing job and configuration parameter values. Another reason to use this class as a superclass is that it will shield your plug-in from future changes to the Plug-in API. For example, if we decide that a new method is needed in the `Plugin` interface we will also try to add a default implementation in the `AbstractPlugin` class.

Important

A plug-in must also have public no-argument constructor. Otherwise, BASE will not be able to create new instances of the plug-in class.

The `net.sf.basedb.core.plugin.Plugin` interface

This interface defines the following methods and must be implemented by all plug-ins.

```
public Plugin.MainType getMainType();
```

Return information about the main type of plug-in. The `Plugin.MainType` is an enumeration with five possible values:

- **ANALYZE**: An analysis plug-in
- **EXPORT**: A plug-in that exports data
- **IMPORT**: A plug-in that imports data
- **INTENSITY**: A plug-in that calculates the original spot intensities from raw data
- **OTHER**: Any other type of plug-in

The returned value is stored in the database but is otherwise not used by the core. Client applications (such as the web client) will probably use this information to group the plug-ins, i.e., a button labeled **Export** will let you select among the export plug-ins.

Example 25.2. A typical implementation just return one of the values

```
public Plugin.MainType getMainType()
{
    return Plugin.MainType.OTHER;
}
```

```
public boolean supportsConfigurations();
```

If this method returns true, the plug-in can have different configurations, (i.e. `PluginConfiguration`). Note that this method may return true even if the `InteractivePlugin` interface is not implemented. The `AbstractPlugin` returns true for this method, which is the old way before the introduction of this method.

```
public boolean requiresConfiguration();
```

If this method returns true, the plug-in must have a configuration to be able to run. For example, some of the core import plug-ins must have information about the file format, to be able to import any data. The `AbstractPlugin` returns false for this method, which is the old way before the introduction of this method.

```
public Collection<Permissions> getPermissions();
```

Return a collection of permissions that the plug-in needs to be able to function as expected. This method may return null or an empty collection. In this case the plug-in permission system is not used and the plug-in always gets the same permissions as the logged in user. If permissions are specified the plug-in should list all permissions it requires. Permissions that are not listed are denied.

Note

The final assignment of permissions to a plug-in is always at the hands of a server administrator. He/she may decide to disable the plug-in permission system or revoke some of the requested permissions. The permissions returned by this method is only a recommendation that the server administrator may or may not accept. See Section 21.1.6, “Plug-in permissions” (page 184) for more information about plug-in permissions.

```
public void init (SessionControl sc,

    ParameterValues configuration,
    ParameterValues job)

throws BaseException;
```

Prepare the plug-in for execution or configuration. If the plug-in needs to do some initialization this is the place to do it. A typical implementation however only stores the passed parameters in instance variables for later use. Since it is not possible to know what the user is going to do at this stage, we recommend lazy initialisation of all other resources.

The parameters passed to this method has vital information that is needed to execute the plug-in. The `SessionControl` is a central core object holding information about the logged in user and is used to create `DbControl` objects which allows a plug-in to connect to the database to read, add or update information. The two `ParameterValues` objects contain information about the configuration and job parameters to the plug-in. The configuration object holds all parameters stored together with a `PluginConfiguration` object in the database. If the plug-in is started without a configuration this object is null. The job object holds all parameters that are stored together with a `Job` object in the database. This object is null if the plug-in is started without a job.

The difference between a configuration parameter and a job parameter is that a configuration is usually something an administrator sets up, while a job is an actual execution of a plug-in. For example, a configuration for an import plug-in holds the regular expressions needed to parse a text file and find the headers, sections and data lines, while the job holds the file to parse.

The `AbstractPlugin` contains an implementation of this method that saves the passed objects in protected instance variables. If you override this method we recommend that you also call `super.init()`.

Example 25.3. The `AbstractPlugin` implementation of `Plugin.init()`

```
protected SessionControl sc = null;
protected ParameterValues configuration = null;
protected ParameterValues job = null;
/**
 * Store copies of the session control, plug-in and job configuration. These
 * are available to subclasses in the {@link #sc}, {@link #configuration}
 * and {@link #job} variables. If a subclass overrides this method it is
 * recommended that it also calls super.init(sc, configuration, job).
 */
public void init(SessionControl sc,
    ParameterValues configuration, ParameterValues job)
    throws BaseException
{
    this.sc = sc;
    this.configuration = configuration;
    this.job = job;
}
```

```
public void run (Request request,

    Response response,
    ProgressReporter progress);
```

Run the plug-in.

The `request` parameter is of historical interest only. It has no useful information and can be ignored.

The `progress` parameter should be used by a plug-in to report its progress back to the core. The core will usually send the progress information to the database, which allows users to see exactly how the plug-in is progressing from the web interface. This parameter is allowed to be null, but BASE will always use a progress reporter. The plug-in should try to not over-use the

progress reporter. The default implementation used by BASE has a time threshold so that calls that occur too often with too little change in the progress are ignored. A good starting point is to divide the work into 100 pieces each representing 1% of the work, i.e., if the plug-in should export 100 000 items it should report progress after every 1000 items.

The response parameter is used to tell the core if the plug-in was successful or failed. Not setting a response is considered a failure by the core. From the run method it is only allowed to use one of the `Response.setDone()`, `Response.setError()` or `Response.setContinue()` methods.

Important

It is also considered bad practice to let exceptions escape out from this method. Always use `try...catch` to catch exceptions and use `Response.setError()` to report the error back to the core.

Example 25.4. Here is a skeleton for the `run()` method

```
public void run(Request request, Response response, ProgressReporter progress)
{
    // Open a connection to the database
    // sc is set by init() method
    DbControl dc = sc.newDbControl();
    try
    {
        // Insert code for plug-in here

        // Commit the work
        dc.commit();
        response.setDone("Plug-in ended successfully");
    }
    catch (Throwable t)
    {
        // All exceptions must be caught and sent back
        // using the response object
        response.setError(t.getMessage(), Arrays.asList(t));
    }
    finally
    {
        // IMPORTANT!!! Make sure opened connections are closed
        if (dc != null) dc.close();
    }
}
```

```
public void done();
```

Clean up all resources after executing the plug-in. This method must not throw any exceptions.

Example 25.5. The `AbstractPlugin` contains an implementation of the `done()` method simply sets the parameters passed to the `init()` method to null

```
/**
 * Clears the variables set by the init method. If a subclass
 * overrides this method it is recommended that it also calls super.done().
 */
public void done()
{
    configuration = null;
    job = null;
    sc = null;
}
```

The `net.sf.basedb.core.plugin.InteractivePlugin` interface

If you want the plug-in to be able to interact with the user you must also implement this interface. This is probably the case for most plug-ins. Among the core plug-ins shipped with BASE the `SpO-`

`tImageCreator` is one plug-in that does not interact with the user. Instead, the web client has special JSP pages that handles all the interaction, creates a job for it and sets the parameters. This approach can also be used for other plug-ins if, for example, an extension is used to provide the gui.

The `InteractivePlugin` has three main tasks:

1. Tell a client application where the plug-in should be plugged in.
2. Ask the users for configuration and job parameters.
3. Validate parameter values entered by the user and store those in the database.

This requires that the following methods are implemented.

```
public Set<GuiContext> getGuiContexts();
```

Return information about where the plug-in should be plugged in. Each place is identified by a `GuiContext` object, which is an `Item` and a `Type`. The item is one of the objects defined by the `Item` enumeration and the type is either `Type.LIST` or `Type.ITEM`, which corresponde to the list view and the single-item view in the web client.

For example, the `GuiContext = (Item.REPORTER, Type.LIST)` tells a client application that this plug-in can be plugged in whenever a list of reporters is displayed. The `GuiContext = (Item.REPORTER, Type.ITEM)` tells a client application that this plug-in can be plugged in whenever a single reporter is displayed. The first case may be appropriate for a plug-in that imports or exports reporters. The second case may be used by a plug-in that updates the reporter information from an external source (well, it may make sense to use this in the list case as well).

The returned information is copied by the core at installation time to make it easy to ask for all plug-ins for a certain `GuiContext`.

A typical implementation creates a static unmodifiable `Set` which is returned by this method. It is important that the returned set cannot be modified. It may be a security issue if a misbehaving client application does that.

Example 25.6. A typical implementation of `getGuiContexts`

```
// From the net.sf.basedb.plugins.RawDataFlatFileImporter plug-in
private static final Set<GuiContext> guiContexts =
    Collections.singleton(new GuiContext(Item.RAWBIOASSAY, GuiContext.Type.ITEM));

public Set<GuiContext> getGuiContexts()
{
    return guiContexts;
}
```

```
public String isInContext(GuiContext context,
                          Object item);
```

This method is called to check if a particular item is usable for the plug-in. This method is invoked to check if a plug-in can be used in a given context. If invoked from a list context the `item` parameter is `null`. The plug-in should return `null` if it finds that it can be used. If the plug-in can't be used it must decide if the reason should be a warning or an error condition.

A warning is issued by returning a string with the warning message. It should be used when the plug-in can't be used because it is unrelated to the current task. For example, a plug-in for importing Genepix data should return a warning when somebody wants to import data to an Agilent raw bioassay.

An error message is issued by throwing an exception. This should be used when the plug-in is related to the current task but still can't do what it is supposed to do. For example, trying to import raw data if the logged in user doesn't have write permission to the raw bioassay.

As a rule of thumb, if there is a chance that another plug-in might be able to perform the same task a warning should be used. If it is guaranteed that no other plug-in can do it an error message should be used.

Here is a real example from the `RawDataFlatFileImporter` plug-in which imports raw data to a `RawBioAssay`. Thus, `GuiContext = (Item.RAWBIOASSAY, Type.ITEM)`, but the plug-in can only import data if the logged in user has write permission, there is no data already, and if the raw bioassay has the same raw data type as the plug-in has been configured for.

Example 25.7. A realistic implementation of the `isInContext()` method

```
/**
 * Returns null if the item is a {@link RawBioAssay} of the correct
 * {@link RawDataType} and doesn't already have spots.
 * @throws PermissionDeniedException If the raw bioassay already has raw data
 * or if the logged in user doesn't have write permission
 */
public String isInContext(GuiContext context, Object item)
{
    String message = null;
    if (item == null)
    {
        message = "The object is null";
    }
    else if (!(item instanceof RawBioAssay))
    {
        message = "The object is not a RawBioAssay: " + item;
    }
    else
    {
        RawBioAssay rba = (RawBioAssay)item;
        String rawDataType = (String)configuration.getValue("rawDataType");
        RawDataType rdt = rba.getRawDataType();
        if (!rdt.getId().equals(rawDataType))
        {
            // Warning
            message = "Unsupported raw data type: " + rba.getRawDataType().getName();
        }
        else if (!rdt.isStoredInDb())
        {
            // Warning
            message = "Raw data for raw data type '" + rdt + "' is not stored in the database";
        }
        else if (rba.hasData())
        {
            // Error
            throw new PermissionDeniedException("The raw bioassay already has data.");
        }
        else
        {
            // Error
            rba.checkPermission(Permission.WRITE);
        }
    }
    return message;
}
```

```
public RequestInformation getRequestInformation(GuiContext context,
                                              String command)
throws BaseException;
```

Ask the plug-in for parameters that need to be entered by the user. The `GuiContext` parameter is one of the contexts returned by the `getGuiContexts` method. The `command` is a string telling

the plug-in what command was executed. There are two predefined commands but as you will see the plug-in may define its own commands. The two predefined commands are defined in the `net.sf.basedb.core.plugin.Request` class.

`Request.COMMAND_CONFIGURE_PLUGIN`

Used when an administrator is initiating a configuration of the plug-in.

`Request.COMMAND_CONFIGURE_JOB`

Used when a user has selected the plug-in for running a job.

Given this information the plug-in must return a `RequestInformation` object. This is simply a title, a description, and a list of parameters. Usually the title will end up as the input form title and the description as a help text for the entire form. Do not put information about the individual parameters in this description, since each parameter has a description of its own.

Example 25.8. When running an import plug-in it needs to ask for the file to import from and if existing items should be updated or not

```
// The complete request information
private RequestInformation configure Job;

// The parameter that asks for a file to import from
private PluginParameter<File> file Parameter;

// The parameter that asks if existing items should be updated or not
private PluginParameter<Boolean> updateExistingParameter;

public RequestInformation getRequestInformation(GuiContext context, String command)
    throws BaseException
{
    RequestInformation requestInformation = null;
    if (command.equals(Request.COMMAND_CONFIGURE_PLUGIN))
    {
        requestInformation = getConfigurePlugin();
    }
    else if (command.equals(Request.COMMAND_CONFIGURE_JOB))
    {
        requestInformation = getConfigureJob();
    }
    return requestInformation;
}

/**
 * Get (and build) the request information for starting a job.
 */
private RequestInformation getConfigureJob()
{
    if (configureJob == null)
    {
        // A file is required
        fileParameter = new PluginParameter<File>(
            "file",
            "File",
            "The file to import the data from",
            new FileParameterType(null, true, 1)
        );

        // The default value is 'false'
        updateExistingParameter = new PluginParameter<Boolean>(
            "updateExisting",
            "Update existing items",
            "If this option is selected, already existing items will be updated " +
            " with the information in the file. If this option is not selected " +
            " existing items are left untouched.",
            new BooleanParameterType(false, true)
        );

        List<PluginParameter<?>> parameters =
            new ArrayList<PluginParameter<?>>(2);
        parameters.add(fileParameter);
        parameters.add(updateExistingParameter);

        configureJob = new RequestInformation
        (
            Request.COMMAND_CONFIGURE_JOB,
            "Select a file to import items from",
            "Description",
            parameters
        );
    }
    return configureJob;
}
```

As you can see it takes a lot of code to put together a `RequestInformation` object. For each parameter you need one `PluginParameter` object and one `ParameterType` object. To make life a little easier, a `ParameterType` can be reused for more than one `PluginParameter`.

```
StringParameterType stringPT = new StringParameterType(255, null, true);
PluginParameter one = new PluginParameter("one", "One", "First string", stringPT);
PluginParameter two = new PluginParameter("two", "Two", "Second string", stringPT);
// ... and so on
```

The `ParameterType` is an abstract base class for several subclasses each implementing a specific type of parameter. The list of subclasses may grow in the future, but here are the most important ones currently implemented.

Note

Most parameter types include support for supplying a predefined list of options to select from. In that case the list will be displayed as a drop-down list for the user, otherwise a free input field is used.

`StringParameterType`

Asks for a string value. Includes an option for specifying the maximum length of the string.

`FloatParameterType`, `DoubleParameterType`, `IntegerParameterType`, `LongParameterType`

Asks for numerical values. Includes options for specifying a range (min/max) of allowed values.

`BooleanParameterType`

Asks for a boolean value.

`DateParameterType`

Asks for a date.

`FileParameterType`

Asks for a file item.

`ItemParameterType`

Asks for any other item. This parameter type requires that a list of options is supplied, except when the item type asked for matches the current `GuiContext`, in which case the currently selected item is used as the parameter value.

`PathParameterType`

Ask for a path to a file or directory. The path may be non-existing and should be used when a plug-in needs an output destination, i.e., the file to export to, or a directory where the output files should be placed.

You can also create a `PluginParameter` with a null name and `ParameterType`. In this case, the web client will not ask for input from the user, instead it is used as a section header, allowing you to group parameters into different sections which increase the readability of the input parameters page.

```
PluginParameter firstSection = new PluginParameter(null, "First section", null, null);
PluginParameter secondSection = new PluginParameter(null, "Second section", null, null);
// ...

parameters.add(firstSection);
parameters.add(firstParameterInFirstSection);
parameters.add(secondParameterInFirstSection);

parameters.add(secondSection);
parameters.add(firstParameterInSecondSection);
parameters.add(secondParameterInSecondSection);
```

```
public void configure(GuiContext context,
    Request request,
    Response response);
```

Sends parameter values entered by the user for processing by the plug-in. The plug-in must validate that the parameter values are correct and then store them in database.

Important

No validation is done by the core, except converting the input to the correct object type, i.e. if the plug-in asked for a `Float` the input string is parsed and converted to a `Float`. If you have extended the `AbstractPlugin` class it is very easy to validate the parameters with the `AbstractPlugin.validateRequestParameters()` method. This method takes the same list of `PluginParameter`:s as used in the `RequestInformation` object and uses that information for validation. It returns null or a list of `Throwable`:s that can be given directly to the `response.setError()` methods.

When the parameters have been validated, they need to be stored in the database. Once again, it is very easy, if you use one of the `AbstractPlugin.storeValue()` or `AbstractPlugin.storeValues()` methods.

The `configure` method works much like the `Plugin.run()` method. It must return the result in the `Response` object, and should not throw any exceptions.

Example 25.9. Configuration implementation building on the examples above

```
public void configure(GuiContext context, Request request, Response response)
{
    String command = request.getCommand();
    try
    {
        if (command.equals(Request.COMMAND_CONFIGURE_PLUGIN))
        {
            // TODO
        }
        else if (command.equals(Request.COMMAND_CONFIGURE_JOB))
        {
            // Validate user input
            List<Throwable> errors =
                validateRequestParameters(getConfigureJob().getParameters(), request);
            if (errors != null)
            {
                response.setError(errors.size() +
                    " invalid parameter(s) were found in the request", errors);
                return;
            }

            // Store user input
            storeValue(job, request, fileParameter);
            storeValue(job, request, updateExistingParameter);

            // We are happy and done
            File file = (File) job.getValue("file");
            response.setSuggestedJobName("Import data from file " + file.getName());
            response.setDone("Job configuration complete", Job.ExecutionTime.SHORT);
        }
    }
    catch (Throwable ex)
    {
        response.setError(ex.getMessage(), Arrays.asList(ex));
    }
}
```

Note that the call to `response.setDone()` has a second parameter `Job.ExecutionTime.SHORT`. It is an indication about how long time it will take to execute the plug-in. This is of interest for job queue managers which probably does not want to start too many long-running jobs at

the same time blocking the entire system. Please try to use this parameter wisely and not use `Job.ExecutionTime.SHORT` out of old habit all the time.

The `Response` class also has a `setContinue()` method which tells the core that the plug-in needs more parameters, i.e. the core will then call `getRequestInformation()` again with the new command, let the user enter values, and then call `configure()` with the new values. This process is repeated until the plug-in reports that it is done or an error occurs.

Tip

You do not have to store all values the plug-in asked for in the first place. You may even choose to store different values than those that were entered. For example, you might ask for the mass and height of a person and then only store the body mass index, which is calculated from those values.

An important note is that during this iteration it is the same instance of the plug-in that is used. However, no parameter values are stored in the database until the plug-in sends a `response.setDone()`. After that, the plug-in instance is usually discarded, and a job is placed in the job queue. The execution of the plug-in happens in a new instance and maybe on a different server. This means that a plug-in can't store state from the configuration phase internally and expect it to be there in the execution phase. Everything the plug-in needs to do its job must be stored as parameters in the database.

The only exception to the above rule is if the plug-in answers with `Response.setExecuteImmediately()` or `Response.setDownloadImmediately()`. Doing so bypasses the entire job queue system and requests that the job is started immediately. This is a permission that has to be granted to each plug-in by the server administrator. If the plug-in has this permission, the same object instance that was used in the configuration phase is also used in the execution phase. This is the only case where a plug-in can retain internal state between the two phases.

25.2.2. How the BASE core interacts with the plug-in when...

This section describes how the BASE core interacts with the plug-in in a number of use cases. We will outline the order the methods are invoked on the plug-in.

Installing a plug-in

When a plug-in is installed the core is eager to find out information about the plug-in. To do this it calls the following methods in this order:

1. A new instance of the plug-in class is created. The plug-in must have a public no-argument constructor.
2. Calls are made to `Plugin.getMainType()`, `Plugin.supportsConfigurations()` and `Plugin.requiresConfiguration()` to find out information about the plug-in. This is the only time these methods are called. The information that is returned by them are copied and stored in the database for easy access.

Note

The `Plugin.init()` method is never called during plug-in installation.

3. If the plug-in implements the `InteractivePlugin` interface the `InteractivePlugin.getGuiContexts()` method is called. This is the only time this method is called and the information it returns are copied and stored in the database.
4. If the server admin decided to use the plug-in permission system, the `Plugin.getPermissions()` method is called. The returned information is copied and stored in the database.

Configuring a plug-in

The plug-in must implement the `InteractivePlugin` interface and the `Plugin.supportsConfigurations()` method must return `TRUE`. The configuration is done with a wizard-like interface (see Section 21.2.1, “Configuring plug-in configurations” (page 187)). The same plug-in instance is used throughout the entire configuration sequence.

1. A new instance of the plug-in class is created. The plug-in must have a public no-argument constructor.
2. The `Plugin.init()` method is called. The `job` parameter is `null`.
3. The `InteractivePlugin.getRequestInformation()` method is called. The `context` parameter is `null` and the `command` is the value of the string constant `Request.COMMAND_CONFIGURE_PLUGIN(_config_plugin)`.
4. The web client process the returned information and displays a form for user input. The plug-in will have to wait some time while the user enters data.
5. The `InteractivePlugin.configure()` method is called. The `context` parameter is still `null` and the `request` parameter contains the parameter values entered by the user.
6. The plug-in must validate the values and decide whether they should be stored in the database or not. We recommend that you use the methods in the `AbstractPlugin` class for this.
7. The plug-in can choose between three different responses:
 - `Response.setDone()`: The configuration is complete. The core will write any configuration changes to the database, call the `Plugin.done()` method and then discard the plug-in instance.
 - `Response.setError()`: There was one or more errors. The web client will display the error messages for the user and allow the user to enter new values. The process continues with step 4 (page 236).
 - `Response.setContinue()`: The parameters are correct but the plug-in wants more parameters. The process continues with step 3 (page 236) but the `command` has the value that was passed to the `setContinue()` method.

Checking if a plug-in can be used in a given context

If the plug-in is an `InteractivePlugin` it has specified in which contexts it can be used by the information returned from `InteractivePlugin.getGuiContexts()` method. The web client uses this information to decide whether, for example, a **Run plugin** button should be displayed on a page or not. However, this is not always enough to know whether the plug-in can be used or not. For example, a raw data importer plug-in cannot be used to import raw data if the raw bioassay already has data. So, when the user clicks the button, the web client will load all plug-ins that possibly can be used in the given context and let each one of them check whether they can be used or not.

1. A new instance of the plug-in class is created. The plug-in must have a public no-argument constructor.
2. The `Plugin.init()` method is called. The `job` parameter is `null`. The `configuration` parameter is `null` if the plug-in does not have any configuration parameters.
3. The `InteractivePlugin.isInContext()` is called. If the `context` is a list context, the `item` parameter is `null`, otherwise the current item is passed. The plug-in should return `null` if it can be used under the current circumstances, or a message explaining why not.
4. After this, `Plugin.done()` is called and the plug-in instance is discarded. If there are several configurations for a plug-in, this procedure is repeated for each configuration.

Creating a new job

If the web client found that the plug-in could be used in a given context and the user selected the plug-in, the job configuration sequence is started. It is a wizard-like interface identical to the configuration wizard. In fact, the same JSP pages, and calling sequence is used. See the section called “Configuring a plug-in” (page 236). We do not repeat everything here. There are a few differences:

- The `job` parameter is not null, but it does not contain any parameter values to start with. The plug-in should use this object to store job-related parameter values. The `configuration` parameter is null if the plug-in is started without configuration. In any case, the configuration values are write-protected and cannot be modified.
- The first call to `InteractivePlugin.getRequestInformation()` is done with `Request.COMMAND_CONFIGURE_JOB (_configjob)` as the command. The `context` parameter reflects the current context.
- When calling `Response.setDone()` the plug-in should use the variant that takes an estimated execution time. If the plug-in has support for immediate execution or download (export plug-ins only), it can also respond with `Response.setExecuteImmediately()` or `Response.setDownloadImmediately()`.

If the plug-in requested and was granted immediate execution or download the same plug-in instance is used to execute the plug-in. This may be done with the same or a new thread. Otherwise, a new job is added to the job queue, the parameter value are saved and the plug-in instance is discarded after calling the `Plugin.done()` method.

Executing a job

Normally, the creation of a job and the execution of it are two different events. The execution may as well be done on a different server. See Section 20.3, “Installing job agents” (page 171). This means that the execution takes place in a different instance of the plug-in class than what was used for creating the job. The exception is if a plug-in supports immediate execution or download. In this case the same instance is used, and it is, of course, always executed on the web server.

1. A new instance of the plug-in class is created. The plug-in must have a public no-argument constructor.
2. The `Plugin.init()` method is called. The `job` parameter contains the job configuration parameters. The `configuration` parameter is null if the plug-in does not have any configuration parameters.
3. The `Plugin.run()` method is called. It is finally time for the plug-in to do the work it has been designed for. This method should not throw any exceptions. Use the `Response.setDone()` method to report success, the `Response.setError()` method to report errors or the `Response.setContinue()` method to respond to a shutdown signal and tell the core to resume the job once the system is up and running again. The `Response.setContinue()` can also be used when the system is not shutting down. The job will then be put back into the job queue and executed again later.
4. In all cases the `Plugin.done()` method is called and the plug-in instance is discarded.

25.2.3. Abort a running a plug-in

BASE includes a simple signalling system that can be used to send signals to plug-ins. The system was primarily developed to allow a user to kill a plug-in when it is executing. Therefore, the focus of this chapter will be how to implement a plug-in to make it possible to kill it during its execution.

Since we don't want to do this by brute force such as destroying the process or stopping thread the plug-in executes in, cooperation is needed by the plug-in. First, the plug-in must implement

the `SignalTarget` interface. From this, a `SignalHandler` can be created. A plug-in may choose to implement its own signal handler or use an existing implementation. BASE, for example, provides the `ThreadSignalHandler` implementation that supports the `ABORT` signal. This is a simple implementation that just calls `Thread.interrupt()` on the plug-in worker thread. This may cause two different effects:

- The `Thread.interrupted()` flag is set. The plug-in must check this at regular intervals and if the flag is set it must cleanup, rollback open transactions and exit as soon as possible.
- If the plug-in is waiting in a blocking call that is interruptable, for example `Thread.sleep()`, an `InterruptedException` is thrown. This should cause the same actions as if the flag was set to happen.

Not all blocking calls are interruptable

For example calling `InputStream.read()` may leave the plug-in waiting in a non-interruptable state. In this case there is nothing BASE can do to wake it up again.

Example 25.10. A plug-in that uses the `ThreadSignalHandler`

```
private ThreadSignalHandler signalHandler;
public SignalHandler getSignalHandler()
{
    signalHandler = new ThreadSignalHandler();
    return signalHandler;
}

public void run(Request request, Response response, ProgressReporter progress)
{
    if (signalHandler != null) signalHandler.setWorkerThread(null);
    beginTransaction();
    boolean done = false;
    boolean interrupted = false;
    while (!done && !interrupted)
    {
        try
        {
            done = doSomeWork(); // NOTE! This must not take forever!
            interrupted = Thread.interrupted();
        }
        catch (InterruptedException ex)
        {
            // NOTE! Try-catch is only needed if thread calls
            // a blocking method that is interruptable
            interrupted = true;
        }
    }
    if (interrupted)
    {
        rollbackTransaction();
        response.setError("Aborted by user", null);
    }
    else
    {
        commitTransaction();
        response.setDone("Done");
    }
}
```

Other signal handler implementations are `ProgressReporterSignalHandler` and `EnhancedThreadSignalHandler`. The latter handler also has support for the `SHUTDOWN` signal which is sent to plug-in when the system is shutting down. Clever plug-ins may use this to enable them to be restarted when the system is up and running again. See that javadoc for information about how to use it. For more information about the signalling system as a whole, see Section 28.3.10, “Sending signals (to plug-ins)” (page 336).

25.2.4. Using custom JSP pages for parameter input

This is an advanced option for plug-ins that require a different interface for specifying plug-in parameters than the default list showing one parameter at a time. This feature is used by setting the `RequestInformation.getJspPage()` property when constructing the request information object. If this property has a non-null value, the web client will send the browser to the specified JSP page instead of to the generic parameter input page.

When setting the JSP page you can either specify an absolute path or only the filename of the JSP file. If only the filename is specified, the JSP file is expected to be located in a special location, generated from the package name of your plug-in. If the plug-in is located in the package `org.company` the JSP file must be located in `<base-dir>/www/plugins/org/company/`.

An absolute path starts with `'/'` and may or may not include the root directory of the BASE installation. If, for example, BASE is installed to `http://your.base.server.com/base`, the following absolute paths are equivalent `/base/path/to/file.jsp`, `/path/to/file.jsp`.

In both cases, please note that the browser still thinks that it is showing the regular parameter input page at the usual location: `<base-dir>/www/common/plugin/index.jsp`. All links in your JSP page should be relative to that directory.

Even if you use your own JSP page we recommend that you use the built-in facility for passing the parameters back to the plug-in. For this to work you must:

- Generate the list of `PluginParameter` objects as usual.
- Name all your input fields in the JSP like: `parameter:name-of-parameter`

```
// Plug-in generate PluginParameter
StringParameterType stringPT = new StringParameterType(255, null, true);
PluginParameter one = new PluginParameter("one", "One", "First string", stringPT);
PluginParameter two = new PluginParameter("two", "Two", "Second string", stringPT);

// JSP should name fields as:
First string: <input type="text" name="parameter:one"><br>
Second string: <input type="text" name="parameter:two">
```

- Send the form to `index.jsp` with the `ID`, `cmd` and `requestId` parameters as shown below.

```
<form action="index.jsp" method="post">
<input type="hidden" name="ID" value="%ID%">
<input type="hidden" name="requestId" value="%request.getParameter("requestId")%">
<input type="hidden" name="cmd" value="SetParameters">
...
</form>
```

The `ID` is the session ID for the logged in user and is required. The `requestId` is the ID for this particular plug-in/job configuration sequence. It is optional, but we recommend that you use it since it protects your plug-in from getting mixed up with other plug-in configuration wizards. The `cmd=SetParameters` tells BASE to send the parameters to the plug-in for validation and saving.

Values are sent as strings to BASE that converts them to the proper value type before they are passed on to your plug-in. However, there is one case that can't be accurately represented with custom JSP pages, namely 'null' values. A null value is sent by not sending any value at all. This is not possible with a fixed form. It is of course possible to add some custom JavaScript that adds and removes form elements as needed, but it is also possible to let the empty string represent null. Just include a hidden parameter like this if you want an empty value for the 'one' parameter converted to null:

```
<input type="hidden" name="parameter:one:emptyIsNull" value="1">
```

If you want a **Cancel** button to abort the configuration this should be linked to a page reload with with the url: `index.jsp?ID=<%=ID%>&cmd=CancelWizard`. This allows BASE to clean up resources that has been put in global session variables.

In your JSP page you will probably need to access some information like the `SessionControl`, `Job` and possible even the `RequestInformation` object created by your plug-in.

```
// Get session control and its ID (required to post to index.jsp)
final SessionControl sc = Base.getExistingSessionControl(pageContext, true);
final String ID = sc.getId();

// Get information about the current request to the plug-in
PluginConfigurationRequest pcRequest =
    (PluginConfigurationRequest) sc.getSessionSetting("plugin.configure.request");
PluginDefinition plugin =
    (PluginDefinition) sc.getSessionSetting("plugin.configure.plugin");
PluginConfiguration pluginConfig =
    (PluginConfiguration) sc.getSessionSetting("plugin.configure.config");
PluginDefinition job =
    (PluginDefinition) sc.getSessionSetting("plugin.configure.job");
RequestInformation ri = pcRequest.getRequestInformation();
```

25.3. Import plug-ins

A plug-in becomes an import plugin simply by returning `Plugin.MainType.IMPORT` from the `Plugin.getMainType()` method.

25.3.1. Autodetect file formats

BASE has built-in functionality for autodetecting file formats. Your plug-in can be part of that feature if it reads its data from a single file. It must also implement the `AutoDetectingImporter` interface.

The `net.sf.basedb.core.plugin.AutoDetectingImporter` interface

```
public boolean isImportable(InputStream in)

throws BaseException;
```

Check the input stream if it seems to contain data that can be imported by the plugin. Usually it means scanning a few lines for some header matching a predefined string or regular expression.

The `AbstractFlatFileImporter` can be used for text-based files and implements this method by reading the headers from the input stream and checking if it stopped at an unknown type of line or not:

```
public final boolean isImportable(InputStream in)
    throws BaseException
{
    FlatFileParser ffp = getInitializedFlatFileParser();
    ffp.setInputStream(in);
    try
    {
        ffp.nextSection();
        FlatFileParser.LineType result = ffp.parseHeaders();
        if (result == FlatFileParser.LineType.UNKNOWN)
        {
            return false;
        }
        else
        {

```

```

        return isImportable(ffp);
    }
}
catch (IOException ex)
{
    throw new BaseException(ex);
}
}

```

The `AbstractFlatFileImporter` also has functions for setting the character set and automatic unwrapping of compressed files. See the javadoc for more information.

Note that the input stream doesn't have to be a text file (but you can't use the `AbstractFlatFileImporter` then). It can be any type of file, for example a binary or an XML file. In the case of an XML file you would need to validate the entire input stream in order to be 100% sure that it is a valid xml file, but we recommend that you only check the first few XML tags, for example, the `<!DOCTYPE >` declaration and/or the root element tag.

Try casting to `ImportInputStream`

In many cases (but not all) the auto-detect functionality uses a `ImportInputStream` as the `in` parameter. This class contains some metadata about the file the input stream is originating from. The most useful feature is the possibility to get information about the character set used in the file. This makes it possible to open text files using the correct character set.

```

String charset = Config.getCharset(); // Default value
if (in instanceof ImportInputStream)
{
    ImportInputStream iim = (ImportInputStream)in;
    if (iim.getCharacterSet() != null) charset = iim.getCharacterSet();
}
Reader reader = new InputStreamReader(in, Charset.forName(charset));

```

```

public void doImport(InputStream in,

    ProgressReporter progress)

throws BaseException;

```

Parse the input stream and import all data that is found. This method is of course only called if the `isImportable()` has returned true. Note however that the input stream is reopened at the start of the file. It may even be the case that the `isImportable()` method is called on one instance of the plugin and the `doImport()` method is called on another. Thus, the `doImport()` can't rely on any state set by the `isImportable()` method.

Call sequence during autodetection

The call sequence for autodetection resembles the call sequence for checking if the plug-in can be used in a given context.

1. A new instance of the plug-in class is created. The plug-in must have a public no-argument constructor.
2. The `Plugin.init()` method is called. The `job` parameter is null. The configuration parameter is null if the plug-in does not have any configuration parameters.
3. If the plug-in is interactive the `InteractivePlugin.isInContext()` is called. If the context is a list context, the `item` parameter is null, otherwise the current item is passed. The plug-in should return null if it can be used under the current circumstances, or a message explaining why not.
4. If the plug-in can be used the `AutoDetectingImporter.isImportable()` method is called to check if the selected file is importable or not.

5. After this, `Plugin.done()` is called and the plug-in instance is discarded. If there are several configurations for a plug-in, this procedure is repeated for each configuration. If the plug-in can be used without a configuration the procedure is also repeated without configuration parameters.
6. If a single plug-in was found the user is taken to the regular job configuration wizard. A new plug-in instance is created for this. If more than one plug-in was found the user is presented with a list of the plug-ins. After selecting one of them the regular job configuration wizard is used with a new plug-in instance.

25.3.2. The `AbstractFlatFileImporter` superclass

The `AbstractFlatFileImporter` is a very useful abstract class to use as a superclass for your own import plug-ins. It can be used if your plug-in uses regular text files that can be parsed by an instance of the `net.sf.basedb.util.FlatFileParser` class. This class parses a file by checking each line against a few regular expressions. Depending on which regular expression matches the line, it is classified as a header line, a section line, a comment, a data line, a footer line or unknown. Header lines are inspected as a group, but data lines individually, meaning that it consumes very little memory since only a few lines at a time needs to be loaded.

The `AbstractFlatFileImporter` defines `PluginParameter` objects for each of the regular expressions and other parameters used by the parser. It also implements the `Plugin.run()` method and does most of the ground work for instantiating a `FlatFileParser` and parsing the file. What you have to do in your plugin is to put together the `RequestInformation` objects for configuring the plugin and creating a job and implement the `InteractivePlugin.configure()` method for validating and storing the parameters. You should also implement or override some methods defined by `AbstractFlatFileImporter`.

Here is what you need to do:

- Implement the `InteractivePlugin` methods. See the section called “The `net.sf.basedb.core.plugin.InteractivePlugin` interface” (page 228) for more information. Note that the `AbstractFlatFileImporter` has defined many parameters for regular expressions used by the parser already. You should just pick them and put in your `RequestInformation` object.

```
// Parameter that maps the items name from a column
private PluginParameter<String> nameColumnMapping;

// Parameter that maps the items description from a column
private PluginParameter<String> descriptionColumnMapping;

private RequestInformation getConfigPluginParameters(GuiContext context)
{
    if (configurePlugin == null)
    {
        // To store parameters for CONFIGURE_PLUGIN
        List<PluginParameter<?>> parameters =
            new ArrayList<PluginParameter<?>>();

        // Parser regular expressions - from AbstractFlatFileParser
        parameters.add(parserSection);
        parameters.add(headerRegexpParameter);
        parameters.add(dataHeaderRegexpParameter);
        parameters.add(dataSplitterRegexpParameter);
        parameters.add(ignoreRegexpParameter);
        parameters.add(dataFooterRegexpParameter);
        parameters.add(minDataColumnsParameter);
        parameters.add(maxDataColumnsParameter);

        // Column mappings
        nameColumnMapping = new PluginParameter<String>(
            "nameColumnMapping",
            "Name",
            "Mapping that picks the items name from the data columns",
```

```

        new StringParameterType(255, null, true)
    );

    descriptionColumnMapping = new PluginParameter<String>(
        "descriptionColumnMapping",
        "Description",
        "Mapping that picks the items description from the data columns",
        new StringParameterType(255, null, false)
    );

    parameters.add(mappingSection);
    parameters.add(nameColumnMapping);
    parameters.add(descriptionColumnMapping);

    configurePlugin = new RequestInformation
    (
        Request.COMMAND_CONFIGURE_PLUGIN,
        "File parser settings",
        "",
        parameters
    );

    }
    return configurePlugin;
}

```

- Implement/override some of the methods defined by `AbstractFlatFileParser`. The most important methods are listed below.

```
protected FlatFileParser getInitializedFlatFileParser()
```

```
throws BaseException;
```

The method is called to create a `FlatFileParser` and set the regular expressions that should be used for parsing the file. The default implementation assumes that your plug-in has used the built-in `PluginParameter` objects and has stored the values at the configuration level. You should override this method if you need to initialise the parser in a different way. See for example the code for the `PrintMapFlatFileImporter` plug-in which has a fixed format and doesn't use configurations.

```

@Override
protected FlatFileParser getInitializedFlatFileParser()
    throws BaseException
{
    FlatFileParser ffp = new FlatFileParser();
    ffp.setSectionRegexp(Pattern.compile("\\[(.+?)\\]"));
    ffp.setHeaderRegexp(Pattern.compile("(.)=(.*)"));
    ffp.setDataSplitterRegexp(Pattern.compile(", "));
    ffp.setDataFooterRegexp(Pattern.compile(""));
    ffp.setMinDataColumns(12);
    return ffp;
}

```

```
protected boolean isImportable(FlatFileParser ffp)
```

```
throws IOException;
```

This method is called from the `isImportable(InputStream)` method, AFTER `FlatFileParser.nextSection()` and `FlatFileParser.parseHeaders()` has been called a single time and if the `parseHeaders` method didn't stop on an unknown line. The default implementation of this method always returns `TRUE`, since obviously some data has been found. A subclass may override this method if it wants to do more checks, for example, make that a certain header is present with a certain value. It may also continue parsing the file. Here is a code example from the `PrintMapFlatFileImporter` which checks if a `FormatName` header is present and contains either `TAM` or `MwBr`.

```
/**
 * Check that the file is a TAM or MwBr file.
 * @return TRUE if a FormatName header is present and contains "TAM" or "MwBr", FALSE
 *         otherwise
 */
@Override
protected boolean isImportable(FlatFileParser ffp)
{
    String formatName = ffp.getHeader("FormatName");
    return formatName != null &&
        (formatName.contains("TAM") || formatName.contains("MwBr"));
}
```

```
protected void begin(FlatFileParser ffp)
```

```
throws BaseException;
```

This method is called just before the parsing of the file begins. Override this method if you need to initialise some internal state. This is, for example, a good place to open a `DbControl` object, read parameters from the job and configuration and put them into more useful variables. The default implementation does nothing, but we recommend that `super.begin()` is always called.

```
// Snippets from the RawDataFlatFileImporter class
private DbControl dc;
private RawDataBatcher batcher;
private RawBioAssay rawBioAssay;
private Map<String, String> columnMappings;
private int numInserted;

@Override
protected void begin()
    throws BaseException
{
    super.begin();

    // Get DbControl
    dc = sc.newDbControl();
    rawBioAssay = (RawBioAssay) job.getValue(rawBioAssayParameter.getName());

    // Reload raw bioassay using current DbControl
    rawBioAssay = RawBioAssay.getById(dc, rawBioAssay.getId());

    // Create a batcher for inserting spots
    batcher = rawBioAssay.getRawDataBatcher();

    // For progress reporting
    numInserted = 0;
}
```

```
protected void handleHeader(FlatFileParser.Line line)
```

```
throws BaseException;
```

This method is called once for every header line that is found in the file. The `line` parameter contains information about the header. The default implementation of this method does nothing.

```
@Override
protected void handleHeader(Line line)
    throws BaseException
{
    super.handleHeader(line);
    if (line.name() != null && line.value() != null)
    {
        rawBioAssay.setHeader(line.name(), line.value());
    }
}
```

```
}
```

```
protected void handleSection(FlatFileParser.Line line)
```

```
throws BaseException;
```

This method is called once for each section that is found in the file. The `line` parameter contains information about the section. The default implementation of this method does nothing.

```
protected abstract void beginData()
```

```
throws BaseException;
```

This method is called after the headers has been parsed, but before the first line of data. This is a good place to add code that depends on information in the headers, for example, put together column mappings.

```
private Mapper reporterMapper;
private Mapper blockMapper;
private Mapper columnMapper;
private Mapper rowMapper;
// ... more mappers

@Override
protected void beginData()
{
    boolean cropStrings = ("crop".equals(job.getValue("stringTooLongError")));

    // Mapper that always return null; used if no mapping expression has been entered
    Mapper nullMapper = new ConstantMapper((String)null);

    // Column mappers
    reporterMapper = getMapper(ffp, (String)configuration.getValue("reporterIdColumnMapping"),
        cropStrings ? ReporterData.MAX_EXTERNAL_ID_LENGTH : null, nullMapper);
    blockMapper = getMapper(ffp, (String)configuration.getValue("blockColumnMapping"),
        null, nullMapper);
    columnMapper = getMapper(ffp, (String)configuration.getValue("columnColumnMapping"),
        null, nullMapper);
    rowMapper = getMapper(ffp, (String)configuration.getValue("rowColumnMapping"),
        null, nullMapper);
    // ... more mappers: metaGrid coordinate, X-Y coordinate, extended properties
    // ...
}
```

```
protected abstract void handleData(FlatFileParser.Data data)
```

```
throws BaseException;
```

This method is abstract and must be implemented by all subclasses. It is called once for every data line in the the file.

```
// Snippets from the RawDataFlatFileImporter class
@Override
protected void handleData(Data data)
    throws BaseException
{
    // Create new RawData object
    RawData raw = batcher.newRawData();

    // External ID for the reporter
    String externalId = reporterMapper.getValue(data);

    // Block, row and column numbers
    raw.setBlock(blockMapper.getInt(data));
    raw.setColumn(columnMapper.getInt(data));
    raw.setRow(rowMapper.getInt(data));
}
```

```
// ... more: metaGrid coordinate, X-Y coordinate, extended properties

// Insert raw data to the database
batcher.insert(raw, externalId);
numInserted++;
}
```

```
protected void end(boolean success);
```

Called when the parsing has ended, either because the end of file was reached or because an error has occurred. The subclass should close any open resources, ie. the `DbControl` object. The `success` parameter is `true` if the parsing was successful, `false` otherwise. The default implementation does nothing.

```
@Override
protected void end(boolean success)
    throws BaseException
{
    try
    {
        // Commit if the parsing was successful
        if (success)
        {
            batcher.close();
            dc.commit();
        }
    }
    catch (BaseException ex)
    {
        // Well, now we got an exception
        success = false;
        throw ex;
    }
    finally
    {
        // Always close... and call super.end()
        if (dc != null) dc.close();
        super.end(success);
    }
}
```

```
protected String getSuccessMessage();
```

This is the last method that is called, and it is only called if everything went successfully. This method allows a subclass to generate a short message that is sent back to the database as a final progress report. The default implementation returns `null`, which means that no message will be generated.

```
@Override
protected String getSuccessMessage()
{
    return numInserted + " spots inserted";
}
```

The `AbstractFlatFileImporter` has a lot of other methods that you may use and/or override in your own plug-in. Check the javadoc for more information.

Configure by example

The `ConfigureByExample` is a tagging interface that can be used by plug-ins using the `FlatFileParser` class for parsing. The web client detects if a plug-in implements this interface and if the list of parameters includes a section parameter with the name `parserSection` a **Test with file**

buttons is activated. This button will take the user to a form which allows the user to enter values for the parameters defined in the `AbstractFlatFileImporter` class. Parameters for column mappings must have the string "Mapping" in their names.

25.4. Export plug-ins

Export plug-ins are plug-ins that takes data from BASE, and prepares it for use with some external entity. Usually this means that data is taken from the database and put into a file with some well-defined file format. An export plug-in should return `MainType.EXPORT` from the `Plugin.getMainType()` method.

25.4.1. Immediate download of exported data

An export plug-in may want to give the user a choice between saving the exported data in the BASE file system or to download it immediately to the client computer. With the basic plug-in API the second option is not possible. The `ImmediateDownloadExporter` is an interface that extends the `Plugin` interface to provide this functionality. If your export plug-in wants to provide immediate download functionality it must implement the `ImmediateDownloadExporter` interface.

The `ImmediateDownloadExporter` interface

```
public void doExport (ExportOutputStream out,  
  
    ProgressReporter progress);
```

Perform the export. The plug-in should write the exported data to the `out` stream. If the `progress` parameter is not null, the progress should be reported at regular interval in the same manner as in the `Plugin.run()` method.

The `ExportOutputStream` class

The `ExportOutputStream` is an extension to the `java.io.OutputStream`. Use the regular `write()` methods to write data to it. It also has some additional methods, which are used for setting metadata about the generated file. These methods are useful, for example, when generating HTTP response headers.

Note

These methods must be called before starting to write data to the `out` stream.

```
public void setContentLength(long contentLength);
```

Set the total size of the exported data. Don't call this method if the total size is not known.

```
public void setMimeType(String mimeType);
```

Set the MIME type of the file that is being generated.

```
public void setCharacterSet(String charset);
```

Sets the character set used in text files. For example, UTF-8 or ISO-8859-1.

```
public void setFilename(String filename);
```

Set a suggested name of the file that is being generated.

Call sequence during immediate download

Supporting immediate download also means that the method call sequence is a bit altered from the standard sequence described in the section called “Executing a job” (page 237).

- The plug-in must call `Response.setDownloadImmediately()` instead of `Response.setDone()` in `Plugin.configure()` to end the job configuration wizard. This requests that the core starts an immediate download.

Note

Even if an immediate download is requested by the plug-in this feature may have been disabled by the server administrator. If so, the plug-in can choose if the job should be added to job queue or if this is an error condition.

- If immediate download is granted the web client will keep the same plug-in instance and call `ImmediateDownloadExporter.doExport()`. In this case, the `Plugin.run()` is never called. After the export, `Plugin.done()` is called as usual.
- If immediate download is not granted and the job is added to the job queue the regular job execution sequence is used.

25.4.2. The `AbstractExporterPlugin` class

This is an abstract superclass that will make it easier to implement export plug-ins that support immediate download. It defines `PluginParameter` objects for asking a user about a path where the exported data should be saved and if existing files should be overwritten or not. If the user leaves the path empty the immediate download functionality should be used. It also contains implementations of both the `Plugin.run()` method and the `ImmediateDownloadExporter.doExport()` method. Here is what you need to do in your own plug-in code (code examples are taken from the `HelpExporter`):

- Your plug-in should extend the `AbstractExporterPlugin` class:

```
public class HelpExporter
    extends AbstractExporterPlugin
    implements InteractivePlugin
```

- You need to implement the `InteractivePlugin.getRequestInformation()` method. Use the `getSaveAsParameter()` and `getOverwriteParameter()` methods defined in the superclass to create plug-in parameters that asks for the file name to save to and if existing files can be overwritten or not. You should also check if the administrator has enabled the immediate execution functionality for your plug-in. If not, the only option is to export to a file in the BASE file system and the filename is a required parameter.

```
// Selected parts of the getRequestConfiguration() method
...
List<PluginParameter<?>> parameters =
    new ArrayList<PluginParameter<?>>();
...
PluginDefinition pd = job.getPluginDefinition();
boolean requireFile = pd == null ?
    false : !pd.getAllowImmediateExecution();

parameters.add(getSaveAsParameter(null, null, defaultPath, requireFile));
parameters.add(getOverwriteParameter(null, null));

configureJob = new RequestInformation
(
    Request.COMMAND_CONFIGURE_JOB,
    "Help exporter options",
    "Set Client that owns the helptexts, " +
```

```

    "the file path where the export file should be saved",
    parameters
  );
  ....
  return configureJob;

```

- You must also implement the `configure()` method and check the parameters. If no filename has been given, you should check if immediate execution is allowed and set an error if it is not. If a filename is present, use the `pathCanBeUsed()` method to check if it is possible to save the data to a file with that name. If the file already exists it can be overwritten if the `OVERWRITE` is `TRUE` or if the file has been flagged for removal. Do not forget to store the parameters with the `storeValue()` method.

```

// Selected parts from the configure() method
if (request.getParameterValue(SAVE_AS) == null)
{
    if (!request.isAllowedImmediateExecution())
    {
        response.setError("Immediate download is not allowed. " +
            "Please specify a filename.", null);
        return;
    }
    Client client = (Client)request.getParameterValue("client");
    response.setDownloadImmediately("Export help texts for client application " +
        client.getName(), ExecutionTime.SHORTEST, true);
}
else
{
    if (!pathCanBeUsed((String)request.getParameterValue(SAVE_AS),
        (Boolean)request.getParameterValue(OVERWRITE)))
    {
        response.setError("File exists: " +
            (String)request.getParameterValue(SAVE_AS), null);
        return;
    }
    storeValue(job, request, ri.getParameter(SAVE_AS));
    storeValue(job, request, ri.getParameter(OVERWRITE));
    response.setDone("The job configuration is complete", ExecutionTime.SHORTEST);
}

```

- Implement the `performExport()` method. This is defined as abstract in the `AbstractExporter-Plugin` class. It has the same parameters as the `ImmediateDownloadExporter.doExport()` method and they have the same meaning. The only difference is that the out stream can be linked to a file in the BASE filesystem and not just to the HTTP response stream.
- Optionally, implement the `begin()`, `end()` and `getSuccessMessage()` methods. These methods do nothing by default.

The call sequence for plug-ins extending `AbstractExporterPlugin` is:

1. Call `begin()`.
2. Call `performExport()`.
3. Call `end()`.
4. Call `getSuccessMessage()` if running as a regular job. This method is never called when doing an immediate download since there is no place to show the message.

25.5. Analysis plug-ins

A plug-in becomes an analysis plug-in simply by returning `Plugin.MainType.ANALYZE` from the `Plugin.getMainType()` method. The information returned from `InteractivePlugin.getGuiContexts()` should include: `[Item.BIOASSAYSET, Type.ITEM]` or

[Item.DERIVEDBIOASSAYSET, Type.ITEM] since web client doesn't look for analysis plug-ins in most other places. If the plug-in can work on a subset of the bioassays it may also include [Item.BIOASSAY, Type.LIST] among the contexts. This will make it possible for a user to select bioassays from the list and then invoke the plug-in. The following code examples are taken from an analysis plug-in that is used in an experiment context.

```
private static final Set<GuiContext> guiContexts =
    Collections.singleton(new GuiContext(Item.BIOASSAYSET, GuiContext.Type.ITEM));

public Set<GuiContext> getGuiContexts()
{
    return guiContexts;
}
```

If the plug-in depends on a specific raw data type or on the number of channels, it should check that the current bioassayset is of the correct type in the `InteractivePlugin.isInContext()` method. It is also a good idea to check if the current user has permission to use the current experiment. This permission is needed to create new bioassaysets or other data belonging to the experiment.

```
public boolean isInContext(GuiContext context, Object item)
{
    if (item == null)
    {
        message = "The object is null";
    }
    else if (!(item instanceof BioAssaySet))
    {
        message = "The object is not a BioAssaySet: " + item;
    }
    else
    {
        BioAssaySet bas = (BioAssaySet)item;
        int channels = bas.getRawDataType().getChannels();
        if (channels != 2)
        {
            message = "This plug-in requires 2-channel data, not " + channels + "-channel.";
        }
        else
        {
            Experiment e = bas.getExperiment();
            e.checkPermission(Permission.USE);
        }
    }
}
```

The plug-in should always include a parameter asking for the current bioassay set when the `InteractivePlugin.getRequestInformation()` is called with `command = Request.COMMAND_CONFIGURE_JOB`.

```
private static final RequestInformation configurePlugin;
private RequestInformation configureJob;
private PluginParameter<BioAssaySet> bioAssaySetParameter;

public RequestInformation getRequestInformation(GuiContext context, String command)
    throws BaseException
{
    RequestInformation requestInformation = null;
    if (command.equals(Request.COMMAND_CONFIGURE_PLUGIN))
    {
        requestInformation = getConfigurePlugin(context);
    }
    else if (command.equals(Request.COMMAND_CONFIGURE_JOB))
    {
        requestInformation = getConfigureJob(context);
    }
}
```

```

        return requestInformation;
    }

    private RequestInformation getConfigJob(GuiContext context)
    {
        if (configureJob == null)
        {
            bioAssaySetParameter; = new PluginParameter<BioAssaySet>(
                "bioAssaySet",
                "Bioassay set",
                "The bioassay set used as the source for this analysis plugin",
                new ItemParameterType<BioAssaySet>(BioAssaySet.class, null, true, 1, null)
            );

            List<PluginParameter<?>> parameters = new ArrayList<PluginParameter<?>>();
            parameters.add(bioAssaySetParameter);
            // Add more plug-in-specific parameters here...

            configureJob = new RequestInformation(
                Request.COMMAND_CONFIGURE_JOB,
                "Configure job",
                "Set parameter for plug-in execution",
                parameters
            );
        }
        return configureJob;
    }
}

```

Of course, the `InteractivePlugin.configure()` method needs to validate and store the bioassay set parameter as well:

```

public void configure(GuiContext context, Request request, Response response)
{
    String command = request.getCommand();
    try
    {
        if (command.equals(Request.COMMAND_CONFIGURE_PLUGIN))
        {
            // Validate and store configuration parameters
            response.setDone("Plugin configuration complete");
        }
        else if (command.equals(Request.COMMAND_CONFIGURE_JOB))
        {
            List<Throwable> errors =
                validateRequestParameters(configureJob.getParameters(), request);
            if (errors != null)
            {
                response.setError(errors.size() +
                    " invalid parameter(s) were found in the request", errors);
                return;
            }
            storeValue(job, request, bioAssaySetParameter);
            // Store other plugin-specific parameters

            response.setDone("Job configuration complete", Job.ExecutionTime.SHORT);
        }
    }
    catch (Throwable ex)
    {
        // Never throw exception, always set response!
        response.setError(ex.getMessage(), Arrays.asList(ex));
    }
}

```

Now, the typical `Plugin.run()` method loads the specified bioassay set and its spot data. It may do some filtering and recalculation of the spot intensity value(s). In most cases it will store the result as a child bioassay set with one bioassay for each bioassay in the parent bioassay set. Here is an example, which just copies the intensity values, while removing those with a negative value in either channel.

```

public void run(Request request, Response response, ProgressReporter progress)
{
    DbControl dc = sc.newDbControl();
    try
    {
        BioAssaySet source = (BioAssaySet)job.getParameter("bioAssaySet");
        // Reload with current DbControl
        source = BioAssaySet.getById(dc, source.getId());
        int channels = source.getRawDataType().getChannels();

        // Create transformation and new bioassay set
        Job j = Job.getById(dc, job.getId());
        Transformation t = source.newTransformation(j);
        t.setName("Copy spot intensities >= 0");
        dc.saveItem(t);

        BioAssaySet result = t.newProduct(null, "new", true);
        result.setName("After: Copying spot intensities");
        dc.saveItem(result);

        // Get query for source data
        DynamicSpotQuery query = source.getSpotData();

        // Do not return spots with intensities < 0
        for (int ch = 1; ch <= channels; ++ch)
        {
            query.restrict(
                Restrictions.gteq(
                    Dynamic.column(VirtualColumn.channel(ch)),
                    Expressions.integer(0)
                )
            );
        }

        // Create batcher and copy data
        SpotBatcher batcher = result.getSpotBatcher();
        int spotsCopied = batcher.insert(query);
        batcher.close();

        // Commit and return
        dc.commit();
        response.setDone("Copied " + spotsCopied + " spots.");
    }
    catch (Throwable t)
    {
        response.setError(t.getMessage(), Arrays.asList(t));
    }
    finally
    {
        if (dc != null) dc.close();
    }
}

```

See Section 28.5, “The Dynamic API” (page 338) for more examples of using the analysis API.

25.5.1. The `AbstractAnalysisPlugin` class

This class is an abstract base class. It is a useful class for most analysis plug-ins used in an experiment context to inherit from. Its main purpose is to define `PluginParameter` objects that are commonly used in analysis plug-ins. This includes:

- The source bioassay set: `getSourceBioAssaySetParameter()`, `getCurrentBioAssaySet()`, `getSourceBioAssaySet()`
- The optional restriction of which bioassays to use. All bioassays in a bioassay set will be used if this parameter is empty. This is useful when the plugin only should run on a subset of bioassays in a bioassay set: `getSourceBioAssaysParameter()`, `getSourceBioAssays()`

- The name and description of the child bioassay set that is going to be created by the plug-in: `getChildNameParameter()`, `getChildDescriptionParameter()`
- The name and description of the transformation that represents the execution of the plug-in: `getTransformationNameParameter()`, `getTransformationName()`

25.5.2. The AnalysisFilterPlugin interface

The `net.sf.basedb.core.plugin.AnalysisFilterPlugin` is a tagging interface, with no methods, that all analysis plug-ins that only filters data should implement. The benefit is that they will be linked from the **Filter bioassay set** button and not just the **Run analysis** button. They will also get a different icon in the experiment outline to make filtering transformations appear different from other transformations.

The interface exists purely for making the user interaction better. There is no harm in not implementing it since the plug-in will always appear in from the **Run analysis** button. On the other hand, it doesn't cost anything to implement the interface since it doesn't have any methods.

25.6. Other plug-ins

25.6.1. Authentication plug-ins

This may be converted to an extension point in the future

There are certain plans to convert the authentication mechanism to an extension point in the future. There are several benefits with this:

- Easier installation since code doesn't have to be installed in the WEB-INF/lib or WEB-INF/classes directory.
- Allow support for multiple authentication modules. Users could either select a specific module, or each module could be tried in turn or some kind of auto-discovery can maybe be implemented.

See ticket #1599: Convert authentication plug-in system to an extension point¹ for more information.

BASE provides a plug-in mechanism for authenticating users (validating the username and password) when they are logging in. This plug-in mechanism is not the same as the regular plug-in API. That is, you do not have worry about user interaction or implementing the `Plugin` interface.

Internal vs. external authentication

BASE can authenticate users in two ways. Either it uses the internal authentication or the external authentication. With internal authentication BASE stores logins and passwords in its own database. With external authentication this is handled by some external application. Even with external authentication it is possible to let BASE cache the logins/passwords. This makes it possible to login to BASE if the external authentication server is down.

Note

An external authentication server can only be used to grant or deny a user access to BASE. It cannot be used to give a user permissions, or put a user into groups or different roles inside BASE.

The external authentication service is only used when a user logs in. Now, one or more of several things can happen:

¹ <http://base.thep.lu.se/ticket/1599>

- The ROOT user is logging on. Internal authentication is always used for the root user and the authenticator plug-in is never used.
- The login is correct and the user is already known to BASE. If the plug-in supports extra information (name, email, phone, etc.) and the `auth.synchronize` setting is `TRUE` the extra information is copied to the BASE server.
- The login is correct, but the user is not known to BASE. This happens the first time a user logs in. BASE will create a new user account. If the driver supports extra information, it is copied to the BASE server (even if `auth.synchronize` is not set). The new user account will get the default quota and be added to the all roles and groups which has been marked as *default*.
- If password caching is enabled, the password is copied to BASE. If an expiration timeout has been set, an expiration date will be calculated and set on the user account. The expiration date is only checked when the external authentication server is down.
- The authentication server says that the login is invalid or the password is incorrect. The user will not be logged in. If a user account with the specified login already exists in BASE, it will be disabled.
- The authentication driver says that something else is wrong. If password caching is enabled, internal authentication will be used. Otherwise the user will not be logged in. An already existing account is not modified or disabled.

The Authenticator interface

To be able to use external authentication you must create a class that implements the `net.sf.based.core.authentication.Authenticator` interface. Specify the name of the class in the `auth.driver` setting in `base.config` and its initialisation parameters in the `auth.init` setting. The class can either be installed on Tomcat's class path (eg. `WEB-INF/lib`) or on an external path. In the latter case the `auth.jarpath` must be set in `base.config`.

Your class must have a public no-argument constructor. The BASE application will create only one instance of the class for lifetime of the BASE server. It must be thread-safe since it may be invoked by multiple threads at the same time. Here are the methods that you must implement

```
public void init (String settings)

throws AuthenticationException;
```

This method is called just after the object has been created with its argument taken from the `auth.init` setting in your `base.config` file. This method is only called once for an instance of the object. The syntax and meaning of the parameter is driver-dependent and should be documented by the plug-in. It is irrelevant for the BASE core.

```
public boolean supportsExtraInformation();
```

This method should simply return `TRUE` or `FALSE` depending on if the plug-in supports extra user information or not. The only required information about a user is a unique ID and the login. Extra information includes name, address, phone, email, etc.

```
public AuthenticationInformation authenticate (String login,

                                             String password)

throws UnknownLoginException, InvalidPasswordException, LoginException, AuthenticationException;
```

Try to authenticate a login/password combination. The plug-in should return an `AuthenticationInformation` object if the authentication is successful or throw an exception if not. There are three exceptions to choose from:

- `UnknownLoginException`: This exception should be thrown if the login is not known to the external authentication system.
- `InvalidPasswordException`: This exception should be thrown if the login is known but the password is invalid. In case it is considered a security issue to reveal that a login exists, the plugin may throw an `UnknownLoginException` or `LoginException` instead.
- `LoginException`: This exception should be thrown if the login failed but it is not known if the cause is an incorrect login or password. The authenticator implementation must specify an error message that is displayed to the user.
- `AuthenticationException`: In case there is another problem, such as the authentication service being down. This exception triggers the use of cached passwords if caching has been enabled.

Configuration settings

The configuration settings for the authentication driver are located in the `base.config` file. Here is an overview of the settings. For more information read the section called “Authentication section” (page 419).

`auth.driver`

The class name of the authentication plug-in.

`auth.jarpath`

The path to the JAR file containing the authentication plug-in. This should be left empty if the plug-in is installed in the `WEB-INF/lib` directory.

`auth.init`

Initialisation parameters sent to the plug-in when calling the `Authenticator.init()` method.

`auth.synchronize`

If extra user information is synchronized at login time or not. This setting is ignored if the driver does not support extra information.

`auth.cachepasswords`

If passwords should be cached by BASE or not. If the passwords are cached a user may login to BASE even if the external authentication server is down.

`auth.daystocache`

How many days to cache the passwords if caching has been enabled. A value of 0 caches the passwords for ever.

25.6.2. Secondary file storage plugins

Primary vs. secondary storage

BASE has support for storing files in two locations, the primary storage and the secondary storage. The primary storage is always disk-based and must be accessible by the BASE server as a path on the file system. The path to the primary storage is configured by the `userfiles` setting in the `base.config` file. The primary storage is internal to the core. Client applications don't get access to read or manipulate the files directly from the file system.

The secondary storage can be anything that can store files. It could, for example, be another directory, a remote FTP server, or a tape based archiving system. A file located in the secondary storage is not accessible by the core, client applications or plug-ins. The secondary storage can only be accessed by the secondary storage controller. The core (and client) applications uses flags on the file items to handle the interaction with the secondary storage.

Each file has an action attribute which default's to `File.Action.NOTHING`. It can take two other values:

1. `File.Action.MOVE_TO_SECONDARY`
2. `File.Action.MOVE_TO_PRIMARY`

All files with the action attribute set to `MOVE_TO_SECONDARY` should be moved to the secondary storage by the controller, and all files with the action attribute set to `MOVE_TO_PRIMARY` should be brought back to primary storage.

The moving of files between primary and secondary storage doesn't happen immediately. It is up to the server administrator to configure how often and at what times the controller should check for files that should be moved. This is configured by the `secondary.storage.interval` and `secondary.storage.time` settings in the `base.config` file.

The SecondaryStorageController interface

All you have to do to create a secondary storage controller is to create a class that implements the `net.sf.basedb.core.SecondaryStorageController` interface. In your `base.config` file you then specify the class name in the `secondary.storage.driver` setting and its initialisation parameters in the `secondary.storage.init` setting.

Your class must have a public no-argument constructor. The BASE application will create only one instance of the class for lifetime of the BASE server. Here are the methods that you must implement:

```
public void init (String settings) ;
```

This method is called just after the object has been created with its argument taken from the `secondary.storage.init` setting in your `base.config` file. This method is only called once for an object.

```
public void run () ;
```

This method is called whenever the core thinks it is time to do some management of the secondary storage. How often the `run()` method is called is controlled by the `secondary.storage.interval` and `secondary.storage.time` settings in the `base.config` file. When this method is called the controller should:

- Move all files which has `action=MOVE_TO_SECONDARY` to the secondary storage. When the file has been moved call `File.setLocation(Location.SECONDARY)` to tell the core that the file is now in the secondary storage. You should also call `File.setAction(File.Action.NOTHING)` to reset the action attribute.
- Restore all files which has `action=MOVE_TO_PRIMARY`. The core will set the location attribute automatically, but you should call `File.setAction(File.Action.NOTHING)` to reset the action attribute.
- Delete all files from the secondary storage that are not present in the database with `location=Location.SECONDARY`. This includes files which has been deleted and files that have been moved offline or re-uploaded.

As a final act the method should send a message to each user owning files that has been moved from one location to the other. The message should include a list of files that has been moved to the secondary storage and a list of files moved from the secondary storage and a list of files that has been deleted due to some of the reasons above.

```
public void close() {}
```

This method is called when the server is closing down. After this the object is never used again.

Configuration settings

The configuration settings for the secondary storage controller is located in the `base.config` file. Here is an overview of the settings. For more information read Appendix B, *base.config reference* (page 418).

`secondary.storage.driver`

The class name of the secondary storage plug-in.

`secondary.storage.init`

Initialisation parameters sent to the plug-in by calling the `init()` method.

`secondary.storage.interval`

Interval in seconds between each execution of the secondary storage controller plug-in.

`secondary.storage.time`

Time points during the day when the secondary storage controller plugin should be executed.

25.6.3. File unpacker plug-ins

The BASE web client has integrated support for unpacking of compressed files. See Section 7.2.1, “Upload a new file” (page 51). Behind the scenes, this support is provided by plug-ins. The standard BASE distribution comes with support for ZIP files (`net.sf.basedb.plugins.ZipFileUnpacker`) and TAR files (`net.sf.basedb.plugins.TarFileUnpacker`).

To add support for additional compressed formats you have to create a plug-in that implements the `net.sf.basedb.util.zip.FileUnpacker` interface. The best way to do this is to extend the `net.sf.basedb.util.zip.AbstractFileUnpacker` which implements all methods in the `Plugin` and `InteractivePlugin` interfaces. This leaves you with the actual unpacking of the files as the only thing to implement.

No support for configurations

The integrated upload in the web interface only works with plug-ins that does not require a configuration to run.

Methods in the FileUnpacker interface

```
public String getFormatName();
```

Return a short string naming the file format. For example: ZIP files or TAR files.

```
public Set<String> getExtensions();
```

Return a set of strings with the file extensions that are most commonly used with the compressed file format. For example: [zip, jar]. Do not include the dot in the extensions. The web client and the `AbstractFlatFileUnpacker.isInContext()` method will use this information to automatically guess which plug-in to use for unpacking the files.

```
public Set<String> getMimeTypes();
```

Return a set of string with the MIME types that commonly used with the compressed file format. For example: [application/zip, application/java-archive]. This information is used by

the `AbstractFlatFileUnpacker.isInContext()` method to automatically guess which plug-in to use for unpacking the files.

```
public int unpack(DbControl dc,
    Directory dir,
    InputStream in,
    File sourceFile,
    boolean overwrite,
    AbsoluteProgressReporter progress)
throws IOException, BaseException;
```

Unpack the files and store them in the BASE file system.

- Do not `close()` or `commit()` the `DbControl` passed to this method. This is done automatically by the `AbstractFileUnpacker` or by the web client.
- The `dir` parameter is the root directory where the unpacked files should be placed. If the compressed file contains subdirectories the plug-in must create those subdirectories unless they already exists.
- If the `overwrite` parameter is `FALSE` no existing file should be overwritten unless the file is `OFFLINE` or marked as removed (do not forget to clear the removed attribute).
- The `in` parameter is the stream containing the compressed data. The stream may come directly from the web upload or from an existing file in the BASE file system.
- The `sourceFile` parameter is the file item representing the compressed file. This item may already be in the database, or a new item that may or may not be saved in the database at the end of the transaction. The information in this parameter can be used to discover the options for file type, character set, MIME type, etc. that was selected by the user in the upload dialog. The `PackUtil` has a useful method that can be used for copying information.
- The `progress` parameter, if not `null`, should be used to report the progress back to the calling code. The plug-in should count the number of bytes read from the `in` stream. If it is not possible by other means the stream can be wrapped by a `net.sf.basedb.util.InputStreamTracker` object which has a `getNumRead()` method.

When the compressed file is uncompressed during the file upload from the web interface, the call sequence to the plug-in is slightly altered from the standard call sequence described in the section called “Executing a job” (page 237).

- After the plug-in instance has been created, the `Plugin.init()` method is called with `null` values for both the configuration and job parameters.
- Then, the `unpack()` method is called. The `Plugin.run()` method is never called in this case.

25.6.4. File packer plug-ins

BASE has support for compressing and downloading a set of selected files and/or directories. This functionality is provided by a plug-in, the `PackedFileExporter`. This plug-in doesn't do the actual packing itself. This is delegated to classes implementing the `net.sf.basedb.util.zip.FilePacker` interface.

BASE ships with a number of packing methods, including ZIP and TAR. To add support for other methods you have to provide an implementation of the `FilePacker` interface. Then, create a new configuration for the `PackedFileExporter` and enter the name of your class in the configuration wizard.

The `FilePacker` interface is not a regular plug-in interface (ie. it is not a subinterface to `Plugin`). This means that you don't have to mess with configuration or job parameters. Another difference

is that your class must be installed in Tomcat's classpath (ie. in one of the `WEB-INF/classes` or `WEB-INF/lib` folders).

This may be converted to an extension point in the future

There are certain plans to convert the packing mechanism to an extension point in the future. The main reason is easier installation since code doesn't have to be installed in the `WEB-INF/lib` or `WEB-INF/classes` directory. See ticket #1600: Convert file packing plug-in system to an extension point² for more information.

Methods in the `FilePacker` interface

```
public String getDescription();
```

Return a short description the file format that is suitable for use in dropdown lists in client applications. For example: `Zip-archive (.zip)` or `TAR-archive (.tar)`.

```
public String getFileExtension();
```

Return the default file extension of the packed format. The returned value should not include the dot. For example: `zip` or `tar`.

```
public String getMimeType();
```

Return the standard MIME type of the packed file format. For example: `application/zip` or `application/x-tar`.

```
public void setOutputStream(OutputStream out)
```

```
throws IOException;
```

Sets the outputstream that the packer should write the packed files to.

```
public void pack(String entryName,
```

```
    InputStream in,
```

```
    long size,
```

```
    long lastModified)
```

```
throws IOException;
```

Add another file or directory to the packed file. The `entryName` is the name of the new entry, including path information. The `in` is the stream to read the file data from. If `in` is `null` then the entry denotes a directory. The `size` parameter gives the size in bytes of the file (zero for empty files or directories). The `lastModified` is that time the file was last modified or 0 if not known.

```
public void close()
```

```
throws IOException;
```

Finish the packing. The packer should release any resources, flush all data and close all output streams, including the `out` stream set in the `setOutputStream` method.

25.6.5. Logging plug-ins

BASE provides a plug-in mechanism for logging changes that are made to items. This plug-in mechanism is not the same as the regular plug-in API. That is, you do not have worry about user interaction or implementing the `Plugin` interface.

This may be converted to an extension point in the future

There are certain plans to convert the logging mechanism to an extension point in the future. There are several benefits with this:

- Easier installation since code doesn't have to be installed in the WEB-INF/lib or WEB-INF/classes directory.
- Allow support for multiple logging modules. Specialized logging modules could for example, be used to log additional information to files.

See ticket #1601: Convert logging plug-in system to an extension point³ for more information.

The logging mechanism works on the data layer level and hooks into callbacks provided by Hibernate. `EntityLogger`s are used to extract relevant information from Hibernate and create log entries. While it is possible to have a generic logger it is usually better to have different implementations depending on the type of entity that was changed. For example, a change in a child item should, for usability reasons, be logged as a change in the parent item. Entity loggers are created by a `LogManagerFactory`. All changes made in a single transaction are usually collected by a `LogManager` which is also created by the factory.

The LogManagerFactory interface

Which `LogManagerFactory` to use is configured in `base.config` (See the section called “Change history logging section” (page 421)). A single factory instance is created when BASE starts and is used for the lifetime of the virtual machine. The factory implementation must of course be thread-safe. Here is a list of the methods the factory must implement:

```
public LogManager getLogManager(LogControl logControl);
```

Creates a log manager for a single transaction. Since a transaction is not thread-safe the log manager implementation doesn't have to be either. The factory has the possibility to create new log managers for each transaction.

```
public boolean isLoggable(Object entity);
```

Checks if changes to the given entity should be logged or not. For performance reasons, it usually makes sense to not log everything. For example, the database logger implementation only logs changes if the entity implements the `LoggableData` interface. The return value of this method should be consistent with `getEntityLogger()`.

```
public EntityLogger getEntityLogger(LogManager logManager,
                                   Object entity);
```

Create or get an entity logger that knows how to log changes to the given entity. If the entity should not be logged, `null` can be returned. This method is called for each modified item in the transaction.

The LogManager interface

A new log manager is created for each transaction. The log manager is responsible for collecting all changes made in the transaction and store those changes in the appropriate place. The interface doesn't define any methods for this collection, since each implementation may have very different needs.

³ <http://base.thep.lu.se/ticket/1599>

```
public LogControl getLogControl();
```

Get the log control object that was supplied by the BASE core when the transaction was started. The log controller contains methods for accessing information about the transaction, such as the logged in user, executing plug-in, etc. It can also be used to execute queries against the database to get even more information.

Warning

Be careful about the queries that are executed by the log controller. Since all logging code is executed at flush time in callbacks from Hibernate we are not allowed to use the regular session. Instead, all queries are sent through the stateless session. The stateless session has no caching functionality which means that Hibernate will use extra queries to load associations. Our recommendation is to avoid queries that return full entities, use scalar queries instead to just load the values that are needed.

```
public void afterCommit();
```

```
,
```

```
public void afterRollback();
```

Called after a successful commit or after a rollback. Note that the connection to the database has been closed at this time and it is not possible to save any more information to it at this time.

The EntityLogger interface

An entity logger is responsible for extracting the changes made to an entity and converting it to something that is useful as a log entry. In most cases, this is not very complicated, but in some cases, a change in one entity should actually be logged as a change in a different entity. For example, changes to annotations are handled by the `AnnotationLogger` which which log it as a change on the parent item.

```
public void logChanges(LogManager logManager,
    EntityDetails details);
```

This method is called whenever a change has been detected in an entity. The `details` variable contains information about the entity and, to a certain degree, what changes that has been made.

25.7. How BASE load plug-in classes

All plug-ins should be installed in the location specified by the `plugins.dir` setting in `base.config`. While it is possible to also install them in a location that is on the classpath, for example `<base-dir>/www/WEB-INF/lib`, it is nothing that we recommend. The rest of the information in this section only applies to plug-ins that have been installed in the `plugins.dir` location.

If the above recommendation has been followed BASE will use it's own classloader to load the plug-in classes. This have several benefits:

- New plug-ins can be installed and existing plug-ins can be updated without restarting the web server. If the `plugins.autounload` setting in `base.config` has been enabled all you have to do to update a plug-in is to replace the JAR file with a new version. BASE will automatically load the new classes the next time the plug-in is used. If the option isn't enabled, the server admin has to manually update the plug-in from the web interface first.

- Plug-ins may use it's own 3-rd party libraries without interfering with other plug-ins. This may be important because a plug-in may depend on a certain version of a library while another plug-in may depend on a different version. Since BASE is using different class-loaders for different plug-ins this is not a problem.

The classloading scheme used by BASE also means plug-in developers must pay attention to a few things:

- A plug-in can only access/use classes from it's own JAR file, BASE core classes, Java system classes and from JAR files listed in the plug-in's `MANIFEST.MF` file. See Section 25.1, “How to organize your plug-in project” (page 222).
- A plug-in can also access other plug-ins, but only via the methods and interfaces defined in BASE. In the following example we assume that there are two plug-ins, `ex.MyPlugin` and `ex.MyOtherPlugin`, located in two different JAR files. The code below is executing in the `ex.MyPlugin`:

```
// Prepare to load MyOtherPlugin
SessionControl sc = ...
DbControl dc = ...
PluginDefinition def = PluginDefinition.getByClassName(dc, "ex.MyOtherPlugin");

// Ok
Plugin other = def.newInstance();

// Not ok; fails with ClassCastException
MyOtherPlugin other = (MyOtherPlugin)def.newInstance();

// Ok; since now we are using the correct class loader
MyOtherPlugin other = def.newInstance(MyOtherPlugin.class);
```

The first call succeeds because it uses the `Plugin` interface which is defined by BASE. This class is loaded by the web servers class loader and is the same for all plug-ins.

The second call fails because BASE uses a different classloader to load the `ex.MyOtherPlugin` class. This class is not (in Java terms) the same as the `ex.MyOtherPlugin` class loaded by the classloader that loaded the `ex.MyPlugin` class. If, on the other hand, both plug-ins are located in the same JAR file BASE uses the same classloader and the second call will succeed.

The third call succeeds because now that we specify the class as an argument, BASE uses that classloader instead.

Another option is that the `ex.MyPlugin` lists the JAR file where `ex.MyOtherPlugin` is located in it's `MANIFEST.MF` file. Then, the following code can be used: `MyOtherPlugin other = new MyOtherPlugin();`

Tomcat includes a good document describing how classloading is implemented in Tomcat: <http://tomcat.apache.org/tomcat-6.0-doc/class-loader-howto.html>. BASE's classloading scheme isn't as complex as Tomcat's, but it very similar to how Tomcat loads different web applications. The figure on the linked document could be extended with another level with separate classloaders for each plug-in as child classloaders to the web application classloaders.

As of BASE 2.13 the default search order for classes has been changed. The default is now to first look in the plug-ins class path (eg. in the same JAR file and in files listed in the `MANIFEST.MF` file). Only if the class is not found the search is delegated to the parent class loader. This behaviour can be changed by setting `X-Delegate-First: true` in the `MANIFEST.MF` file. If this property is set the parent class loader is search first. This is the same as in BASE 2.12 and earlier.

Note

The benefit with the new search order is that plug-ins may use a specific version of any external package even if the same package is part of the BASE distribution. This was not possible before since the package in the BASE distribution was loaded first.

25.8. Example plug-ins (with download)

We have created some example plug-ins which demonstrates how to use the plug-in system and how to create an interactive plug-in that can ask a user for one or more parameters. You can download a tar file with the source and compiled plug-in code from the BASE plug-ins website: <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.examplecode>

Chapter 26. Extensions developer

26.1. Overview

The BASE application includes an extensions mechanism that makes it possible to dynamically add functions to the GUI without having to edit the JSP code. It is, for example, possible to add menu items in the menu and toolbar buttons in selected toolbars.

Go to the Administrative Plug-ins & extensions Overview menu to display a list of possible extension points and all installed extensions. From this page, if you are logged in with enough permissions, it is also possible to configure the extensions system, enable/disable extensions, etc. Read more about this in Chapter 21, *Plug-ins and extensions* (page 178).

Extensions can come in two forms, either as an XML file in the *BASE Extensions XML* format or as a JAR file. A JAR file is required when the extension needs to execute custom-made code or use custom resources such as icons, css stylesheets, or JSP files.

More reading

- Chapter 21, *Plug-ins and extensions* (page 178).
- Section 28.6, “The Extensions API” (page 338).

26.1.1. Download code examples

The code examples in this chapter can be downloaded from the BASE plug-ins site: <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.examplecode>

26.1.2. Terminology

Extension point

An extensions point is a place in the BASE core or web client interface where it is possible to extend the functionality with custom extensions. An extension point has an ID which must be unique among all existing extension points. Extension points registered by the BASE web client all starts with `net.sf.basedb.clients.web` prefix. The extension point also defines an `Action` subclass that all extensions must implement.

Extension

An extensions is a custom addition to the BASE web client interface or core API. This can mean a new menu item in the menu or a new button in a toolbar. An extension must provide an `ActionFactory` that knows how to create actions that fits the requirements from the extension point the extension is extending.

Action

An `Action` is an interface definition which provides an extension point enough information to make it possible to render the action as HTML. An action typically has methods such as, `getTitle()`, `getIcon()` and `getOnClick()`.

Action factory

An `ActionFactory` is an object that knows how to create actions of some specific type, for example menu item actions. Action factories are part of an extension definition and can usually be configured with parameters from the XML file. BASE ships with several implementations of action factories for all defined extension points. Still, if your extension needs a different implementation you can easily create your own factory.

Renderer

A `Renderer` is an object that knows how to convert the information in an action to HTML. The use of renderers are optional. Some extension points use a "hard-coded" approach that renders

the actions directly on the JSP file. Some extension points uses a locked renderer, while other extension points provides a default renderer, but allows extensions to supply their own if they want to. Renderers are mostly used by the web client extensions, not so much by the core extensions.

Renderer factory

A `RendererFactory` is an object that knows how to create renderers. Renderer factories can be part of both extension points and extensions. In most other aspects renderer factories are very similar to action factories.

Error handler factory

An `ErrorHandlerFactory` is an object that knows how to handle error that occur when executing extensions. An error handler factory is defined as part of an extension point and handles all errors related to the extensions of that extension point. In most other aspects error handler factories are similar to renderer and action factories. If the extension point doesn't define an error handler factory, the system will select a default that only writes a message to the log file `LoggingErrorHandlerFactory`.

Client context

A `ClientContext` is an object which contains information about the current user session. It is for example, possible to get information about the logged in user, the currently active item, etc.

In the BASE web client the context is always a `JspContext`. Wherever a `ClientContext` object is provided as a parameter it is always safe to cast it to a `JspContext` object. Extension points in the core usually use a `ClientContext`.

The context can also be used by an extension to request that a specific javascript or stylesheet file should be included in the HTML code.

26.2. Hello world as an extension

We will use the classical Hello world as the first simple example of an extension. This extension will add a new menu item in the menu which displays a popup with the text "Hello world!" when selected. Copy the XML code below and save it to a file in the `plugins.dir` directory. The filename must end with `.xml`. Install the extension by going through the installation wizard at [Administrative Plug-ins & extensions Overview](#).

When the extension has been installed you should have a new menu item: **Extensions Hello world!** which pops up a message in a Javascript window.

Note

You may have to logout and login again to see the new menu item.

```
<?xml version="1.0" encoding="UTF-8" ?>
<extensions xmlns="http://base.thep.lu.se/extensions.xsd">
  <extension
    id="net.sf.basedb.clients.web.menu.extensions.helloworld"
    extends="net.sf.basedb.clients.web.menu.extensions"
  >
    <index>1</index>
    <about>
      <name>Hello world</name>
      <description>
        The very first extensions example. Adds a "Hello world"
        menu item that displays "Hello world" in a javascript
        popup when selected.
      </description>
    </about>
    <action-factory>
      <factory-class>
```

```
        net.sf.basedb.clients.web.extensions.menu.FixedMenuItemFactory
    </factory-class>
    <parameters>
        <title>Hello world!</title>
        <tooltip>This is to test the extensions system</tooltip>
        <onClick>alert('Hello world!')</onClick>
        <icon>/images/info.gif</icon>
    </parameters>
</action-factory>
</extension>
</extensions>
```

The `<extensions>` tag is the root tag and is needed to set up the namespace and schema validation.

The `<extension>` defines a new extension. It must have an `id` attribute that is unique among all installed extensions and an `extends` attribute which id the ID of the extension point. For the `id` attribute we recommend using the same naming conventions as for java packages. See Java naming conventions from Oracle¹.

The `<about>` tag is optional and can be used to provide meta information about the extension. We recommend that all extensions are given at least a `<name>`. Other supported subtags are:

- `<description>`
- `<version>`
- `<copyright>`
- `<contact>`
- `<email>`
- `<url>`

Global about tag

`<about>` tag can also be specified as a first-level tag (eq. as a child to `<extensions>`). This can be useful when an XML file defines more than one extension and you don't want to repeat the same information for every extension. You can still override the information for specific extensions by including new values in the extension's `<about>` tag.

The `<action-factory>` tag is required and so is the `<factory-class>` subtag. It tells the extension system which factory to use for creating actions. The `FixedMenuItemFactory` is a very simple factory that is shipped with BASE. This factory always creates the same menu item, no matter what. Another factory for menu items is the `PermissionMenuItemFactory` which can create menu items that depends on the logged in user's permissions. It is for example, possible to hide or disable the menu item if the user doesn't have enough permissions. If none of the supplied factories suits you it is possible to supply your own implementation. More about this later.

The `<parameters>` subtag is used to provide initialisation parameters to the factory. Different factories supports different parameters and you will have to check the javadoc documentation for each factory to get information about which parameters that are supported.

Tip

In case the factory is poorly documented you can always assume that public methods the start with `set` and take a single `String` as an argument can be used as a parameter. The parameter tag to use should be the same as the method name, minus the `set` prefix and with the first letter in lowercase. For example, the method `setIcon(String icon)` corresponds to the `<icon>` parameter.

¹ <http://www.oracle.com/technetwork/java/codeconventions-135099.html>

26.2.1. Extending multiple extension points with a single extension

A single extension can extend multiple extension points as long as their action classes are compatible. This is for, for example, the case when you want to add a button to more than one toolbar. To do this use the `<extends>` tag with multiple `<ref>` tags. You can skip the `extends` attribute in the main tag.

```
<extension
  id="net.sf.basedb.clients.web.menu.extensions.history-edit"
  >
  <extends>
    <ref index="2">net.sf.basedb.clients.web.tabcontrol.edit.sample</ref>
    <ref index="2">net.sf.basedb.clients.web.tabcontrol.edit.extract</ref>
  </extends>
  ...
</extension>
```

This is a feature of the XML format only. Behind the scenes two extensions will be created (one for each extension point). The extensions will share the same action and renderer factory instances. Since the id for an extension must be unique a new id will be generated by combining the original id with the parts of the id's from the extension points.

26.3. Custom action factories

Some times the factories shipped with BASE are not enough, and you may want to provide your own factory implementation. In this case you will have to create a class that implements the `ActionFactory` interface. Here is a very simple example that does the same as the previous "Hello world" example.

```
package net.sf.basedb.examples.extensions;

import net.sf.basedb.clients.web.extensions.JspContext;
import net.sf.basedb.clients.web.extensions.menu.MenuItemAction;
import net.sf.basedb.clients.web.extensions.menu.MenuItemBean;
import net.sf.basedb.util.extensions.ActionFactory;
import net.sf.basedb.util.extensions.InvokationContext;

/**
 * First example of an action factory where everything is hardcoded.
 * @author nicklas
 */
public class HelloWorldFactory
  implements ActionFactory<MenuItemAction>
{
  private MenuItemAction[] helloWorld;

  // A public, no-argument constructor is required
  public HelloWorldFactory()
  {
    helloWorld = new MenuItemAction[1];
  }

  // Return true enable the extension, false to disable it
  public boolean prepareContext(
    InvokationContext<? super MenuItemAction> context)
  {
    return true;
  }
}
```

```
// An extension may create one or more actions
public MenuItemAction[] getActions(
    InvokationContext<? super MenuItemAction> context)
{
    // This cast is always safe with the web client
    JspContext jspContext = (JspContext)context.getClientContext();
    if (helloWorld[0] == null)
    {
        MenuItemBean bean = new MenuItemBean();
        bean.setTitle("Hello factory world!");
        bean.setIcon(jspContext.getRoot() + "/images/info.gif");
        bean.setOnClick("alert('Hello factory world!')");
        helloWorld[0] = bean;
    }
    return helloWorld;
}
```

And here is the XML configuration file that goes with it.

```
<?xml version="1.0" encoding="UTF-8" ?>
<extensions xmlns="http://base.thep.lu.se/extensions.xsd">
    <extension
        id="net.sf.basedb.clients.web.menu.extensions.helloworldfactory"
        extends="net.sf.basedb.clients.web.menu.extensions"
        >
        <index>2</index>
        <about>
            <name>Hello factory world</name>
            <description>
                A "Hello world" variant with a custom action factory.
                Everything is hard-coded into the factory.
            </description>
        </about>
        <action-factory>
            <factory-class>
                net.sf.basedb.examples.extensions.HelloWorldFactory
            </factory-class>
        </action-factory>
    </extension>
</extensions>
```

To install this extension you need to put the compiled `HelloWorldFactory.class` and the XML file inside a JAR file. The XML file must be located at `META-INF/extensions.xml` and the class file at `net/sf/basedb/examples/extensions/HelloWorldFactory.class`.

The above example is a bit artificial and we have not gained anything. Instead, we have lost the ability to easily change the menu since everything is now hardcoded into the factory. To change, for example the title, requires that we recompile the java code. It would be more useful if we could make the factory configurable with parameters. The next example will make the icon and message configurable, and also include the name of the currently logged in user. For example: "Greetings <name of logged in user>!".

```
package net.sf.basedb.examples.extensions;

import net.sf.basedb.clients.web.extensions.AbstractJspActionFactory;
import net.sf.basedb.clients.web.extensions.menu.MenuItemAction;
import net.sf.basedb.clients.web.extensions.menu.MenuItemBean;
import net.sf.basedb.core.DbControl;
import net.sf.basedb.core.SessionControl;
import net.sf.basedb.core.User;
import net.sf.basedb.util.extensions.ClientContext;
import net.sf.basedb.util.extensions.InvokationContext;
import net.sf.basedb.util.extensions.xml.PathSetter;
```

```
import net.sf.basedb.util.extensions.xml.VariableSetter;

/**
 * Example menu item factory that creates a "Hello world" menu item
 * where the "Hello" part can be changed by the "prefix" setting in the
 * XML file, and the "world" part is dynamically replaced with the name
 * of the logged in user.
 *
 * @author nicklas
 */
public class HelloUserFactory
    extends AbstractJspActionFactory<MenuItemAction>
{
    // To store the URL to the icon
    private String icon;

    // The default prefix is Hello
    private String prefix = "Hello";

    // A public, no-argument constructor is required
    public HelloUserFactory()
    {}

    /**
     * Creates a menu item that displays: {prefix} {name of user}!
     */
    public MenuItemAction[] getActions(
        InvokationContext<? super MenuItemAction> context)
    {
        String userName = getUserName(context.getClientContext());
        MenuItemBean helloUser = new MenuItemBean();
        helloUser.setTitle(prefix + " " + userName + "!");
        helloUser.setIcon(icon);
        helloUser.setOnClick("alert('" + prefix + " " + userName + "!'");
        return new MenuItemAction[] { helloUser };
    }

    /**
     * Get the name of the logged in user.
     */
    private String getUserName(ClientContext context)
    {
        SessionControl sc = context.getSessionControl();
        DbControl dc = context.getDbControl();
        User current = User.getById(dc, sc.getLoggedInUserId());
        return current.getName();
    }

    /**
     * Sets the icon to use. Path conversion is enabled.
     */
    @VariableSetter
    @PathSetter
    public void setIcon(String icon)
    {
        this.icon = icon;
    }

    /**
     * Sets the prefix to use. If not set, the
     * default value is "Hello".
     */
    public void setPrefix(String prefix)
    {
        this.prefix = prefix == null ? "Hello" : prefix;
    }
}
```

There are two new parts in this factory. The first is the `getUserName()` method which is called from `getActions()`. Note that the `getActions()` method always creates a new `MenuItemBean`. It can no longer be cached since the title and javascript code depends on which user is logged in.

The second new part is the `setIcon()` and `setPrefix()` methods. The extensions system uses java reflection to find the existence of the methods if `<icon>` and/or `<prefix>` tags are present in the `<parameters>` tag for a factory, the methods are automatically called with the value inside the tag as its argument.

The `VariableSetter` and `PathSetter` annotations on the `setIcon()` are used to enable "smart" conversions of the value. Note that in the XML file you only have to specify `/images/info.gif` as the URL to the icon, but in the hardcoded factory you have to do: `jspContext.getRoot() + "/images/info.gif"`. In this case, it is the `PathSetter` which automatically adds the JSP root directory to all URLs starting with `/`. The `VariableSetter` can do the same thing but you would have to use `$ROOT$` instead. Eg. `$ROOT$/images/info.gif`. The `PathSetter` only looks at the first character, while the `VariableSetter` looks in the entire string.

Here is an example of an extension configuration that can be used with the new factory.

```
<extensions xmlns="http://base.thep.lu.se/extensions.xsd">
  <extension
    id="net.sf.basedb.clients.web.menu.extensions.hellouser"
    extends="net.sf.basedb.clients.web.menu.extensions"
  >
    <index>3</index>
    <about>
      <name>Greetings user</name>
      <description>
        A "Hello world" variant with a custom action factory
        that displays "Greetings {name of user}" instead. We also
        make the icon configurable.
      </description>
    </about>
    <action-factory>
      <factory-class>
        net.sf.basedb.examples.extensions.HelloUserFactory
      </factory-class>
      <parameters>
        <prefix>Greetings</prefix>
        <icon>/images/take_ownership.png</icon>
      </parameters>
    </action-factory>
  </extension>
</extensions>
```

Be aware of multi-threading issues

When you are creating custom action and renderer factories be aware that multiple threads may use a single factory instance at the same time. Action and renderer objects only need to be thread-safe if the factories re-use the same objects.

26.4. Custom images, JSP files, and other resources

Some times your extension may need other resources. It can for example be an icon, a javascript file, a JSP file or something else. Fortunately this is very easy. You need to put the extension in a JAR file. As usual the extension definition XML file should be at `META-INF/extensions.xml`. Everything you put in the JAR file inside the `resources/` directory will automatically be extracted by the extension system to a directory on the web server. Here is another "Hello world" example which uses a custom JSP file to display the message. There is also a custom icon.


```
<extensions xmlns="http://base.thep.lu.se/extensions.xsd">
  <extension
    id="net.sf.basedb.clients.web.menu.extensions.hellojspworld"
    extends="net.sf.basedb.clients.web.menu.extensions"
  >
    <index>4</index>
    <about>
      <name>Hello JSP world</name>
      <description>
        This example uses a custom JSP file to display the
        "Hello world" message instead of a javascript popup.
      </description>
    </about>
    <action-factory>
      <factory-class>
        net.sf.basedb.clients.web.extensions.menu.FixedMenuItemFactory
      </factory-class>
      <parameters>
        <title>Hello JSP world!</title>
        <tooltip>Opens a JSP file with the message</tooltip>
        <onClick>
          Main.openPopup('$HOME$/hello_world.jsp?ID=' + getSessionId(), 'HelloJspWorld', 400,
300)
        </onClick>
        <icon>~/images/world.png</icon>
      </parameters>
    </action-factory>
  </extension>
</extensions>
```

The JAR file should have have the following contents:

```
META-INF/extensions.XML
resources/hello_world.jsp
resources/images/world.png
```

When this extension is installed the `hello_world.jsp` and `world.png` files are automatically extracted to the web servers file system. Each extension is given a unique `HOME` directory to make sure that extensions doesn't interfere with each other. The URL to the home directory is made available in the `$HOME$` variable. All factory settings that have been annotated with the `VariableSetter` will have their values scanned for `$HOME$` which is replaced with the real URL. It is also possible to use the `$ROOT$` variable to get the root URL for the BASE web application. Never use `/base/...` since users may install BASE into another path.

The tilde (`~`) in the `<icon>` tag value is also replaced with the `HOME` path. Note that this kind of replacement is only done on factory settings that have been annotated with the `PathSetter` annotation and is only done on the first character.

Note

Unfortunately, the custom JSP file can't use classes that are located in the extension's JAR file. The reason is that the JAR file is not known to Tomcat and Tomcat will not look in the `plugins.dir` folder to try to find classes. There are currently two possible workarounds:

- Place classes needed by JSP files in a separate JAR file that is installed into the `WEB-INF/lib` folder. The drawback is that this requires a restart of Tomcat.
- Use an X-JSP file instead. This is an experimental feature. See Section 26.4.2, "X-JSP files" (page 272) for more information.

26.4.1. Javascript and stylesheets

It is possible for an extension to use a custom javascript or stylesheet. However, this doesn't happen automatically and may not be enabled for all extension points. If an extension needs

this functionality the action factory or renderer factory must call `JspContext.addScript()` or `JspContext.addStylesheet()` from the `prepareContext()` method.

The `AbstractJspActionFactory` and `AbstractJspRendererFactory` factory can do this. All factories shipped with BASE extends one of those classes and we recommend that custom-made factories also does this.

Factories that are extending one of those two classes can use `<script>` and `<stylesheet>` tags in the `<parameters>` section for an extensions. Each tag may be used more than one time. The values are subject to path and variable substitution.

```
<action-factory>
  <factory-class>
    ... some factory class ...
  </factory-class>
  <parameters>
    <script>~/scripts/custom.js</script>
    <stylesheet>~/css/custom.css</stylesheet>
    ... other parameters ...
  </parameters>
</action-factory>
```

If scripts and stylesheets has been added to the JSP context the extension system will, *in most cases*, include the proper HTML to link in the requested scripts and/or stylesheet.

Use UTF-8 character encoding

The script and stylesheet files should use use UTF-8 character encoding. Otherwise they may not work as expected in BASE.

All extension points doesn't support custom scripts/stylesheets

In some cases the rendering of the HTML page has gone to far to make is possible to include custom scripts and stylesheets. This is for example the case with the extensions menu. Always check the documentation for the extension point if scripts and stylesheets are supported or not.

26.4.2. X-JSP files

The drawback with a custom JSP file is that it is not possible to use classes from the extension's JAR file in the JSP code. The reason is that the JAR file is not known to Tomcat and Tomcat will not look in the `plugins.dir` folder to try to find classes.

One workaround is to place classes that are needed by the JSP files in a separate JAR file that is placed in `WEB-INF/lib`. The drawback with this is that it requires a restart of Tomcat. It is also a second step that has to be performed manually by the person installing the extension and is maybe forgotten when doing an update.

Another workaround is to use an X-JSP file. This is simply a regular JSP file that has a `.xjsp` extension instead of `.jsp`. The `.xjsp` extension will trigger the use of a different compiler that knows how to include the extension's JAR file in the class path.

X-JSP is experimental

The X-JSP compiler depends on functionality that is internal to Tomcat. The JSP compiler is not part of any open specification and the implementation details may change at any time. This means that the X-JSP compiler may or may not work with future versions of Tomcat. We have currently tested it with Tomcat 6.0.14 only. It will most likely not work with other servlet containers.

Adding support for X-JSP requires that a JAR file with the X-JSP compiler is installed into Tomcat's internal `/lib` directory. It is an optional step and not all BASE installations may have the compiler installed. See Section 21.1.4, “Installing the X-JSP compiler” (page 183).

26.5. Custom renderers and renderer factories

It is always the responsibility of the extension point to render an action. The need for custom renderers is typically very small, at least if you want your extensions to blend into the look and feel of the BASE web client. Most customizations can be probably be handled by stylesheets and images. That said, you may still have a reason for using a custom renderer.

Renderer factories are not very different from action factories. They are specified in the same way in the XML file and uses the same method for initialisation, including support for path conversion, etc. The difference is that you use a `<renderer-factory>` tag instead of an `<action-factory>` tag.

```
<renderer-factory>
  <factory-class>
    ... some factory class ...
  </factory-class>
  <parameters>
    ... some parameters ...
  </parameters>
</renderer-factory>
```

A `RendererFactory` also has a `prepareContext()` method that can be used to tell the web client about any scripts or stylesheets the extension needs. If your renderer factory extends the `AbstractJspRendererFactory` class you will not have to worry about this since you can configure scripts and stylesheets in the XML file.

A render factory must also implement the `getRenderer()` which should return a `Renderer` instance. The extension system will then call the `Renderer.render()` method to render an action. This method may be called multiple times if the extension created more than one action.

The renderers responsibility is to generate the HTML that is going to be sent to the web client. To do this it needs access to the `JspContext` object that was passed to the renderer factory. Here is a simple outline of both a renderer factory and renderer.

```
// File: MyRendererFactory.java
public class MyRendererFactory
    extends AbstractJspRendererFactory<MyAction>
{
    public MyRendererFactory()
    {}

    @Override
    public MyRenderer getRenderer(InvocationContext context)
    {
        return new MyRenderer((JspContext)context.getClientContext());
    }
}

// File: MyRenderer.java
public class MyRenderer
    implements Renderer<MyAction>
{
```

```
private final JspContext context;
public MyRenderer(JspContext context)
{
    this.context = context;
}

/**
 * Generates HTML (unless invisible):
 * <a class="[clazz]" style="[style]" onclick="[onClick]">[title]</a>
 */
public void render(MyAction action)
{
    if (!action.isVisible()) return;
    Writer out = context.getOut();
    try
    {
        out.write("<a");
        if (action.getClazz() != null)
        {
            out.write(" class=\"" + action.getClazz() + "\"");
        }
        if (action.getStyle() != null)
        {
            out.write(" style=\"" + action.getStyle() + "\"");
        }
        if (action.getOnClick() != null)
        {
            out.write(" href=\"" + action.getOnClick() + "\"");
        }
        out.write(">");
        out.write(HTML.encodeTags(action.getTitle()));
        out.write("</a>\n");
    }
    catch (IOException ex)
    {
        throw new RuntimeException(ex);
    }
}
```

26.6. Extension points

The BASE web client ships with a number of predefined extension points. Adding more extension points to the existing web client requires some minor modifications to the regular JSP files. But this is not what this chapter is about. This chapter is about defining new extension points as part of an extension. It is nothing magical about this and the process is the same as for the regular extension points in the web client.

The first thing you need to do is to start writing the XML definition of the extension point. Here is an example from the web client:

```
<extensions
  xmlns="http://base.thep.lu.se/extensions.xsd"
>
  <extension-point
    id="net.sf.basedb.clients.web.menu.extensions"
    >
    <action-class>net.sf.basedb.clients.web.extensions.menu.MenuItemAction</action-class>
    <name>Menu: extensions</name>
    <description>
      Extension point for adding extensions to the 'Extensions' menu.
      Extensions should provide MenuItemAction instances. The rendering
      is internal and extensions can't use their own rendering factories.
      The context will only include information about the currently logged
      in user, not information about the current page that is displayed.
      The reason for this is that the rendered menu is cached as a string
    </description>
  </extension-point>
</extensions>
```

```
        in the user session. The menu is not updated on every page request.  
        This extension point doesn't support custom stylesheets or javascripts.  
    </description>  
</extension-point>  
</extensions>
```

The `<extensions>` tag is the root tag and is needed to set up the namespace and schema validation.

The `<extension-point>` defines a new extension point. It must have an `id` attribute that is unique among all installed extension points. We recommend using the same naming conventions as for java packages. See Java naming conventions from Oracle².

Document the extension point!

The `<name>` and `<description>` tags are optional, but we strongly recommend that values are provided. The description tag should be used to document the extension point. Pay special attention to the support (or lack of support) for custom scripts, stylesheets and renderers.

The `<action-class>` defines the interface or class that extensions must provide to the extension point. This must be a class or interface that is a subclass of the `Action` interface. We generally recommend that interfaces are used since this gives more implementation flexibility for action factories, but a regular class may work just as well.

The action class is used to carry information about the action, such as a title, which icon to use, a tooltip text, a javascript snippet that is invoked on click events, etc. The action class may be as simple or complex as you like.

Web client extension points

This is a note for the core developers. Extension points that are part of the web client should always define the action as an interface. We recommend that `getId()`, `getClazz()` and `getStyle()` attributes are always included if this makes sense. It is usually also a good idea to include `isVisible()` and `isEnabled()` attributes.

Now, if you are a good citizen you should also provide at least one implementation of an action factory that can create the objects of the desired type of action. The factory should of course be configurable from the XML file.

If you are lazy or if you want to immediately start testing the JSP code for the extension point, it may be possible to use one of the debugger action factories in the `net.sf.basedb.util.extensions.debug` package.

- **ProxyActionFactory:** This action factory can only be used if your action class is an interface and all important methods starts with `get` or `is`. The proxy action factory uses Java reflection to create a dynamic proxy class in runtime. It will map all `getX()` and `isY()` methods to retrieve the values from the corresponding parameter in the XML file. For example, `getIcon()` will retrieve the value of the `<icon>` tag and `isVisible()` from the `<visible>`. The factory is smart enough to convert the string to the correct return value for `int`, `long`, `float`, `double` and `boolean` data types and their corresponding object wrapper types, if this is needed.
- **BeanActionFactory:** This action factory can be used if you have created a bean-like class that implements the desired action class. The factory will create an instance of the class specified by the `<beanClass>` parameter. The factory will then use Java reflection to find `set` method for the other parameters. If there is a parameter `<icon>` the factory first looks for a `setIcon(String)` method. If it can't find that it will see if there is a `getIcon()` method which has a return type, `T`. If so, a second attempt is made to find a `setIcon(T)` method. The factory is smart enough to convert the string value from the XML file to the correct return value for `int`, `long`, `float`, `double` and `boolean` data types and their corresponding object wrapper types, if this is needed.

² <http://www.oracle.com/technetwork/java/codeconventions-135099.html>

It is finally time to write the JSP code that actually uses the extension point. It is usually not very complicated. Here is an example which lists snippets from a JSP file:

```
// 1. We recommend using the extensions taglib (and the BASE core taglib)
<%@ taglib prefix="ext" uri="/WEB-INF/extensions.tld" %>
<%@ taglib prefix="base" uri="/WEB-INF/base.tld" %>

// 2. Prepare the extension point
SessionControl sc = Base.getExistingSessionControl(pageContext, true);
JspContext jspContext = ExtensionsControl.createContext(sc, pageContext);
ExtensionsInvoker invoker = ExtensionsControl.useExtensions(jspContext,
    "my.domain.name.extensionspoint");

// 3. Output scripts and stylesheets
<base:page title="My new extension point">
    <base:head>
        <ext:scripts context="<%=jspContext%>" />
        <ext:stylesheets context="<%=jspContext%>" />
    </base:head>
    <base:body>
        ....

// 4a. Using a taglib for rendering with the default renderer
<ext:render extensions="<%=invoker%>" context="<%=jspContext%>" />

// 4b. Or, use the iterator and a more hard-coded approach
<%
Iterator it = invoker.iterate();
while (it.hasNext())
{
    MyAction action = (MyAction)it.next();
    String html = action.getTitle() +
        ....
    out.write(html);
}
%>
```

26.6.1. Error handlers

An extension points may define a custom error handler. If not, the default error handler is used which simply writes a message to the log file. If you want to use a different error handler, create a `<error-handler-factory>` tag inside the extension point definition. The `<factory-class>` is a required subtag and must specify a class with a public no-argument constructor that implements the `ErrorHandlerFactory` interface. The `<parameters>` subtag is optional and can be used to specify initialization parameters for the factory just as for action and renderer factories.

```
<extensions
  xmlns="http://base.thep.lu.se/extensions.xsd"
>
  <extension-point
    id="net.sf.basedb.clients.web.menu.extensions"
  >
    <action-class>net.sf.basedb.clients.web.extensions.menu.MenuItemAction</action-class>
    <name>Menu: extensions</name>
    <error-handler-factory>
      <factory-class>
        ... some factory class ...
      </factory-class>
      <parameters>
        ... initialization parameters ...
      </parameters>
    </error-handler-factory>
  </extension-point>
</extensions>
```

26.7. Custom servlets

It is possible for an extension to include servlets without having to register those servlets in Tomcat's `WEB-INF/web.xml` file. The extension needs to be in a JAR file as usual. The servlet class should be located in the JAR file following regular Java conventions. Eg. The class `my.domain.ServletClass` should be located at `my/domain/ServletClass.class`. You also need to create a second XML file that contains the servlet definitions at `META-INF/servlets.xml`. The format for defining servlets in this file is very similar to how servlets are defined in the `web.xml` file. Here is an example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<servlets xmlns="http://base.thep.lu.se/servlets.xsd">
  <servlet>
    <servlet-name>HelloWorld</servlet-name>
    <servlet-class>net.sf.basedb.examples.extensions.HelloWorldServlet</servlet-class>
    <init-param>
      <param-name>template</param-name>
      <param-value>Hello {user}! Welcome to the Servlet world!</param-value>
    </init-param>
  </servlet>
</servlets>
```

The `<servlets>` tag is the root tag and is needed to set up the namespace and schema validation. This may contain any number of `<servlet>` tags, each one defining a single servlet.

The `<servlet-name>` tag contains the name of the servlet. This is a required tag and must be unique among the servlets defined by this extension. Other extensions may use the same name without any problems.

The `<servlet-class>` tag contains the name of implementing class. This is required and the class must implement the `Servlet` interface and have a public, no-argument constructor. We recommend that servlet implementations instead extends the `HttpServlet` class. This will make the servlet programming easier.

A servlet may have any number `<init-param>` tags, containing initialisation parameters for the servlet. Here is the code for the servlet references in the above example.

```
public class HelloWorldServlet
    extends HttpServlet
{
    private String template;
    public HelloWorldServlet()
    {}

    @Override
    public void init()
        throws ServletException
    {
        ServletConfig cfg = getServletConfig();
        template = cfg.getInitParameter("template");
        if (template == null) template = "Hello {user}.";
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        final SessionControl sc = Base.getExistingSessionControl(request, true);
        final DbControl dc = sc.newDbControl();
        try
```

```
{
    User current = User.getById(dc, sc.getLoggedInUserId());
    PrintWriter out = response.getWriter();
    out.print(template.replace("{user}", current.getName()));
}
finally
{
    if (dc != null) dc.close();
}
}
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
{
    doGet(req, resp);
}
}
```

Invoking the servlet is done with a URL that is constructed like: `$HOME$/[servlet-name].servlet`, where `$HOME$` is the home directory of the extension. An alternate URL that doesn't require the `.servlet` extension is available: `$SERVLET_HOME$/[servlet-name]`, where `$SERVLET_HOME$` is the home directory of servlets for the extension. Note that this directory is on a different sub-path than the `$HOME$` directory.

Extra path information is supported if it is inserted between the servlet name and the `.servlet` extension: `$HOME$/[servlet-name]/[extra/path/info].servlet`, `$SERVLET_HOME$/[servlet-name]/[extra/path/info]`

Query parameters are supported as normal: `$HOME$/[servlet-name].servlet?param1=value¶m2=value`, `$SERVLET_HOME$/[servlet-name]?param1=value¶m2=value`

```
<extension
  id="net.sf.basedb.clients.web.menu.extensions.helloervletworld"
  extends="net.sf.basedb.clients.web.menu.extensions"
>
<index>5</index>
<about>
  <name>Hello Servlet world</name>
  <description>
    This example uses a custom Servlet page to display the
    "Hello world" message instead of a javascript popup.
  </description>
</about>
<action-factory>
  <factory-class>
    net.sf.basedb.clients.web.extensions.menu.FixedMenuItemFactory
  </factory-class>
  <parameters>
    <title>Hello Servlet world!</title>
    <tooltip>Opens a Servlet generated page with the message</tooltip>
    <onClick>
      Main.openPopup('$HOME$/HelloWorld.servlet?ID=' + getSessionId(), 'HelloServletWorld',
400, 300)
    </onClick>
    <icon>~/images/servlet.png</icon>
  </parameters>
</action-factory>
</extension>
```

Note

To keep things as simple as possible, a new instance of the servlet class is created for each request. If the servlet needs complex or expensive initialisation, that should be externalised to other classes that the servlet can use.

26.8. Extension points defined by BASE

In this section, we will give an overview of the extension points defined by BASE. Most extension points are used in the web client to add buttons and menu items, but there are a few for the core API as well.

26.8.1. Menu: extensions

Menu items can be added to the top-level Extensions menu. Actions should implement the interface: `MenuItemAction`

The `MenuItemAction.getMenuType()` provides support for `MENUITEM`, `SUBMENU` and `SEPARATOR` menus. Which of the other properties that are needed depend on the menu type. Read the javadoc for more information. This extension point doesn't support custom javascripts or stylesheets and the rendering is internal (eg. extensions can't provide their own renderers).

BASE ships with two action factories: `FixedMenuItemFactory` and `PermissionMenuItemFactory`. The fixed factory provides a menu that is the same for all users. The permission factory can disable or hide a menu depending on the logged in user's role-based permissions. The title, icon, etc. can have different values depending on if the menu item is disabled or enabled.

26.8.2. Toolbars

Most toolbars on all list and single-item view pages can be extended with extra buttons. Actions should implement the interface: `ButtonAction`. Button actions are very simple and only need to provide things like a title, tooltip, on-click script, etc. This extension point has support for custom javascript, stylesheets and renderers. The default renderer is `ToolbarButtonRendererFactory`.

BASE ships with two action factories: `FixedButtonFactory` and `PermissionButtonFactory`. The fixed factory provides a toolbar button that is the same for all users. The permission factory can disable or hide a button depending on the logged in user's role-based permissions. The title, icon, etc. can have different values depending on if the menu item is disabled or enabled.

26.8.3. Edit dialogs

Most item edit dialogs can be extended with additional tabs. Actions should implement the interface: `TabAction`. The actions are, in principle, simple and only need to provide a title and content (HTML). The action may also provide javascripts for validation, etc. This extension point has support for custom stylesheets and javascript. Rendering is fixed and can't be overridden.

BASE ships with two action factories: `FixedTabFactory` and `IncludeContentTabFactory`. The fixed factory provides a tab with fixed content that is the same for all users and all items. This factory is not very useful in a real scenario. The other factory provides content by including the output from another resource, eg. a JSP page, a servlet, etc. The current context is stored in a request-scoped attribute under the key given by `JspContext.ATTRIBUTE_KEY`. A JSP or servlet should use this to hook into the current flow. Here is a code example:

```
// Get the JspContext that was created on the main edit page
final JspContext jspContext = (JspContext)request.getAttribute(JspContext.ATTRIBUTE_KEY);

// The current item is found in the context. NOTE! Can be null if a new item
final BasicItem item = (BasicItem)jspContext.getCurrentItem();

// Get the DbControl and SessionControl used to handle the request (do not close!)
final DbControl dc = jspContext.getDbControl();
final SessionControl sc = dc.getSessionControl();
```

The extra tab need to be paired with an extension that is invoked when the edit form is saved. Each edit-dialog extension point has a corresponding on-save extension point. Actions should implement the interface: `OnSaveAction`. This interface define three callback methods that BASE will call when saving an item. The `OnSaveAction.onSave()` method is called first, but not until all regular properties have been updated. If the transaction is committed the `OnSaveAction.onCommit()` method is also called, otherwise the `OnSaveAction.onRollback()` is called. The `onSave()` method can throw an exception that will be displayed to the user. The other callback method should not throw exceptions at all, since that may result in undefined behaviour and can be confusing for the user.

26.8.4. Bioassay set: Tools

The bioassay set listing for an experiment has a **Tools** column which can be extended by extensions. This extension point is similar to the toolbar extension points and actions should implement the interface: `ButtonAction`.

Note that the list can contain `BioAssaySet`, `Transformation` and `ExtraValue` items. The factory implementation need to be aware of this if it uses the `JspContext.getItem()` method to examine the current item.

26.8.5. Bioassay set: Overview plots

The bioassay set page has a tab **Overview plots**. The contents of this tab is supposed to be some kind of images that have been generated from the data in the current bioassay set. What kind of plots that can be generated typically depends on the kind of data you have. BASE ships with an extension (`MAPlotFactory`) that creates MA plots and Correction factor plots for 2-channel bioassays. Actions should implement the interface: `OverviewPlotAction`. A single action generates a sub-tab in the **Overview plots** tab. The sub-tab may contain one or more images. Each image is defined by a `PlotGenerator` which sets the size of the image and provides an URL to a servlet that generates the actual image. It is recommended that the servlet cache images since the data in a bioassay set never changes. The BASE core API provides a system-managed file cache that is suitable for this. Call `Application.getStaticCache()` to get a `StaticCache` instance. See the source code for the core `PlotServlet` for details of how to use the cache.

26.8.6. Services

A service is a piece of code that is loaded when the BASE web server starts up. The service is then running as long as the BASE web server is running. It is possible to manually stop and start services. This extension point is different from most others in that it doesn't affects the visible interface. Since services are loaded at startup time, this also means that the context passed to `ActionFactory` methods will not have any `ClientContext` associated with it (eg. the `InvocationContext.getClientContext()` always return null). There is also no meaning for extensions to specify a `RendererFactory`. Service actions should implement the interface: `ServiceControllerAction`. The interface provides `start()` and `stop()` methods for controlling the service. BASE doesn't ship with any service, but there is an FTP service available at the BASE plug-ins site: <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.ftp>³

26.8.7. Connection managers

This extension point adds support for using external files in BASE. This is a core extension point and is available independently of the web client. Actions should implement the interface: `ConnectionManagerFactory`.

The `getDisplayname()` and `getDescription()` methods are used in the gui when a user manually selects which connection manager to use. The `supports(URI)` is used when auto-selecting a connection manager based on the URI of the resource.

³ <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.ftp>

BASE ships with factory that supports HTTP and HTTPS file references: `HttpConnectionManagerFactory`. The BASE plug-ins site has an connection manager that support the Hadoop distributed file system (HDFS): <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.hdfs>⁴

26.8.8. Fileset validators

See also

- Section 28.3.9, “Using files to store data” (page 330)
- Section 28.2.7, “Experimental platforms and item subtypes” (page 304)

In those cases where files are used to store data instead of importing it to the database, BASE can use extensions to check that the supplied files are valid and also to extract metadata from the files. For example, the `CelValidationAction` is used to check if a file is a valid Affymetrix CEL file and to extract data headers and the number of spots from it.

Validation and metadata extraction actions should implement the `ValidationAction` interface. This is a core extension point and is available independently of the web client.

This extension point is a bit more complex than most other extension points. To begin with, the factory class will be called with the owner of the file set as the current item. Eg. the `ClientContext.getCurrentItem()` should return a `FileStoreEnabled` item. It is recommended that the factory performs a pre-filtering of the items to avoid calling the actual validation code on unsupported files. For example, the `CelValidationFactory` will check that the item is a `RawBioAssay` item using the Affymetrix platform.

Each file in the file set is then passed to the `ValidationAction.acceptFile(FileSetMember)` which may accept or reject the file. If the file is accepted it may be accepted for immediate validation or later validation. The latter option is useful in more complex scenarios where files need to be validated as a group. If the file is accepted the `ValidationAction.validateAndExtractMetadata()` is called, which is where the real work should happen.

The extensions for this extension point is also called when a file is replaced or removed from the file set. The calling sequence to set up the validation is more or less the same as described above, but the last step is to call `ValidationAction.resetMetadata()` instead of `ValidationAction.validateAndExtractMetadata()`.

Use the `SingleFileValidationAction` class.

Most validators that work on a single file at a time may find the `SingleFileValidationAction` class useful. It should simplify the task of making sure that only the desired file type is validated. See the source code of the `CelValidationAction` class for an example.

⁴ <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.hdfs>

Chapter 27. Web services

This chapter is an introduction of web services in BASE. It is recommended to begin your reading with the first section in this chapter and then you can move on to either the second section for more information how to develop client applications, or to the third section if you think there are some services missing and you want to know how to proceed to develop a new one.

Before moving on to develop client applications or new services there are few things that need to be explained first.

1. Items in BASE are not send directly by the web services, most of them are to complex for this should be possible. Instead is each item type represented by an info class that can hold the type's less complex properties.
2. BASE offers a way for services to allow the client applications to put their own includes and restrictions on a query before it is executed. For those who intend to develop services it is recommended to have a look in javadoc for the `QueryOptions` class. This is on the first hand for the service developers but it can be useful for client developers to also know that this may be available in some services.

27.1. Available services

Web services can, at the moment, be used to provide some information and data related to experiments in BASE, for example, information about raw bioassays or bioassay set data. The subsection below gives an overview of the services that are currently present in BASE short description for each. More detailed information can be found in the javadoc and WSDL-files. Each service has it's own class and WSDL-file.

27.1.1. Services

`SessionService, SessionClient`

Provides methods to manage a sessions. This is the main entry point to the BASE web services. This contains methods for logging in and out and keeping the session alive to avoid automatic logout due to inactivity.

`ProjectService, ProjectClient`

Service related to projects. You can list available projects and select one to use as the active project.

`ExperimentService, ExperimentClient`

Service related to experiments. List your experiments and find out which raw bioassays that are part of it and which bioassay sets have been created as part of the analysis. Find reporter lists that are part of the experiment and get information about the experimental factors.

`BioAssaySetService, BioAssaySetClient`

Services related to bioassay sets. Get access data files that are attached to bioassay sets. Data from the database must first be exported and saved as a file. Find annotation values on bioassay sets.

`RawBioAssayService, RawBioAssayClient`

Services related to raw bioassays. Find out which raw data files that are present and download them. Find annotation values on raw bioassays.

`ArrayDesignService, ArrayDesignClient`

Services related to array design. Find out which data files that are present and download them. Find annotation values on array designs.

AnnotationTypeService, AnnotationTypeClient

Services related to annotation types. Find out which annotation types that can be used for different types of items.

ReporterService, ReporterClient

Services related to reporters and reporter lists. Get information about all admin-defined extended properties. Download reporter information for reporters.

FileService, FileClient

Services related to files. Download files.

27.2. Client development

How to develop client applications for the web services in BASE depends on which program language you are using. BASE comes with a simple client API for java for the existing services. If you use this API, you don't have to worry about WSDL files, stubs and skeletons and other web services related stuff. Just use it the client API as any other java API.

The client API can be downloaded with example code from the BASE plug-ins website¹. The package contains all external JAR files you need, the WSDL files (in case you still want them) and some example code that logs in to a BASE server, lists projects and experiments and then logs out again. Here is a short example of how to login to a BASE server, list the experiments and then logout.

```
String serviceUrl = "http://your.base.server/base2/services";
String login = "mylogin";
String password = "mypassword";

// Create new session
SessionClient session = new SessionClient(serviceUrl, null, null);

// Login
session.login(login, password, null);

// Get all projects and print out name and ID
ExperimentClient ex = new ExperimentClient(session);
ExperimentInfo[] experiments = ex.getExperiments(new QueryOptions());

if (experiments != null && experiments.length > 0)
{
    for (ExperimentInfo info : experiments)
    {
        System.out.println("name=" + info.getName() + "; id=" + info.getId());
    }
}

// Logout
session.logout();
```

If you want to use another language than Java or you don't want to use our client API, you probably need the WSDL files. These can be found in the client API package discussed above and also in the BASE core distribution in the `<base-dir>/misc/wsdl` directory. The WSDL files can also be generated on the fly by the BASE server by appending `?wsdl` to the url for a specific service. For example, `http://your.base.server/base2/services/Session?wsdl`.

27.2.1. Receiving files

Some methods can be used to download files or exported data. Since this kind of data can be binary data the usual return methods can't be used. BASE uses a method commonly known as *web*

¹ <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.examples.webservices>

services with attachments using MTOM (SOAP Message Transmission Optimization Mechanism) to send file data. Read the MTOM Guide² from Apache if you want to know more about this.

With the client API it is relatively easy to download a file. Here is a short program example that downloads the CEL files for all raw bioassays in an experiment.

```
int experimentId = ...
SessionClient session = ...
String fileType = "affymetrix.cel";

// Create clients for experiment and raw bioassay
ExperimentClient ec = new ExperimentClient(session);
RawBioAssayClient rc = new RawBioAssayClient(session);

// Get all raw bioassays in the experiment
RawBioAssayInfo[] rawInfo = ec.getRawBioAssays(experimentId, new QueryOptions());
if (rawInfo == null && rawInfo.length == 0) return;

for (RawBioAssayInfo info : rawInfo)
{
    // We receive the file contents as an InputStream
    InputStream download = rc.downloadRawDataByType(info.getId(), fileType);

    // Save to file with the same name as the raw bioassay + .cel
    // assume that there are no duplicates
    File saveTo = new File(info.getName() + ".cel");
    FileUtil.copy(download, new FileOutputStream(saveTo));
}
```

If you are using another programming language than Java or doesn't want to use the client API you must know how to get access to the received file. The data is sent as a binary attachment to an element in the XML. It is in the interest of the client developer to know how to get access to a received file and to make sure that the programming language/web services framework that is used supports MTOM. Below is a listing which shows an example of a returned message from the `RawBioAssayService.downloadRawDataByType()` service.

```
--MIMEBoundaryurn_uuid_1526E5ADD9FC4431651195044149664
Content-Type: application/xop+xml; charset=UTF-8; type="application/soap+xml"
Content-Transfer-Encoding: binary
Content-ID: <0.urn:uuid:1526E5ADD9FC4431651195044149665@apache.org>

<ns:downloadRawDataByTypeResponse xmlns:ns="http://server.ws.basedb.sf.net">
  <ns:return>
    <Test.cel:Test.cel xmlns:Test.cel="127.0.0.1">
      <xop:Include href="cid:1.urn:uuid:1526E5ADD9FC4431651195044149663@apache.org"
        xmlns:xop="http://www.w3.org/2004/08/xop/include" />
    </Test.cel:Test.cel>
  </ns:return>
</ns:downloadRawDataByTypeResponse>
--MIMEBoundaryurn_uuid_1526E5ADD9FC4431651195044149664
Content-Type: text/plain
Content-Transfer-Encoding: binary
Content-ID: <1.urn:uuid:1526E5ADD9FC4431651195044149663@apache.org>

... binary file data is here ...
```

Here is a program listing, that shows how to pick up the file. This is the actual implementation that is used in the web service client that comes with BASE. The `InputStream` returned from this method is the same `InputStream` that is returned from, for example, the `RawBioAssayClient.downloadRawDataByType()` method.

² <http://axis.apache.org/axis2/java/core/docs/mtom-guide.html>

```
// From AbstractRPCClient.java
protected InputStream invokeFileBlocking(String operation, Object... args)
    throws AxisFault, IOException
{
    //Get the original response element as sent from the server-side
    OMElement response = getService().invokeBlocking(getOperation(operation), args);

    //The file element returned from the service is the first element of the response
    OMElement fileElement = response.getFirstElement();

    //The data node always in the first element.
    OMElement dataElement = fileElement.getFirstElement();
    if (dataElement == null) return null;

    //Get the binary node and pick up the inputstream.
    OMText node = (OMText)dataElement.getFirstOMChild();
    node.setBinary(true);
    DataHandler dataHandler = (DataHandler)node.getDataHandler();
    return dataHandler.getInputStream();
}
```

27.3. Services development

This list should work as guide when creating new web service in BASE.

1. Create a new class that extends `AbstractRPCService`
2. Place the new service in same package as the abstract class, `net.sf.basedb.ws.server`
3. Write the routines/methods the service should deploy.

Never return void from methods

For server-side exceptions to be propagated to the client the web services method mustn't be declared as *void*. We suggest that in cases where there is no natural return value, the session ID is returned, for example:

```
public String myMethod(String ID, ...more parameters...)
{
    // ... your code here
    return ID;
}
```

4. Make the Ant build-file creates a WSDL-file when the services are compiled (see below). This step is not needed for BASE to work but may be appreciated by client application developers.
5. Register the service in the `<base-dir>/src/webservices/server/META-INF/services.xml` file. This is an XML file listing all services and is needed for BASE (Axis) to pick up the new service and expose it to the outside world. Below is an example of hoe the `Session` service is registered.

Example 27.1. How to register a service in `services.xml`

```
<service name="Session" scope="application">
  <description>
    This service handles BASE sessions (including login/logout)
  </description>
  <messageReceivers>
    <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
      class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
    <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
      class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver" />
  </messageReceivers>
  <parameter name="ServiceClass"
    locked="false">net.sf.basedb.ws.server.SessionService</parameter>
</service>
```

27.3.1. Generate WSDL-files

When a new service is created it can be a good idea to also get a WSDL-file generated when the web services are compiled. The WSDL-file will be a help for those developers who intend to create client applications to the new service. It is a one-liner in the Ant build file to do this and not very complicated. To create a WSDL file for the new web service add a line like the one below to the `webservices.wsdl` target. Replace *SessionService* with the name of the new service class.

```
<webservices.wsdl serviceClassName="SessionService"/>
```

27.4. Example web service client (with download)

We have created a simple Java client that uses web services to get information about projects and experiments from a BASE server. The example code can also download raw data files attached to an experiment. The example code can be used as a starting point for developers wanting to do their own client. You can download a tar file with the source and compiled code³ from the BASE plug-ins website.

³ <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.examples.webservices>

Chapter 28. The BASE API

28.1. The Public API of BASE

Not all public classes and methods in the `base-*.jar` files and other JAR files shipped with BASE are considered as *Public API*. This is important knowledge since we will always try to maintain backwards compatibility for classes that are part of the public API. For other classes, changes may be introduced at any time without notice or specific documentation. In other words:

Only use the public API when developing plug-ins and extensions

This will maximize the chance that your code will continue to work with the next BASE release. If you use the non-public API you do so at your own risk.

See the BASE API javadoc¹ for information about what parts of the API that contributes to the public API. Methods, classes and other elements that have been tagged as `@deprecated` should be considered as part of the internal API and may be removed in a subsequent release without warning.

Keeping the backwards compatibility is an aim only. It may not always be possible. See Appendix I, *API changes that may affect backwards compatibility* (page 443) to read more about changes that have been introduced by each release that may affect existing code.

28.1.1. What is backwards compatibility?

There is a great article about this subject on http://wiki.eclipse.org/index.php/Evolving_Java-based_APIs. This is what we will try to comply with. If you do not want to read the entire article, here are some of the most important points:

Binary compatibility

For example:

- We cannot change the number or types of parameters to a method or constructor.
- We cannot add or change methods to interfaces that are intended to be implemented by plug-in or client code.

Contract compatibility

For example:

- We cannot change the implementation of a method to do things differently than before. For example, allow `null` as a return value when it was not allowed before.

Note

Sometimes there is a very fine line between what is considered a bug and what is considered a feature. For example, if the actual implementation does not do what the javadoc says, do we change the code or do we change the documentation? This has to be considered from case to case and depends on the age of the code and if we expect plug-ins and clients to be affected by it or not.

¹ ../../../api/index.html

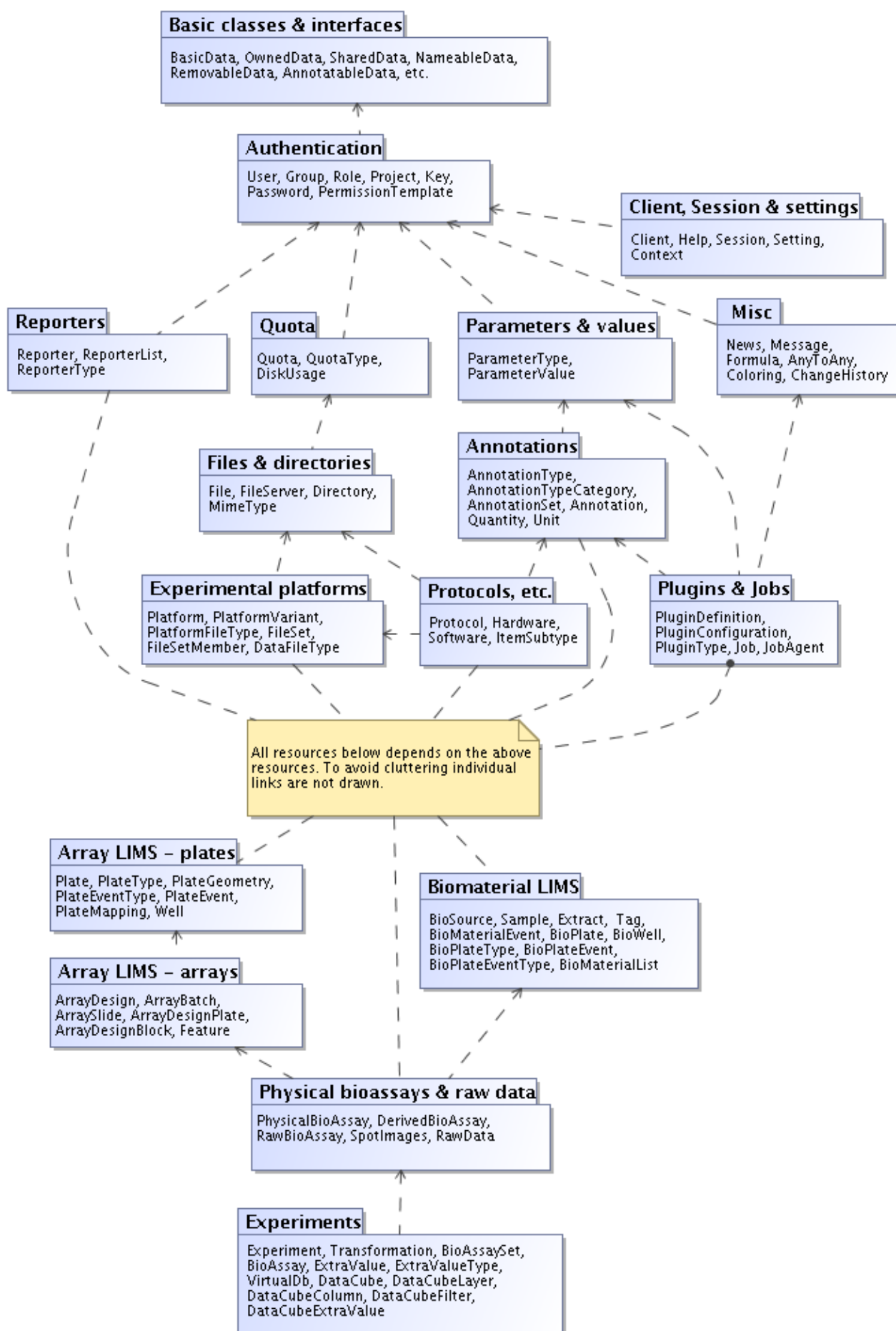
Source code compatibility

This is not an important matter and is not always possible to achieve. In most cases, the problems are easy to fix. Example:

- Adding a class may break a plug-in or client that import classes with `. *` if the same class name exists in another package.

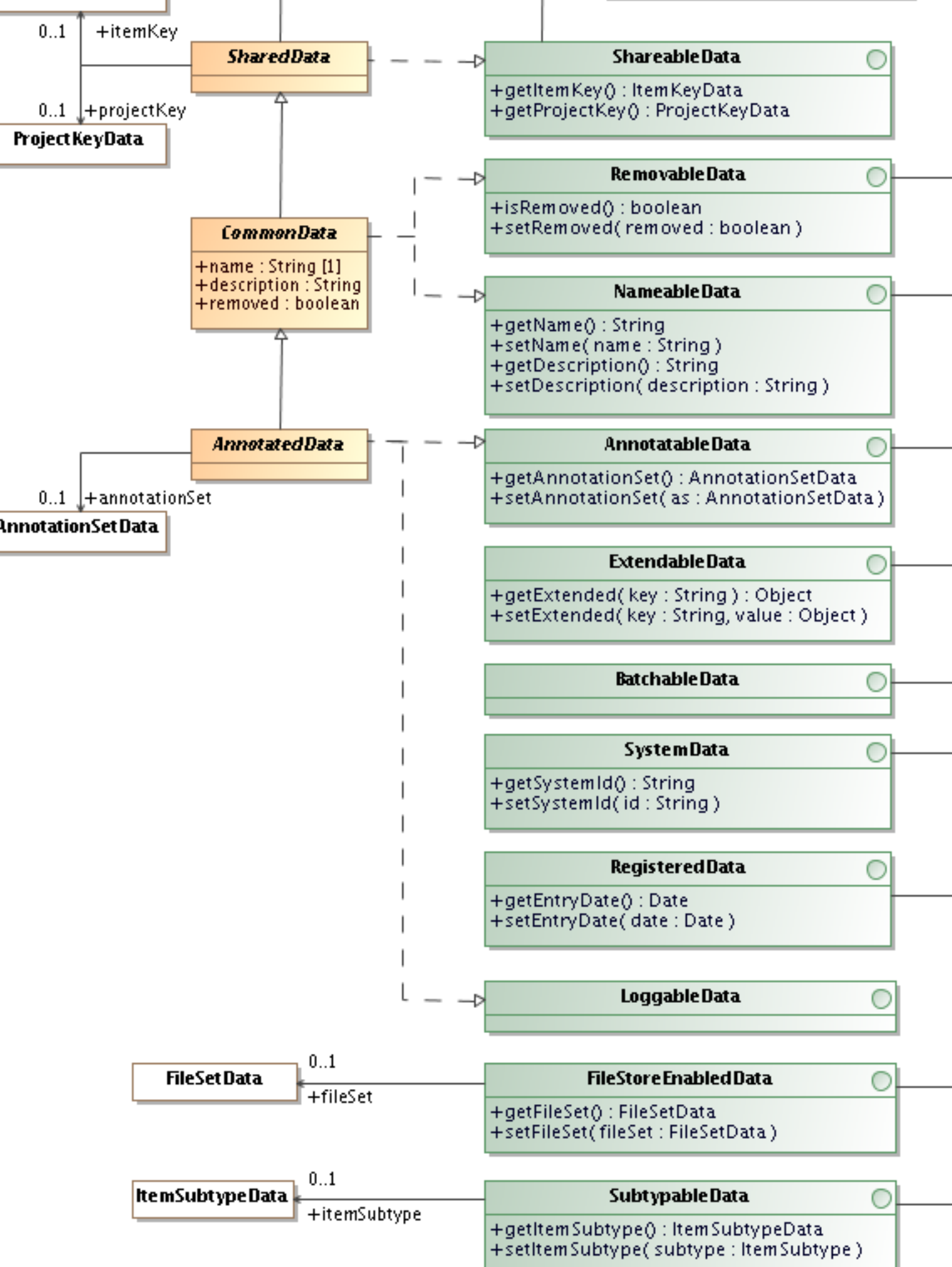
28.2. The Data Layer API

This section gives an overview of the entire data layer API. The figure below show how different modules relate to each other.



28.2.1. Basic classes and interfaces

This document contains information about the basic classes and interfaces in this package. They are important since all data-layer classes must inherit from one of the already existing abstract base classes or implement one or more of the existing interfaces. They contain code that is common to all classes, for example implementations of the `equals()` and `hashCode()` methods or how to link with the owner of an item.



Classes

BasicData

The root class. It overrides the `equals()`, `hashCode()` and `toString()` methods from the `Object` class. It also defines the `id` and `version` properties. All data layer classes must inherit from this class or one of its subclasses.

OwnedData

Extends the `BasicData` class and adds an `owner` property. The owner is a required link to a `UserData` object, representing the user that is the owner of the item.

SharedData

Extends the `OwnedData` class and adds properties (`itemKey` and `projectKey`) that holds access permission information for an item. Access permissions are held in `ItemKeyData` and/or `ProjectKeyData` objects. These objects only exist if the item has been shared.

CommonData

This is a convenience class for items that extends the `SharedData` class and implements the `NameableData` and `RemoveableData` interfaces. This is one of the most common situations.

AnnotatedData

This is a convenience class for items that can be annotated. Annotations are held in `AnnotationSetData` objects. The annotation set only exists if annotations have been created for the item.

Interfaces

IdentifiableData

All items are identifiable, which means that they have a unique `id`. The `id` is unique for all items of a specific type (ie. class). The `id` is a number that is automatically generated by the database and has no other meaning outside of the application. The `version` property is used for detecting and preventing concurrent modifications to an item.

OwnableData

An ownable item is an item which has an owner. The owner is represented as a required link to a `UserData` object.

ShareableData

A shareable item is an item which can be shared to other users, groups or projects. Access permissions are held in `ItemKeyData` and/or `ProjectKeyData` objects.

NameableData

A nameable item is an item that has a name (required) and a description (optional). The name doesn't have to be unique, except in a few special cases (for example, the name of a file).

RemoveableData

A removable item is an item that can be flagged as removed. This doesn't remove the information about the item from the database, but can be used by client applications to hide items that the user is not interested in. A `trashcan` function can be used to either restore or permanently remove items that have the flag set.

SystemData

A system item is an item which has an additional `id` in the form of string. A system `id` is required when we need to make sure that we can get a specific item without knowing the numeric `id`. Example of such items are the root user and the everyone group. A system `id` is generally constructed like: `net.sf.basedb.core.User.ROOT`. The system `id`s are defined in the core layer by each item class.

DiskConsumableData

This interface is used by items which occupy a lot of disk space and should be part of the quota system, for example files. The required `DiskUsageData` contains information about the size, location, owner etc. of the item.

AnnotatableData

This interface is used by items which can be annotated. Annotations are name/value pairs that are attached as extra information to an item. All annotations are contained in an `AnnotationSetData` object.

ExtendableData

This interface is used by items which can have extra administrator-defined columns. The functionality is similar to annotations. It is not as flexible, since it is a global configuration, but has better performance. BASE will generate extra database columns to store the data in the tables for items that can be extended.

BatchableData

This interface is a tagging interface which is used by items that needs batch functionality in the core.

RegisteredData

This interface is used by items which registered the date they were created in the database. The registration date is set at creation time and can't be modified later. Since this didn't exist prior to BASE 2.10, null values are allowed on all pre-existing items. Note! For backwards compatibility reasons with existing code in `BioMaterialEventData` the method name is `getEntryDate()`.

LoggableData

This is a tagging interface that indicates that the `DbLogManagerFactory` logging implementation should log changes made to items that implements it.

FileStoreEnabledData

This interface is implemented by all items that can have files with related data attached to them. The file types that can be used for a specific item are usually determined by the main type, the subtype or platform.

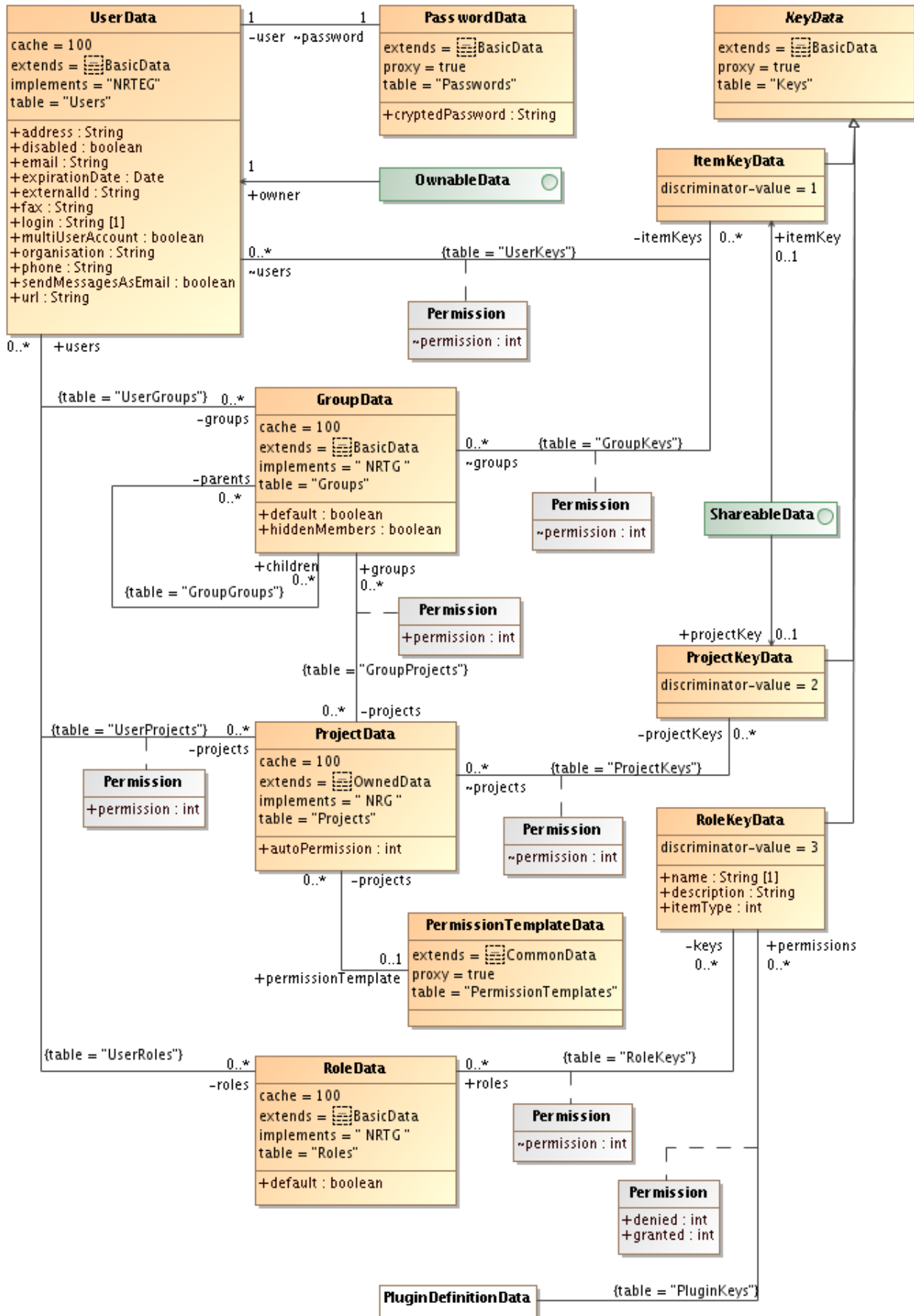
SubtypableData

This interface should be implemented by all items that can be subtyped. Unless otherwise noted the subtype is always an optional link to a `ItemSubtypeData`. item. In the simplest form, the subtype is a kind of an annotation, but for items that also implements the `FileStoreEnabledData` interface, the subtype can be used to specify the file types that are applicable for each item.

28.2.2. User authentication and access control

This section gives an overview of user authentication and how groups, roles and projects are used for access control to items.

Figure 88.2 User authentication and access control



Users and passwords

The `UserData` class holds information about users. We keep the passwords in a separate table and use proxies to avoid loading password data each time a user is loaded to minimize security risks. It is only if the password needs to be changed that the `PasswordData` object is loaded. The one-to-one mapping between user and password is controlled by the password class, but a cascade attribute on the user class makes sure that the password is deleted when a user is deleted.

Groups, roles, projects and permission template

The `GroupData`, `RoleData` and `ProjectData` classes holds information about groups, roles and projects respectively. A user may be a member of any number of groups, roles and/or projects. New users are automatically added as members of all groups and roles that has the `default` property set.

The membership in a project comes with an attached permission values. This is the highest permission the user has in the project. No matter what permission an item has been shared with the user will not get higher permission. Groups may be members of other groups and also in projects. A `PermissionTemplateData` is just a holder for permissions that users can use when sharing items. The template is never part of the actual permission control mechanism.

Group membership is always accounted for, but the core only allows one project at a time to be use, this is the *active project*. When a project is active new items that are created are automatically shared according to the settings for the project. There are two cases. If the project has a permission template, the new item is given the same permissions as the template has. If the project doesn't have a permission template, the new item is shared to the active project with the permission given by the `autoPermission` property. Note that in the first case the new item may or may not be shared to the active project depending on if the template is shared to the project or not.

Note that the permission template is only used (by the core) when creating new items. The permissions held by the template are copied and when the new item has been saved to the database there is no longer any reference back to the template that was used to create it. This means that changes to the template does not affect already existing items and that the template can be deleted without problems.

Keys

The `KeyData` class and it's subclasses `ItemKeyData`, `ProjectKeyData` and `RoleKeyData`, are used to store information about access permissions to items. To get permission to manipulate an item a user must have access to a key giving that permission. There are three types of keys:

`ItemKey`

Is used to give a user or group access to a specific item. The item must be a `ShareableData` item. The permissions are usually set by the owner of the item. Once created an item key cannot be changed. This allows the core to reuse a key if the permissions match exactly, ie. for a given set of users/groups/permissions there can be only one item key object.

`ProjectKey`

Is used to give members of a project access to a specific item. The item must be a `ShareableData` item. Once created a project key cannot be changed. This allows the core to reuse a key if the permissions match exactly, ie. for a given set of projects/permissions there can be only one project key object.

`RoleKey`

Is used to give a user access to all items of a specific type, ie. `READ` all `SAMPLES`. The installation will make sure that there already exists a role key for each type of item, and it is not possible to add new or delete existing keys. Unlike the other two types this key can be modified.

A role key is also used to assign permissions to plug-ins. If a plug-in has been specified to use permissions the default is to deny everything. The mapping to the role key is used to grant

permissions to the plugin. The `granted` value gives the plugin access to all items of the related item type regardless of if the user that is running the plug-in has the permission or not. The `denied` values denies access to all items of the related item type even if the logged in user has the permission. Permissions that are not granted nor denied are checked against the logged in users regular permissions. Permissions to items that are not linked are always denied.

Permissions

The `permission` property appearing in many classes is an integer values describing the permission:

Value	Permission
1	Read
3	Use
7	Restricted write
15	Write
31	Delete
47 (=32+15)	Set owner
79 (=64+15)	Set permissions
128	Create
256	Denied

The values are constructed so that `READ -> USE -> RESTRICTED_WRITE -> WRITE -> DELETE` are chained in the sense that a higher permission always implies the lower permissions also. The `SET_OWNER` and `SET_PERMISSION` both implies `WRITE` permission. The `DENIED` permission is only valid for role keys, and if specified it overrides all other permissions.

When combining permission for a single item the permission codes for the different paths are OR-ed together. For example a user has a role key with `READ` permission for `SAMPLES`, but also an item key with `USE` permission for a specific sample. Of course, the resulting permission for that sample is `USE`. For other samples the resulting permission is `READ`.

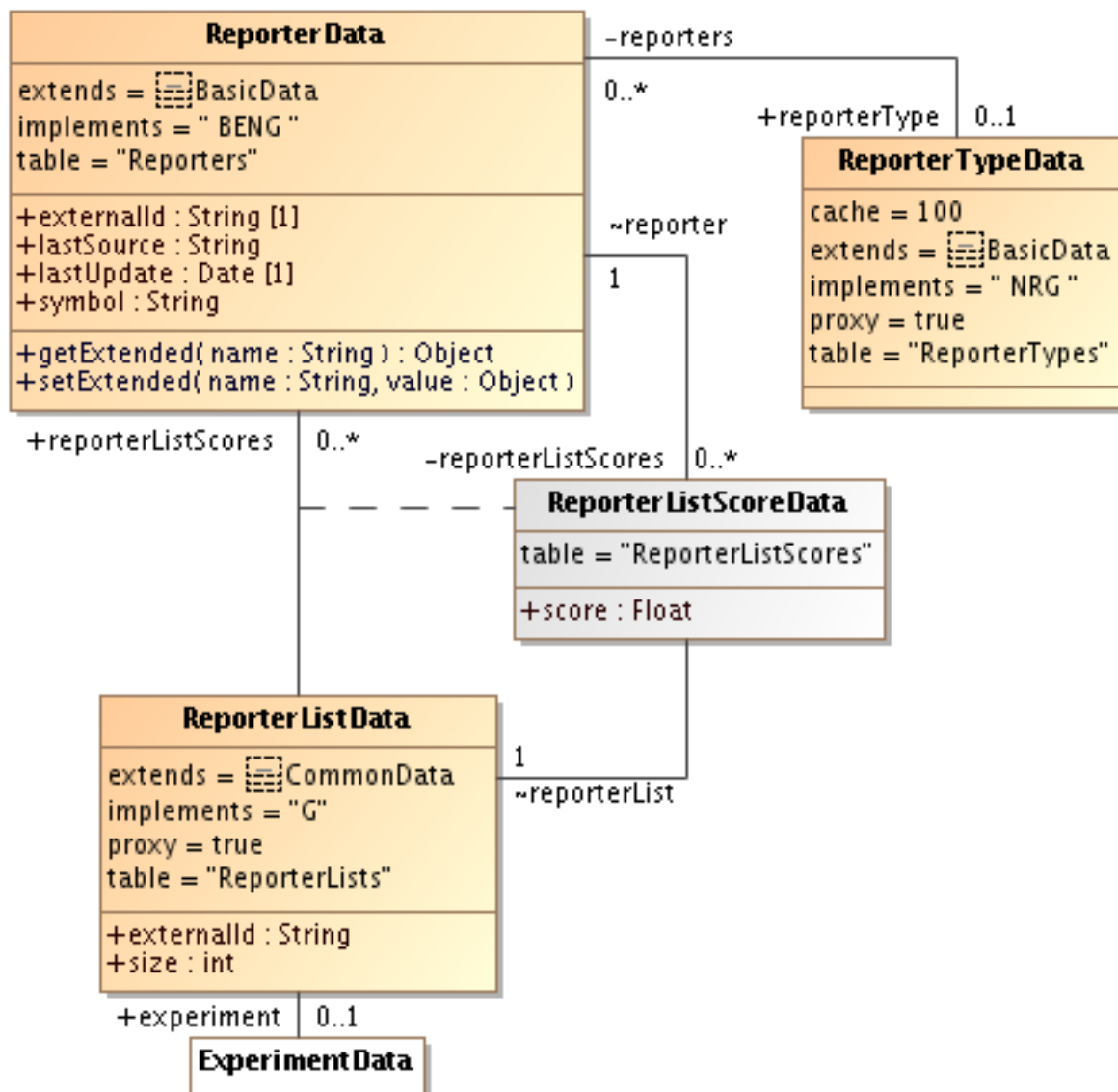
If the user is also a member of a project which has `WRITE` permission for the same sample, the user will have `WRITE` permission when working with that project.

The `RESTRICTED_WRITE` permission is in most cases the same as the `WRITE` permission. So far the `RESTRICTED_WRITE` permission is only given to users to their own `UserData` object so they can change their address and other contact information, but not quota, expiration date and other administrative information.

28.2.3. Reporters

This section gives an overview of reporters in BASE.

Figure 28.4. Reporters



Reporters

The `ReporterData` class holds information about reporters. The `externalId` is a required property that must be unique among all reporters. The external ID is the value BASE uses to match reporters when importing data from files.

The `ReporterData` is an *extendable* class, which means that the server administrator can define additional columns (=annotations) in the reporters table. These are accessed with the `ReporterData.getExtended()` and `ReporterData.setExtended()` methods. See Appendix C, *extended-properties.xml reference* (page 426) for more information about this.

The `ReporterData` is also a *batchable* class which means that there is no corresponding class in the core layer. Client applications and plug-ins should work directly with the `ReporterData` class. To help manage the reporters there is the `Reporter` and `ReporterBatcher` classes. The main reason for this is to increase the performance and lower the memory usage by bypassing internal caching in the core and Hibernate. Performance is also increased by the batchers which uses more efficient SQL against the database than Hibernate.

The `lastUpdate` property holds the data and time the reporter information was last updated. The value is managed automatically by the `ReporterBatcher` class. That goes for `lastSource` property too,

which holds information about where the last update comes from. By default this is set to the name of the logged in user, but it can be changed by calling `ReporterBatcher.setUpdateSource(String source)` before the batcher commits the updates to the database. The source-string should have the format:

```
[ITEM_TYPE] : [ITEM_NAME]
```

where, in the file-case, ITEM_TYPE is File and ITEM_NAME is the file's name.

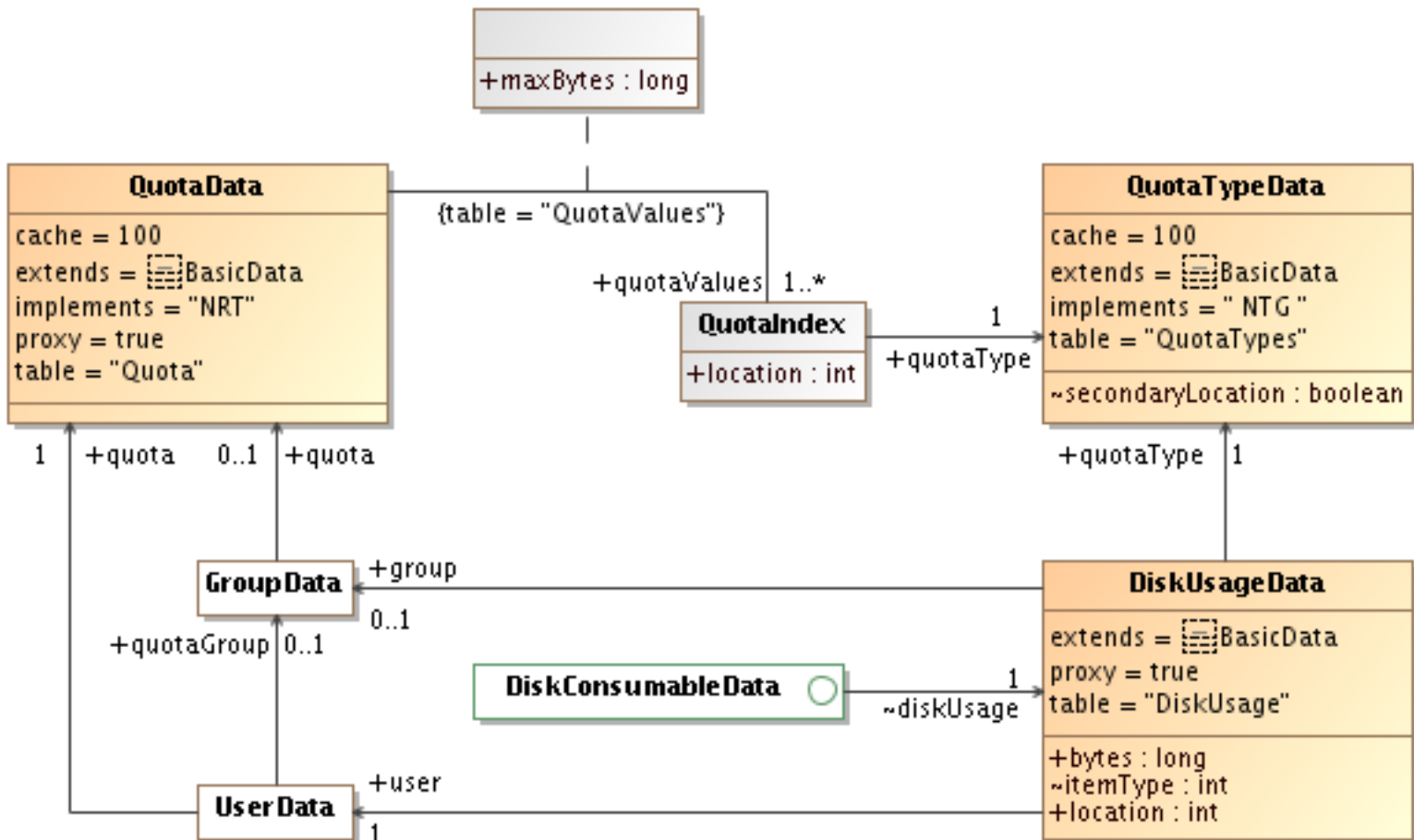
Reporter lists

Reporter lists can be used to group reporters that are somehow related to each other. This could for example be a list of interesting reporters found in the analysis of an experiment. Each reporter in the list may optionally be assigned a score. The meaning of the score value is not interpreted by BASE.

28.2.4. Quota and disk usage

This section gives an overview of quota system in BASE and how the disk usage is kept track of.

Figure 28.5. Quota and disk usage



Quota

The `QuotaData` holds information about a single quota registration. The same quota may be used by many different users and groups. This object encapsulates allowed quota values for different types of quota types and locations. BASE defines several quota types (file, raw data and experiment), and locations (primary, secondary and offline).

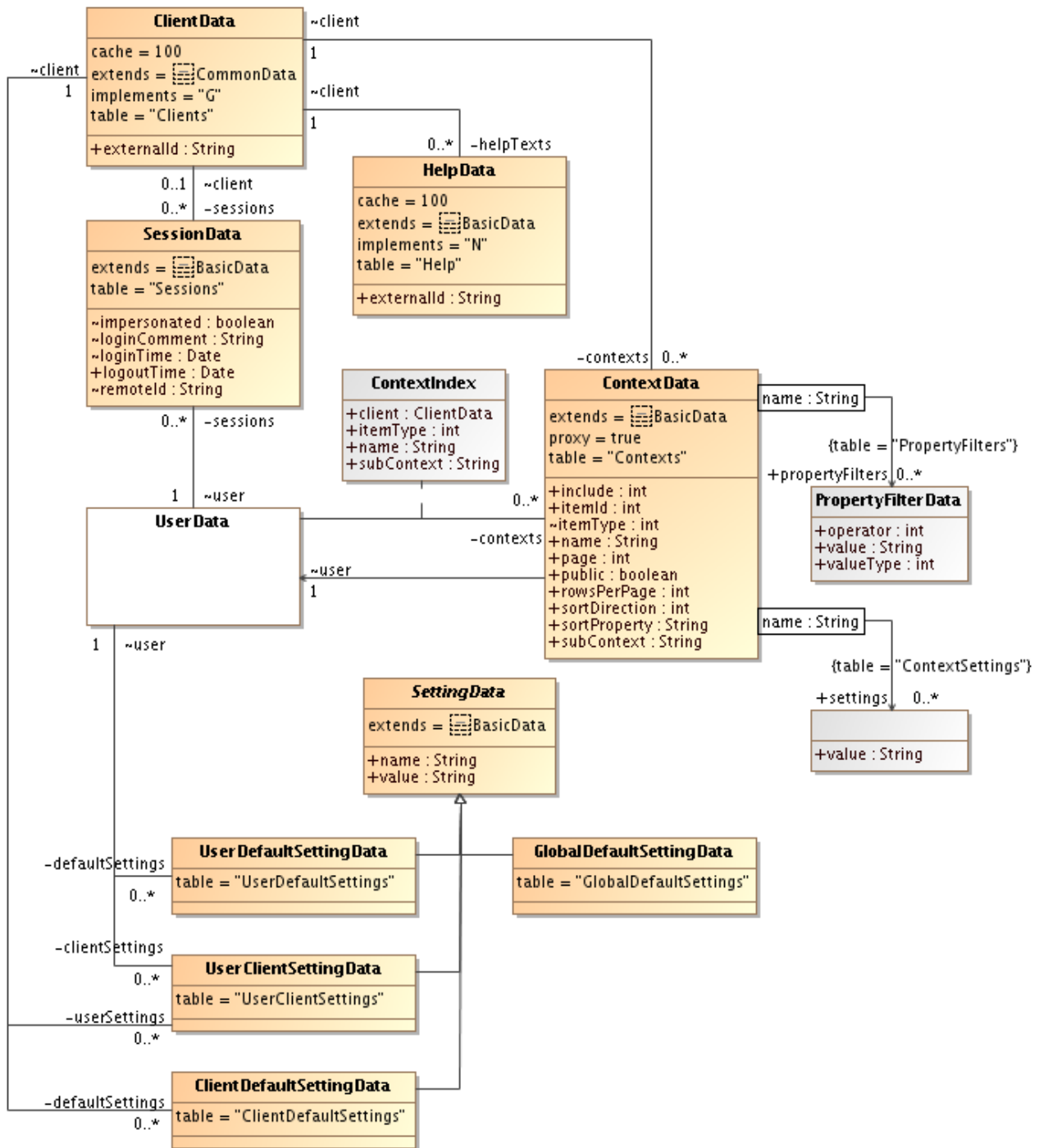
The `quotaValues` property is a map from `QuotaIndex` to maximum byte values. This map must contain at least one entry for the total quota at the primary location.

Disk usage

A `DiskConsumableData` (for example a file) item is automatically linked to a `DiskUsageData` item. This holds information about the number of bytes, the location and quota type the item uses. It also holds information about which user and group (optional) that should be charged for the disk usage. The user is always the owner of the item.

28.2.5. Client, session and settings

This section gives an overview of client applications, sessions and settings.

Figure 28.6. Client, sessions and settings

Clients

The `ClientData` class holds information about a client application. The `externalId` property is a unique identifier for the application. To avoid ID clashes the ID should be constructed in the same way as Java packages, for example `net.sf.basedb.clients.web` is the ID for the web client application.

A client application doesn't have to be registered with BASE to be able to use it. But we recommend it since:

- The permission system allows an admin to specify exactly which users that may use a specific application.
- The application can't store any context-sensitive or application-specific settings unless it is registered.
- The application can store context-sensitive help in the BASE database.

Sessions

A session represents the time between login and logout for a single user. The `SessionData` object is entirely managed by the BASE core, and should be considered read-only for client applications.

Settings

There are two types of settings: context-sensitive settings and regular settings. The regular settings are simple key-value pairs of strings and can be used for almost anything. There are four subtypes:

- Global default settings: Settings that are used by all users and client applications on the BASE server. These settings are read-only except for administrators. BASE has not yet defined any settings of this type.
- User default settings: Settings that are valid for a single user for any client application. BASE has not yet defined any settings of this type.
- Client default settings: Settings that are valid for all users using a specific client application. Each client application is responsible for defining it's own settings. Settings are read-only except for administrators.
- User client settings: Settings that are valid for a single user using a specific client application. Each client application is responsible for defining it's own settings.

The context-sensitive settings are designed to hold information about the current status of options related to the listing of items of a specific type. This includes:

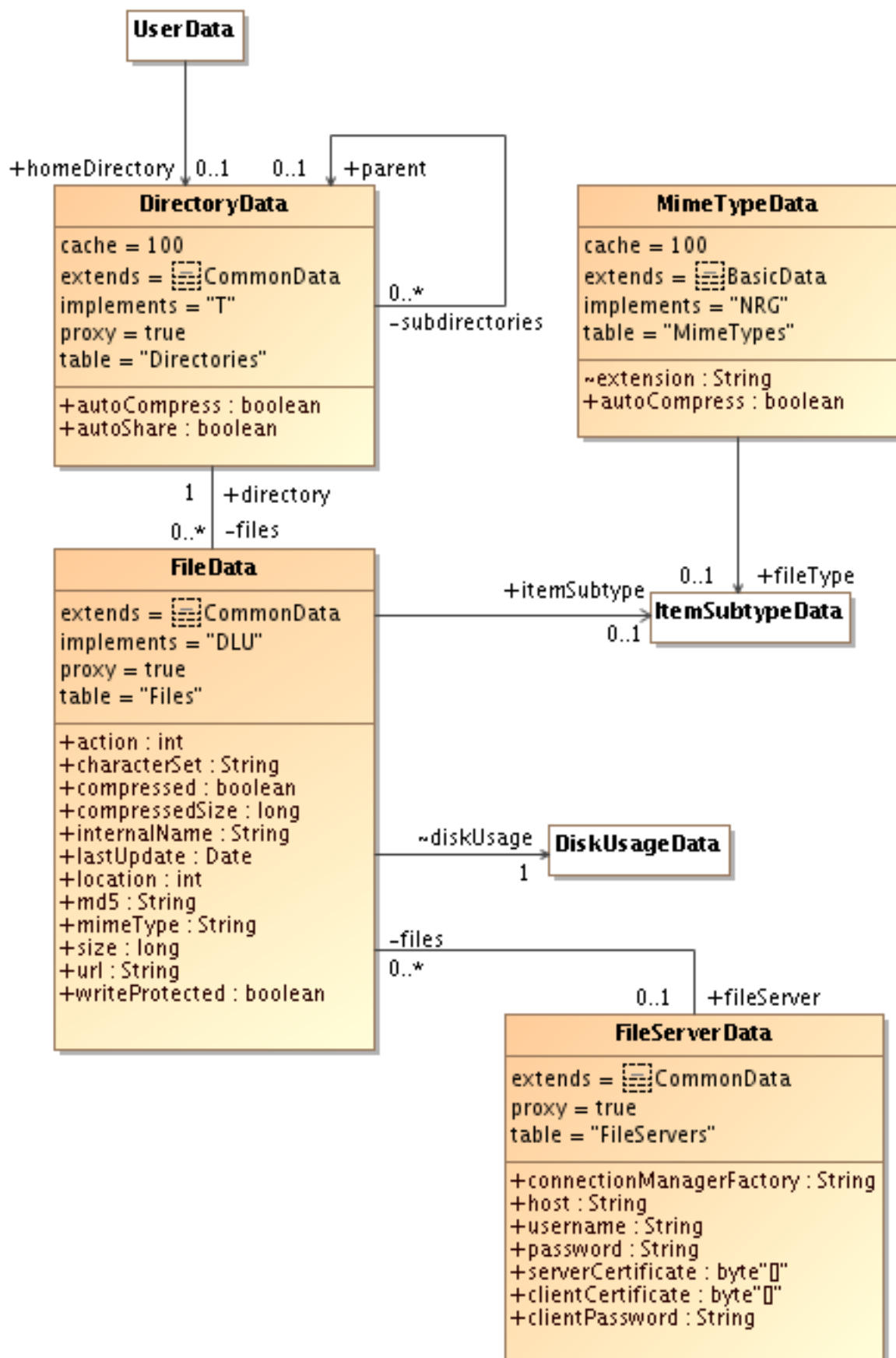
- Current filtering options (as 1 or more `PropertyFilterData` objects).
- Which columns and direction to use for sorting.
- The number of items to display on each page, and which page that is the current page.
- Simple key-value settings related to a given context.

Context-sensitive settings are only accessible if a client application has been registered. The settings may be named to make it possible to store several presets and to quickly switch between them. In any case, BASE maintains a current default setting with an empty name. An administrator may mark a named setting as public to allow other users to use it.

28.2.6. Files and directories

This section covers the details of the BASE file system.

Figure 28.7. Files and directories



The `DirectoryData` class holds information about directories. Directories are organised in the usual way as a tree structure. All directories must have a parent directory, except the system-defined root directory.

The `FileData` class holds information about a file. The actual file contents is stored on disk in the directory specified by the `userfiles` setting in `base.config`. The `internalName` property is the name of the file on disk, but this is never exposed to client applications. The filenames and directories on the disk doesn't correspond to the the filenames and directories in BASE.

The `url` property is used for file items which are stored in an external location. In this case there is no local file data on the BASE server.

The `location` property can take three values:

- 0 = The file is offline, ie. there is no file on the disk
- 1 = The file is in primary storage, ie. it is located on the disk and can be used by BASE
- 2 = The file is in secondary storage, ie. it has been moved to some other place and can't be used by BASE immediately.
- 3 = The file is an external file whose location is referenced by the `url` property. If the file is protected by passwords or certificates the file item may reference a `FileServerData` object. Note that an external file in most cases can be used by client applications/plugin-ins as if the file was stored locally on the BASE server.

The `action` property controls how a file is moved between primary and secondary storage. It can have the following values:

- 0 = Do nothing
- 1 = If the file is in secondary storage, move it back to the primary storage
- 2 = If the file is in primary storage, move it to the secondary storage

The actual moving between primary and secondary storage is done by an external program. See the section called “Secondary storage controller”(page 421) and Section 25.6.2, “Secondary file storage plugins” (page 255) for more information.

The `md5` property can be used to check for file corruption when it is moved between primary and secondary storage or when a user re-uploads a file that has been offline.

BASE can store files in a compressed format. This is handled internally and is not visible to client applications. The `compressed` and `compressedSize` properties are used to store information about this. A file may always be compressed if the users says so, but BASE can also do this automatically if the file is uploaded to a directory with the `autoCompress` flag set or if the file has MIME type with the `autoCompress` flag set.

The `FileServerData` class holds information about an external file server. If the `connectionManagerFactory` isn't set BASE automatically selects a factory based on the URL of the file. There is built-in support for HTTP and HTTPS, but it is possible to install extensions for support for other protocols. The `host` property can be set to override the host part of the URL from the file. See Section 26.8.7, “Connection managers” (page 280) for more information about connection managers.

The `username` and `password` properties are used if the server requires the user to be logged in. BASE has built-in support for Basic and Digest authentication. The `serverCertificate` can be used with HTTPS servers that uses a non-trusted certificate to tell BASE to trust the server anyway. In most cases, this is only needed if the server uses a self-signed certificate, but could, for example, also be used if a trusted site has forgot to renew an expired certificate. The server certificate should be

an X.509 certificate in either binary or text format. The `clientCertificate` and `clientPassword` properties are used for servers that require that users present a valid client certificate before they are allowed access. The client certificate is usually issued by the server operator and must be in PKCS #12 format.

The `FileTypeData` class holds information about file types. It is used only to make it easier for users to organise their files.

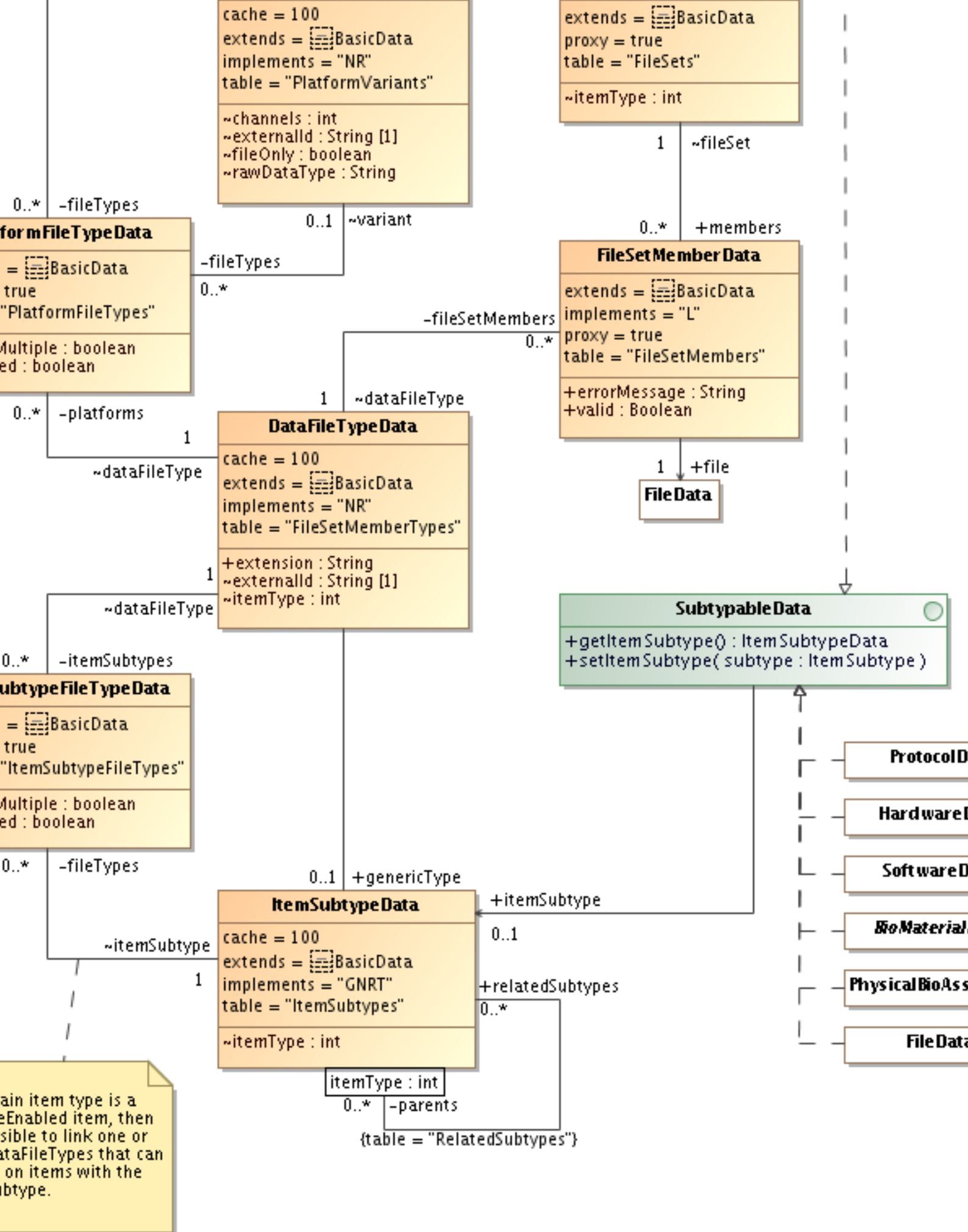
The `MimeTypeData` is used to register mime types and map them to file extensions. The information is only used to lookup values when needed. Given the filename we can set the `File.mimeType` and `File.fileType` properties. The MIME type is also used to decide if a file should be stored in a compressed format or not. The extension of a MIME type must be unique. Extensions should be registered without a dot, ie *html*, not *.html*.

28.2.7. Experimental platforms and item subtypes

This section gives an overview of experimental platforms and how they are used to enable data storage in files instead of in the database. In some senses item subtypes are related to platforms so they are also included here.

See also

- Section 28.3.9, “Using files to store data” (page 330)
- Section D.1, “Default platforms and variants installed with BASE” (page 430)
- Section 26.8.8, “Fileset validators” (page 281)



Platforms

The `PlatformData` holds information about a platform. A platform can have one or more `PlatformVariant`s. Both the platform and variant are identified by an external ID that is fixed and can't be changed. *Affymetrix* is an example of a platform. If the `fileOnly` flag is set data for the platform can only be stored in files and not imported into the database. If the flag is not set data can be imported into the database. In the latter case, the `rawDataType` property can be used to lock the platform to a specific raw data type. If the value is `null` the platform can use any raw data type.

Each platform and its variant can be connected to one or more `DataFileTypeData` items. This item describes the kind of files that are used to hold data for the platform and/or variant. The file types are re-usable between different platforms and variants. Note that a file type may be attached to either only a platform or to a platform with a variant. File types attached to platforms are inherited by the variants. The variants can only define additional file types, not remove or redefine file types that has been attached to the platform.

The file type is also identified by a fixed, non-changable external ID. The `itemType` property tells us what type of item the file holds data for (ie. array design or raw bioassay). It also links to a `ItemSubtype` which is the generic type of data in the file. This allows us to query the database for, as an example, files with the generic type `FileType.RAW_DATA`. If we are in an *Affymetrix* experiment we will get the CEL file, for another platform we will get another file.

The `required` flag in `PlatformFileTypeData` is used to signal that the file is a required file. This is not enforced by the core. It is intended to be used by client applications for creating a better GUI and for validation of an experiment.

The `allowMultiple` flag in `PlatformFileTypeData` controls if it should be possible to store more than one file of the given type in file type. Again, this is not enforced by the core, but only a recommendation to client applications. The setting is also used for validation of an experiment.

Item subtypes

The `ItemSubtypeData` class describes a subtype for a main `itemType`. In the simplest form the subtype is a kind of annotation that is used mainly for creating a better user experience. If the main item type is also implementing the `FileStoreEnabledData` interface, it is possible to register associations to the file types that can be used together with a given item subtype. The `required` and `allowMultiple` have are used in the same way as in the `PlatformFileTypeData` class.

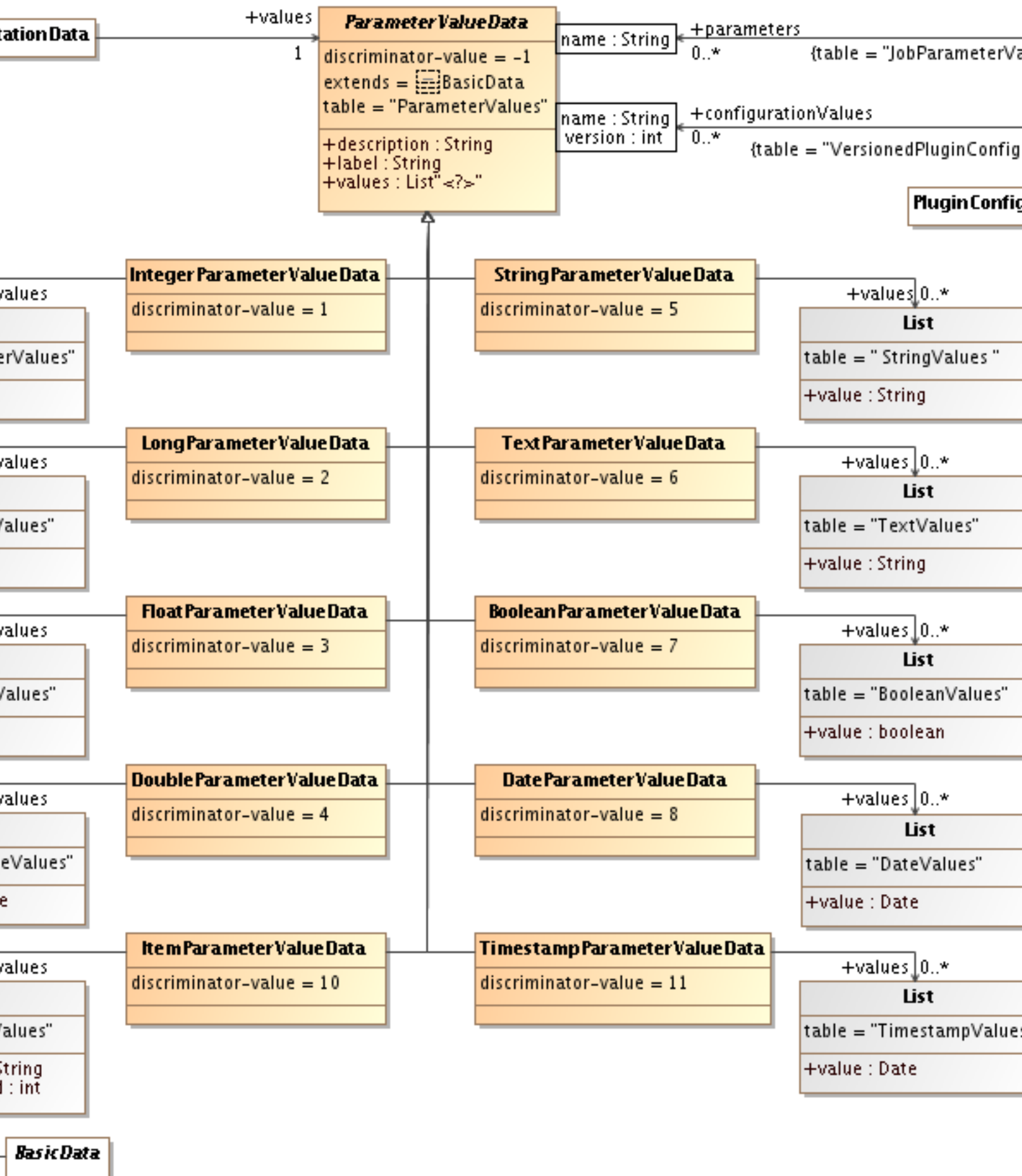
A subtype can be related to other subtypes. This is used to "chain" together groups of item subtypes. For example, *Hybridization* is a subtype for *PHYSICALBIOASSAY*, which is related to the *Labeled extract* (*EXTRACT*) subtype which is related to the *Label* (*TAG*) subtype. In addition, there are also several protocol and hardware subtypes mixed into this. The relationship between subtypes makes it possible for client applications to filter out unrelated stuff, and to validate experiments.

FileStoreEnabled items and data files

An item must implement the `FileStoreEnabledData` interface to be able to store data in files instead of in the database. The interface creates a link to a `FileSetData` object, which can hold several `FileSetMemberData` items. Each member points to specific `FileData` item.

28.2.8. Parameters

This section gives an overview the generic parameter system in BASE that is used to store annotation values, plugin configuration values, job parameter values, etc.



The parameter system is a generic system that can store almost any kind of simple values (string, numbers, dates, etc.) and also links to other items. It is, for example, used to store configuration parameters to plug-ins and jobs as well as annotation values to annotatable items. The `ParameterValueData` class is an abstract base class that can hold multiple values (all must be of the same type). Unless only a specific type of values should be stored, this is the class that should be used when creating references for storing parameter values. It makes it possible for a single relation to use any kind of values or for a collection reference to mix multiple types of values. A typical use case maps a `Map` with the parameter name as the key:

```
private Map<String, ParameterValueData<?>> configurationValues;
/**
    Link parameter name with it's values.
    @hibernate.map table="`PluginConfigurationValues`" lazy="true" cascade="all"
    @hibernate.collection-key column="`pluginconfiguration_id`"
    @hibernate.collection-index column="`name`" type="string" length="255"
    @hibernate.collection-many-to-many column="`value_id`"
        class="net.sf.basedb.core.data.ParameterValueData"
*/
public Map<String, ParameterValueData<?>> getConfigurationValues()
{
    return configurationValues;
}
void setConfigurationValues(Map<String, ParameterValueData<?>> configurationValues)
{
    this.configurationValues = configurationValues;
}
```

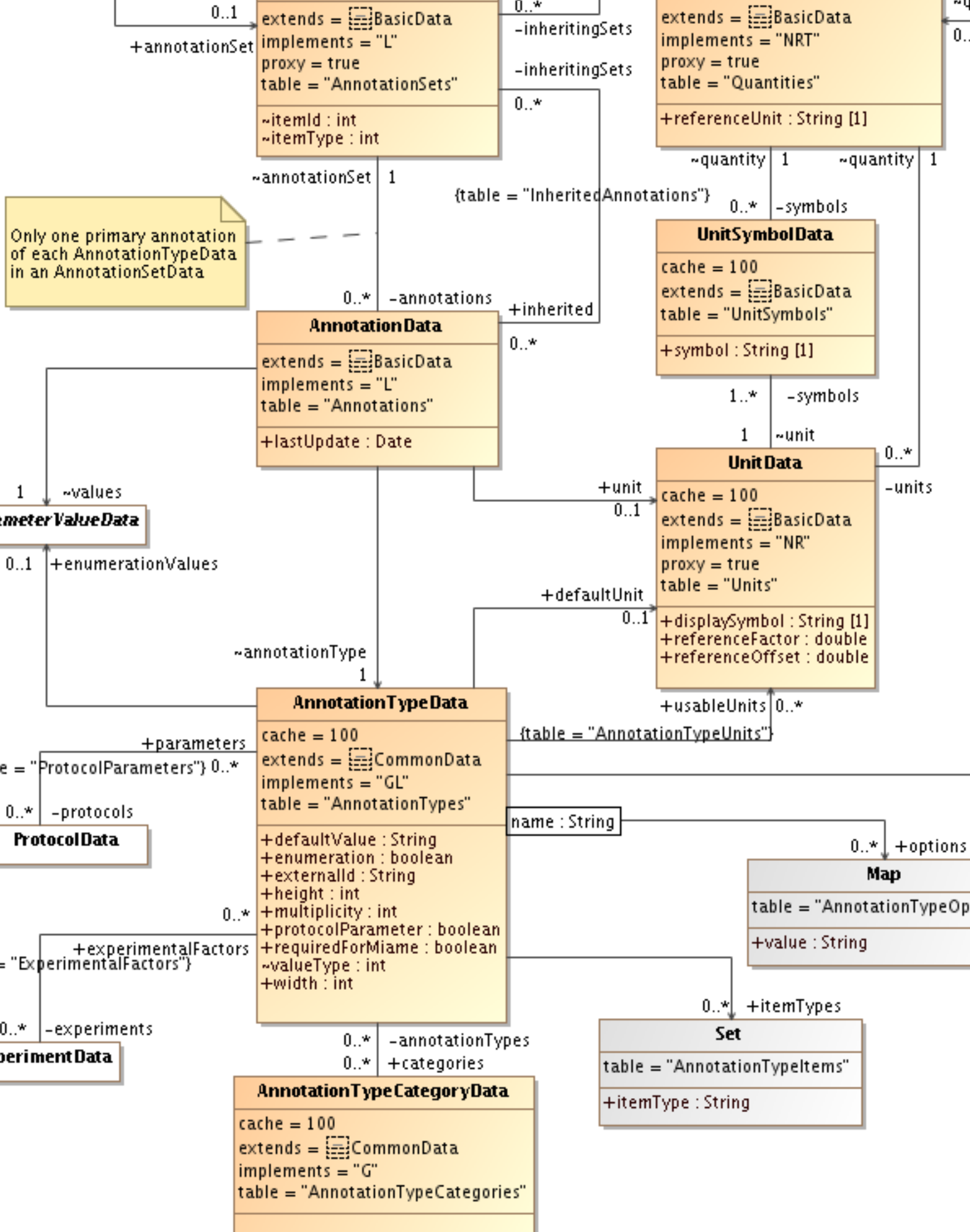
Now it is possible for the collection to store all types of values:

```
Map<String, ParameterValueData<?>> config = ...
config.put("names", new StringParameterValueData("A", "B", "C"));
config.put("sizes", new IntegerParameterValueData(10, 20, 30));

// When you later load those values again you have to cast
// them to the correct class.
List<String> names = (List<String>)config.get("names").getValues();
List<Integer> sizes = (List<Integer>)config.get("sizes").getValues();
```

28.2.9. Annotations

This section gives an overview of how the BASE annotation system works.



Annotations

An item must implement the `AnnotatableData` interface to be able to use the annotation system. This interface gives a link to a `AnnotationSetData` item. This class encapsulates all annotations for the item. There are two types of annotations:

- *Primary annotations* are annotations that explicitly belong to the item. An annotation set can contain only one primary annotation of each annotation type. The primary annotation are linked with the annotations property. This property is a map with an `AnnotationTypeData` as the key.
- *Inherited annotations* are annotations that belong to a parent item, but that we want to use on another item as well. Inherited annotations are saved as references to either a single annotation or to another annotation set. Thus, it is possible for an item to inherit multiple annotations of the same annotation type.

The `AnnotationData` class is also just a placeholder. It connects the annotation set and annotation type with a `ParameterValueData` object. This is the object that holds the actual annotation values.

Annotation types

Instances of the `AnnotationTypeData` class defines the various annotations. It must have a `valueType` property which cannot be changed. The value of this property controls which `ParameterValueData` subclass is used to store the annotation values, ie. `IntegerParameterValueData`, `StringParameterValueData`, etc. The `multiplicity` property holds the maximum allowed number of values for an annotation, or 0 if an unlimited number is allowed.

The `itemTypes` collection holds the codes for the types of items the annotation type can be used on. This is checked when new annotations are created but already existing annotations are not affected if the collection is modified.

Annotation types with the `protocolParameter` flag set are treated a bit differently. They will not show up as annotations to items with a type found in the `itemTypes` collection. Instead, a protocol parameter should be attached to a protocol. Then, when an item is using that protocol it becomes possible to add annotation values for the annotation types specified as protocol parameters. It doesn't matter if the item's type is found in the `itemTypes` collection or not.

The `options` collection is used to store additional options required by some of the value types, for example a max string length for string annotations or the max and min allowed value for integer annotations.

The `enumeration` property is a boolean flag indicating if the allowed values are predefined as an enumeration. In that case those values are found in the `enumerationValues` property. The actual subclass is determined by the `valueType` property.

Most of the other properties are hints to client applications how to render the input field for the annotation.

Units

Numerical annotation values can have units. A unit is described by a `UnitData` object. Each unit belongs to a `QuantityData` object which defines the class of units. For example, if the quantity is *weight*, we can have units, *kg*, *mg*, *µg*, etc. The `UnitData` contains a factor and offset that relates all units to a common reference defined by the `QuantityData` class. For example, *1 meter* is the reference unit for distance, and we have $1 \text{ meter} * 0.001 = 1 \text{ millimeter}$. In this case, the factor is *0.001* and the offset 0. Another example is the relationship between kelvin and Celsius, which is $1 \text{ kelvin} + 273.15 = 1 \text{ }^{\circ}\text{Celsius}$. Here, the factor is 1 and the offset is *+273.15*. The `UnitSymbolData` is used to make it possible to assign alternative symbols to a single unit. This is needed to simplify input where it may be hard to know what to type to get *m* or *°C*. Instead, *m2* and *C* can be used as alternative symbols.

The creator of an annotation type may select a `QuantityData`, which can't be changed later, and a default `UnitData`. When entering annotation values a user may select any unit for the selected quantity (unless annotation type owner has limited this by selecting `usableUnits`). Before the values are stored in the database, they are converted to the default unit. This makes it possible to compare and filter on annotation values using different units. For example, filtering with `>5mg` also finds items that are annotated with `2g`.

The core should automatically update the stored annotation values if the default unit is changed for an annotation type, or if the reference factor for a unit is changed.

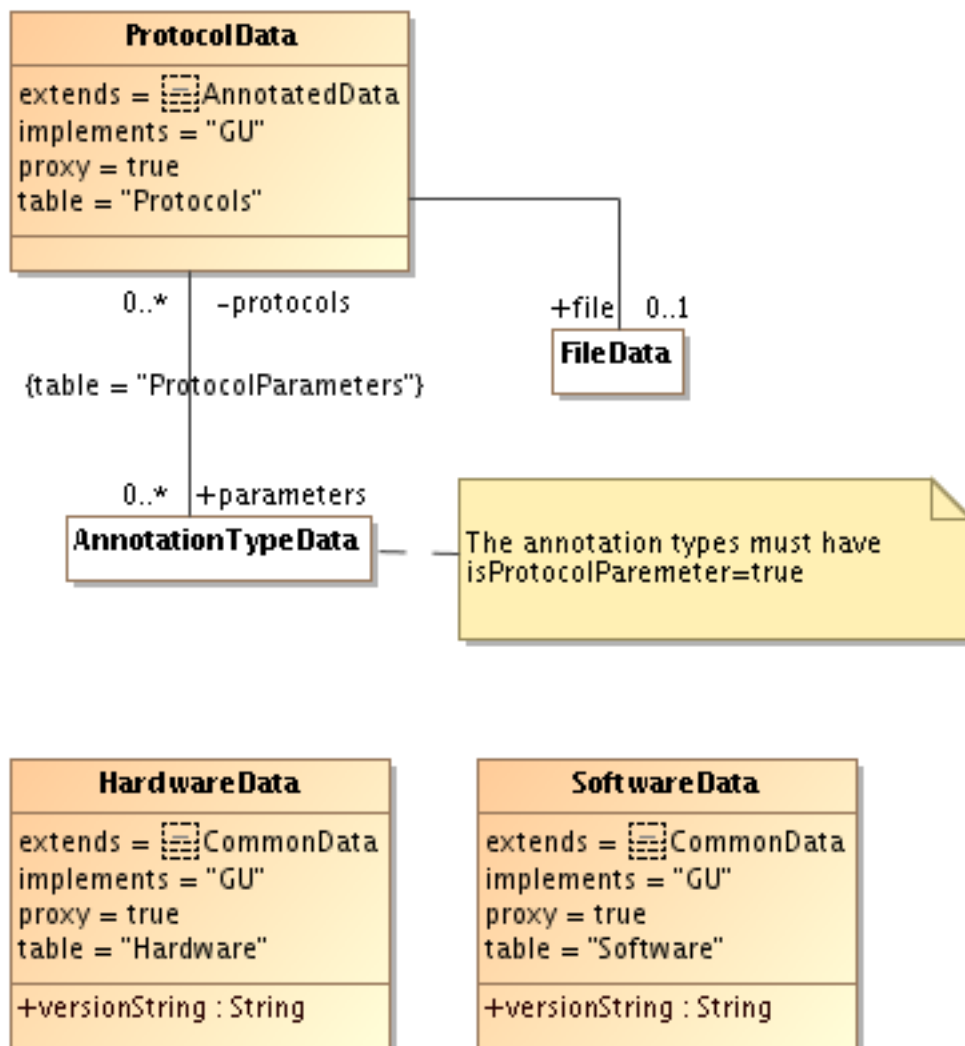
Categories

The `AnnotationTypeCategoryData` class defines categories that are used to group annotation types that are related to each other. This information is mainly useful for client applications when displaying forms for annotating items, that wish to provide a clearer interface when there are many (say 50+) annotations type for an item. An annotation type can belong to more than one category.

28.2.10. Protocols, hardware and software

This section gives an overview of how protocols that describe various processes, such as sampling, extraction and scanning, are used in BASE.

Figure 28.11. Protocols, hardware and software



Protocols

A protocol is something that defines a procedure or recipe for some kind of action, such as sampling, extraction and scanning. The subtype of the protocol is used to determine what the protocol is used for. In BASE we only store a short name and description. It is possible to attach a file that provides a longer description of the procedure.

Parameters

The procedure described by the protocol may have parameters that are set independently each time the protocol is used. It could for example be a temperature, a time or something else. The definition of parameters is done by creating annotation types and attaching them to the protocol. It is only possible to attach annotation types which has the `protocolParameter` property set to `true`. The same annotation type can be used for more than one protocol, but only do this if the parameters actually has the same meaning.

Hardware and software

BASE is pre-installed with a set of subtypes for hardware and software. They are typically used to filter the registered hardware and software depending on what a user is doing. For example, when adding raw data to BASE a user can select a scanner. The GUI will display the hardware that has been registered as *scanner* subtype. Other subtypes are *hybridization station* and *print robot*. An administrator may register more subtypes.

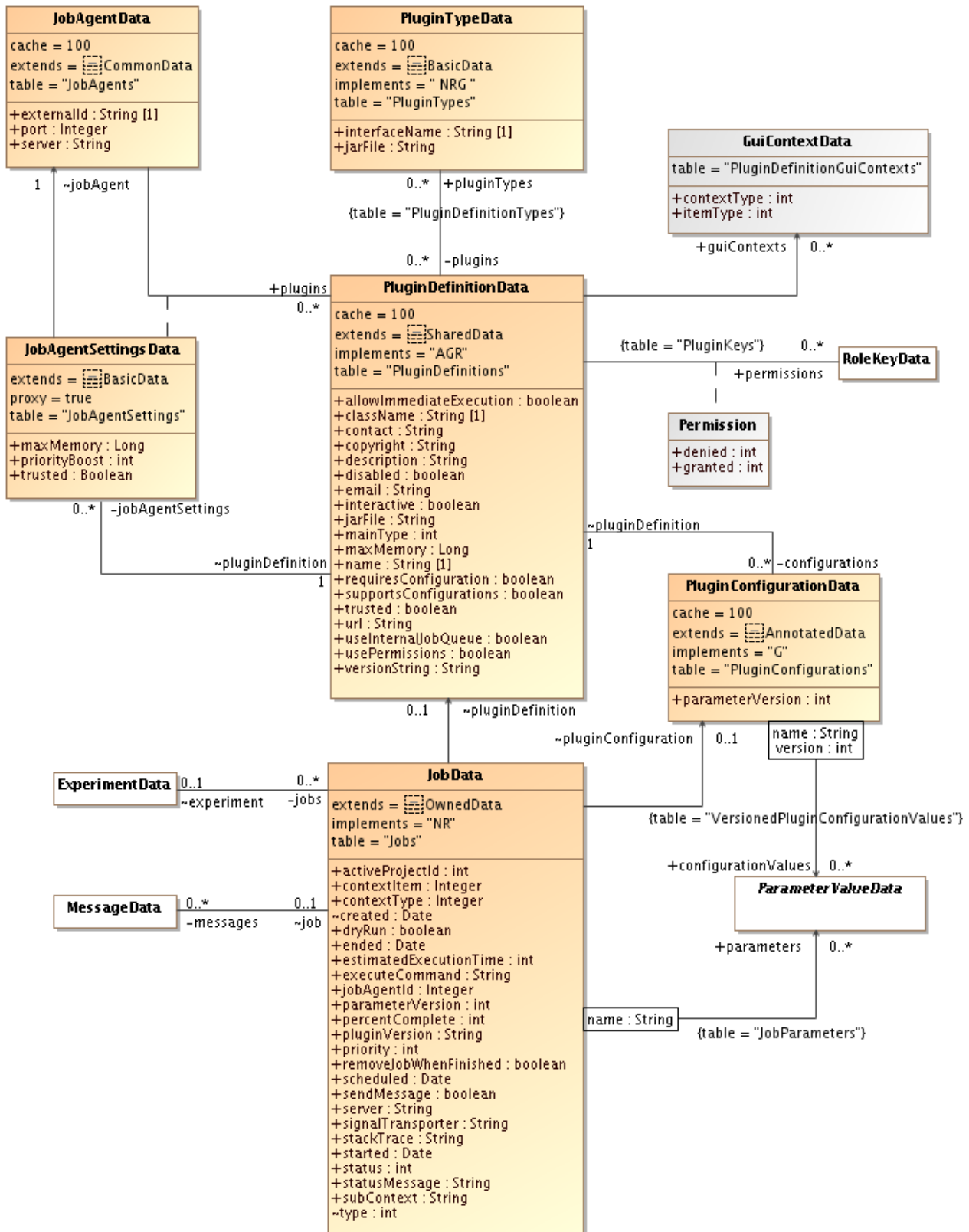
28.2.11. Plug-ins, jobs and job agents

This section gives an overview of plug-ins, jobs and job agents.

See also

- Section 21.1, “Managing plug-ins and extensions” (page 178)
- Section 20.3, “Installing job agents” (page 171)

Figure 28.12. Plug-ins, jobs and job agents



Plug-ins

The `PluginDefinitionData` holds information of the installed plugin classes. Much of the information is copied from the plug-in itself from the `About` object and by checking which interfaces it implements.

There are five main types of plug-ins:

- `IMPORT` (`mainType = 1`): A plug-in that imports data to BASE.
- `EXPORT` (`mainType = 2`): A plug-in that exports data from BASE.
- `INTENSITY` (`mainType = 3`): A plug-in that calculates intensity values from raw data.
- `ANALYZE` (`mainType = 4`): A plug-in that analyses data.
- `OTHER` (`mainType = 5`): Any other plug-in.

A plug-in may have different configurations. The flags `supportsConfigurations` and `requiresConfiguration` are used to specify if a plug-in must have or can't have any configurations. Configuration parameter values are versioned. Each time anyone updates a configuration the version number is increased and the parameter values are stored as a new entity. This is required because we want to be able to know exactly which parameters a job were using when it was executed. When a job is created we also store the parameter version number (`JobData.parameterVersion`). This means that even if someone changes the configuration later we will always know which parameters the job used.

The `PluginTypeData` class is used to group plug-ins that share some common functionality, by implementing additional (optional) interfaces. For example, the `AutoDetectingImporter` should be implemented by import plug-ins that supports automatic detection of file formats. Another example is the `AnalysisFilterPlugin` interface which should be implemented by all analysis plug-ins that only filters data.

Jobs

A job represents a single invocation of a plug-in to do some work. The `JobData` class holds information about this. A job is usually executed by a plug-in, but doesn't have to be. The `status` property holds the current state of a job.

- `UNCONFIGURED` (`status = 0`): The job is not yet ready to be executed.
- `WAITING` (`status = 1`): The job is waiting to be executed.
- `PREPARING` (`status = 5`): The job is about to be executed but hasn't started yet.
- `EXECUTING` (`status = 2`): The job is currently executing.
- `ABORTING` (`status = 6`): The job is executing but an `ABORT` signal has been sent requesting it to abort and finish.
- `DONE` (`status = 3`): The job finished successfully.
- `ERROR` (`status = 4`): The job finished with an error.

Job agents

A job agent is a program running on the same or a different server that is regularly checking for jobs that are waiting to be executed. The `JobAgentData` holds information about a job agent and

the `JobAgentSettingsData` links the agent with the plug-ins the agent is able to execute. The job agent will only execute jobs that are owned by users or projects that the job agent has been shared to with at least use permission. The `priorityBoost` property can be used to give specific plug-ins higher priority. Thus, for a job agent it is possible to:

- Specify exactly which plug-ins it will execute. For example, it is possible to dedicate one agent to only run one plug-in.
- Give some plug-ins higher priority. For example a job agent that is mainly used for importing data should give higher priority to all import plug-ins. Other types of jobs will have to wait until there are no more data to be imported.
- Specify exactly which users/groups/projects that may use the agent. For example, it is possible to dedicate one agent to only run jobs for a certain project.



Biomaterials

There are three main types of biomaterials: `BioSourceData`, `SampleData` and `ExtractData`. All types of are derived from the base class `BioMaterialData`. The reason for this is that they all share common functionality such as pooling and events. By using a common base class we do not have to create duplicate classes for keeping track of events and parents.

The `BioSourceData` is the simplest of the biomaterials. It cannot have parents and can't participate in events. It's only used as a (non-required) parent for samples.

The `MeasuredBioMaterialData` class is used as a base class for the other biomaterial types. It introduces quantity measurements and can store original and remaining quantities. They are both optional. If an original quantity has been specified the core automatically calculates the remaining quantity based on the events a biomaterial participates in.

All measured biomaterial have at least one event associated with them, the *creation event*, which holds information about the creation of the biomaterial. A measured biomaterial can be created in three ways:

- From a single item of the same type or the parent type. Biosource is the parent type of samples and sample is the parent type of extracts. The `parentType` property must be set to the correct parent type and the `parent` property is set to point to the parent item. The parent information is also always duplicated in the `sources` collection of the `BioMaterialEventData` object representing the creation event. It is the responsibility of the core to make sure that everything is properly synchronized and that remaining quantities are calculated.
- From multiple items of the same type, i.e pooling. In this case the `parentType` property is set, but the `parent` property is null. All source biomaterials are contained in the `sources` collection. The core is still responsible for keeping everything synchronized and to update remaining quantities.
- As a standalone biomaterial without parents. The `parentType` property should be null, as should the `parent` property and the `sources` collection.

Bioplates and plate types

Biomaterial (except biosource) may optionally be placed on `BioPlateData`s. A bioplate is something that collects multiple biomaterial as a unit. A bioplate typically has a `PlateGeometryData` that determines the number of locations on the plate (`BioWellData`). A single well can hold a single biomaterial at a time.

The bioplate must be of a specific `BioPlateTypeData`. The type can be used to put limitations on how the plate can be used. For example, it can be limited to a single type of biomaterial. It is also possible to lock wells so that the biomaterial in them can't be changed. Supported lock modes are:

- *Unlocked*: Wells are unlocked and the biomaterial may be changed any number of times.
- *Locked-after-move*: The well is locked after it has been used one time and the biomaterial that was put in it has been moved to another plate.
- *Locked-after-add*: The well is locked after biomaterial has been put into it. It is not possible to remove the biomaterial.
- *Locked-after-create*: The well is locked once it has been created. Biomaterial must be put into wells before the plate is saved to the database.

Biomaterial and plate events

An event represents something that happened to one or more biomaterials, for example the creation of another biomaterial. The `BioMaterialEventData` holds information about entry and event dates,

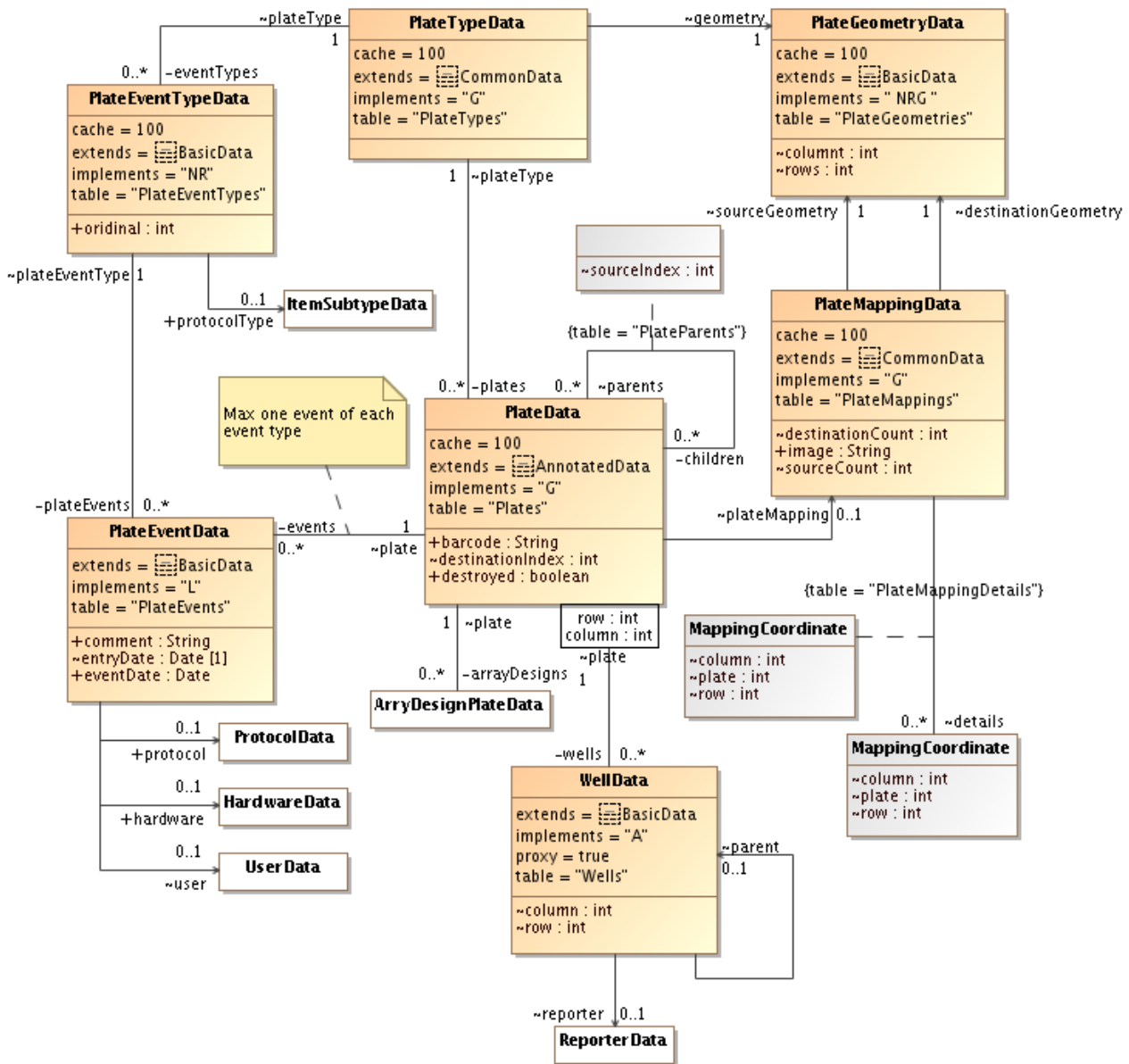
protocols used, the user who is responsible, etc. There are three types of events represented by the `eventType` property.

1. *Creation event*: This event represents the creation of a (measured) biomaterial. The `sources` collection contains information about the biomaterials that were used to create the new biomaterial. All sources must be of the same type. There can only be one source of the parent type. These rules are maintained by the core.
2. *Bioassay event*: This event represents the creation of a bioassay. This event type is needed because we want to keep track of quantities for extracts. This event has a `PhysicalBioAssayData` as a product instead of a biomaterial. The `sources` collection can only contain extracts. If the bioassay can hold extracts in multiple positions the `position` property in `BioMaterialEventSourceData` can be used to track which extract that was put in each position. It is allowed to put multiple extracts in the same position, but then the usually need to use different `TagData` items. However, this is not enforced by the core.
3. *Other event*: This event represents some other important information about a single biomaterial that affected the remaining quantity. This event type doesn't have any sources.

It is also possible to register events that applies to one or more bioplates using the `BioPlateEventData` class. The `BioPlateEventParticipantData` class holds information about each plate that is part of the event. The `role` property is a textual description of what happened to the plate. Eg. a move event, may have one *source* plate and one *destination* plate. It is recommended (but not required) that all biomaterial that are affected by the plate event are linked via a `BioMaterialEventData` to a `BioPlateEventParticipantData`. This will make it easier to keep track of the history of individual biomaterial items. Biomaterial events that are linked in this way are also automatically updated if the bioplate event is modified (eg. selecting a protocol, event date, etc.).

28.2.13. Array LIMS - plates

Figure 28.14. Array LIMS - plates



Plates

The **PlateData** is the main class holding information about a single plate. The associated **PlateGeometryData** defines how many rows and columns there are on a plate. Since this information is used to create wells, and for various other checks it is not possible to change the number of rows or columns once a geometry has been created.

All plates must have a **PlateTypeData** which defines the geometry and a set of event types (see below).

If the destroyed flag of a plate is set it is not allowed to use the plate for a plate mapping or to create array designs. However, it is possible to change the flag to not destroyed.

The barcode is intended to be used as an external identifier of the plate. But, the core doesn't care about the value or if it is unique or not.

Plate events

The plate type defines a set of `PlateEventTypeData` objects, each one representing a particular event a plate of this type usually goes through. For a plate of a certain type, it is possible to attach exactly one event of each event type. The event type defines an optional protocol type, which can be used by client applications to filter a list of protocols for the event. The core doesn't check that the selected protocol for an event is of the same protocol type as defined by the event type.

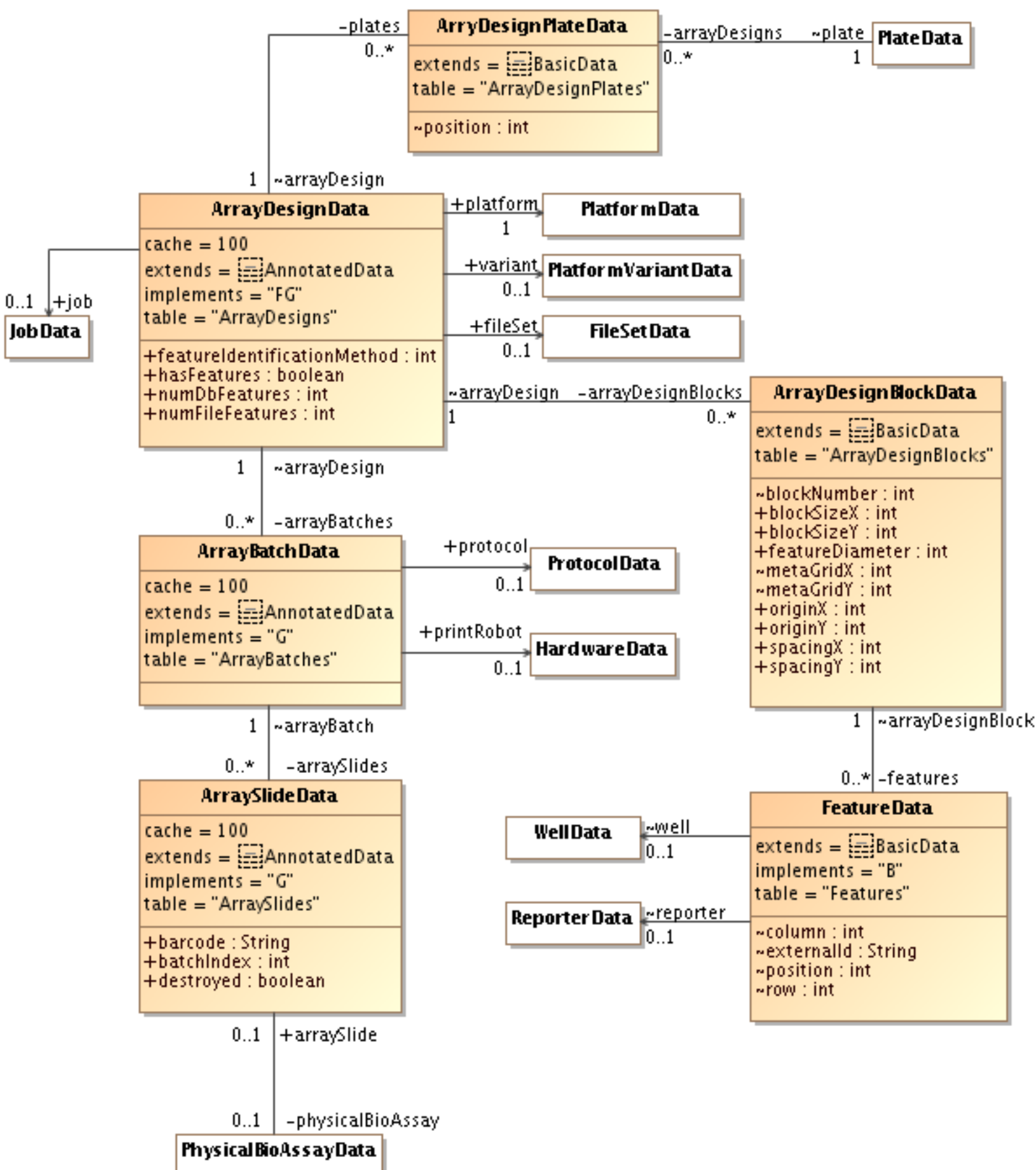
The ordinal value can be used as a hint to client applications in which order the events actually are performed in the lab. The core doesn't care about this value or if several event types have the same value.

Plate mappings

A plate can be created either from scratch, with the help of the information in a `PlateMappingData`, from a set of parent plates. In the first case it is possible to specify a reporter for each well on the plate. In the second case the mapping code creates all the wells and links them to the parent wells on the parent plates. Once the plate has been saved to the database, the wells cannot be modified (because they are used downstream for various validation, etc.)

The details in a plate mapping are simply coordinates that for each destination plate, row and column define a source plate, row and column. It is possible for a single source well to be mapped to multiple destination wells, but for each destination well only a single source well can be used.

Figure 28.15 Array LIMS Data Tables



Array designs

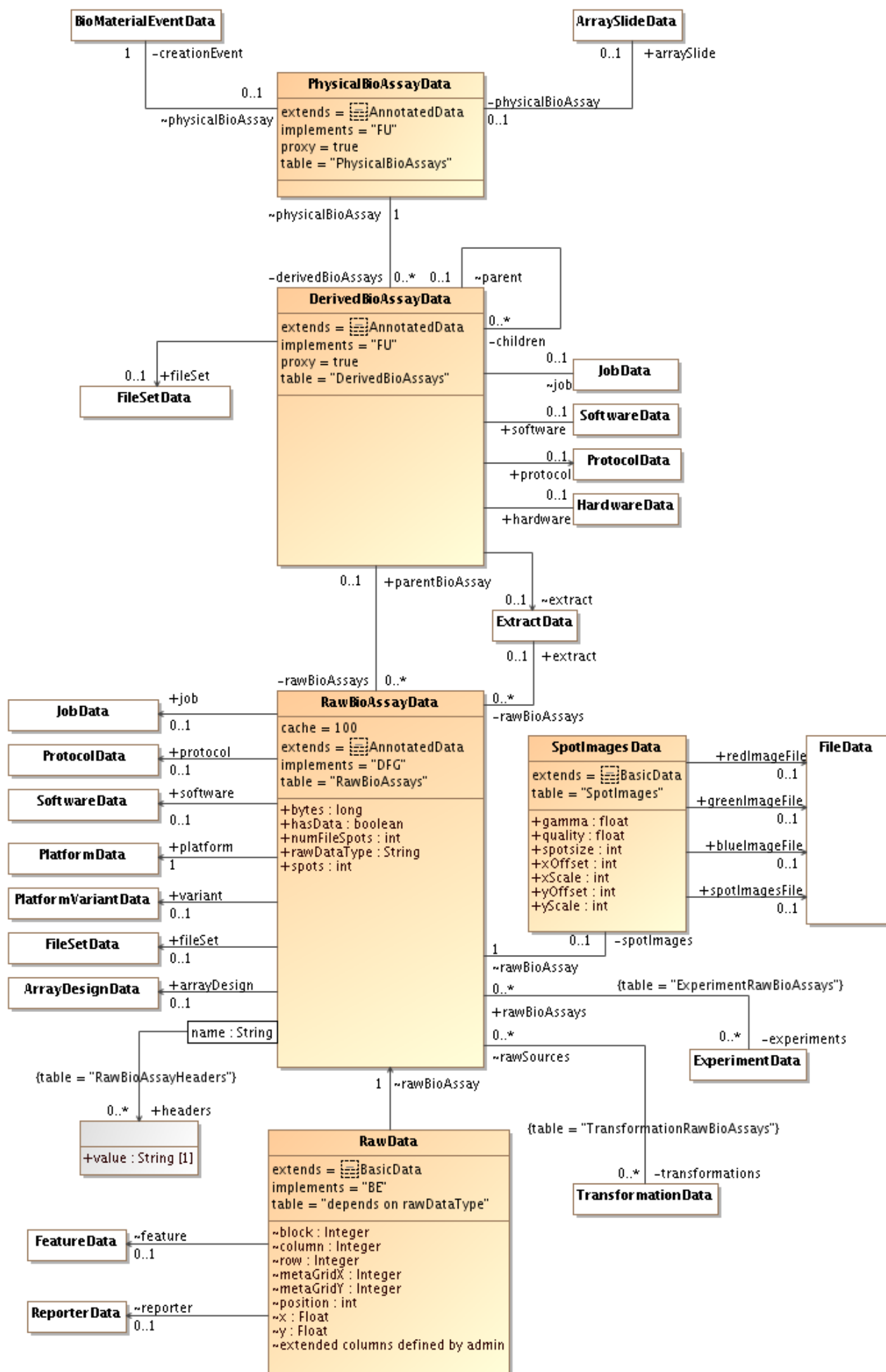
Array designs are stored in `ArrayDesignData` objects and can be created either as standalone designs or from plates. In the first case the features on an array design are described by a reporter map. A reporter map is a file that maps a coordinate (block, meta-grid, row, column), position or an external ID on an array design to a reporter. Which method to use is given by the `ArrayDesign.featureIdentificationMethod` property. The coordinate system on an array design is divided into blocks. Each block can be identified either by a `blockNumber` or by meta coordinates. This information is stored in `ArrayDesignBlockData` items. Each block contains several `FeatureData` items, each one identified by a row and column coordinate. Platforms that doesn't divide the array design into blocks or doesn't use the coordinate system at all must still create a single super-block that holds all features.

Array designs that are created from plates use a print map file instead of a reporter map. A print map is similar to a plate mapping but maps features (instead of wells) to wells. The file should specify which plate and well a feature is created from. Reporter information will automatically be copied by BASE from the well.

It is also possible to skip the importing of features into the database and just keep the data in the original files instead. This is typically done for Affymetrix CDF files.

Array slides

The `ArraySlideData` represents a single array. Arrays are usually printed several hundreds in a batch, represented by a `ArrayBatchData` item. The `batchIndex` is the ordinal number of the array in the batch. The barcode can be used as a means for external programs to identify the array. BASE doesn't care if a value is given or if they are unique or not. If the `destroyed` flag is set it prevents a slide from being used by a hybridization.



Physical bioassays

A `PhysicalBioAssayData` item connect the array slides from the Array LIMS part with extracts from the biomaterials part. The `creationEvent` is used to register which extracts that were used on the bioassay. The relation to slides is a one-to-one relation. A slide can only be used on a single physical bioassay and a bioassay can only use a single slide. The relation is optional from both sides.

Further processing of the bioassay is registered as a series of `DerivedBioAssayData` items. For microarray experiments the first step is typically a scanning of the hybridization. Information about the software/hardware and protocol used can be registered. Any data files generated by the process can be registered with the `FileSetData` item. If more than one processing step is required child derived bioassays can be created that describe each additional step.

If the root physical bioassay has multiple extracts in multiple positions, the `extract` property of a derived bioassay is used to link with the extract that the specific derived bioassay represents. If the link is null the derived bioassay represents all extracts on the physical bioassay.

Raw data

A `RawBioAssayData` object represents the raw data that is produced by analysing the data from the physical bioassay. You may register which software that was used, the protocol and any parameters (through the annotation system).

Files with the analysed data values can be attached to the associated `FileSetData` object. The platform and, optionally, the variant has information about the file types that can be used for that platform. If the platform file types support metadata extraction, headers, the number of spots, and other information may be automatically extracted from the raw data file(s).

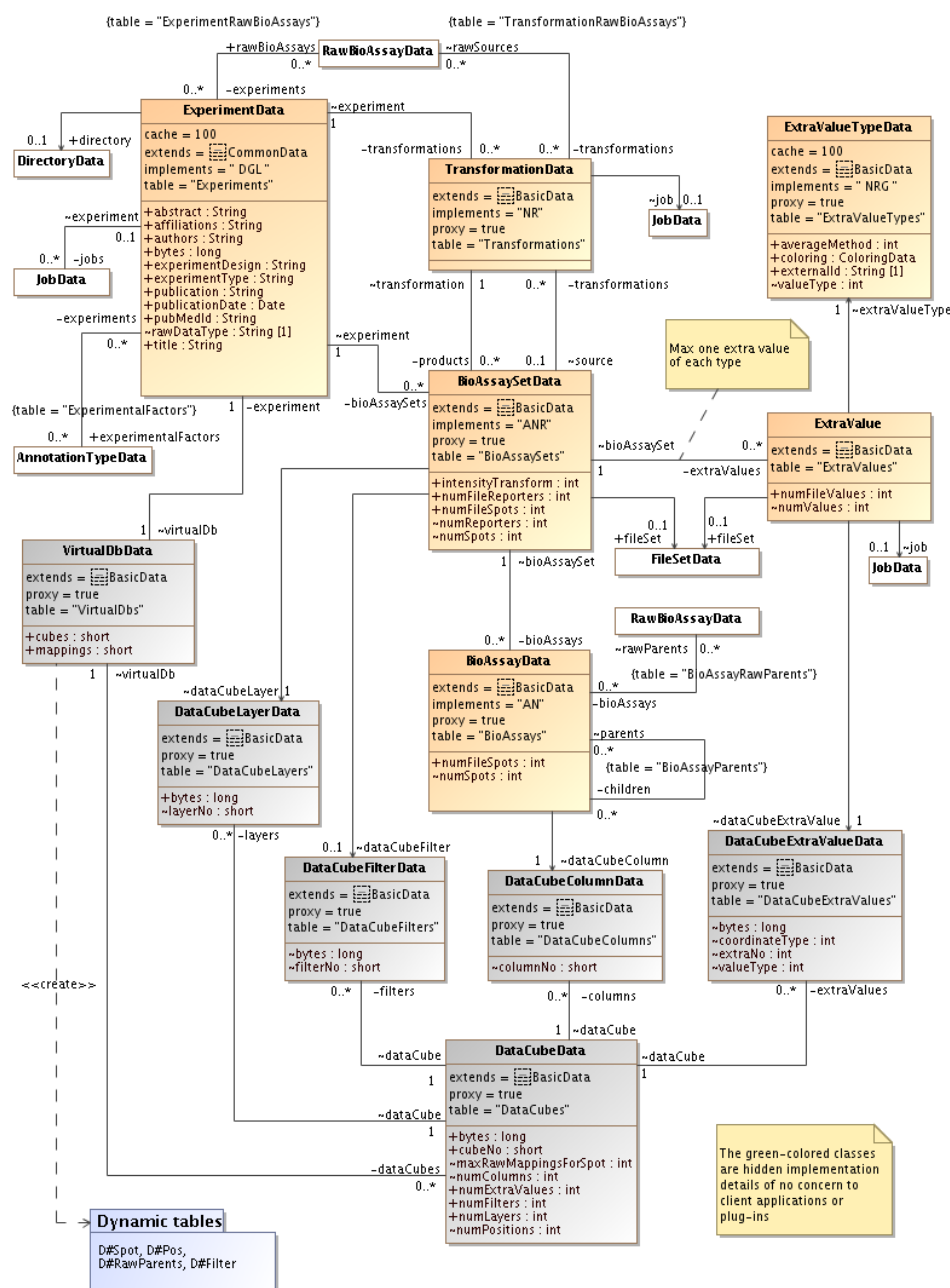
If the platform support it, raw data can also be imported into the database. This is handled by `batchers` and `RawData` objects. Which table to store the data in depends on the `rawDataType` property. The properties shown for the `RawData` class in the diagram are the mandatory properties. Each raw data type defines additional properties that are specific to that raw data type.

Spot images

Spot images can be created if you have the original image files. BASE can use 1-3 images as sources for the red, green and blue channel respectively. The creation of spotimages requires that x and y coordinates are given for each raw data spot. The scaling and offset values are used to convert the coordinates to pixel coordinates. With this information BASE is able to cut out a square from the source images that, theoretically, contains a specific spot and nothing else. The spot images are gamma-corrected independently and then put together into PNG images that are stored in a zip file.

28.2.16. Experiments and analysis

Figure 28.17. Experiments



Experiments

The `ExperimentData` class is used to collect information about a single experiment. It links to any number of `RawBioAssayData` items, which must all be of the same `RawDataType`.

Annotation types that are needed in the analysis must be connected to the experiment as experimental factors and the annotation values should be set on or inherited by each raw bioassay that is part of the experiment.

The directory connected to the experiment is the default directory where plugins that generate files should store them.

Bioassay sets, bioassays and transformations

Each line of analysis starts with the creation of a *root* `BioAssaySetData`, which holds the intensities calculated from the raw data. A bioassayset can hold one intensity for each channel. The number of channels is defined by the raw data type. For each raw bioassay used a `BioAssayData` is created.

Information about the process that calculated the intensities are stored in a `TransformationData` object. The root transformation links with the raw bioassays that are used in this line of analysis and to a `JobData` which has information about which plug-in and parameters that was used in the calculation.

Once the root bioassayset has been created it is possible to again apply a transformation to it. This time the transformation links to a single source bioassayset instead of the raw bioassays. As before, it still links to a job with information about the plug-in and parameters that does the actual work. The transformation must make sure that new bioassays are created and linked to the bioassays in the source bioassayset. This above process may be repeated as many times as needed.

Data to a bioassay set can only be added to it before it has been committed to the database. Once the transaction has been committed it is no longer possible to add more data or to modify existing data.

Virtual databases, datacubes, etc.

The above processes requires a flexible storage solution for the data. Each experiment is related to a `VirtualDb` object. This object represents the set of tables that are needed to store data for the experiment. All tables are created in a special part of the BASE database that we call the *dynamic database*. In MySQL the dynamic database is a separate database, in Postgres it is a separate schema.

A virtual database is divided into data cubes. A data cube can be seen as a three-dimensional object where each point can hold data that in most cases can be interpreted as data for a single spot from an array. The coordinates to a point is given by *layer*, *column* and *position*. The layer and column coordinates are represented by the `DataCubeLayerData` and `DataCubeColumnData` objects. The position coordinate has no separate object associated with it.

Data for a single bioassay set is always stored in a single layer. It is possible for more than one bioassay set to use the same layer. This usually happens for filtering transformations that doesn't modify the data. The filtered bioassay set is then linked to a `DataCubeFilterData` object, which has information about which data points that passed the filter.

All data for a bioassay is stored in a single column. Two bioassays in different bioassaysets (layers) can only have the same column if one is the parent of the other.

The position coordinate is tied to a reporter.

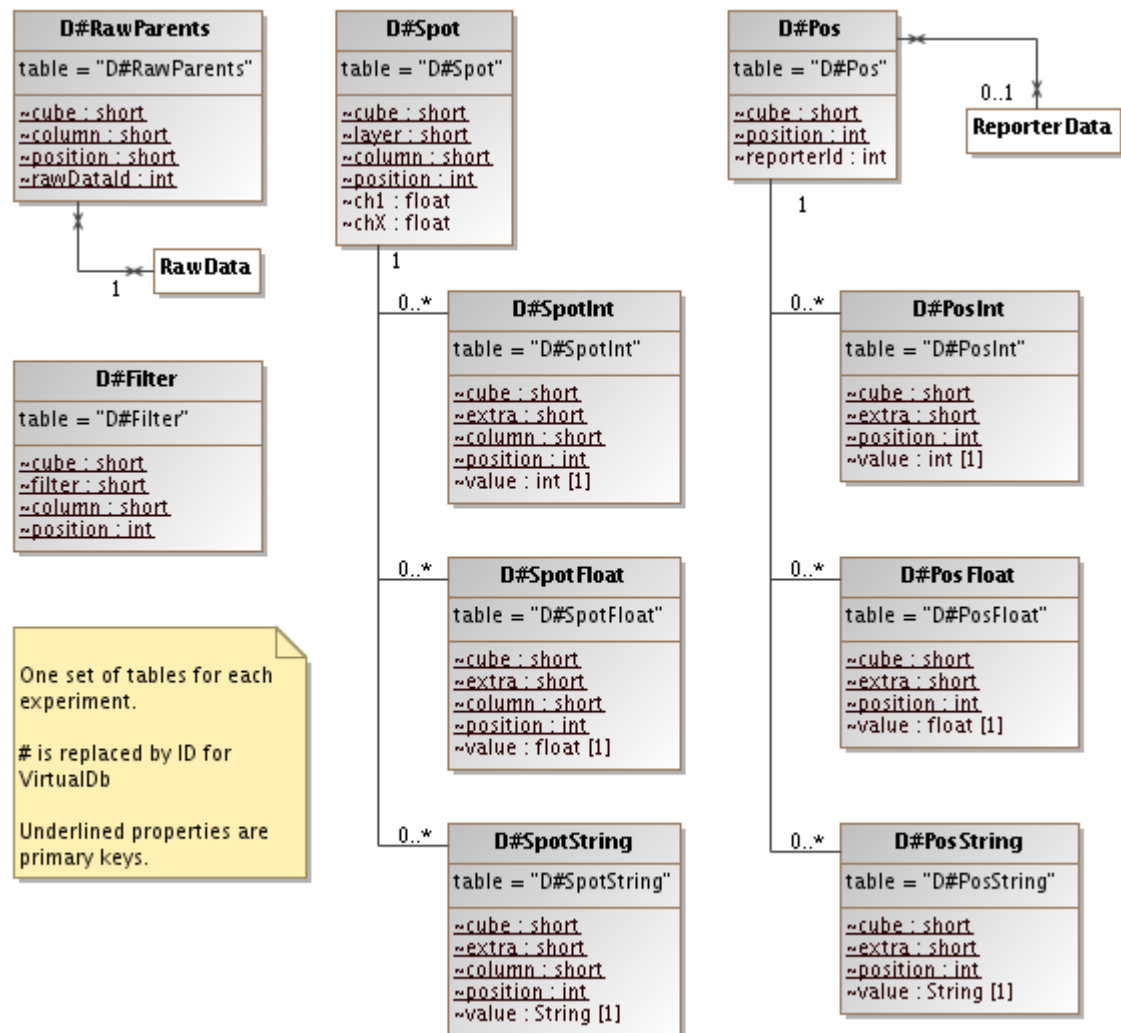
A child bioassay set may use the same data cube as it's parent bioassay set if all of the following conditions are true:

- All positions are linked to the same reporter as the positions in the parent bioassay set.
- All data points are linked to the same (possible many) raw data spots as the corresponding data points in the parent bioassay set.
- The bioassays in the child bioassay set each have exactly one parent in the parent bioassay set. No parent bioassay may be the parent of more than one child bioassay.

If any of the above conditions are not true, a new data cube must be created for the child bioassay set.

The dynamic database

Figure 28.18. The dynamic database



Each virtual database consists of several tables. The tables are dynamically created when needed. For each table shown in the diagram the # sign is replaced by the id of the virtual database object at run time.

There are no classes in the data layer for these tables and they are not mapped with Hibernate. When we work with these tables we are always using batcher classes and queries that works with integer, floats and strings.

The D#Spot table

This is the main table which keeps the intensities for a single spot in the data cube. Extra values attached to the spot are kept in separate tables, one for each type of value (D#SpotInt, D#SpotFloat and D#SpotString).

The D#Pos table

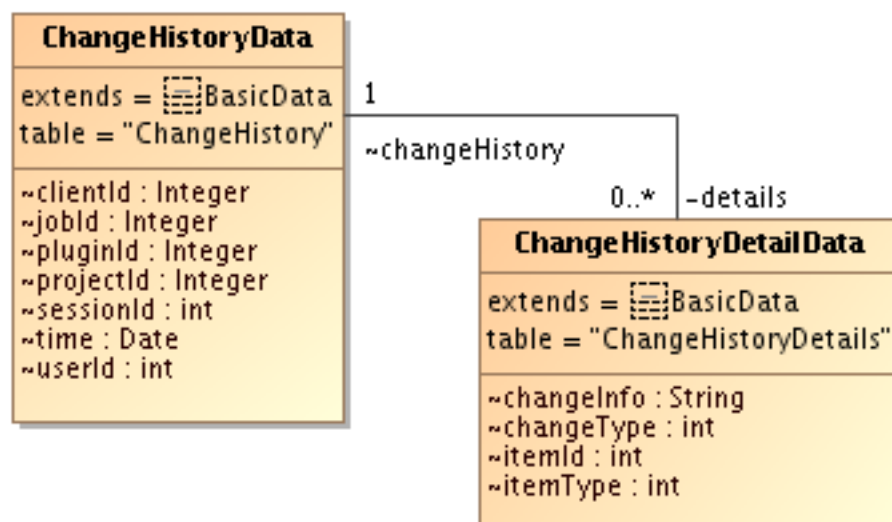
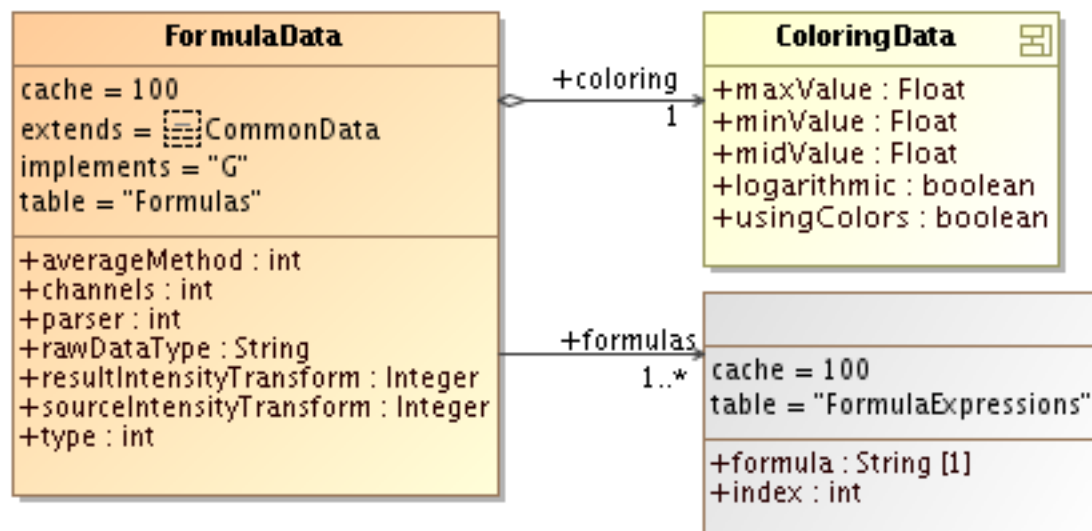
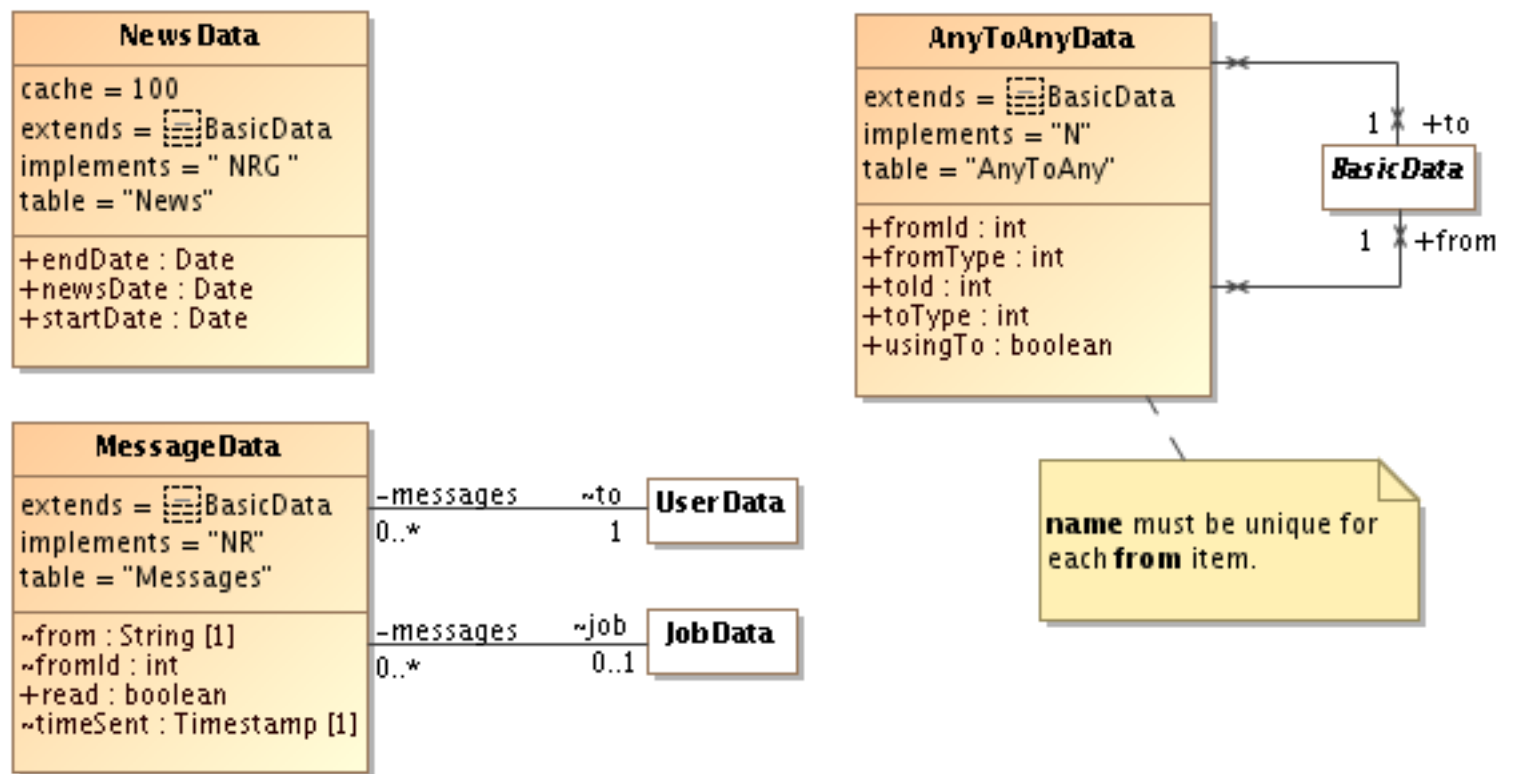
This table stores the reporter id for each position in a cube. Extra values attached to the position are kept in separate tables, one for each type of value (D#PosInt, D#PosFloat and D#PosString).

The D#Filter table

This table stores the coordinates for the spots that remain after filtering. Note that each filter is related to a bioassayset which gives the cube and layer values. Each row in the filter table then adds the column and position allowing us to find the spots in the D#Spot table.

The D#RawParents table

This table holds mappings for a spot to the raw data it is calculated from. We don't need the layer coordinate since all layers in a cube must have the same mapping to raw data.



28.3. The Core API

This section gives an overview of various parts of the core API.

28.3.1. Authentication and sessions

This documentation is only available in the old format which may not be up-to-date with the current implementation. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/core/authentication.html>

28.3.2. Access permissions

This documentation is only available in the old format which may not be up-to-date with the current implementation. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/core/accesspermissions.html>

28.3.3. Data validation

TODO

28.3.4. Transaction handling

TODO

28.3.5. Create/read/write/delete operations

This documentation is only available in the old format which may not be up-to-date with the current implementation. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/core/itemhandling.html>

28.3.6. Batch operations

This documentation is only available in the old format which may not be up-to-date with the current implementation. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/core/batchprocessing.html>

28.3.7. Quota

TODO

28.3.8. Plugin execution / job queue

This documentation is only available in the old format which may not be up-to-date with the current implementation. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/core/plugins.html>

28.3.9. Using files to store data

BASE has support for storing data in files instead of importing it into the database. Files can be attached to any item that implements the `FileStoreEnabled` interface. For example, `RawBioAssay`, `ArrayDesign` and a few other classes. The ability to store data in files is not a replacement for

storing data in the database. It is possible (for some platforms/raw data types) to have data in files and in the database at the same time. There are three cases:

- Data in files only
- Data in the database only
- Data in both files and in the database

Not all three cases are supported for all types of data. This is controlled by the `Platform` class, which may disallow that data is stored in the database. To check this call `Platform.isFileOnly()` and/or `Platform.getRawDataType()`. If the `isFileOnly()` method returns `true`, the platform can't store data in the database. If the value is `false` more information can be obtained by calling `getRawDataType()`, which may return:

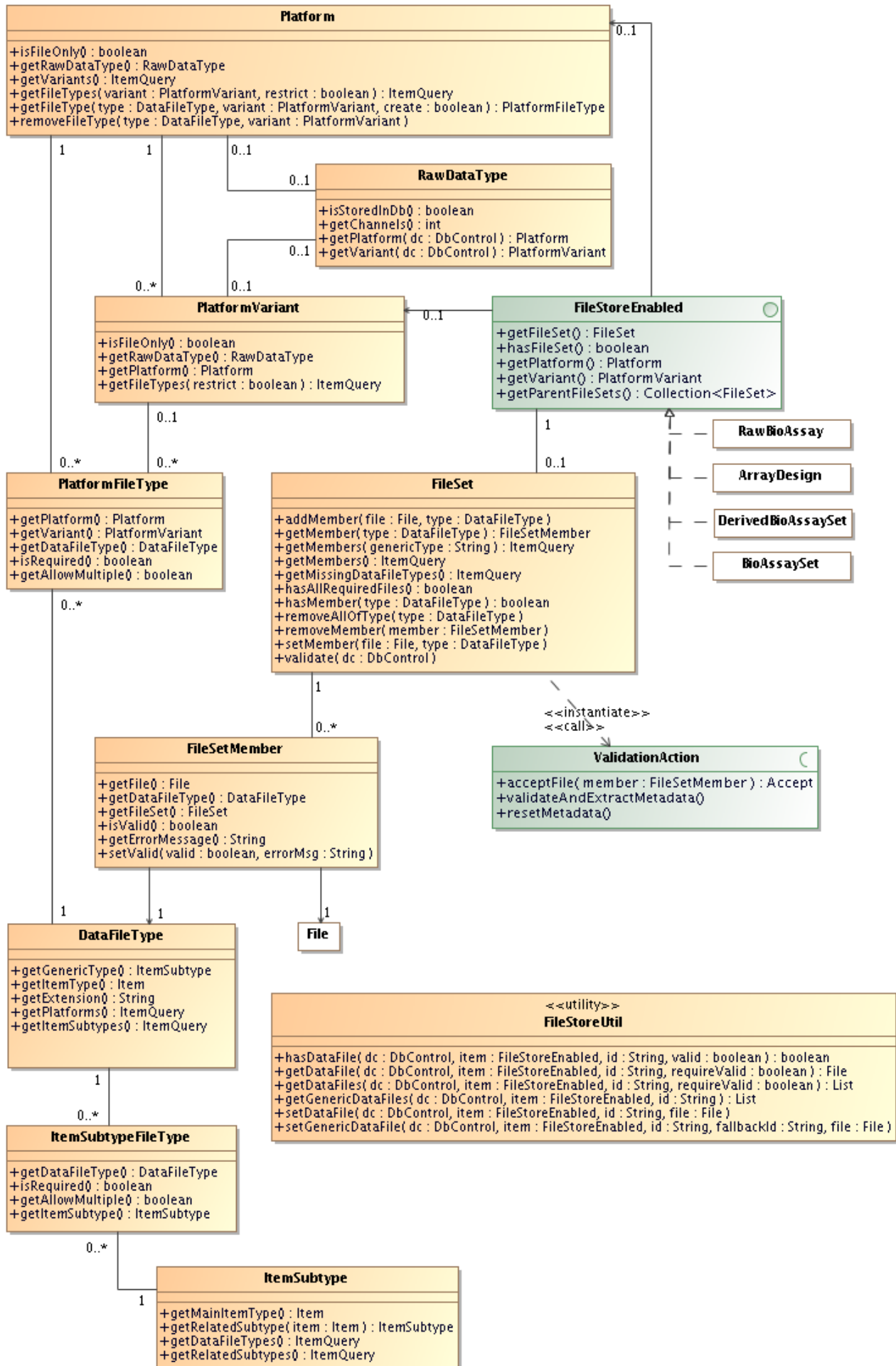
- `null`: The platform can store data with any raw data type in the database.
- A `RawDataType` that has `isStoredInDb() == true`: The platform can store data in the database but only data with the specified raw data type.
- A `RawDataType` that has `isStoredInDb() == false`: The platform can't store data in the database.

Some `FileStoreEnabled` items doesn't have a platform (for example, `DerivedBioAssay`). In this case, the file storage ability is controlled by the subtype of the item. See `getDataFileTypes()` method in the `ItemSubtype` class.

For backwards compatibility reasons, each `Platform` that can only store data in files will create "virtual" raw data type objects internally. These raw data types all return `false` from the `RawDataType.isStoredInDb()` method. They also have a back-link to the platform/variant that created it: `RawDataType.getPlatform()` and `RawDataType.getVariant()`. These two methods will always return `null` when called on a raw data type that can be stored in the database.

See also

- Section 28.2.7, “Experimental platforms and item subtypes” (page 304)
- Section 26.8.8, “Fileset validators” (page 281)
- Section D.1, “Default platforms and variants installed with BASE” (page 430)



This is rather large set of classes and methods. The ultimate goal is to be able to create links between a `FileStoreEnabled` item and `File` items and to provide some metadata about the files. The `FileStoreUtil` class is one of the most important ones. It is intended to make it easy for plug-in (and other) developers to access the files without having to mess with platform or file type objects. The API is best described by a set of use-case examples.

Use case: Asking the user for files for a given item

A client application must know what types of files it makes sense to ask the user for. In some cases, data may be split into more than one file so we need a generic way to select files.

Given that we have a `FileStoreEnabled` item we want to find out which `DataFileType` items that can be used for that item. The `Base.getDataFileTypes()` can be used for this. You'll need to supply information about the platform, variant and subtype of the item. The method will create a query that returns a list of `DataFileType` items, each one representing a specific file type that we should ask the user about. Examples:

1. The `Affymetrix` platform defines `CEL` as a raw data file and `CDF` as an array design (reporter map) file. If we have a `RawBioAssay` the query will only return the `CEL` file type and the client can ask the user for a `CEL` file.
2. The `Generic` platform defines `PRINT_MAP` and `REPORTER_MAP` for array designs. If we have an `ArrayDesign` the query will return those two items.
3. The `Scan` subtype defines `MICROARRAY_IMAGE` for derived bioassays.

It might also be interesting to know the currently selected file for each file type and if the file is required and if multiple files are allowed. Here is a simple code example that may be useful to start from:

```
DbControl dc = ...
FileStoreEnabled item = ...
Platform platform = item.getPlatform();
PlatformVariant variant = item.getVariant();
Itemsubtype subtype = item instanceof Subtypable ?
    ((Subtypable)item).getItemSubtype() : null;

// Get list of DataFileTypes used by the platform
ItemQuery<DataFileType> query =
    Base.getDataFileTypes(item.getType(), item, platform, variant, subtype);
List<DataFileType> types = query.list(dc);

// Always check hasFileSet() method first to avoid
// creating the file set if it doesn't exists
FileSet fileSet = item.hasFileSet() ?
    null : item.getFileSet();

for (DataFileType type : types)
{
    // Get the current file, if any
    FileSetMember member = fileSet == null || !fileSet.hasMember(type) ?
        null : fileSet.getMember(type);
    File current = member == null ?
        null : member.getFile();

    // Check if a file is required by the platform/subtype
    PlatformFileType pft = platform == null ?
        null : platform.getFileType(type, variant, false);
    ItemSubtypeFileType ift = subtype == null ?
        null : subtype.getAssociatedDataFileType(type, false);
    boolean isRequired = pft == null ?
        false : pft.isRequired();
    isRequired |= ift == null ?
        false : ift.isRequired();
}
```

```
// Now we can do something with this information to
// let the user select a file ...
}
```

Also remember to catch `PermissionDeniedException`

The above code may look complicated, but this is mostly because of all checks for `null` values. Remember that many things are optional and may return `null`. Another thing to look out for is `PermissionDeniedException`:s. The logged in user may not have access to all items. The above example doesn't include any code for this since it would have made it too complex.

Use case: Link, validate and extract metadata from the selected files

When the user has selected the file(s) we must store the links to them in the database. This is done with a `FileSet` object. A file set can contain any number of files. Call `FileSet.setMember()` or `FileSet.addMember()` to store a file in the file set. If a file already exists for the given file type it is replaced if the `setMember` method is called. The following program example assumes that we have a map where `File`:s are related to `DataFileType`:s. When all files have been added we call `FileSet.validate()` to validate the files and extract metadata.

```
DbControl dc = ...
FileStoreEnabled item = ...
Map<DataFileType, File> files = ...

// Store the selected files in the fileset
FileSet fileSet = item.getFileSet();
for (Map.Entry<DataFileType, File> entry : files)
{
    DataFileType type = entry.getKey();
    File file = entry.getValue();
    fileSet.setMember(type, file);
}

// Validate the files and extract metadata
fileSet.validate(dc);
```

Validation and extraction of metadata is important since we want data in files to be equivalent to data in the database. The validation and metadata extraction is initiated by the core when the `FileSet.validate()` is called. The validation and metadata extraction is handled by extensions so the actual outcome depends on what has been installed on the BASE server.

Note

The `FileSet.validate()` method doesn't throw any exceptions. Instead, all validation errors are returned a list of `Throwable`:s. The validation result is also stored for each file and can be access with `FileSetMember.isValid()` and `FileSetMember.getErrorMessage()`.

Here is the general outline of what is going on in the core:

1. The core calls the main `ExtensionsManager` and initiates the action factory for all file set validator extensions.
2. After inspecting the current item and file set, the factories create one or more `ValidationAction`:s.
3. For each file in the file set, the `ValidationAction.acceptFile()` method is called on each action, which is supposed to either accept or deny validation of the file.
4. If the file is accepted the `ValidationAction.validateAndExtractMetadata()` method is called.

Only one instance of each validator class is created

The validation is not done until all files have been added to the fileset. If the same validator is used for more than one file, the same instance is reused. Eg. the `acceptFile()` is called one time for each file. Depending on the return value, the `validateAndExtractMetadata()` may be called either immediately or not until all files have been processed.

Use case: Import data into the database

This should be done by existing plug-ins in the same way as before. A slight modification is needed since it is good if the importers are made aware of already selected files in the `FileSet` to provide good default values. The `FileStoreUtil` class is very useful in cases like this:

```
RawBioAssay rba = ...
DbControl dc = ...

// Get the current raw data file, if any
List<File> rawDataFiles =
    FileStoreUtil.getGenericDataFiles(dc, rba, FileType.RAW_DATA);
File defaultFile = rawDataFiles.size() > 0 ?
    rawDataFiles.get(0) : null;

// Create parameter asking for input file - use current as default
PluginParameter<File> fileParameter = new PluginParameter<File>(
    "file",
    "Raw data file",
    "The file that contains the raw data that you want to import",
    new FileParameterType(defaultFile, true, 1)
);
```

An import plug-in should also save the file that was used to the file set:

```
RawBioassay rba = ...
// The file the user selected to import from
File rawDataFile = (File) job.getValue("file");

// Save the file to the fileset. The method will check which file
// type the platform uses as the raw data type. As a fallback the
// GENERIC_RAW_DATA type is used
FileStoreUtil.setGenericDataFile(dc, rba, FileType.RAW_DATA,
    DataFileType.GENERIC_RAW_DATA, rawDataFile);
```

Use case: Using raw data from files in an experiment

Just as before, an experiment is still locked to a single `RawDataType`. This is a design issue that would break too many things if changed. If data is stored in files the experiment is also locked to a single `Platform`. This has been designed to have as little impact on existing plug-ins as possible. In most cases, the plug-ins will continue to work as before.

A plug-in (using data from the database that needs to check if it can be used within an experiment can still do:

```
Experiment e = ...
RawDataType rdt = e.getRawDataType();
if (rdt.isStoredInDb())
{
    // Check number of channels, etc...
    // ... run plug-in code ...
}
```

A newer plug-in which uses data from files should do:

```
Experiment e = ...
DbControl dc = ...
RawDataType rdt = e.getRawDataType();
if (!rdt.isStoredInDb())
{
    // Check that platform/variant is supported
    Platform p = rdt.getPlatform(dc);
    PlatformVariant v = rdt.getVariant(dc);
    // ...

    // Get data files
    File aFile = FileStoreUtil.getDataFile(dc, ...);

    // ... run plug-in code ...
}
```

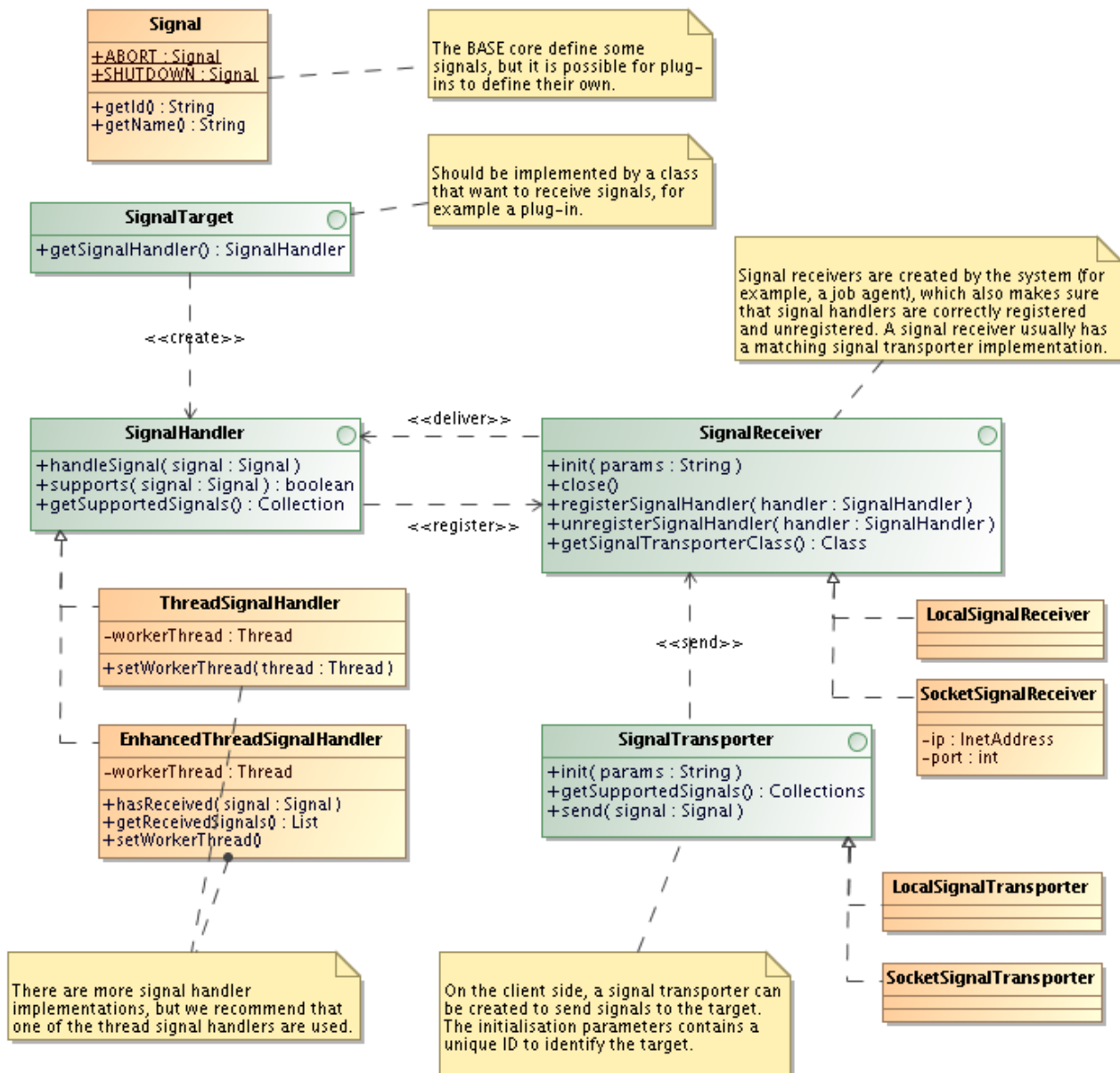
28.3.10. Sending signals (to plug-ins)

BASE has a simple system for sending signals between different parts of a system. This signalling system was initially developed to be able to kill plug-ins that a user for some reason wanted to abort. The signalling system as such is not limited to this and it can be used for other purposes as well. Signals can of course be handled internally in a single JVM but also sent externally to other JVM:s running on the same or a different computer. The transport mechanism for signals is decoupled from the actual handling of them. If you want to, you could implement a signal transporter that sends signal as emails and the target plug-in would never know.

The remainder of this section will focus mainly on the sending and transportation of signals. For more information about handling signals on the receiving end, see Section 25.2.3, “Abort a running a plug-in” (page 237).

Diagram of classes and methods

Figure 28.21. The signalling system



The signalling system is rather simple. An object that wishes to receive signals must implement the `SignalTarget`. Its only method is `getSignalHandler()`. A `SignalHandler` is an object that knows what to do when a signal is delivered to it. The target object may implement the `SignalHandler` itself or use one of the existing handlers.

The difficult part here is to be aware that a signal is usually delivered by a separate thread. The target object must be aware of this and know how to handle multiple threads. As an example we can use the `ThreadSignalHandler` which simply calls `Thread.interrupt()` to deliver a signal. The target object that uses this signal handler must know that it should check `Thread.interrupted()` at regular intervals from the worker thread. If that method returns true, it means that the `ABORT` signal has been delivered and the main thread should clean up and exit as soon as possible.

Even if a signal handler could be given directly to the party that may be interested in sending a signal to the target this is not recommended. This would only work when sending signals within the same

virtual machine. The signalling system includes `SignalTransporter` and `SignalReceiver` objects that are used to decouple the sending of signals with the handling of signals. The implementation usually comes in pairs, for example `SocketSignalTransporters` and `SocketSignalReceiver`.

Setting up the transport mechanism is usually a system responsibility. Only the system know what kind of transport that is appropriate for it's current setup. Ie. should signals be delivered by TCP/IP sockets, only internally, or should a delivery mechanism based on web services be implemented? If a system wants to receive signals it must create an appropriate `SignalReceiver` object. Within BASE the internal job queue set up it's own signalling system that can be used to send signals (eg. kill) running jobs. The job agents do the same but uses a different implementation. See the section called "Internal job queue section" (page 420) for more information about how to configure the internal job queue's signal receiver. In both cases, there is only one signal receiver instance active in the system.

Let's take the internal job queue as an example. Here is how it works:

- When the internal job queue is started, it will also create a signal receiver instance according to the settings in `base.config`. The default is to create `LocalSignalReceiver` which can only be used inside the same JVM. If needed, this can be changed to a `SocketSignalReceiver` or any other user-provided implementation.
- When the job queue has found a plug-in to execute it will check if it also implements the `SignalTarget` interface. If it does, a signal handler is created and registered with the signal receiver. This is actually done by the BASE core by calling `PluginExecutionRequest.registerSignalReceiver()` which also makes sure that the the ID returned from the registration is stored in the database together with the job item representing the plug-in to execute.
- Now, when the web client see's a running job which has a non-empty signal transporter property, the **Abort** button is activated. If the user clicks this button the BASE core uses the information in the database to create `SignalTransporter` object. This is simply done by calling `Job.getSignalTransporter()`. The created signal transporter knows how to send a signal to the signal receiver it was first registered with. When the signal arrives at the receiver it will find the handler for it and call `SignalHandler.handleSignal()`. This will in it's turn trigger some action in the signal target which soon will abort what it is doing and exit.

28.4. The Query API

This documentation is only available in the old format which may not be up-to-date with the current implementation. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/query/index.html>

28.5. The Dynamic API

This documentation is only available in the old format which may not be up-to-date with the current implementation. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/dynamic/index.html>

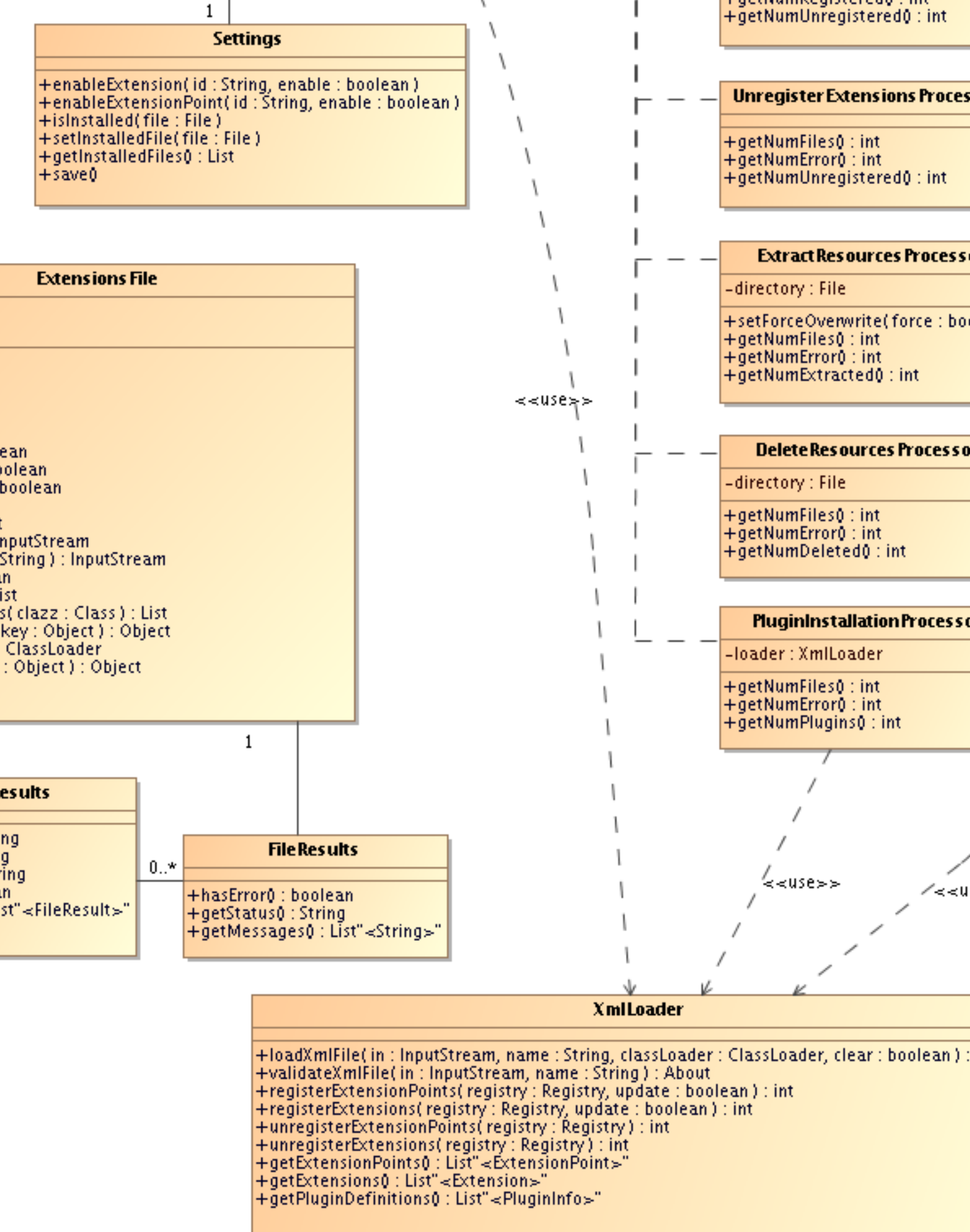
28.6. The Extensions API

28.6.1. The core part

The *Extensions API* is divided into two parts. A core part and a web client specific part. The core part can be found in the `net.sf.basedb.util.extensions` package and it's sub-packages, and consists of two sub-parts:

- An `ExtensionsManager` that keeps track of the JAR files on the file system containing extensions code. The manager can detect new, updated and deleted files and is used to load metadata information about the extensions and register them in the `Registry` so that they can be used. The manager is also used to install plug-ins.
- A set of interface definitions which forms the core of the Extensions API. The interfaces defines, for example, what an `Extension` is, what an `ActionFactory` should do and a few other things.

Let's start by looking at the extensions manager and related classes.



The BASE application is using a single manager and a single registry (handled by the `Application` class). The manager has been configured to look for extensions and plug-ins in the directory specified by the `plugins.dir` setting in `base.config`. Theoretically, a single manager can handle multiple directories, but we do not use that feature. The BASE core also includes some special files that are added with the `addURI()` method. They contain definitions for the core extensions and core plug-ins and are shipped as XML files that reside inside the BASE core JAR files.

The `ExtensionsManager.scanForChanges()` method is called to initiate a check for new, updated and deleted files. The manager uses the `XmlLoader` to find information about each JAR or XML file it finds in the directory. After the scan, the `ExtensionsManager.GetFiles()` method can be used to find more information about each individual file, for example, if it is a new or modified file, if it contains valid extension definitions and information about the author, etc. This information is used by the installation wizard in the web client to display a dialog where the user can select which extensions to install. See Figure 21.2, “Extensions and plug-ins installation wizard” (page 180). Note that no installation or other actions take place at this stage.

The `ExtensionsManager.processFiles()` method is called to actually do something. It needs an `ExtensionsFileProcessor` implementation as an argument. As you can see in the diagram above there are multiple implementations (all are not shown in the diagram), each with a very specific task. A processor is usually also paired with a filter to target it at files that fulfill some criteria, for example, only at valid extension files that have been updated. Typically, the `ExtensionsManager.processFiles()` method needs to be called multiple times with different processor implementations to perform a full installation of an extension or plug-in. Here is a list of the various processors currently in use in BASE.

`RegisterExtensionsProcessor`

Is used to register extensions with the registry. Can be paired with different filters depending on when it is used. At BASE startup an `InstalledFilter` is used so that only installed extensions are registered.

`UnregisterExtensionsProcessor`

Is used to unregister extensions when a file has been deleted. This should always be paired with for example, a `DeletedFilter`.

`UnregisterExtensionsProcessor`

Is used to unregister extensions when a file has been deleted. This should always be paired with for example, a `DeletedFilter`.

`ExtractResourcesProcessor`

Is used to extract files from the JAR file to a local directory. This is currently used by the web client to extract JSP files, images, etc. that are needed by web client extensions.

`DeleteResourcesProcessor`

The opposite of `ExtractResourcesProcessor`.

`PluginInstallationProcessor`

Is used to install and register plug-ins.

`DisablePluginsProcessor`

Is used to disable plug-ins from extensions that have been removed.

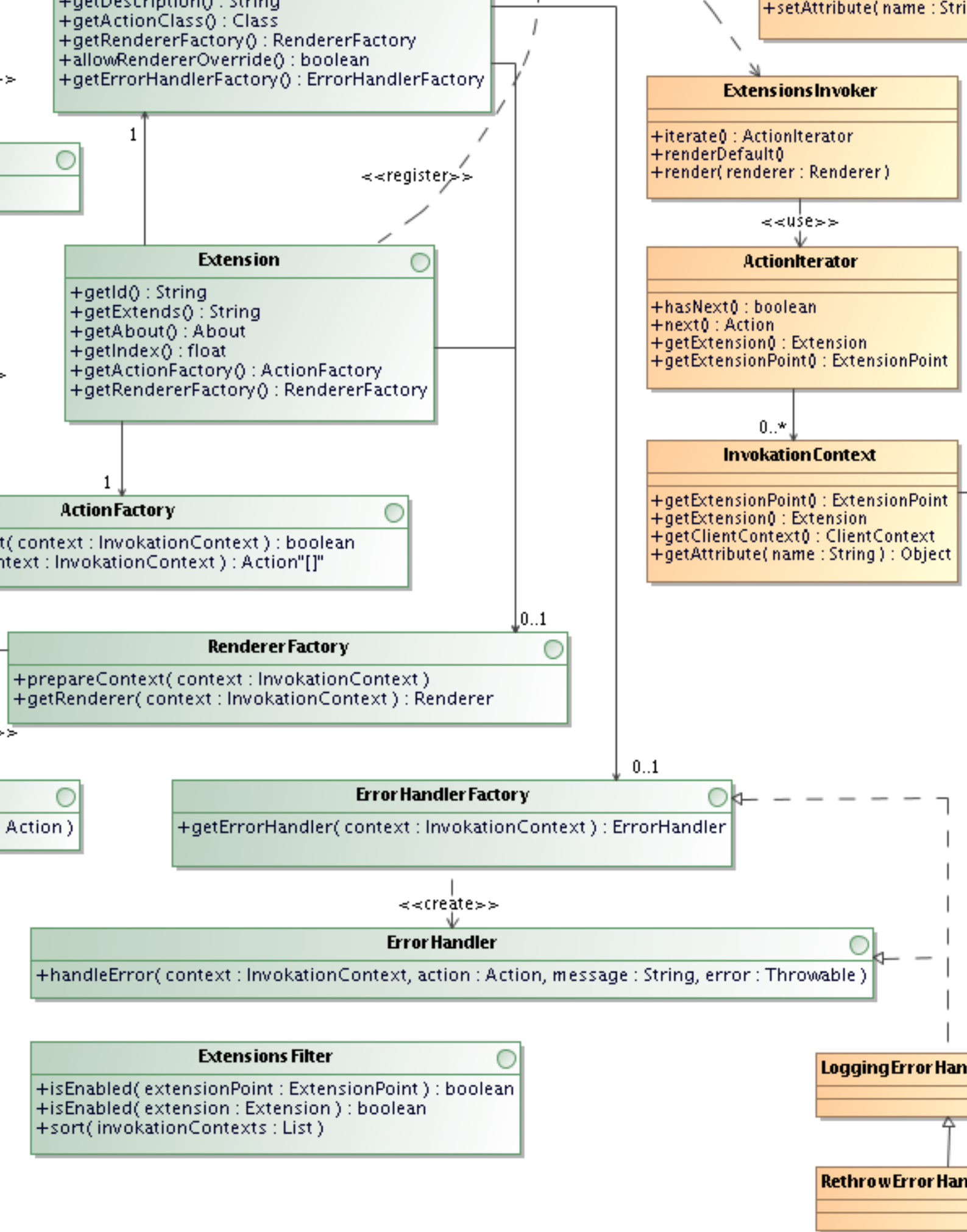
`MarkAsProcessedProcessor`

This is usually the final processor that is called and resets the timestamp on all processed files so that the next time `ExtensionsManager.scanForChanges()` is called it will know what has been modified.

Note

This list contains the core processors only. The web client part is using some additional processors to perform, for example, servlet registration.

The result of the processing can be collected with a `ProcessResults` object and is then displayed to the user as in Figure 21.3, “Extensions and plug-ins installation results” (page 181). All of the above is only used when BASE is starting up and initializing the extensions system or when the server administrator is performing a manual installation or update. The next diagram shows the part of the extensions system that is used when actually using the extensions for what they are intended to do (the web client adds some extra features to this as well, but that is discussed later).



The `Registry` is one of the main classes in the extension system. All extension points and extensions must be registered before they can be used. Typically, you will first register extension points and then extensions, because an extension can't be registered until the extension point it is extending has been registered.

An `ExtensionPoint` is an ID and a definition of an `Action` class. The other options (name, description, renderer factory, etc.) are optional. An `Extension` that extends a specific extension point must provide an `ActionFactory` instance that can create actions of the type the extension point requires.

Example 28.1. The menu extensions point

The `net.sf.basedb.clients.web.menu.extensions` extension point requires `MenuItemAction` objects. An extension for this extension point must provide a factory that can create `MenuItemAction`s. BASE ships with default factory implementations, for example the `FixedMenuItemFactory` class, but an extension may provide its own factory implementation if it wants to.

Call the `Registry.useExtensions()` method to use extensions from one or several extension points. This method will find all extensions for the given extension points. If a filter is given, it checks if any of the extensions or extension points has been disabled. It will then call `ActionFactory.prepareContext()` for all remaining extensions. This gives the action factory a chance to also disable the extension, for example, if the logged in user doesn't have a required permission. The action factory may also set attributes on the context. The attributes can be anything that the extension point may make use of. Check the documentation for the specific extension point for information about which attributes it supports. If there are any renderer factories, their `RendererFactory.prepareContext()` is also called. They have the same possibility of setting attributes on the context, but can't disable an extension.

After this, an `ExtensionsInvoker` object is created and returned to the extension point. Note that the `ActionFactory.getActions()` has not been called yet, so we don't know if the extensions are actually going to generate any actions. The `ActionFactory.getActions()` is not called until we have got ourselves an `ActionIterator` from the `ExtensionsInvoker.iterate()` method and starts to iterate. The call to `ActionIterator.hasNext()` will propagate down to `ActionFactory.getActions()` and the generated actions are then available with the `ActionIterator.next()` method.

The `ExtensionsInvoker.renderDefault()` and `ExtensionsInvoker.render()` are just convenience methods that will make it easier to render the actions. The first method will of course only work if the extension point is providing a renderer factory, that can create the default renderer.

Be aware of multi-threading issues

When you are creating extensions you must be aware that multiple threads may access the same objects at the same time. In particular, any action factory or renderer factory has to be thread-safe, since only one exists for each extension. Action and renderer objects should be thread-safe if the factories re-use the same objects.

Any errors that happen during usage of an extension is handled by an `ErrorHandler`. The core provides two implementations. We usually don't want the errors to show up in the gui so the `LoggingErrorHandlerFactory` is the default implementation that only writes to the log file. The `RethrowErrorHandlerFactory` error handler can be used to re-throw exceptions which usually means that they trickle up to the gui and are shown to the user. It is also possible for an extension point to provide its own implementation of an `ErrorHandlerFactory`.

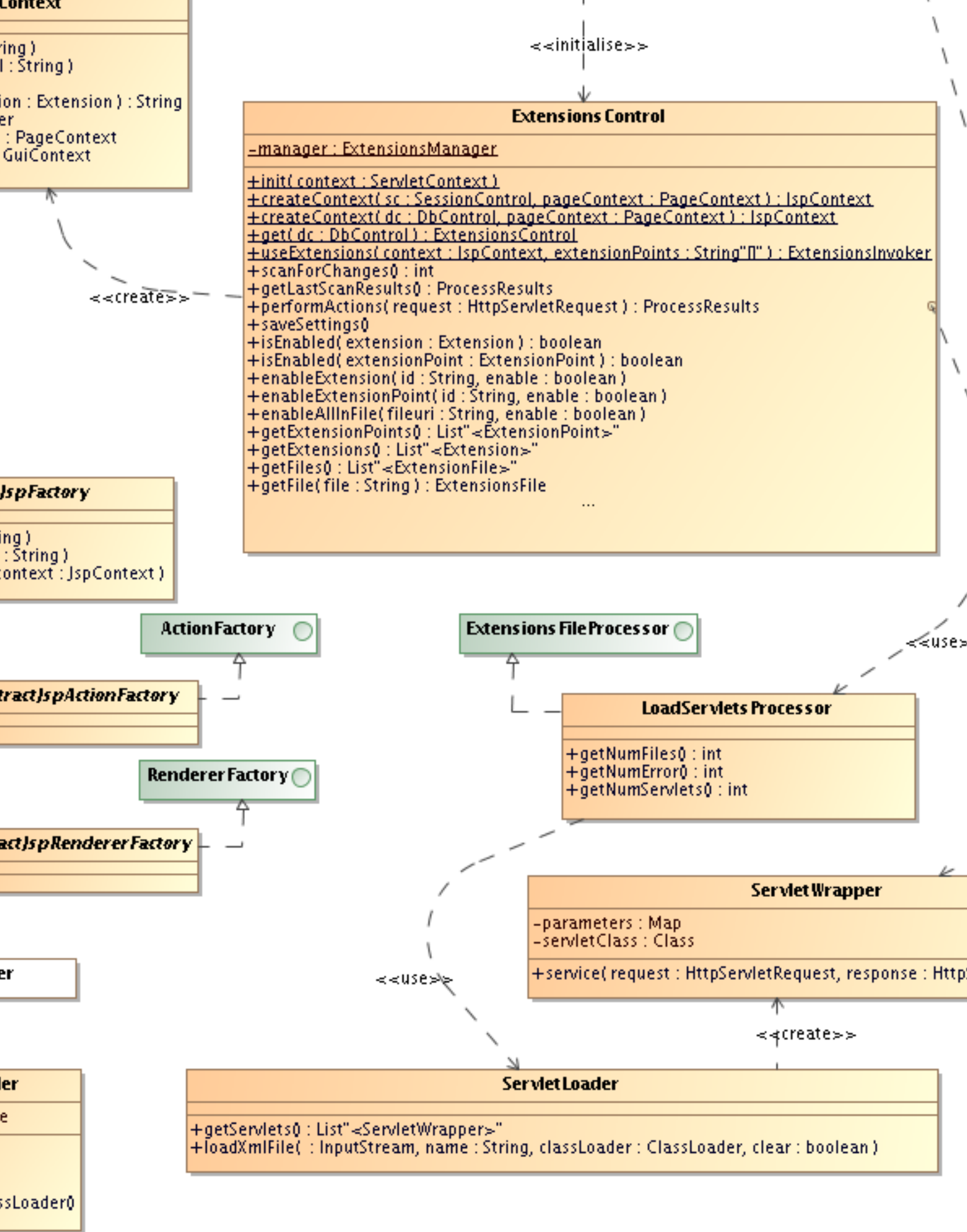
28.6.2. The web client part

The web client specific parts of the Extensions API can be found in the `net.sf.basedb.client.web.extensions` package and its subpackages. The top-level package contains the `ExtensionsControl` class which is used to administrate the extension system. It is more or

less a wrapper around the `ExtensionsManager` provided by the core, but adds permission control and a few other things.

In the top-level package there are also some abstract classes that may be useful to extend for developers creating their own extensions. For example, we recommend that all action factories extend the `AbstractJspActionFactory` class. All web client extension points use the `JspContext` class instead of `ClientContext`. The JSP context provides some extra information about the current request.

The sub-packages to `net.sf.basedb.client.web.extensions` are mostly specific to a single extension point or to a specific type of extension point. The `net.sf.basedb.client.web.extensions.menu` package, for example, contains classes that are/can be used for extensions adding menu items to the Extensions menu. See Section 26.8, “Extension points defined by BASE” (page 279) for more information about the extension points defined by BASE.



When the Tomcat web server is starting up, the `ExtensionsServlet` is automatically loaded. This servlet has as two purposes:

- Initialise the extensions system by calling `ExtensionsControl.init()`. This will result in an initial scan for installed extensions. This means that the extension system is up and running as soon as the first user logs in to BASE. The processing scheme is slightly different from what is done when the core is setting up the initial manager. The major additions are:
 - The `LoadServletsProcessor` is used to load servlets that has been defined in `META-INF/servlets.xml`.
 - The `ExtractResourcesProcessor` is used to extract all files from `resources/*` in the JAR file to `www/extensions/jar-name.jar/` in Tomcat's web application directory.
- Act as a proxy for custom servlets defined by the extensions. URL:s ending with `.servlet` has been mapped to the `ExtensionsServlet`. When a request is made it will extract the name of the extension's JAR file from the URL, get the corresponding `ServletWrapper` and then invoke the custom servlet. More information can be found in Section 26.7, "Custom servlets" (page 277).

Using extensions only involves calling the `ExtensionsControl.createContext()` and `ExtensionsControl.useExtensions()` methods. This returns an `ExtensionsInvoker` object as described in the previous section.

To render the actions it is possible to either use the `ExtensionsInvoker.iterate()` method and generate HTML from the information in each action. Or (the better way) is to use a renderer together with the `Render` taglib.

To get information about the installed extensions, change settings, enabled/disable extensions, performing a manual installation, etc. use the `ExtensionsControl.get()` method. This will create a permission-controlled object. All users has read permission, administrators has write permission.

Note

The permission we check for is `WRITE` permission on the web client item. This means it is possible to give a user permissions to manage the extension system by assigning `WRITE` permission to the web client entry in the database. Do this from `Administrate Clients`.

The `XJspCompiler` is mapped to handle the compilation `.xjsp` files which are regular JSP files with a different extension. The difference is that the `XJSP` compiler include the extension's JAR file on the class path, which means that the JSP file can use classes that would otherwise be impossible. This feature is experimental and requires installing an extra JAR into Tomcat's lib directory. See Section 21.1.4, "Installing the X-JSP compiler" (page 183) for more information.

28.7. Other useful classes and methods

TODO

Chapter 29. Write documentation

29.1. User, administrator and developer documentation with Docbook

This chapter is for those who intend to contribute to the BASE user documentation. The chapter contains explanations of how the documentation is organized, what the different parts is about and other things that will make it easier to know where to insert new text.

The documentation is written with the docbook standard, which is a bunch of defined XML elements and XSLT style sheets that are used to format the text. Later on in this chapter is a reference, over those docbook elements that are recommended to use when writing BASE documentation. Further information about docbook can be found in the on-line version of O'Reilly's DocBook: The Definitive Guide¹ by Norman Walsh and Leonard Mueller.

29.1.1. Documentation layout

The book, which is the main element in this docbook documentation, is divided into four separated parts, depending on who the information is directed to. What kind of documentation each one of these parts contains or should contain is described here below.

Overview documentation

The overview part contains, like the name says, an overview of BASE. For example an explanation about what the purpose with BASE is, the terms that are used in the program and other general things that anyone that is interested in the program wants/needs to know before exploring it further.

User documentation

This part contains information that are relevant for the common BASE-user. More or less should everything that a power user role or an user role needs to know be included here.

Administrator documentation

Things that only an administrator-role can do is documented in this part. It can be how to install the program, how to configure the server for best performance or other subjects that are related to the administration.

Developer documentation

Documentation concerning the participation of BASE development should be placed in this part, e.g. coding standards, the java doc from the source code, how to develop a plug-in etc.

In addition to the four main parts, there is also a FAQ part and an Appendix part.

29.1.2. Getting started

Before writing any documentation in BASE there are a couple of things, some rules and standards, to have in mind.

Organization of source files

The source files of the documentation are located in `<base-dir>/doc/src/docbook`. The different parts of the documentation are organized into separate folders and each one of the folders contains

¹ <http://www.docbook.org/tdg/en/html/>

one index file and one file for each chapter in that part. The index file joins the chapters, in current part/folder, together and does not really contain any text. The documentation root directory also contains an `index.xml` file which joins the index files from the different parts together.

Create new chapter/file

Most files in the documentation, except the index files, represents a chapter. Here is how to create a new chapter and include it in the main documentation:

1. Create a new XML-file in the folder for the part where the new chapter should be included. Give it a name that is quite similar to the new chapter's title but use `_` instead of blank space and keep it down to one or a few words.
2. Begin to write the chapter's body, here is an example:

Example 29.1. Example of a chapter

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE chapter PUBLIC
  "-//Dawid Weiss//DTD DocBook V3.1-Based Extension for XML and graphics inclusion//EN"
  "../../../lib/docbook/preprocess/dweiss-docbook-extensions.dtd">

<chapter id="example_chapter">
  <?dbhtml dir="example" filename="index.html" ?>
  <title>Example of a chapter </title>
  <para>
    ...
  </para>
  <sect1 id="example_chapter.sect1">
    <title>Example of top level section </title>
    <para>
      ...
    </para>
    <sect2 id="example_chapter.sect1.sect2">
      <title>Example of second level section </title>
      <para>
        ...
      </para>
      <sect3 id="example_chapter.sect1.sect2.sect3">
        <title>Example of third level section</title>
        <para>
          ...
        </para>
      </sect3>
    </sect2>
  </sect1>
</chapter>
```

Note

Do not forget to include the `<?dbhtml>` tag with the `dir` and/or `filename` attributes set to target folder and filename for the chapter.

3. The last step is to get the new chapter included in the documentation. This is done by including the file's name in the file `index.xml` that's located in the current part's folder. The following example shows how the chapter that was created above is included in the user documentation part (as the last chapter).

Example 29.2. Include a chapter

```
<part id="user">
  <?dbhtml dir="user" filename="index.html" ?>
  <title>User documentation</title>
  <include file="overview.xml"/>
  <include file="webclient.xml"/>
  <include file="project_permission.xml"/>
  <include file="file_system.xml"/>
  <include file="jobs.xml"/>
  <include file="reporters.xml"/>
  <include file="annotations.xml"/>
  <include file="platforms.xml" />
  <include file="subtypes.xml" />
  <include file="protocols.xml"/>
  <include file="wares.xml"/>
  <include file="array_lims.xml"/>
  <include file="biomaterials.xml"/>
  <include file="experiments_analysis.xml"/>
  <include file="import_data.xml"/>
  <include file="export_data.xml"/>
  <include file="example_chapter.xml"/>
</part>
```

Order of chapters

The chapters will come in the same order as they are included in the index file.

Controlling chunking

We have configured docbook to create a new sub-directory for each `<chapter>` and a new output file for each `<sect1>` tag. In most cases this gives each page a relatively good size. Not too long and not too short. However, if a chapter contains many small `<sect1>` sections (for example, Chapter 3, *Resources* (page 9)), you end up with many pages with just a few lines of text on each page. This is not so good and can be avoided by adding `chunked="0"` as an attribute to the `<chapter>` tag, for example:

```
<chapter id="resources" chunked="0">
```

This will stop the chunking of `<sect1>` sections in this chapter.

On the other hand, if you have a `<sect1>` that contains many long `<sect2>` sections you might want to put each `<sect2>` section in a separate chunk:

```
<sect1 id="sect.with.large.sect2" chunked="1">
```

The id attribute

Common to all elements is that they have an `id` attribute that can be used to identify the element with. The value must be unique for the entire documentation. Most of the elements that are used inside the BASE documentation do not need to have an `id` if they do not are used in any cross references from other part of the text.

There are some elements that always should have the `id` set. Which these elements are and how the `id` should look like for each one of those is described below. All values should be as short as possible and associated to the current element. The `id` value should only consist of the lowercase characters a-z, '_' instead of blank spaces and '.' to symbolize the different levels in the document.

chapter

The chapter should have an `id` that is identical or almost identical to the chapter's title.


```
chapter_title
```

sect1-sect5

The creation of a section's id is done by combining an appropriate part of the parent chapter and/or section id with the current section's title (or part of the title). This rule may seem a little "fuzzy" but the aim is to not create too long id's yet they must still be unique. using the upper level's id and add the section's title or a part of the title. For a <sect2> the id could for example be created like this:

```
sect1_title.sect2_title
```

examples

The naming of an example's id is a bit different compare to the chapter's and section's id. It should begin with the chapter's id followed by **examples** and the caption of the example.

```
chapter_title.examples.example_caption
```

figures

The figure's id should have the following layout

```
chapter_title.figures.figure_name
```

Help text for the BASE web client

This documentation is also use to create the help texts that show up in the BASE web client when clicking on the question mark icons. The parts of the text that also should be used as help texts in the web client must be inside <helptext> tags. These texts will be exported to a XML-file at the same time as the HTML-documentation is generated. The generated XML-file is compatible with the help text importer in BASE and will be imported when running the update script or installation script.

Note

The <helptext> must be outside the <para> tag but inside a <sect> tag to work properly. Do not put any <sect> tags inside the helptext. This will make the automatic chapter and section numbering confused.

The tag supports the following attributes

external_id

Is used to identify and to find a help text in the web client. The BASE web client already has several IDs to help texts defined, it could therefore be a good idea to have a look in the program to see if there already is an external id to use for the particular help text.

title

Is directly connected to the name/title property for a help text item in BASE.

webonly

If set to a non-zero value, the contents of the tag will not be outputted in the HTML or PDF documentation. This is useful for minor functionality that is not important enough to be mentioned in the documentation but has a help icon in the web client.

Example 29.3. How to use the help text tag

```
<sect1 id="....">
  <helptext external_id="helptexts.external.id" title="The title">
    The text that also should be used as a helptext in the program.
  <seeother>
    <other external_id="other.external.id">Related info here...</other>
  </seeother>
</helptext>
</sect1>
```

Skip parts of a help text

From time to time, it may happen that you find that some parts of the text inside a `<helptext>` tag does not make sense. It may, for example, be a reference to an image or an example, or a link to another chapter or section. Put a `<nohelp>` tag around the problematic part to avoid it from being outputted to the help texts.

```
<nohelp>See <xref linkend="chapter11" /></nohelp>
```

Link to other help texts

You can use `<seeother>` and `<other>` to create links between different help texts. The `<seeother>` tag does not have any attributes and is just a container for one or more `<other>` tags. Each tag requires a single attribute.

`external_id`

The external ID of the other help text to link to.

```
<seeother>
  <other external_id="userpreferences.password">Change password</other>
  <other external_id="userpreferences.other">Other information</other>
</seeother>
```

We recommend that you place the links at the end of the help text section. The links will not show up in the HTML or PDF version.

Import help texts into BASE

Import the generated XML-file manually by uploading it from the `data` directory to the BASE-server's file system and then use the help text importer plug-in to import the help text items from that file.

The help texts can also be imported by running the TestHelp test program. In short, here are the commands you need to import the help texts:

```
ant docbook
ant test
cd build/test
./run.sh TestHelp
```

Build the documentation

Those who have checked out the documentation source from repository or got the source from a distribution package needs to compile it to get the PDF and HTML documentation files. The compilation of the documentation source requires, beside Ant, that the XML-parser Xsltproc is installed on the local computer. More information about XsltProc and how it is installed, can be found at <http://xmlsoft.org/XSLT/index.html>.

Note

There is an xml-parser in newer java-versions that can be used instead of XsltProc but the compilation will most likely take much much longer time. Therefore it's not recommended to be used when generating/compiling the documentation in BASE.

There are two different types of format that is generated from the documentation source. The format to view the documentation on-line will be the one with chunked HTML pages where each chapter and section of first level are on separate pages/files. The other format is a PDF-file that are most useful for printing and distribution. Those two types of output are generated with the ant-target: **ant docbook**. This documentation is also generated with **ant dist**, which will put the output files in the right location for distribution with BASE.

29.1.3. Docbook tags to use

The purpose with this section is to give an overview of those docbook elements that are most common in this documentation and to give some example on how they should be used. There will not be any detailed explanation of the tags here, instead the reader is recommended to get more information from Docbook's documentation ² or other references.

Text elements

Define	Element to use	Comments
Chapter	<code><chapter></code>	See Example 29.1, "Example of a chapter" (page 349).
Title	<code><title></code>	See Example 29.1, "Example of a chapter" (page 349).
Paragraph	<code><para></code>	Used almost everywhere around real text.
Top-level subsection	<code><sect1></code>	See the section called "The id attribute" (page 350).
Second level section	<code><sect2></code>	See the section called "The id attribute" (page 350).
Third level section	<code><sect3></code>	See the section called "The id attribute" (page 350).

Code elements

These elements should be used to mark up words and phrases that have a special meaning in a coding environment, like method names, class names and user inputs, etc.

Define	Element to use	Comment
Class name	<code><classname></code>	The name of a (Java) class. The <code>docapi</code> attribute can be used to link the class to it's Javadoc (for the BASE API). This is done by setting the attribute to the package name of the class, like <pre><classname docapi="net.sf.baseb.core">DbContr classname</pre>
Interface name	<code><interfacename></code>	The name of an (Java) interface. Has a <code>docapi</code> -attribute which has the same functionality as in <code>classname</code> above.
User input	<code><userinput></code>	Text that is entered by a user.
Variable name	<code><varname></code>	The name of a variable in a program.
Constant	<code><constant></code>	The name of a constant in a program.
Method definition	<code><methodsynopsis></code>	See Example 29.4, "Method with no arguments and a return value" (page 354).

² <http://www.docbook.org/tdg/en/html/docbook.html>

Define	Element to use	Comment
Modifier of a method	<modifier>	- " -
Classification of return value	<type>	- " -
Method name	<methodname>	- " -
No parameter/type	<void>	- " -
Define a parameter	<methodparam>	See Example 29.5, “ Method with arguments and no return value ” (page 354).
Parameter type	<type>	- " -
Parameter name	<parameter>	- " -

Follow one of the examples below to insert a method definition in the document.

Example 29.4. Method with no arguments and a return value

```
<methodsynopsis language="java">
  <modifier>public</modifier>
  <type>Plugin.MainType</type>
  <methodname>getMainType</methodname>
  <void />
</methodsynopsis>
```

which is rendered as:

```
public Plugin.MainType getMainType();
```

Example 29.5. Method with arguments and no return value

```
<methodsynopsis language="java">
  <modifier>public</modifier>
  <void />
  <methodname>init</methodname>
  <methodparam>
    <type>SessionControl</type>
    <parameter>sc</parameter>
  </methodparam>
  <methodparam>
    <type>ParameterValues</type>
    <parameter>configuration</parameter>
  </methodparam>
  <methodparam>
    <type>ParameterValues</type>
    <parameter>job</parameter>
  </methodparam>
</methodsynopsis>
```

which is rendered as:

```
public void init (SessionControl sc,
                 ParameterValues configuration,
                 ParameterValues job);
```

Gui elements

Docbook has some elements that can be used to symbolize GUI items in a program. Following list contains the ones that are most common in this document.

Define	Element to use	Comment
Button	<guibutton>	
Label	<guilabel>	

Define	Element to use	Comment
Menu choice	<code><menuchoice></code>	
Menu	<code><guimenu></code>	
Submenu	<code><guisubmenu></code>	
Menu item	<code><guimenuitem></code>	
Icon	<code><guiicon></code>	

Example 29.6. Describe a menu choice

```
<menuchoice>
  <guimenu>Administrate</guimenu>
  <guisubmenu>Plug-ins & extensions</guisubmenu>
  <guimenuitem>Overview</guimenuitem>
</menuchoice>
```

In the text it will look like this: Administrate Plug-ins & extensions Overview

Images and figures

Images and figures are normally implemented like the following example. The image-file must be located in `doc/src/docbook/figures`, otherwise the image will not be visible in the generated output files.

Example 29.7. Screen-shot in the documentation

```
<figure id="docbook.figures.homepage">
  <title>The home page</title>
  <screenshot>
    <mediaobject>
      <imageobject>
        <imagedata
          scalefit="1"
          width="100%"
          fileref="figures/homapage.png" format="PNG"
        />
      </imageobject>
    </mediaobject>
  </screenshot>
</figure>
```

which will generate an image like Figure 5.1, “The home page” (page 17).

Warning

When using images in docbook you will always have problems with image resolution and scaling. Since we are generating output for both HTML and PDF it is even worse. What we have found to work is this:

- The screenshots must be saved without any resolution information in them, or with the resolution set to 96 dpi. We have configured PDF to use 96 dpi instead of 72 dpi to make the HTML and PDF output look as similar as possible.
- Scaling in HTML has been disabled. The images will always be the same size (number of pixels) as they actually are. Please, do not make the screenshots too wide!

Tip

Change your BASE preferences, see Section 5.2.4, “Preferences” (page 20), to a smaller font size or use the zoom functionality in the web browser to make more information fit in the same image width.

- For small images, less than the width of the PDF page, do not specify scaling factors or widths.

- Images that are wider than the PDF page will be clipped. To prevent this you must add the following attributes to the `<imagedata>` tag: `scalefit="1" width="100%"`. This will scale down the image so that it fits the available width.
- If you still need to scale the image differently in the PDF use the `width` and `depth` attributes.

Examples and program listing

Following describes how to insert an example in the documentation. The examples in this document are often some kind of program listing but they can still be examples of something else.

Use spaces instead of tabs for indentation

Use spaces for indentation in program listing, this is because of the tab-indentations will sooner or later cause corrupt text.

- The verbatim text is split into several lines if the text contains more than 80 characters. This could give the text an unwanted look and it's therefore recommended to manually insert new lines to have control over layout of the text
- We have added support for syntax highlighting of program examples in the HTML version. To enable it add a language attribute with one of the following values: `java`, `xml` or `sql`. The highlighting engine support more languages. To add support for those in docbook, change the customized.chunked.xsl file. The syntax highlighting engine doesn't handle docbook markup inside the `<programlisting>` tag very well. You should avoid that, by using text-only examples withing a `<![CDATA[...]]>` section. By default, Java program examples include line numbering, but XML examples don't. To disable line numbering for Java add `:nogutter` to the language attribute: `<programlisting language="java:nogutter">`. To enable line numbering for xml add `:gutter` to the language attribute: `<programlisting language="xml:gutter">`.

Example 29.8. Example in the documentation

The code below is used to create Example 25.2, "A typical implementation just return one of the values" (page 226).

```
<example id="net.sf.basedb.core.plugin.Plugin.getMainType">
  <title>A typical implementation just return one of the values</title>
  <programlisting language="java">
public Plugin.MainType getMainType()
{
    return Plugin.MainType.OTHER;
}
  </programlisting>
</example>
```

Admonitions

The admonitions that are used in this document can be found in the table below.

Define	Element to use	Comment
Warning text	<code><warning></code>	
Notification text	<code><note></code>	
A tip	<code><tip></code>	
Important text	<code><important></code>	
Something to be cautious about	<code><caution></code>	

Lists

Following items can be used to define different kind of lists in the documentation. Some common elements for the lists are also described here.

Define	Element	Comment
None-ordered list	<code><itemizedlist></code>	
Term definition list	<code><variablelist></code>	
Ordered list	<code><orderedlist></code>	
List item	<code><listitem></code>	

The example below shows how to create a list for term definition in the text.

Example 29.9. Example how to write a variable list

```
<variablelist>
  <varlistentry>
    <term>Term1</term>
    <listitem>
      <para>
        Definition/explanation of the term
      </para>
    </listitem>
  </varlistentry>

  <varlistentry>
    <term>Term2</term>
    <listitem>
      <para>
        Definition/explanation of the term
      </para>
    </listitem>
  </varlistentry>
</variablelist>
```

which is rendered as:

Term1
Definition/explanation of the term

Term2
Definition/explanation of the term

Link elements

Define	Element	Comment
Cross reference	<code><xref linkend=""></code>	Use this to Link to other parts of the document.
Cross reference with own text	<code><link></code>	Can be used as an alternative to xref
External URLs	<code><ulink url=""></code>	

Example 29.10. Links

```
<xref linkend="docbook.usedtags.links" />
<link linkend="docbook.usedtags.links">Link to this section</link>
<ulink url="http://base.thep.lu.se">Base2's homepage</ulink>
```

The first element will autogenerate the linked section's/chapter's title as a hyperlinked text. As an alternative to xref is link that lets you write your own hyperlinked text. The third and last one should be used to link to any URL outside the document.

29.2. Create UML diagrams with MagicDraw

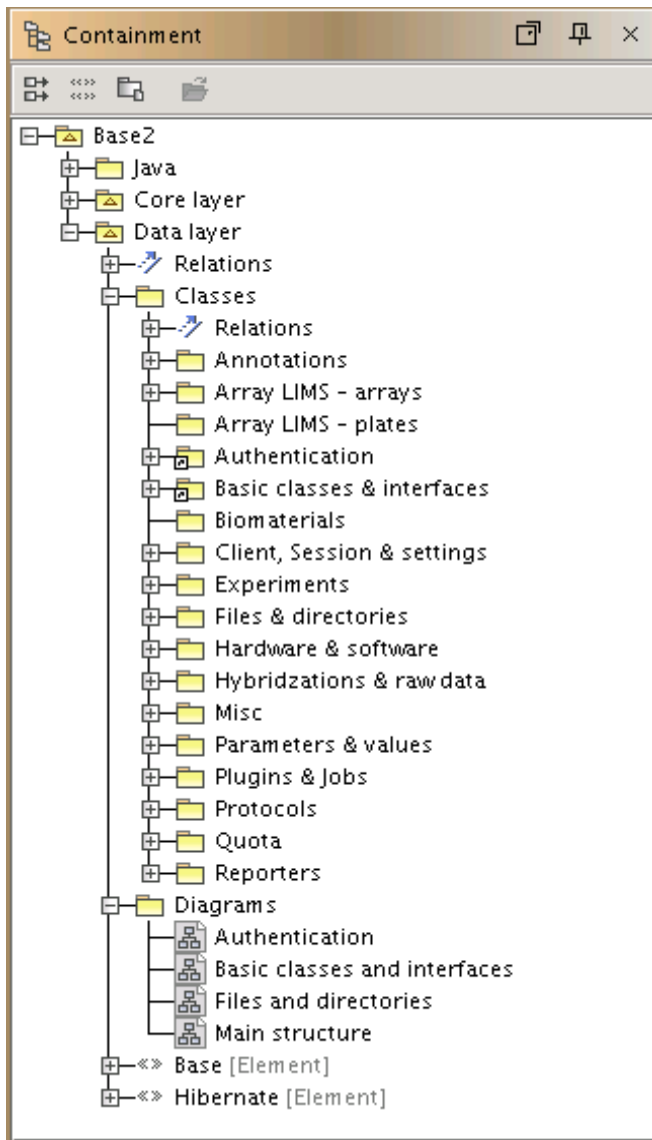
UML or UML-like diagrams are used to document the relationship of classes in the Core API. To create the diagrams we use the community edition (version 12.5) of a program called MagicDraw. This is a Java program and it should be possible to run it on any platform which has at least a Java 1.5 run-time. To get more information about MagicDraw and to download the program go to their website: <http://www.magicdraw.com/>

29.2.1. Organisation

All classes and diagrams are in a single UML file. It can be found at `<base-dir>/doc/src/uml/baseuml.mdzip`

Everything in MagicDraw has been organised into packages and modules. At the top we have the Core layer and the Data layer. The Java module is for classes that related to the Java programming language, such as Map and Set that are not pre-defined by MagicDraw.

Figure 29.1. MagicDraw organisation



29.2.2. Classes

New classes should be added to one of the sub-packages inside the `Data layer/Classes` or `Core layer/Classes` modules. It is very simple:

1. Select the sub-package in the overview and click with the right mouse button.
2. Select the `New element Class` menu item in the menu that pops up.
3. The overview will expand to add a new class. Enter the name of the class and press enter.

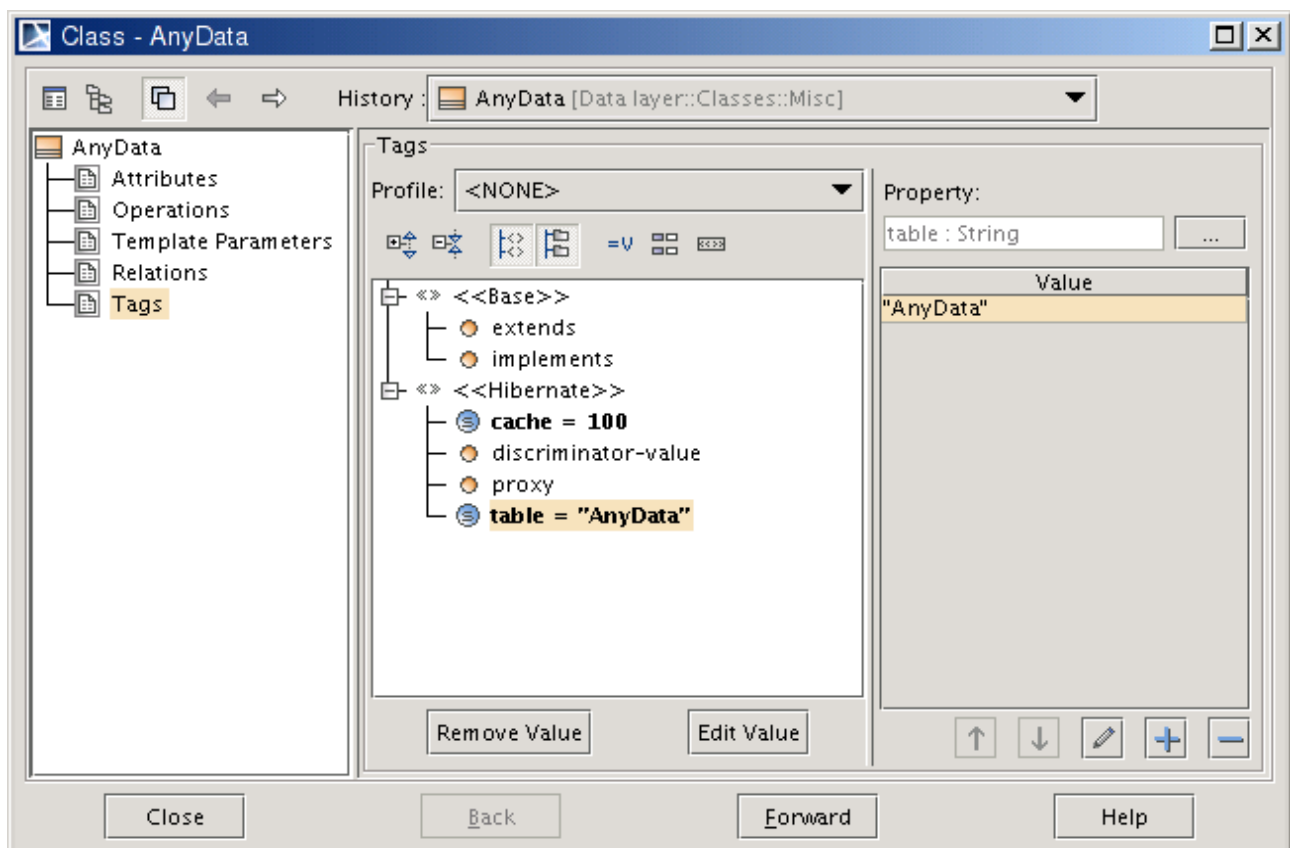
Data layer classes

If you added a class to the data layer you also need to record some important information.

- The database table the data is stored in
- If the second-level cache is enabled or not
- If proxies are enabled or not
- The superclass
- Implemented interfaces
- Simple properties, ie. strings, numbers, dates
- Associations to other classes

To achieve this we have slightly altered the meaning of some UML symbols. For example we use the access modifier symbols (+, ~ and -) to indicate if a property is updatable or not. Some of the information needed is specified as *tagged values* that can be attached to a class. Double-click on the new class to bring up it's properties dialog box. Switch to the **Tags** configuration page.

Figure 29.2. Setting tagged values



We have defined the following tags:

table

The name of the database table where the items should be stored.

cache

The number of items to store in the second-level cache of Hibernate. Only specify a value if caching should be enabled.

proxy

A boolean flag to indicate if proxies should be used or not.

extends

Select the superclass of this class.

implements

Specify which interfaces the class implements. To save space we use the following one-letter abbreviations:

- A = AnnotatableData
- B = BatchableData
- D = DiskConsumableData
- E = ExtendableData
- F = FileAttachableData
- G = RegisteredData
- L = LoggableData
- N = NameableData
- O = OwnableData
- R = RemoveableData
- S = ShareableData
- T = SystemData

discriminator-value

Used for classes that share the underlying database table. The discriminator value is used so that Hibernate knows which subclass to create.

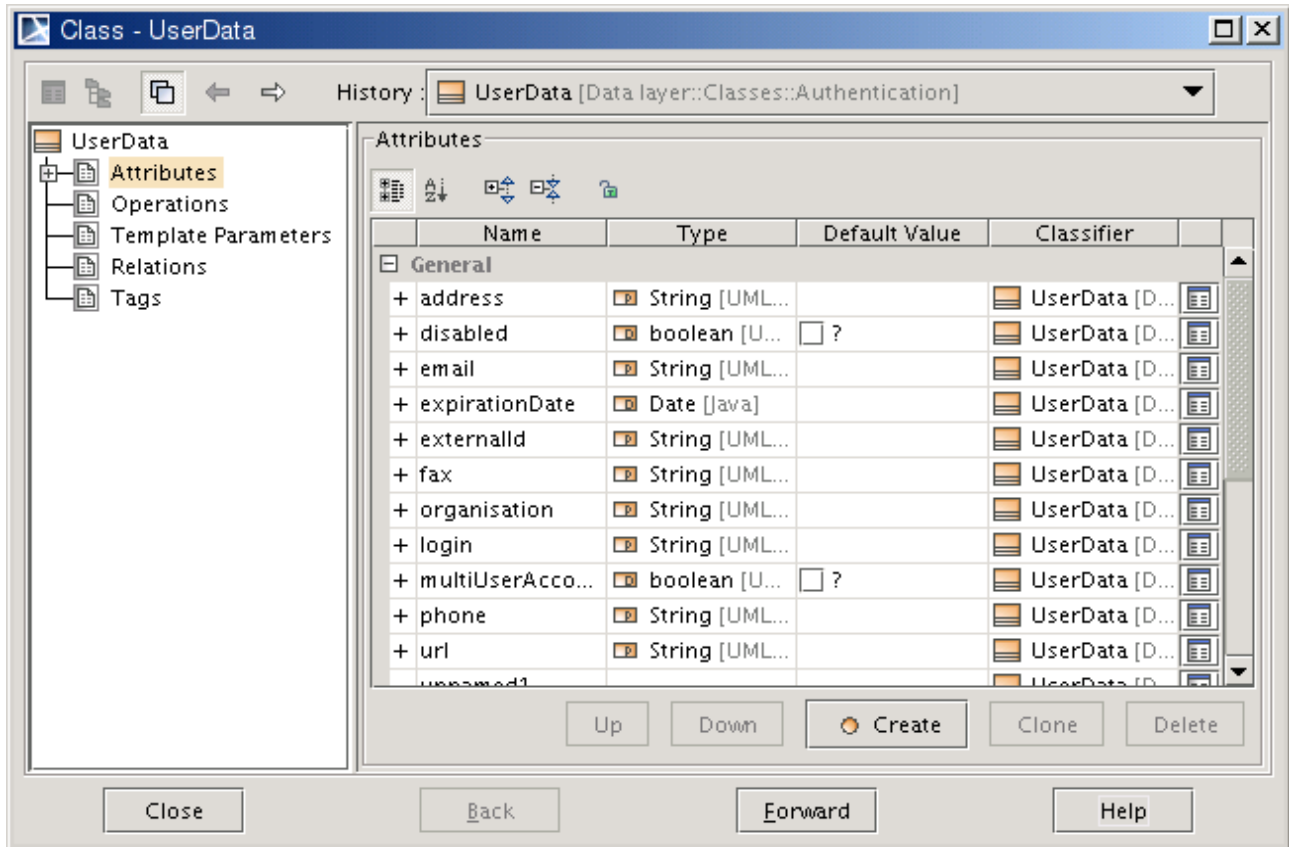
Simple properties are strings, numbers, dates, etc. that are part of an object. Properties are entered as attributes of the class. The easiest way to enter properties are by typing directly in a diagram. It can also be done from the **Attributes** configuration page.

Each attribute must have information about:

- The name of the attribute, ie. the name of the get/set method without the get or set part and starting with a lowercase letter.
- The data type: enter or select the Java object type in the **Type** selection list.
- If null values are allowed or not. Specify a multiplicity of 1 if a non-null value is required, but only if the underlying datatype can hold null values.
- If it is modifiable or not. From the **Visibility** list, select one of the following:
 - public (+): the attribute is modifiable. This translates to public get and set methods.

- package (~): the attribute can only be set once. This translates to public get and set methods and an `update="false"` tag in the Hibernate mapping.
- private (-): the attribute is private (will this ever be used?). This translates to package private get and set methods.

Figure 29.3. Class attributes



Associations to other classes are easiest created from a diagram view by drawing an **Association** link between the two classes. The ends should be given a name, multiplicity and visibility should be selected. For the visibility we use the same options as for attributes, but with a slightly different interpretation.

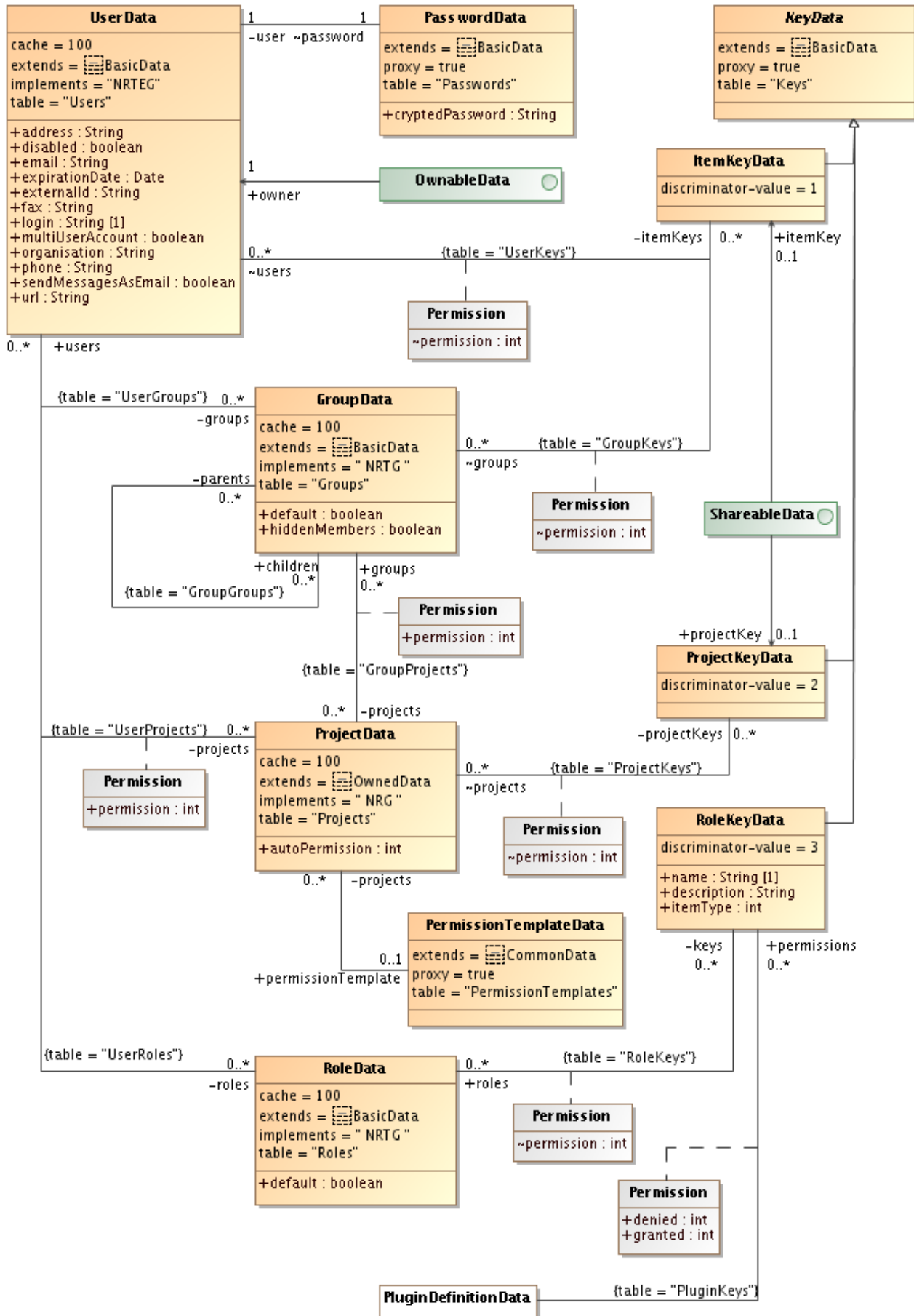
- public (+): the association is modifiable. This translates to public get and set methods for many-to-one associations. Many-to-many associations must have a package private set method since the set or map must never be replaced.
- package (~): same as public but the association cannot be changed once it has been created. For many-to-one associations an `update="false"` tag in the Hibernate mapping should be used. For many-to-many association there is no corresponding tag. It will be the responsibility of the core to make sure no modifications are done.
- private (-): this is the inverse end of an association. Only used for one-to-many and many-to-many associations and translates to package private get and set methods.

If the association involves a join table (many-to-many) the name of that table should be entered as a tagged value to the association.

If the association have values attached to it, use the **Association class** link type and enter information about the values in the attached class.

A lot more can be said about this, but it is probably better to have a look at already existing diagrams if you have any questions. The authentication overview shown below is one of the most complex diagrams that involves many different links.

Figure 90.4 Authentication UML diagram



Core layer classes

TODO

29.2.3. Diagrams

Create a new diagram

New diagrams should be added to one of the sub-packages inside the `Data layer/Diagrams` or `Core layer/Diagrams` modules. It is very simple:

1. Select the sub-package in the overview and click with the right mouse button.
2. Select the New diagram Class diagram menu item in the menu that pops up.
3. The overview will expand to add a new diagram. A new empty diagram frame is also opened on the right part of the screen. Enter the name of the diagram and press enter.

Only class diagrams are fully supported

The community edition of MagicDraw only has full support for class diagrams. The other diagram types has limitations, in the number of objects that can be created.

To display a class in a diagram, simply select the class in the overview and drag it into to the diagram.

Visual appearance and style

We have defined several different display style for classes. To set a style for a class right click on it in a diagram and select the Symbol properties menu item. In the bottom of the pop-up, use the **Apply style** selection list to select one of the predefined styles.

- Data class: Use this style for all primary data classes in a diagram. It will display all info that we are interested in.
- External class: Use this style for related classes that are actually part of another diagram. This style will hide all information except the class name. This style can be used for both data layer and core layer classes.
- Association class: Use this style for classes that hold information related to an association between two other classes. Classes with this style are displayed in a different color. This style can be used for both data layer and core layer classes.
- Core class: Use this style for all primary core classes in a diagram. It will display all info that we are interested in.

Save diagram as image

When the diagram is complete, save it as a PNG image in the `<base-dir>/doc/src/docbook/figures/uml` directory.

29.3. Javadoc

Existing Javadoc documentation is available on-line at: <http://base.thep.lu.se/chrome/site/doc/api/index.html>.

The BASE API is divided into four different parts on the package level.

- Public API - All classes and methods in the package are public. May be used by client applications and plug-ins. In general, backwards compatibility will be maintained.

- **Extension API** - All classes and methods in the package intended for internal extensions only. Not part of the public API and should not be used by client applications or plug-in.
- **Internal API** - All classes and methods in the package are internal. Should never be used by client application or plug-ins.
- **Mixed Public and Internal API** - Contains a mix of public and internal classes. Check the Javadoc for each class/method before using it.

Introduction to the Base API and it's parts can be found on the start page of Base Javadoc. Plugin developers and other external developers should pay most attention to the public API. What we consider to be the public part of the API is discussed in Section 28.1, "The Public API of BASE" (page 287).

29.3.1. Writing Javadoc

This section only covers Javadoc comments, how to write proper none-Javadoc comments are described in Section 30.3.2, "General coding style guidelines" (page 368)

General information about Javadoc and how it is written in a proper way can be found at <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>. The rule when coding in Base is that all packages, classes, interfaces, public methods and public attributes must be commented in Javadoc style. It is also recommended that private and protected methods has some comments, but maybe not as detailed as the public ones. Below follow more specific details what to have in mind when writing Javadoc in the Base project.

General

General things that are common for all Javadoc comments in Base.

- All comments should be in English.
- Do not start each new line of comment with a star.
- If a comment is mostly related to the inner workings of BASE, it should be tagged with

```
@base.internal
```

Package comments

Package comments should be placed in a file named `package.html` in the source code directory.

Is the package public or internal?

This information should be added in the `package.html` file. You must also modify the `build.xml` file. The `doc.javadoc` target contains `<group>` tags which lists all packages that are part of each group.

Class and interface comments

A comment for a class or interface should start with a general description. The class comment should then give information about what the class can be used for, while an interface comment more should inform which kinds of classes that are supposed to implement the interface.

```
@author
```

The first name of the author(s) of the class.

```
@since
```

The BASE verion when this class or interface was added.

```
@see
```

Optional. Links to some related subjects.

```
@base.modified
```

Optional. Some classes has this tag too. This is for give the class-file a time stamp when it is checked in to subversion.

Method comments

A method comment should start with a general description of the method and what it does. The following tags must be present in this order:

```
@param
```

One tag for each parameter of the method. Make sure to tell what values are allowed and what will happen if a disallowed value is passed.

```
@return
```

What is returned by the method. Make sure to tell what values can be returned (ie. if it can be null).

```
@throws
```

One tag for each exception that the method can throw and describe when and why it will be thrown.

```
@since
```

Use only this tag together with methods added in a later version then the one the class was created in. It holds which version the method first was available in.

```
@see
```

Optional. Link to relevant information, one tag for each link.

Attribute comments

If the attribute is a static final, describe what the attribute is for and where it is typically used. Other attributes can often be explained through their getter and setter methods.

Chapter 30. Core developer reference

30.1. Publishing a new release

This documentation is available on the BASE wiki¹.

30.2. Subversion / building BASE

This documentation is available on the BASE wiki².

30.3. Coding rules and guidelines

30.3.1. Development process and other important procedures

This section describes the development process we try to use in the BASE project. It is not carved in stone and deviations may occur. For every new feature or enhancement the procure below should be followed. If you encounter any problems, arrange a group meeting. Someone else may have the solution! The text is biased towards adding new items to BASE, but it should be possible to use the general outline even for other types of features.

1. Group meeting

- The group should have a short meeting and discuss the new or changed feature. Problem areas should be identified, not solved!
- The person who is going to make the analysis, design and development is responsible for taking notes. They should be kept until the analysis and design phase has been finished.
- A follow-up meeting should be held at the end of the analysis phase.
- A single meeting may of course discuss more than one feature.

2. Analysis and design

- Create an diagram of the classes including their properties, links and associations. Use the already existing diagrams and code as a template. The diagram should have information about cache and proxy settings.
- Write a short document about the diagram, especially things that are not obvious and explain any deviations from the recommendations in the coding guidelines.
- Identify things that may affect backwards compatibility. For more information about such things read Section 28.1, “The Public API of BASE” (page 287) and Section 30.3.3, “API changes and backwards compatibility” (page 374).
- Identify what parts of the documentation that needs to changed or added to describe the new feature. This includes, but is not limited to:

¹ <http://base.thep.lu.se/wiki/ReleaseProcedure>

² <http://base.thep.lu.se/wiki/BuildingBase>

- User and administrator documentation, how to use the feature, screenshots, etc.
- Plug-in and core developer documentation, code examples, database schema changes, etc.
- If there are any problems with the existing code, these should be solved at this stage. Write some prototype code for testing if necessary.
- Group meeting to verify that the specified solution is ok, and to make sure everybody has enough knowledge of the solution.

3. Create the classes for the data layer

- If step 2 is properly done, this should not take long.
- Follow the coding guidelines in Section 30.3.4, “Data-layer rules” (page 375).
- At the end of this step, go back and have a look at the diagram/documentation from the analysis and design phase and make sure everything is still correct.

4. Create the corresponding classes in the core layer

- For simple cases this is also easy. Other cases may require more effort.
- If needed, go back to the analysis and design phase and do some more investigations. Make sure the documentation is updated if there are changes.

5. Create test code

- Build on and use the existing test as much as possible.

6. Write code to update existing installations

Important

- If the database schema is changed or if there for some reason is need to update existing data in the database, the `Install.SCHEMA_VERSION` counter must be increased.
- Add code to the `net.sf.basedb.core.Update` class to increase the schema version and modify data in existing installations.

7. Write new and update existing user documentation

- Most likely, users and plug-in developers wants to know about the feature.

Important

Do not forget to update the Appendix I, *API changes that may affect backwards compatibility* (page 443) document if you have introduced any incompatible changes.

30.3.2. General coding style guidelines

Naming

General

All names should be in English. Names should be descriptive and derived from the the domain the code is intended for. Try to avoid names longer than twenty characters.

Packages

Package names must be in all lower case letters. The top-level package of BASE is `net.sf.basedb`.

Classes and Interfaces

Names of classes and interfaces should be a concatenation of one or more words. The initial letter of all words in the name, including the first word, should be upper case letters. The rest of the characters should be lower case. Example:

```
public class SoftwareType
{
    ...
}
```

Constant member variables

Constant member variables, usually defined *static final*, should be named using all upper case characters. Words in the name should be separated by underscore characters. Example:

```
public static final int VERSION_NUMBER = 3;
```

Private member variables

Private member variables should be a concatenation of one or more descriptive words. The initial letter of all words in the name, except the first word, should be upper case letters. The rest of the characters should be lower case. Example:

```
private String loginComment;
```

Methods

Methods should be named using a descriptive statement, usually made up by several words. Typically the first word is a verb, stating the action and the others stating the target and attributes. Lower and upper case letters should then be mixed, with all words in the name except the first one starting with an upper case letter and the rest being in lower case letters. Example:

```
public ItemQuery<Annotation> getAllInheritedAnnotations()
{
    ...
}
```

Mutator (get/set) methods

Avoid direct access to attributes (member variables) from outside of a class. Instead make attributes private and use mutator methods to access them. Prefix the mutator methods with *get* and *set* respectively to fetch or change an attribute. If the getter returns a boolean value prefix the mutator method with (typically) *is*, *has* or *can*. Examples:

```
private String name;
public String getName()
{
    return name;
}
public void setName(String name)
{
    this.name = name;
}

private boolean activated;
public boolean isActivated()
{
    return activated;
}
```

Exceptions

The names of Exceptions must end with the word *Exception*. Example:

```
public class NoMoreNumbersException
    extends Exception
{
    ...
}
```

Layout and comments

Interface layout

Interfaces should only have public members, i.e. static attributes and method prototypes.

White space

All code must be properly indented. In general each new block starts a new indentation level. Use *tab* when indenting.

Code blocks

The starting brace "{" of a code block should be placed on a line by itself at the same indentation level as the preceeding line of code. The first line in the new block should be indented one tab-stop more. The ending brace "}" should be placed at the same indentation level as the matching starting brace. Use braces even if the code block is only one line. Example:

```
public String getName()
{
    if (name == null)
    {
        return "unknown";
    }
    else
    {
        return name;
    }
}
```

Javadoc

Packages, classes, public methods and public attributes should be commented in Javadoc style. It is recommended that private and protected methods also has some comments, but maybe not as detailed as the public ones.

- All comments should be in English.
- Do not start each line of a comment with a star.

More info about Javadoc can be found at: <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>

Package comments

Place package comments in a file named `package.html` in the source code directory.

Class comments

A class comment should start with a general description of the class and what it does. Use the `@author` tag to specify the names of the programmers that was involved in coding the file and the `@since` tag to specify the version of BASE when the class first appeared. Example:

```
/**
 * ...
 * @author Nicklas, Martin
 * @since 2.0
 */
public class BasicItem
{
    ...
}
```

```
}
```

Method comments

A method comment should start with a general description of the method and what it does. Use `@param` to describe each parameter and `@return` to describe what the method returns. Use `@throws` to describe all checked exceptions including when and why they can be thrown. Use `@see` to link to other related method and information.

Attribute comments

If it is a static final attribute, describe what the attribute is for and where it is typically used.

`@base.developer`

The `@base.developer` tag can be used anywhere to add comments that are mainly targeted to the developers of BASE.

`@base.internal`

The `@base.internal` tag can be used at package and class-level documentation to mark that part of the code as an internal API that should not be used by plug-ins and other client code. See Section 28.1, “The Public API of BASE” (page 287).

Inline comments

- All comments should be in English.
- Comment on why instead of what. Your code should be clear enough to answer questions on what it does. It is more important to document why it does it.
- Do not place end line comments at the end of statements.
- Do not use decorated banner like comments, as these are hard to maintain. If more extensive commenting is needed - use Javadoc.
- Avoid using semicolon (;) as part of inline comments. Searching for all comments containing a semicolon is used to find commented out code blocks.

Commented out code

Avoid leaving code that is commented out. It is a distraction when maintaining the code. Sometimes, for example during a big refactoring, it is not possible to fix everything at once. In this case it is allowed to comment out code, but it is recommended that a *TODO* marker (see below) is added to make it easier to find places that need to be fixed later.

Todo comments

If there are parts of the code that cannot be completed at the time the majority of the code is written, place a comment starting with *TODO* (in capital letters), followed with a description of what needs to be done. If there is a ticket in the Trac server, use the ticket number in the comment. Example:

```
public static Date copy(Date value)
{
    return value == null ? null : new Date(value.getTime());
    // TODO (#1234): check if there is a better way to copy
}
```

Subversion comment and GNU licence message

Each file should start with a subversion comment and the GNU licence and copyright message. Non-java files should also include this information, in a format appropriate for the specific file.

```
/*
```

```
$Id: core_ref.xml 5820 2011-10-24 10:46:11Z nicklas $

Copyright (C) Authors contributing to this file.

This file is part of BASE - BioArray Software Environment.
Available at http://base.thep.lu.se/

BASE is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 3
of the License, or (at your option) any later version.

BASE is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with BASE. If not, see <http://www.gnu.org/licenses/>.
*/
```

Statements

Package and import statements

The package statement is the first statement to appear in the source file. All declared classes must belong to a named package. It is not allowed for classes to belong to the "default" package, i.e. to omit the package statement in the source file. Exception: Test code may belong to the default package.

Import statements should follow directly after the package statement. Try to avoid preceding class names in code with package names. Instead use import statements.

Wildcards in import statements make it difficult to see dependencies, other than those to packages. Therefore import classes/interfaces explicitly. The only exception is when classes in sub-packages to the current package are accessed. Such items are either imported explicitly or the whole subpackage may be imported with wildcard. However, avoid wildcard imports when sub-packages are large - more than approximately 4 items.

Do not explicitly import packages or classes not used in your program code. Try to maintain an alphabetical order among the imported classes. Group classes from the same package. Most IDE:s have functionality for maintaining import statements. Use it.

Inside a class, attributes and methods should be organised in the following order:

- public static final attributes
- other static attributes
- public static methods
- other static methods
- private attributes
- constructors
- Methods defined by interfaces, grouped by the interface in which they are defined
- Methods that override a method from a superclass, with methods from the topmost superclass first
- Additional methods for the specific class

Classes and interfaces

Class and interface statements should be organized in the following manner:

- class or interface statement
- extends statement indented on a separate row
- implements statement indented on one or more rows
- class body

```
public class SomeClass
    extends SomeBase
    implements Cloneable, SomeInterface
{
    ...
}

public interface Testable
    extends Cloneable
{
    ...
}
```

Methods

If a method throws checked exceptions, these should be declared indented on a separate row directly after the method declaration. Example:

```
public int getNextValue(int previousValue)
    throws NoMoreValuesException
{
    ...
}
```

Local variables

Local variables should be declared and initialized where they are used. They should be declared in the smallest possible scope. Avoid overloading variable names in inner blocks.

Array declarations

The square brackets indicating an array of something should be immediately to the right of whatever class or datatype the array consists of (e.g. `String[] args`) do not use C-style array declarations (`String args[]`).

Conditional statements

- There must always be braces around the block following the condition control structure, even if it's just a single statement block. This doesn't apply to the cases when the single statement is on the same line as the condition.
- Avoid placing statements resulting in side effects (e.g. function calls) in the condition construct.
- Do not jump out of conditional blocks with `break`, `return` or `exit`. The exception is the usage of `break` in switch-statements.
- Use `continue`-statements in `for`- and `while`- loops with caution. It's recommended to mark the statement with a clear comment.

```
// IF-statements:
if (...)
{
    ...
}
```

```
}
else
{
    ...
}

if (...) ...;

// FOR-statements:
for (init; condition; update)
{
    ...
    /* #### CONTINUE-STATEMENT #### */
    if (...) continue;
    ...
}

// WHILE-statement:
while (condition)
{
    ...
}

// DO-WHILE-statement:
do
{
    ...
}
while (condition);

// SWITCH-statement:
switch (operand)
{
    case: ...
    {
        ...
        break;
    }
    default:
    {
        ...
    }
}

// Exception blocks:
try
{
    ...
}
catch (SpecialException se)
{
    ...
}
catch (Exception e)
{
    ...
}
finally
{
    ...
}
```

30.3.3. API changes and backwards compatibility

The main rule is to do not introduce any changes that are backwards incompatible. That is, existing client applications and plug-ins should continue to run in the next release of BASE, without the need to change them. It may sound easy but there are many things to watch out for. There is a great article about this subject on http://wiki.eclipse.org/index.php/Evolving_Java-based_APIs.

The Public API

Not all code in BASE is considered to be part of the public API. See Section 28.1, “The Public API of BASE” (page 287) and the javadoc³ for information about the public API. Changes made to the internal API does not have to follow the same rules.

Binary compatibility

This is hardest requirement and means that existing binaries must run with the updated BASE version. The Eclipse document discusses this type of compatibility in great detail.

Contract compatibility

Methods should continue to do things in the same ways as before. Avoid introducing side-effects and expanding/contracting the allowed range of return or parameter values. This may not always be easy to keep this type of compatibility. For example, adding a new option to an enumeration may break code that is not prepared for it.

Internal data structure compatibility

It may not be possible to keep the internal data structures. If they change the update script should convert the old data to the new format. Avoid exposing the internal data structure to client applications. Use wrapper classes, etc, to hide as much as possible.

Source code compatibility

This is not an important issue and in most cases the problems are easy to fix.

Do not forget to log changes!

Any change that may affect backwards compatibility must be logged in Appendix I, *API changes that may affect backwards compatibility* (page 443).

30.3.4. Data-layer rules

The coding guidelines for this package has been slightly modified from the the general coding guidelines. Here is a short list with the changes.

Class and interface names

Class names should follow the general guidelines, but should in most cases end with `Data`.

```
public class SampleData
    extends CommonData
    implements DiskConsumableData
{
    ...
}
```

Attributes and methods order

Inside a class, attributes and methods should be organised in related groups, ie. the private attribute is together with the getter and setter methods that uses that attribute. This makes it easy to re-use existing code with copy-and-paste operations.

```
public static int long MAX_ADDRESS_LENGTH = 255;
private String address;
/**
 * @hibernate.property column="`address`" type="string" length="255" not-null="false"
 */
public String getAddress()
{
    return address;
}
public void setAddress(String address)
```

³ ../../../../api/index.html

```
{
    this.address = address;
}

private int row;
/**
    @hibernate.property column="`row`" type="int"
 */
public int getRow()
{
    return row;
}
public void setRow(int row)
{
    this.row = row;
}
```

Extend/implement the basic classes and interfaces

Each data-class must inherit from one of the already existing abstract base classes. They contain code that is common to all classes, for example implementations of the `equals()` and `hashCode()` methods or how to link with the owner of an item. For information about which classes/interfaces that can be used see Section 28.2.1, “Basic classes and interfaces” (page 290).

Define a public no-argument constructor

Always define a public no-argument constructor. No other constructors are needed. If we want to use other persistence mechanisms or serializability in the future this type of constructor is probably the most compatible. The constructor should be empty and not contain any code. Do not initialise properties or create new objects for internal use. Most of the time the object is loaded by Hibernate and Hibernate will ensure that it is properly initialised by calling all setter methods.

For example, a many-to-many relation usually has a `Set` or a `Map` to hold the links to the other objects. Do not create a new `HashSet` or `HashMap` in the constructor. Wait until the getter method is called and only create a new object if Hibernate hasn't already called the setter method with it's own object. See the code example below. There is also more information about this in Many-to-many and one-to-many mappings (page 383).

```
// From GroupData.java
public GroupData()
{}

private Set<UserData> users;
public Set<UserData> getUsers()
{
    if (users == null) users = new HashSet<UserData>();
    return users;
}
```

See also:

- "Hibernate in action", chapter 3.2.3 "Writing POJOs", page 67-69
- Hibernate reference documentation: 4.1.1. Implement a no-argument constructor⁴

Object identity

We use database identity to compare objects, ie. two objects are considered equal if they are of the same class and have the same id, thus representing the same database row. All this stuff is implemented by the `BasicData` class. Therefore it is required that all classes are subclasses of this class. It is recommended that the `equals()` or `hashCode()` methods are not overridden by any of the subclasses. We would have liked to make them final, but then the proxy feature of Hibernate would not work.

Avoid mixing saved and unsaved objects

The approach used for object identity may give us a problem if we mix objects which hasn't been saved to the database, with objects loaded from the database. Our recommendation is to avoid that, and save any objects to the database before adding them to sets, maps or any other structure that uses the `equals()` and `hashCode()` methods.

To be more specific, the problem arises because the following two rules for hash codes are contradicting when the hash code is based on the database id:

1. The hash code of an object mustn't change.
2. Equal objects must have equal hash code.

For objects in the database, the hash code is based on the id. For new objects, which doesn't have an id yet, we fall back to the system hash code. But, what happens when we save the new object to the database? If nobody has asked for the hash code it is safe to use the id, otherwise we must stick with the system hash code. Now, imagine that we load the same object from the database in another Hibernate session. What will now happen? The loaded object will have its hash code based on the id but the original object is still using the system hash code, which most likely is not the same as the id. Yet, the `equals()` method returns true. This is a violation of the contract for the `equals()` method. If these two objects are used in a set it may cause unexpected behaviour. Therefore, do not put new objects in a set, or other collection, that calls the `hashCode()` method before the object is saved to the database.

See also:

- "Hibernate in action", chapter 3.4 "Understanding object identity", page 87-90
- "Hibernate in action", chapter 4.1.4 "The scope of object identity", page 119-121
- "Hibernate in action", chapter 4.1.6 "Implementing `equals()` and `hashCode()`", page 122-126
- Hibernate reference documentation: 4.3. Implementing `equals()` and `hashCode()`⁵

No final methods

No methods should be tagged with the `final` keyword. This is a requirement to be able to use the proxy feature of Hibernate, which we need for performance reasons.

See also:

- Hibernate reference documentation: 4.1.3. Prefer non-final classes⁶
- Hibernate reference documentation: 21.1.3. Single-ended association proxies⁷

Second-level cache

To gain performance we use the second-level cache of Hibernate. It is a transparent feature that doesn't affect the code in any way. The second-level cache is configured in the `hibernate.cfg.xml` and `ehcache.xml` configuration files and not in the individual class mapping files. BASE is shipped with a standard configuration, but different deployment scenarios may have to fine-tune the cache settings for that particular hardware/software setup. It is beyond the scope of this document to discuss this issue.

The second-level cache is suitable for objects that are rarely modified but are often needed. For example, we do not expect the user information represented by the `UserData` object to change very often, but it is displayed all the time as the owner of various items. Before coming up with a good caching strategy we have to answer the following questions:

1. Should objects of this class be cached at all?
2. How long timeout should we use?

3. How many objects should we keep in memory or on disk?

The first question is the most important. Good candidates are classes with few objects that change rarely, but are read often. Also, objects which are linked to by many other objects are good candidates. The `UserData` class is an example which matches all three requirements. The `TagData` class is an example which fulfils the first two. The `BioMaterialEventData` class is on the other hand a bad cache candidate, since it is not linked to any other object than a `BioMaterialData` object.

The answer to the second question depends on how often an object is modified. For most objects this time is probably several days or months, but we would not gain much by keeping objects in the cache for so long. Suddenly, the information has changed and we won't risk that old information is kept that long. We have set the timeout to 1 hour for all classes so far, and we don't recommend a longer timeout. The only exception is for immutable objects, that cannot be changed at all, which may have an infinite timeout.

The answer to the third question depends a lot on the hardware (available memory). With lots of memory we can afford to cache more objects. Caching to disk is not really necessary if the database is on the same machine as the web server, but if it is on another machine we have to consider the network delay to connect to the database versus the disk access time. The default configuration does not use disk cache.

See also:

- "Hibernate in action", chapter 5.3 "Caching theory and practice", page 175-194.
- Hibernate reference documentation: 21.2. The Second Level Cache⁸

Proxies

Proxies are also used to gain performance, and they may have some impact on the code. Proxies are created at runtime (by Hibernate) as a subclass of the actual class but are not populated with data until some method of the object is called. The data is loaded from the database the first time a method other than `getId()` is called. Thus, we can avoid loading data that is not needed at a particular time.

There can be a problem with using the `instanceof` operator with proxies and the table-per-class-hierarchy mapping. For example, if we have the abstract class `Animal` and subclasses `Cat` and `Dog`. The proxy of an `Animal` is a runtime generated subclass of `Animal`, since we do not know if it is a `Cat` or `Dog`. So, `x instanceof Dog` and `x instanceof Cat` would both return false. If we hadn't used a proxy, at least one of them would always be true.

Proxies are only used when a not-null object is linked with many-to-one or one-to-one from another object. If we ask for a specific object by id, or by a query, we will never get a proxy. Therefore, it only makes sense to enable proxies for classes that can be linked from other classes. One-to-one links on the primary key where null is allowed silently disables the proxy feature, since Hibernate doesn't know if there is an object or not without querying the database.

Proxy vs. cache

The goal of a proxy and the second-level cache are the same: to avoid hitting the database. It is perfectly possible to enable both proxies and the cache for a class. Then we would start with a proxy and as soon as a method is called Hibernate would look in the second-level cache. Only if it is not there it would be loaded from the database. But, do we really need a proxy in the first place? Well, I think it might be better to use only the cache or only proxies. But, this also makes it even more important that the cache is configured correctly so there is a high probability that the object is already in the cache.

If a class has been configured to use the second-level cache, we recommend that proxies are disabled. For child objects in a parent-child relationship proxies should be disabled, since they

have no other links to them than from the parent. If a class can be linked as many-to-one from several other classes it makes sense to enable proxies. If we have a long chain of many-to-one relations it may also make sense to enable proxies at some level, even if the second-level cache is used. In that case we only need to create one proxy instead of looking up several objects in the cache. Also, think about how a particular class most commonly will be used in a client application. For example, it is very common to display the name of the owner of an item, but we are probably not interested in displaying quota information for that user. So, it makes sense to put users in the second-level cache and use proxies for quota information.

Batchable classes and stateless sessions

Hibernate has a *stateless session* feature. A stateless session has no first-level cache and doesn't use the second-level cache either. This means that if we load an item with a stateless session Hibernate will always traverse many-to-one and one-to-one associations and load those objects as well, unless they are configured to use proxies.

BASE use stateless sessions for loading `BatchableData` items (reporters, raw data and features) since they are many and we want to use as little memory as possible. Here it is required that proxies are enabled for all items that are linked from any of the batchable items, ie. `RawBioAssay`, `ReporterType`, `ArrayDesignBlock`, etc. If we don't do this Hibernate will generate multiple additional select statements for the same parent item which will affect performance in a bad way.

On the other hand, the proxies created from a stateless session cannot later be initialised. We have to get the ID from the proxy and then load the object using the regular session. But this can also result in lots of additional select statements so if it is known before that we need some information it is recommended that a `FETCH JOIN` query is used so that we get fully initialized objects instead of proxies to begin with.

Here is a table which summarises different settings for the second-level cache, proxies, batch fetching and many-to-one links. Batch fetching and many-to-one links are discussed later in this document.

First, decide if the second-level cache should be enabled or not. Then, if proxies should be enabled or not. The table then gives a reasonable setting for the batch size and many-to-one mappings. NOTE! The many-to-one mappings are the links from other classes to this one, not links from this class.

The settings in this table are not absolute rules. In some cases there might be a good reason for another combination. Please, write a comment about why the recommendations were not followed.

Table 30.1. Choosing cache and proxy settings

Global configuration	Class mapping		Many-to-one mapping
Cache	Proxy	Batch-size	Outer-join
no	no*	yes	true
yes	no*	no	false
no	yes	yes	false
yes	yes	no	false

* = Do not use this setting for classes which are many-to-one linked from a batchable class.

See also:

- "Hibernate in action", chapter 4.4.6 "Selecting a fetching strategy in mappings", page 146-147
- "Hibernate in action", chapter 6.4.1 "Polymorphic many-to-one associations", page 234-236
- Hibernate reference documentation: 21.1.3. Single-ended association proxies⁹

Hibernate mappings

We use Javadoc tags to specify the database mapping needed by Hibernate. The tags are processed by XDoclet at build time which generates the XML-based Hibernate mapping files.

XDoclet doesn't support all mappings

The XDoclet that we use was developed to generate mapping files for Hibernate 2.x. Since then, Hibernate has released several 3.x versions, and the mapping file structure has changed. Some changes can be handled by generating a corresponding 2.x mapping and then converting it to a 3.x mapping at build time using simple search-and-replace operations. One such case is to update the DTD reference to the 3.0 version instead of the 2.0 version. Other changes can't use this approach. Instead we have to provide extra mappings inside an XML files. This is also needed if we need to use some of the new 3.x features that has no 2.x counterpart.

Class mapping

```
/**
 * This class holds information about any data...
 * @author Your name
 * @since 3.0
 * @hibernate.class table="`Anys`" lazy="false" batch-size="10"
 */
public class AnyData
    extends CommonData
{
    // Rest of class code...
}
```

The class declaration must contain a `@hibernate.class` Javadoc entry where Hibernate can find the name of the table where items of this type are stored. The table name should generally be the same as the class name, without the ending `Data` and in a plural form. For example `UserData` `Users`. The back-ticks (```) around the table name tells Hibernate to enclose the name in whatever the actual database manager uses for such things (back-ticks in MySQL, quotes for an ANSI-compatible database).

Always set the lazy attribute

The `lazy` attribute enables/disables proxies for the class. Do not forget to specify this attribute since the default value is `true`. If proxies are enabled, it may also make sense to specify a `batch-size` attribute. Then Hibernate will load the specified number of items in each `SELECT` statement instead of loading them one by one. It may also make sense to specify a batch size when proxies are disabled, but then it would probably be even better to use eager fetching by setting `outer-join="true"` (see many-to-one mapping).

Classes that are linked with a many-to-one association from a batchable class must specify `lazy="true"`. Otherwise the stateless session feature of Hibernate may result in a large number of `SELECT`s for the same item, or even circular loops if two or more items references each other.

Remember to enable the second-level cache

Do not forget to configure settings for the second-level cache if this should be enabled. This is done in the `hibernate.cfg.xml` and `ehcache.xml`.

See also:

- "Hibernate in action", chapter 3.3 "Defining the mapping metadata", page 75-87
- Hibernate reference documentation: 5.1.3. Entity¹⁰

Property mappings

Properties such as strings, integers, dates, etc. are mapped with the `@hibernate.property` Javadoc tag. The main purpose is to define the database column name. The column names

should generally be the same as the get/set method name without the get/set prefix, and with upper-case letters converted to lower-case and an underscore inserted. Examples:

- `getAddress()` `column="`address`"`
- `getLoginComment()` `column="`login_comment`"`

The back-ticks (```) around the column name tells Hibernate to enclose the name in whatever the actual database manager uses for such things (back-ticks in MySQL, quotes for an ANSI-compatible database).

String properties

```
public static int long MAX_STRINGPROPERTY_LENGTH = 255;
private String stringProperty;
/**
 * Get the string property.
 * @hibernate.property column="`string_property`" type="string"
 *                    length="255" not-null="true"
 */
public String getStringProperty()
{
    return stringProperty;
}
public void setStringProperty(String stringProperty)
{
    this.stringProperty = stringProperty;
}
```

Do not use a greater value than 255 for the `length` attribute. Some databases has that as the maximum length for character columns (ie. MySQL). If you need to store longer texts use `type="text"` instead. You can then skip the `length` attribute. Most databases will allow up to 65535 characters or more in a text field. Do not forget to specify the `not-null` attribute.

You should also define a public constant `MAX_STRINGPROPERTY_LENGTH` containing the maximum allowed length of the string.

Numerical properties

```
private int intProperty;
/**
 * Get the int property.
 * @hibernate.property column="`int_property`" type="int" not-null="true"
 */
public int getIntProperty()
{
    return intProperty;
}
public void setIntProperty(int intProperty)
{
    this.intProperty = intProperty;
}
```

It is also possible to use `Integer`, `Long` or `Float` objects instead of `int`, `long` and `float`. We have only used it if null values have some meaning.

Boolean properties

```
private boolean booleanProperty;
/**
 * Get the boolean property.
```

```
@hibernate.property column="`boolean_property`"
    type="boolean" not-null="true"
*/
public boolean isBooleanProperty()
{
    return booleanProperty;
}
public void setBooleanProperty(boolean booleanProperty)
{
    this.booleanProperty = booleanProperty;
}
```

It is also possible to use a `Boolean` object instead of `boolean`. It is only required if you absolutely need null values to handle special cases.

Date values

```
private Date dateProperty;
/**
    Get the date property. Null is allowed.
    @hibernate.property column="`date_property`" type="date" not-null="false"
*/
public Date getDateProperty()
{
    return dateProperty;
}
public void setDateProperty(Date dateProperty)
{
    this.dateProperty = dateProperty;
}
```

Hibernate defines several other date and time types. We have decided to use the `type="date"` type when we are only interested in the date and the `type="timestamp"` when we are interested in both the date and time.

See also:

- "Hibernate in action", chapter 3.3.2 "Basic property and class mappings", page 78-84
- "Hibernate in action", chapter 6.1.1 "Built-in mapping types", page 198-200
- Hibernate reference documentation: 5.1.4. `property`¹¹
- Hibernate reference documentation: 5.2.2. Basic value types¹²

Many-to-one mappings

```
private OtherData other;
/**
    Get the other object.
    @hibernate.many-to-one column="`other_id`" not-null="true" outer-join="false"
*/
public OtherData getOther()
{
    return other;
}
public void setOther(OtherData other)
{
    this.other = other;
}
```

We create a many-to-one mapping with the `@hibernate.many-to-one` tag. The most important attribute is the `column` attribute which specifies the name of the database column to use for

the id of the other item. The back-ticks (```) around the column name tells Hibernate to enclose the name in whatever the actual database manager uses for such things (back-ticks in MySQL, quotes for an ANSI-compatible database).

We also recommend that the `not-null` attribute is specified. Hibernate will not check for null values, but it will generate table columns that allow or disallow null values. See it as an extra safety feature while debugging. It is also used to determine if Hibernate uses `LEFT JOIN` or `INNER JOIN` in SQL statements.

The `outer-join` attribute is important and affects how the cache and proxies are used. It can take three values: `auto`, `true` or `false`. If the value is `true` Hibernate will always use a join to load the linked object in a single select statement, overriding the cache and proxy settings. This value should only be used if the class being linked has disabled both proxies and the second-level cache, or if it is a link between a child and parent in a parent-child relationship. A `false` value is best when we expect the associated object to be in the second-level cache or proxying is enabled. This is probably the most common case. The `auto` setting uses a join if proxying is disabled otherwise it uses a proxy. Since we always know if proxying is enabled or not, this setting is not very useful. See Table 30.1, “Choosing cache and proxy settings” (page 379) for the recommended settings.

See also:

- "Hibernate in action", chapter 3.7 "Introducing associations", page 105-112
- "Hibernate in action", chapter 4.4.5-4.4.6 "Fetching strategies", page 143-151
- "Hibernate in action", chapter 6.4.1 "Polymorphic many-to-one associations", page 234-236
- Hibernate reference documentation: 5.1.7. Mapping one to one and many to one associations¹³

Many-to-many and one-to-many mappings

There are many variants of mapping many-to-many or one-to-many, and it is not possible to give examples of all of them. In the code these mappings are represented by `Set`'s, `Map`'s, `List`'s, or some other collection object. The most important thing to remember is that (in our application) the collections are only used to maintain the links between objects. They are (in most cases) not used for returning objects to client applications, as is the case with the many-to-one mapping.

For example, if we want to find all members of a group we do not use the `GroupData.getUsers()` method, instead we will execute a database query to retrieve them. The reason for this design is that the logged in user may not have access to all users and we must add a permission checking filter before returning the user objects to the client application. Using a query will also allow client applications to specify sorting and filtering options for the users that are returned.

```
// RoleData.java
private Set<UserData> users;
/**
 * Many-to-many from roles to users.
 * @hibernate.set table="`UserRoles`" lazy="true"
 * @hibernate.collection-key column="`role_id`"
 * @hibernate.collection-many-to-many column="`user_id`"
 * class="net.sf.basedb.core.data.UserData"
 */
public Set<UserData> getUsers()
{
    if (users == null) users = new HashSet<UserData>();
    return users;
}
void setUsers(Set<UserData> users)
{
    this.users = users;
}
```

As you can see this mapping is a lot more complicated than what we have seen before. The most important thing is the `lazy` attribute. It tells Hibernate to delay the loading of the related objects until the set is accessed. If the value is false or missing, Hibernate will load all objects immediately. There is almost never a good reason to specify something other than `lazy="true"`.

Another important thing to remember is that the getter method must always return the same object that Hibernate passed to the setter method. Otherwise, Hibernate will not be able to detect changes made to the collection and as a result will have to delete and then recreate all links. To ensure that the collection object is not changed we have made the `setUsers()` method package private, and the `getUsers()` will create a new `HashSet` for us only if Hibernate didn't pass one in the first place.

Let's also have a look at the reverse mapping:

```
// UserData.java
private Set<RoleData> roles;
/**
 * Many-to-many from users to roles
 * @hibernate.set table="`UserRoles`" lazy="true"
 * @hibernate.collection-key column="`user_id`"
 * @hibernate.collection-many-to-many column="`role_id`"
 * class="net.sf.basedb.core.data.RoleData"
 */
Set<RoleData> getRoles()
{
    return roles;
}
void setRoles(Set<RoleData> roles)
{
    this.roles = roles;
}
```

The only real difference here is that both the setter and the getter methods are package private. This is a safety measure because Hibernate will get confused if we modify both ends. Thus, we are forced to always add/remove users to/from the set in the `RoleData` object. The methods in the `UserData` class are never used by us. Note that we do not have to check for null and create a new set since Hibernate will handle null values as an empty set.

So, why do we need the second collection at all? It is never accessed except by Hibernate, and since it is lazy it will always be "empty". The answer is that we want to use the relation in HQL statements. For example:

```
SELECT ... FROM RoleData rle WHERE rle.users ...
SELECT ... FROM UserData usr WHERE usr.roles ...
```

Without the second mapping, it would not have been possible to execute the second HQL statement. The inverse mapping is also important in parent-child relationships, where it is used to cascade delete the children if a parent is deleted (see below).

Do not use the `inverse="true"` setting

Hibernate defines an `inverse="true"` setting that can be used with the `@hibernate.set` tag. If specified, Hibernate will ignore changes made to that collection. However, there is one problem with specifying this attribute. Hibernate doesn't delete entries in the association table, leading to foreign key violations if we try to delete a user. The only solutions are to skip the `inverse="true"` attribute or to manually delete the object from all collections on the non-inverse end. The first alternative is the most efficient since it only requires a single SQL statement. The second alternative must first load all associated objects and then issue a single delete statement for each association.

In the "Hibernate in action" book they have a very different design where they recommend that changes are made in both collections. We don't have to do this since we are only interested in maintaining the links, which is always done in one of the collections.

Parent-child relationships

When one or more objects are tightly linked to some other object we talk about a parent-child relationship. This kind of relationship becomes important when we are about to delete a parent object. The children cannot exist without the parent so they must also be deleted. Luckily, Hibernate can do this for us if we specify a `cascade="delete"` option for the link. This example is a one-to-many link between client and help texts.

```
// ClientData.java
private Set<HelpData> helpTexts;
/**
 * This is the inverse end.
 * @see HelpData#getClient()
 * @hibernate.set lazy="true" inverse="true" cascade="delete"
 * @hibernate.collection-key column="`client_id`"
 * @hibernate.collection-one-to-many class="net.sf.basedb.core.data.HelpData"
 */
Set<HelpData> getHelpTexts()
{
    return helpTexts;
}

void setHelpTexts(Set<HelpData> helpTexts)
{
    this.helpTexts = helpTexts;
}

// HelpData.java
private ClientData client;
/**
 * Get the client for this help text.
 * @hibernate.many-to-one column="`client_id`" not-null="true"
 * update="false" outer-join="false" unique-key="uniquehelp"
 */
public ClientData getClient()
{
    return client;
}

public void setClient(ClientData client)
{
    this.client = client;
}
```

This shows both sides of the one-to-many mapping between parent and children. As you can see the `@hibernate.set` doesn't specify a table, since it is given by the `class` attribute of the `@hibernate.collection-one-to-many` tag.

In a one-to-many mapping, it is always the "one" side that handles the link so the "many" side should always be mapped with `inverse="true"`.

Maps

Another type of many-to-many mapping uses a `Map` for the collection. This kind of mapping is needed when the association between two objects needs additional data to be kept as part of the association. For example, the permission (stored as an integer value) given to users that are members of a project. Note that you should use a `Set` for mapping the inverse end.

```
// ProjectData.java
private Map<UserData, Integer> users;
/**
 * Many-to-many mapping between projects and users including permission values.
 * @hibernate.map table="`UserProjects`" lazy="true"
 * @hibernate.collection-key column="`project_id`"
 * @hibernate.index-many-to-many column="`user_id`"
 *     class="net.sf.basedb.core.data.UserData"
 * @hibernate.collection-element column="`permission`" type="int" not-null="true"
 */
public Map<UserData, Integer> getUsers()
{
    if (users == null) users = new HashMap<UserData, Integer>();
    return users;
}
void setUsers(Map<UserData, Integer> users)
{
    this.users = users;
}

// UserData.java
private Set<ProjectData> projects;
/**
 * This is the inverse end.
 * @see ProjectData#getUsers()
 * @hibernate.set table="`UserProjects`" lazy="true"
 * @hibernate.collection-key column="`user_id`"
 * @hibernate.collection-many-to-many column="`project_id`"
 *     class="net.sf.basedb.core.data.ProjectData"
 */
Set<ProjectData> getProjects()
{
    return projects;
}
void setProjects(Set<ProjectData> projects)
{
    this.projects = projects;
}
```

See also:

- "Hibernate in action", chapter 3.7 "Introducing associations", page 105-112
- "Hibernate in action", chapter 6.2 "Mapping collections of value types", page 211-220
- "Hibernate in action", chapter 6.3.2 "Many-to-many associations", page 225-233
- Hibernate reference documentation: Chapter 7. Collection Mapping¹⁴
- Hibernate reference documentation: Chapter 24. Example: Parent/Child¹⁵

One-to-one mappings

A one-to-one mapping can come in two different forms, depending on if both objects should have the same id or not. We start with the case where the objects can have different id:s and the link is done with an extra column in one of the tables. The example is from the mapping between physical bioassays and arrayslides.

```
// PhysicalBioAssayData.java
private ArraySlideData arrayslide;
/**
 * Get the array slide
 * @hibernate.many-to-one column="`arrayslide_id`" not-null="false"
 *     unique="true" outer-join="false"
 */
public ArraySlideData getArraySlide()
{
}
```

```
        return arrayslide;
    }
    public void setArraySlide(ArraySlideData arrayslide)
    {
        this.arrayslide = arrayslide;
    }

    // ArraySlideData.java
    private PhysicalBioAssayData bioassay;
    /**
     * Get the bioassay.
     * @hibernate.one-to-one property-ref="arraySlide"
     */
    public PhysicalBioAssayData getPhysicalBioAssay()
    {
        return hybridization;
    }
    public void setPhysicalBioAssay(PhysicalBioAssayData bioassay)
    {
        this.bioassay = bioassay;
    }
}
```

As you can see, we use the `@hibernate.many-to-one` mapping with `unique="true"` for the bioassay side. This will force the database to only allow the same array slide to be linked once. Also note that since, `not-null="false"`, null values are allowed and it doesn't matter which end of the relation that is inserted first into the database.

For the array slide end we use a `@hibernate.one-to-one` mapping and specify the name of the property on the other end that we are linking to. One important thing to remember is to keep both ends synchronized. This should usually be done at the core layer and not in the data layer. Doing it in the data layer may effectively disable lazy loading if the synchronization code causes proxy initialization.

The second form of a one-to-one mapping is used when both objects must have the same id (primary key). The example is from the mapping between users and passwords.

```
// UserData.java
/**
 * @hibernate.id column="`id`" generator-class="foreign"
 * @hibernate.generator-param name="property" value="password"
 */
public int getId()
{
    return super.getId();
}
private PasswordData password;
/**
 * Get the password.
 * @hibernate.one-to-one class="net.sf.basedb.core.data.PasswordData"
 * cascade="all" outer-join="false" constrained="true"
 */
public PasswordData getPassword()
{
    if (password == null)
    {
        password = new PasswordData();
        password.setUser(this);
    }
    return password;
}
void setPassword(PasswordData user)
{
    this.password = password;
}

// PasswordData.java
private UserData user;
/**
```

```
    Get the user.
    @hibernate.one-to-one class="net.sf.basedb.core.data.UserData"
*/
public UserData getUser()
{
    return user;
}
void setUser(UserData user)
{
    this.user = user;
}
```

In this case, we use the `@hibernate.one-to-one` mapping in both classes. The `constrained="true"` tag in `UserData` tells Hibernate to always insert the password first, and then the user. This makes it possible to use the (auto-generated) id for the password as the id for the user. This is controlled by the mapping for the `UserData.getId()` method, which uses the foreign id generator. This generator will look at the password property, ie. call `getPassword().getId()` to find the id for the user. Also note the initialisation code and `cascade="all"` tag in the `UserData.getPassword()` method. This is needed to avoid `NullPointerException`s and to make sure everything is created and deleted properly.

See also:

- "Hibernate in action", chapter 6.3.1 "One-to-one association", page 220-225
- Hibernate reference documentation: 5.1.7. Mapping one to one and many to one associations¹⁶

Class documentation

The documentation for the class doesn't have to be very lengthy. A single sentence is usually enough. Provide tags for the author, version, last modification date and a reference to the corresponding class in the `net.sf.basedb.core` package.

```
/**
    This class holds information about any items.

    @author Your name
    @since 2.0
    @see net.sf.basedb.core.AnyItem
    @base.modified $Date: 2007-08-17 09:18:29 +0200 (Fri, 17 Aug 2007) $
    @hibernate.class table="`Anys`" lazy="false"
*/
public class AnyData
    extends CommonData
{
    ...
}
```

Method documentation

Write a short one-sentence description for all public getter methods. You do not have to document the parameters or the setter methods, since it would just be a repetition. Methods defined by interfaces are documented in the interface class. You should not have to write any documentation for those methods.

For the inverse end of an association, which has only package private methods, write a notice about this and provide a link to the non-inverse end.

```
// UserData.java
private String address;
/**
    Get the address for the user.
```

```
@hibernate.property column="`address`" type="string" length="255"
*/
public String getAddress()
{
    return address;
}
public void setAddress(String address)
{
    this.address = address;
}

private Set<GroupData> groups;
/**
    This is the inverse end.
    @see GroupData#getUsers()
    @hibernate.set table="`UserGroups`" lazy="true" inverse="true"
    @hibernate.collection-key column="`user_id`"
    @hibernate.collection-many-to-many column="`group_id`"
    class="net.sf.basedb.core.data.GroupData"
*/
Set<GroupData> getGroups()
{
    return groups;
}
void setGroups(Set<GroupData> groups)
{
    this.groups = groups;
}
```

Field documentation

Write a short one-sentence description for public static final fields. Private fields does not have to be documented.

```
/**
    The maximum length of the name of an item that can be
    stored in the database.
    @see #setName(String)
*/
public static final int MAX_NAME_LENGTH = 255;
```

UML diagram

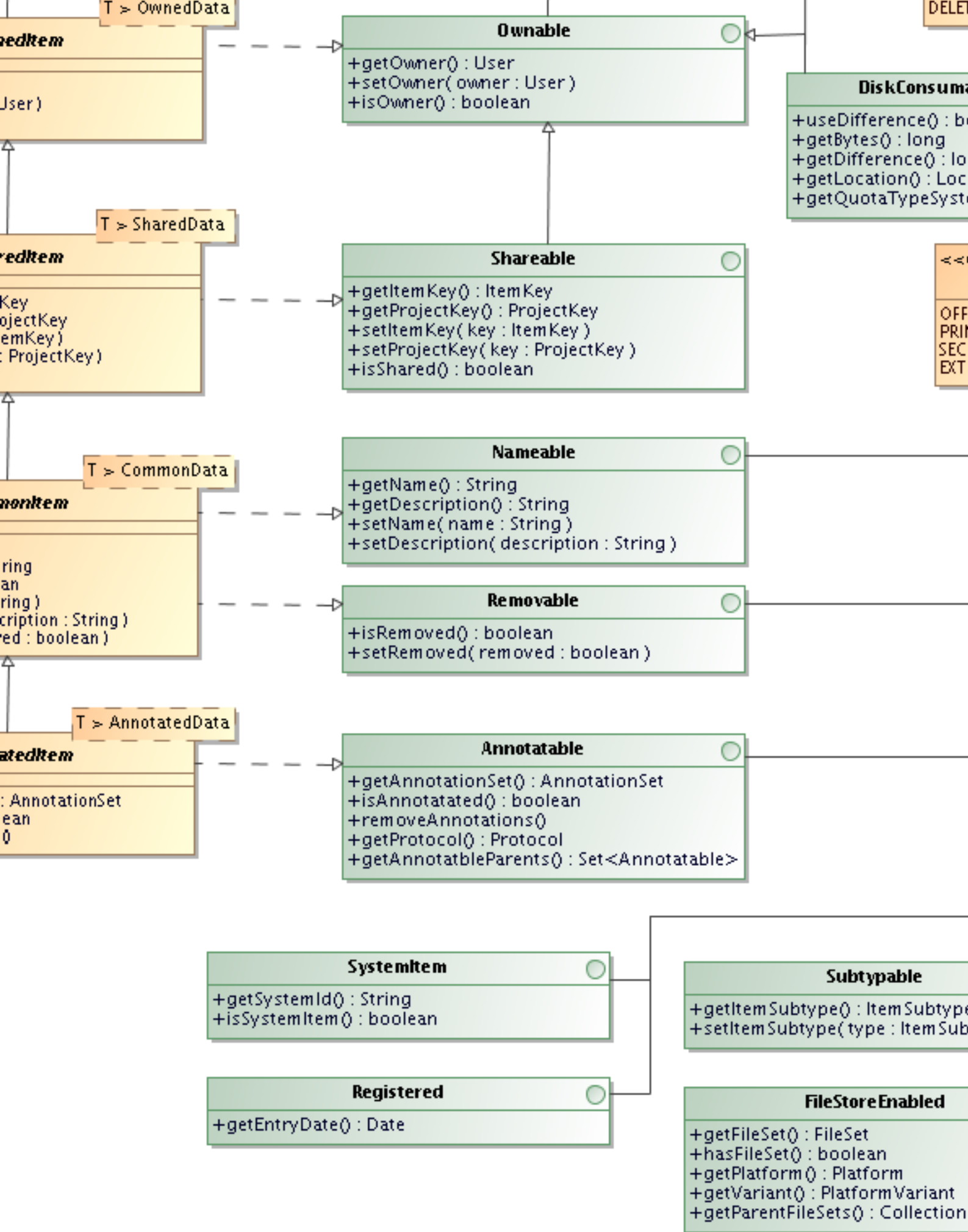
Groups of related classes should be included in an UML-like diagram to show how they are connected and work together. For example we group together users, groups, roles, etc. into an authentication UML diagram. It is also possible that a single class may appear in more than one diagram. For more information about how to create UML diagrams see Section 29.2, “Create UML diagrams with MagicDraw” (page 358).

30.3.5. Item-class rules

This document contains important information about item classes for the BASE developer. Item classes are classes that handles the business logic for the data classes in the `net.sf.basedb.core.data` package. In general there is one item class for each data class. When extending the database and creating new classes it is important that it follows the design of the already existing code.

Basic class and interface hierarchy

To simplify the development of items, we have created a set of abstract classes and interfaces. A real class for an item must inherit from one of those classes and may implement any of the interfaces if needed. The structure is similar to the structure found in the `net.sf.basedb.core.data` package (See Section 28.2, “The Data Layer API” (page 288)).



Access permissions

Each item class must be prepared to handle the access permissions for the logged in user. The base classes will do most of the required work, but not everything. There are four cases which the item class must be aware of:

- Initialise permissions in the `initPermissions()` method.
- Check for *write* permission in setter methods.
- Check for *use* permission when creating associations to other items.
- Make sure the `getQuery()` method returns only items with at least read permission.

Initialise permissions

The permissions for an item are initialised by a call to the `initPermissions()` method. This method is called as soon as the item becomes attached to a `DbControl` object, which is responsible for managing items in the database. The `initPermissions()` method should be overridden by subclasses that needs to grant or deny permissions that is not granted or denied by default. When overriding the `initPermissions()` method it is important to:

- Combine the additional permissions with those that was passed as parameters. Use the binary OR operator (`|`) with the result from the `Permission.grant()` and `Permission.deny()` methods to do this.
- Call `super.initPermissions()`. Otherwise, no permissions will be set all, resulting in an `PermissionDeniedException` almost immediately.

Here is an example from the `OwnedItem` class. If the currently logged in user is the same as the owner of the item, `DELETE`, `SET_OWNER` and `SET_PERMISSION` permissions are granted. Remember that delete permission also implies `READ`, `USE` and `WRITE` permissions.

```
// OwnedItem.java
void initPermissions(int granted, int denied)
{
    UserData owner = getData().getOwner();
    // owner may be null for new items
    if (owner != null && owner.getId() == getSessionControl().getLoggedInUserId())
    {
        granted |= Permission.grant(Permission.DELETE, Permission.SET_OWNER,
            Permission.SET_PERMISSION);
    }
    super.initPermissions(granted, denied);
}
```

Here is another example for `News` items, which grants read permission to anyone (even if not logged in) if today is between the start and end date of the news entry:

```
// News.java
void initPermissions(int granted, int denied)
    throws BaseException
{
    long today = new Date().getTime();
    long startDate = getData().getStartDate().getTime();
    long endDate = getData().getEndDate() == null ? 0 :
        getData().getEndDate().getTime()+24*3600*1000;
    if (startDate <= today && (endDate == 0 || today <= endDate))
```

```
{
    granted |= Permission.grant(Permission.READ);
}
super.initPermissions(granted, denied);
}
```

A third example from the `Help` class which is a child item to `Client`. Normally you will get `READ` permission on all child items if you have `READ` permission on the parent item, and `CREATE`, `WRITE` and `DELETE` permissions if you have `WRITE` permission on the parent item. In this case you don't have to override the `initPermissions()` method if the child class extends the `ChildItem` class. Instead, it should implement the `getSharedParent()` method. The `ChildItem.initPermissions()` will take care of checking the permissions on the parent instead of on the child. Note that this only works if the parent itself hasn't overridden the `initPermissions()` method, since that method is never called in this case.

```
// Help.java
public class Help
    extends ChildItem
    implements Nameable

...

SharedData getSharedParent()
{
    return getData().getClient();
}
```

Permissions granted by the base classes

BasicItem

This class will grant or deny permissions as they are defined by the roles the logged in user is a member of. If a subclass extends directly from this class, it is common that the `initPermissions()` method needs to be overridden.

ChildItem

This class grants `READ` permission if the logged in user has `READ` permission on the parent item, and `CREATE`, `WRITE` and `DELETE` permission if the logged in user has `WRITE` (configurable) permission on the parent item.

OwnedItem

The owner of an item gets `DELETE`, `SET_OWNER` and `SET_PERMISSION` permissions. Delete permission also implies read, use and write permissions. Subclasses to this class usually don't have to override the `initPermissions()` method.

SharedItem

The logged in user gets permissions as specified in the associated `ItemKey` and/or `ProjectKey`. Subclasses to this class usually don't have to override the `initPermissions()` method.

Checking for write permission in setter methods

An item class is required to check for `WRITE` permission in each method that modifies the state from a public method. Example:

```
public void setName(String name)
    throws PermissionDeniedException
{
    checkPermission(Permission.WRITE);
    // ... rest of code
}
```

Warning

If you forget this, an unauthorised user may be able to change the properties of an item. `WRITE` permissions are not checked in any other central place in the core code. Place the permission check on the first line in the method, before any data validation. This will make it easier to spot places where the permission check is forgotten.

Checking for use permission when creating associations

An item class is required to check for `USE` permission on associated objects in each method that modifies the association from a public method. Example from the `Protocol` class:

```
public void setFile(File file)
    throws PermissionDeniedException
{
    checkPermission(Permission.WRITE);
    if (file != null) file.checkPermission(Permission.USE);
    getData().setFile(file == null ? null : file.getData());
}
```

Warning

If you forget this, an unauthorised user may be able to change the association of an item. `USE` permissions are not checked in any other central place in the core code. Place the permission check as early in the method as possible after it has been validated that the value isn't null.

Making sure the `getQuery()` method only returns items with read permission

This method can be one of the most complex ones of the entire class. The query it generates must always be compatible with the `initPermissions()` method. I.e. it must not return any items for which the `initPermissions()` method doesn't grant `READ` permission. And the other way around, if the `initPermissions()` method grants `READ` permission to an item, the query should be able to return it. The simplest case is if you don't override the `initPermissions()` method in such a way that it affects `READ` permissions. In this case you can just create a query and return it as it is. The query implementation will take care of the rest.

```
// Client.java
public static ItemQuery<Client> getQuery()
{
    return new ItemQuery<Client>(Client.class);
}
```

A common case is when an item is the child of another item. Usually the parent is a `Shareable` item which means that we optimally should check the item and project keys on the parent when returning the children. But, this is a rather complex operation, so in this case we have chosen a different approach. The `getQuery()` method of child items must take a parameter of the parent type. The query can then safely return all children of that parent, since having a reference to the parent item, means that `READ` permission is granted. A null value for the parent is allowed, but then we fall back to check for role permissions only (with the help of a `ChildFilter` object).

```
// Help.java
private static final QueryRuntimeFilter RUNTIME_FILTER =
    new QueryRuntimeFilterFactory.ChildFilter(Item.HELP, Item.CLIENT);

public static ItemQuery<Help> getQuery(Client client)
{
    ItemQuery<Help> query = null;
    if (client != null)
    {
```

```
        query = new ItemQuery<Help>(Help.class, null);
        query.restrictPermanent(
            Restrictions.eq(
                Hql.property("client"),
                Hql.entity(client)
            )
        );
    }
    else
    {
        query = new ItemQuery<Help>(Help.class, RUNTIME_FILTER);
    }
    return query;
}
```

There are many other variants of the `getQuery()` method, for example all items having to with the authentication, User, Group, Role, etc. must check the logged in user's membership. We don't show any more examples here. Take a look in the source code if you want more information. You can also read Section 28.4, “The Query API” (page 338) for more examples.

Data validation

An item class must validate all data that is passed to it as parameters. There are three types of validation:

1. Validation of properties that are independent of other properties. For example, the length of a string or the value of number.
2. Validation of properties that depends on other properties on the same object. For example, we have properties for the row and column counts, and then an array of linked objects for each position.
3. Validation of properties that depends on the values of other objects. For example, the login of a user must be unique among all users.

For each of these types of validation we have choosen a strategy that is as simple as possible and doesn't force us to complex requirements on the code for objects. First, we may note that case 1 is very common, case 2 is very uncommon, and case 3 is just a bit more common than case 2.

Case 1 validation

For case 1 we choose to make the validation in the set method for each property. Example:

```
public void setName(String name)
    throws InvalidDataException
{
    checkPermission(Permission.WRITE);
    // Null is not allowed
    if (name == null) throw new InvalidUseOfNullException("name");
    // The name must not be too long
    if (name.length > MAX_NAME_LENGTH)
    {
        throw new StringTooLongException("name", name, MAX_NAME_LENGTH);
    }
    getData().setName(name);
}
// Note! In this case we should actually use NameableUtil instead
```

This will take care of all case 1 validation except that we cannot check properties that doesn't allow null values if the method never is called. To solve this problem we have two strategies:

- Provide a default value that is set in the constructor. For example the name of a new user can be initilised to "New user".

- Use constructor methods with parameters for required objects.

Which strategy to use is decided from case to case. Failure to validate a property will usually result in a database exception, so no real harm is done, except that we don't want to show the ugly error messages to our users. The `News` class uses a mix of the two strategies:

```
// News.java
public static News getNew(DbControl dc, Date startDate, Date newsDate)
{
    News n = dc newItem(News.class);
    n.setName("New news");
    n.setStartDate(startDate);
    n.setNewsDate(newsDate);
    n.getData().setEntryDate(new Date());
    return n;
}
...
public void setStartDate(Date startDate)
    throws PermissionDeniedException, InvalidDataException
{
    checkPermission(Permission.WRITE);
    getData().setStartDate(DateUtil.setNotNullDate(startDate, "startDate"));
}
...
```

Case 2 validation

This case requires interception of saves and updates and a call to the `validate()` method on the item. This is automatically done on items which implements the `Validatable` interface. Internally this functionality is implemented by the `DbControl` class, which keeps a "commit queue" that holds all loaded items that implements the `Validatable` interface. When `DbControl.commit()` is called, the queue is iterated and the `validate()` method is called for each item. Here is another example from the `News` class which must validate that the three dates (`startDate`, `newsDate` and `endDate`) are in proper order:

```
// News.java
void validate()
    throws InvalidDataException, BaseException
{
    super.validate();
    Date startDate = getData().getStartDate();
    Date newsDate = getData().getNewsDate();
    Date endDate = getData().getEndDate();
    if (startDate.after(newsDate))
    {
        throw new InvalidDataException("Invalid date. startDate is after newsDate.");
    }
    if (endDate != null && newsDate.after(endDate))
    {
        throw new InvalidDataException("Invalid date. newsDate is after endDate.");
    }
}
```

Case 3 validation

Usually, we do not bother with checking for this case, but delegates to the database to do the check. The reason that we do not bother to check for this case is that we can't be sure to succeed even if we first check the database. It is possible that during the time between our check and the actual insert or update, another transaction has already inserted another object into the database that violates the check. This is not perfect and the error messages are a bit ugly, but under the circumstances it is the best we can do.

Participating in transactions

Sometimes it is necessary for an item to intercept certain events. For example, the `File` object needs to know if a transaction has been completed or rolled back so it can clean up temporary files that have been used. We have created the `Transactional` interface, which is a tagging interface that tells the core to call certain methods on the item at certain events. The interface doesn't contain any methods, the item class needs to override methods from the `BasicItem` class. The following events/methods have been defined:

Note

The methods are always called for new items and items that are about to be deleted. It is only necessary for an item to implement the `Transactional` interface if it needs to act on `UPDATE` events.

onBeforeCommit(Action)

This method is called before a commit is issued to Hibernate. It should be used by an item when it needs to update dependent objects before anything is written to the database. Note that nothing has been sent to the database yet and new items have not got an id when this method is called. If you override this method you must call `super.onBeforeCommit()` to allow the superclass to do whatever it needs to do. Here is an example from the `OwnedItem` class which sets the owner to the currently logged in user, if no owner has been explicitly specified:

```
void onBeforeCommit(Transactional.Action action)
    throws NotLoggedInException, BaseException
{
    super.onBeforeCommit(action);
    if (action == Transactional.Action.CREATE && getData().getOwner() == null)
    {
        org.hibernate.Session session = getDbControl().getHibernateSession();
        int loggedInUserId = getSessionControl().getLoggedInUserId();
        UserData owner =
            HibernateUtil.loadData(session, UserData.class, loggedInUserId);
        if (owner == null) throw new NotLoggedInException();
        getData().setOwner(owner);
    }
}
```

setProjectDefaults(Project)

This method is called before inserting new items into the database to allow items to propagate default values from the active project. The method is only called when a project is active. Sub-classes should always call `super.setProjectDefaults()` and should only set default values that hasn't been explicitly set by client code (including `setFoo(null)` calls).

Note

With few exceptions a project can only hold `ItemSubtype` items as default values. This means that the item that is going to use the default value should implement the `Subtypable` interface and list the other related item types in the `@SubtypableRelatedItems` annotation.

```
// DerivedBioAssay.java
@Override
@SubtypableRelatedItems({Item.PHYSICALBIOASSAY, Item.DERIVEDBIOASSAYSET, Item.SOFTWARE,
    Item.HARDWARE, Item.PROTOCOL})
public ItemSubtype getItemSubtype()
{
    return getDbControl().getItem(ItemSubtype.class, getData().getItemSubtype());
}

/**
    Set protocol, hardware and software from project default settings.
```

```
*/
@Override
void setProjectDefaults(Project activeProject)
    throws BaseException
{
    super.setProjectDefaults(activeProject);
    if (!hasPermission(Permission.WRITE)) return;

    DbControl dc = getDbControl();
    if (!protocolHasBeenSet)
    {
        ProtocolData protocol =
            (ProtocolData)activeProject.findDefaultRelatedData(dc, this, Item.PROTOCOL, false);
        if (protocol != null)
        {
            getData().setProtocol(protocol);
            protocolHasBeenSet = true;
        }
    }
    if (!hardwareHasBeenSet)
    {
        HardwareData hardware =
            (HardwareData)activeProject.findDefaultRelatedData(dc, this, Item.HARDWARE, false);
        if (hardware != null)
        {
            getData().setHardware(hardware);
            hardwareHasBeenSet = true;
        }
    }
    if (!softwareHasBeenSet)
    {
        SoftwareData software =
            (SoftwareData)activeProject.findDefaultRelatedData(dc, this, Item.SOFTWARE, false);
        if (software != null)
        {
            getData().setSoftware(software);
            softwareHasBeenSet = true;
        }
    }
}
```

onAfterInsert()

This method is called on all items directly after Hibernate has inserted it into the database. This method can be used in place of the `onBeforeCommit()` in case the id is needed.

onAfterCommit(Action)

This method is called after a successful commit has been issued to Hibernate. It should be used by an item which needs to do additional processing. For example the `File` object may need to cleanup temporary files. This method should not use the database and it must not fail, since it is impossible to rollback anything that has already been committed to the database. If the method fails, it should log an exception with the `Application.log()` method.

onRollback(Action)

This method is called after an unsuccessful commit has been issued to Hibernate. The same rules as for the `onAfterCommit()` method applies to this method.

Internally this functionality is implemented by the `DbControl` class, which keeps a "commit queue" that holds all new objects, all objects that are about to be deleted and all objects that implements the `Transactional` interface. When `DbControl.commit()` is called, the queue is iterated and `onBeforeCommit()` is called for each item, and then either `onAfterCommit()` or `onRollback()`. The `Action` parameter is of an enumeration type which can have three different values:

- **CREATE:** This is a new item which is saved to the database for the first time.
- **UPDATE:** This is an existing item, which has been modified.
- **DELETE:** This is an existing item, which is now being deleted from the database

Template code for item classes

The `AnyItem.java`¹⁷ and `AChildItem.java`¹⁸ files contains two complete item classes with lots of template methods. Please copy and paste as much as you want from these, but do not forget to change the specific details.

Class declaration

An item class should extend one of the four classes: `BasicItem`, `OwnedItem`, `SharedItem` and `CommonItem`. Which one depends on what combination of interfaces are needed for that item. The most common situation is probably to extend the `CommonItem` class. Do not forget to include the GNU licence and copyright statement. Also note that the corresponding data layer class is specified as a generics parameter of the superclass.

```
/*
 $Id $

Copyright (C) 2011 Your name

This file is part of BASE - BioArray Software Environment.
Available at http://base.thep.lu.se/

BASE is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 3
of the License, or (at your option) any later version.

BASE is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with BASE. If not, see <http://www.gnu.org/licenses/>.
*/
package net.sf.basedb.core;
import net.sf.basedb.core.data.AnyData;
/**
 This class is used to represent an AnyItem in BASE.

@author Your name
@since 3.0
@see AnyData
@base.modified $Date$
*/
public class AnyItem
  extends CommonItem<AnyData>
{
    ...
}
```

Static methods and fields

`getNew(DbControl)`

This method is used to create a new item. The new item must be created by calling the `DbControl.newItem()`.

```
/**
 Create a new <code>AnyItem</code> item.
```

¹⁷ `../examples/AnyItem.java.txt`

¹⁸ `../examples/AChildItem.java.txt`


```
@param dc The <code>DbControl</code> which will be used for
permission checking and database access
@return The new <code>AnyItem</code> item
@throws BaseException If there is an error
*/
public static AnyItem getNew(DbControl dc)
throws BaseException
{
    AnyItem a = dc.newItem(AnyItem.class);
    a.setName("New any item");
    return a;
}
```

The method must initialise all not-null properties to a sensible default values or it may take values as parameters:

```
// User.java
public static User getNew(DbControl dc, String login, String password)
throws BaseException
{
    User u = dc.newItem(User.class);
    u.setName("New user");
    u.setLogin(login);
    u.setPassword(password);
    int defaultQuotaId = SystemItems.getId(Quota.DEFAULT);
    org.hibernate.Session session = dc.getHibernateSession();
    QuotaData defaultQuota =
        HibernateUtil.loadData(session, QuotaData.class, defaultQuotaId);
    u.getData().setQuota(defaultQuota);
    return u;
}
```

When the default value is an association to another item, use the data object (`QuotaData`) not the item object (`Quota`) to create the association. The reason for this is that the logged in user may not have read permission to the default object. Ie. The logged in user may have permission to create users, but not permission to read quota.

`getById(DbControl, int)`

This method is used to load an item from the database when the id of that item is known. Use the `DbControl.loadItem()` method to load the item. If the item is not found an `ItemNotFoundException` should be thrown.

```
/**
Get an <code>AnyItem</code> item when you know the id.

@param dc The <code>DbControl</code> which will be used for
permission checking and database access.
@param id The id of the item to load
@return The <code>AnyItem</code> item
@throws ItemNotFoundException If an item with the specified
id is not found
@throws PermissionDeniedException If the logged in user doesn't
have read permission to the item
@throws BaseException If there is another error
*/
public static AnyItem getById(DbControl dc, int id)
throws ItemNotFoundException, PermissionDeniedException, BaseException
{
    AnyItem a = dc.loadItem(AnyItem.class, id);
    if (a == null) throw new ItemNotFoundException("AnyItem[id="+id+"]");
    return a;
}
```

getQuery()

See Section , “Making sure the getQuery() method only returns items with read permission” (page 393).

Constructors

Each item class needs only one constructor, which takes an object of the corresponding data class as a parameter. The constructor should never be invoked directly. Use the `DbControl.newItem()` method.

```
AnyItem(AnyData anyData)
{
    super(anyData);
}
```

Core methods

isUsed()

This method is defined by the `BasicItem` class and is called whenever we need to know if there are other items referencing the current item. The main use case is to let client applications know if it is safe to delete an object or not. The default implementation checks `AnyToAny` links between items. A subclass must override this method if it can be referenced by other items. A subclass should always call `super.isUsed()` as a last check if it is not used by any other item. The method should check if it is being used (referenced by) some other item. For example, a `Tag` is used if there is an `Extract` with that tag. The simplest way to check if the item is used is to use a predefined query that counts the number of references.

```
/**
 * Check if:
 * <ul>
 * <li>Some {@link Extract}s are marked with this tag
 * </li>
 * </ul>
 */
public boolean isUsed()
    throws BaseException
{
    org.hibernate.Session session = getDbControl().getHibernateSession();
    org.hibernate.Query query = HibernateUtil.getPredefinedQuery(session,
        "GET_EXTRACTS_FOR_TAG", "count(*)");
    /**
     * SELECT {1}
     * FROM ExtractData ext
     * WHERE ext.tag = :tag
     */
    query.setEntity("tag", this.getData());
    boolean used = HibernateUtil.loadData(Long.class, query) > 0;
    return used || super.isUsed();
}
```

Sometimes it may be harder to decide what counts as using an item or not. Some examples:

- An event for a sample does not count as using the sample, since they have a parent-child relationship. I.e. deleting the sample will also delete all events associated with it. On the other hand, the protocol registered for the event counts as using the protocol, because deleting the protocol should not delete all events.
- As a general rule, if one item is used by a second item, then the second item cannot be used by the first. It could lead to situations where it would be impossible to delete either one of them.

getUsingItems()

Find all items that are referencing this one. This method is related to the `isUsed()` method and is defined in the `BasicItem` class. The default implementation load all items linked via an `Any-ToAny` link that has the `usingTo` flag set to true. A subclass must override this method if it can be referenced to be used by other items. A subclass should always call `super.getUsingItems()` first and then add extra items to the `Set` returned by that call. For example, a `Tag` should load all `Extract:s` with that tag.

```
/**
 * Get all:
 * <ul>
 * <li>{@link Extract}s marked with this tag
 * </li>
 * </ul>
 */
@Override
public Set<ItemProxy> getUsingItems()
{
    Set<ItemProxy> using = super.getUsingItems();
    org.hibernate.Session session = getDbControl().getHibernateSession();

    // Extracts
    org.hibernate.Query query = HibernateUtil.getPredefinedQuery(session,
        "GET_EXTRACTS_FOR_TAG", "ext.id");
    /**
     * SELECT {1}
     * FROM ExtractData ext
     * WHERE ext.tag = :tag
     */
    query.setEntity("tag", this.getData());
    addUsingItems(using, Item.EXTRACT, query);
    return using;
}
```

initPermissions(int, int)

See Section , “Initialise permissions” (page 391).

validate()

See the section called “Data validation” (page 394).

onBeforeCommit(Action), setProjectDefaults(Project), onAfterInsert(), onAfterCommit(Action) , onRollback(Action)

See the section called “Participating in transactions” (page 396).

Getter and setter methods

The `get` methods for basic property types are usually very simple. All that is needed is to return the value. Be aware of date values though, they are mutable and must be copied.

```
/**
 * Get the value of the string property.
 */
public String getStringProperty()
{
    return getData().getStringProperty();
}

/**
 * Get the value of the int property.
 */
public int getIntProperty()
{
    return getData().getIntProperty();
}
```

```
}

/**
 * Get the value of the boolean property.
 */
public boolean isBooleanProperty()
{
    return getData().isBooleanProperty();
}

/**
 * Get the value of the date property.
 * @return A date object or null if unknown
 */
public Date getDateProperty()
{
    return DateUtil.copy(getData().getDateProperty());
}
```

The set methods must always check for WRITE permission and validate the parameters. There are plenty of utility methods to help with this.

```
/**
 * The maximum length of the string property. Check the length
 * against this value before calling {@link #setStringProperty(String)}
 * to avoid exceptions.
 */
public static final int MAX_STRINGPROPERTY_LENGTH =
    AnyData.MAX_STRINGPROPERTY_LENGTH;

/**
 * Set the value of the string property. Null values are not
 * allowed and the length must be shorter than
 * {@link #MAX_STRINGPROPERTY_LENGTH}.
 * @param value The new value
 * @throws PermissionDeniedException If the logged in user
 *     doesn't have write permission
 * @throws InvalidDataException If the value is null or too long
 */
public void setStringProperty(String value)
    throws PermissionDeniedException, InvalidDataException
{
    checkPermission(Permission.WRITE);
    getData().setStringProperty(
        StringUtil.setNotNullString(value, "stringProperty", MAX_STRINGPROPERTY_LENGTH)
    );
}

/**
 * Set the value of the int property. The value mustn't be less than
 * zero.
 * @param value The new value
 * @throws PermissionDeniedException If the logged in user
 *     doesn't have write permission
 * @throws InvalidDataException If the value is less than zero
 */
public void setIntProperty(int value)
    throws PermissionDeniedException, InvalidDataException
{
    checkPermission(Permission.WRITE);
    getData().setIntProperty(
        IntegerUtil.checkMin(value, "intProperty", 0)
    );
}

/**
 * Set the value of the boolean property.
 * @param value The new value
 */
```

```
    @throws PermissionDeniedException If the logged in user
        doesn't have write permission
*/
public void setBooleanProperty(boolean value)
    throws PermissionDeniedException
{
    checkPermission(Permission.WRITE);
    getData().setBooleanProperty(value);
}

/**
    Set the value of the date property. Null values are allowed.
    @param value The new value
    @throws PermissionDeniedException If the logged in user
        doesn't have write permission
*/
public void setDateProperty(Date value)
    throws PermissionDeniedException
{
    checkPermission(Permission.WRITE);
    getData().setDateProperty(DateUtil.setNullableDate(value, "dateProperty"));
}
```

Many-to-one associations

Many-to-one associations require slightly more work. First of all, the item must be connected to a `DbControl` since it is used to load the information from the database and create the new item object. Secondly, we must make sure to check for use permission on the referenced object in the setter method.

```
/**
    Get the associated other item.
    @return The OtherItem item
    @throws PermissionDeniedException If the logged in user
        doesn't have read permission
    @throws BaseException If there is another error
*/
public OtherItem getOtherItem()
    throws PermissionDeniedException, BaseException
{
    return getDbControl().getItem(OtherItem.class, getData().getOtherItem());
}

/**
    Set the associated item. Null is not allowed.
    @param other The other item
    @throws PermissionDeniedException If the logged in user
        doesn't have write permission
    @throws InvalidDataException If the other item is null
    @throws BaseException If there is another error
*/
public void setOtherItem(OtherItem other)
    throws PermissionDeniedException, InvalidDataException, BaseException
{
    checkPermission(Permission.WRITE);
    if (otherItem == null) throw new InvalidUseOfNullException("otherItem");
    getData().setOtherItem(otherItem.getData());
}
```

One-to-many and many-to-many associations

If the association is a one-to-many or many-to-many it becomes a little more complicated again. There are many types of such associations and how they are handled usually depends on if they are sets, maps, lists or any other type of collections. In all cases we need methods for adding and removing items, and a method that returns a `Query` that can list all associated items. The first

example if for parent/child relationship, which is a one-to-many association where the children are mapped as a set.

```
/**
 * Create a child item for this any item.
 * @return The new AChildItem object
 * @throws PermissionDeniedException If the logged in user doesn't have
 *         write permission
 * @throws BaseException If there is another error
 */
public AChildItem newChildItem()
    throws PermissionDeniedException, BaseException
{
    checkPermission(Permission.WRITE);
    return AChildItem.getNew(getDbControl(), this);
}

/**
 * Get a query that will return all child items for this any item.
 * @return A {@link Query} object
 */
public ItemQuery<AChildItem> getChildItems()
{
    return AChildItem.getQuery(this);
}
```

The second example is for the many-to-many associations between users and roles, which is also mapped as a set.

```
// Role.java
/**
 * Add a user to this role.
 * @param user The user to add
 * @throws PermissionDeniedException If the logged in user doesn't
 *         have write permission for the role and
 *         use permission for the user
 * @throws InvalidDataException If the user is null
 */
public void addUser(User user)
    throws PermissionDeniedException, InvalidDataException
{
    checkPermission(Permission.WRITE);
    if (user == null) throw new InvalidUseOfNullException("user");
    user.checkPermission(Permission.USE);
    getData().getUsers().add(user.getData());
}

/**
 * Remove a user from this role.
 * @param user The user to remove
 * @throws PermissionDeniedException If the logged in user doesn't
 *         have write permission for the role and
 *         use permission for the user
 * @throws InvalidDataException If the user is null
 */
public void removeUser(User user)
    throws PermissionDeniedException, InvalidDataException
{
    checkPermission(Permission.WRITE);
    if (user == null) throw new InvalidUseOfNullException("user");
    user.checkPermission(Permission.USE);
    getData().getUsers().remove(user.getData());
}

/**
 * Check if the given user is member of this role or not.
```

```
@param user The user to check
@return TRUE if the user is member, FALSE otherwise
*/
public boolean isMember(User user)
{
    return getData().getUsers().contains(user.getData());
}

/**
    Get a query that returns the users that
    are members of this role. This query excludes users that the logged
    in user doesn't have permission to read.
    @see User#getQuery()
*/
public ItemQuery<User> getUsers()
{
    ItemQuery<User> query = User.getQuery();
    query.joinPermanent(
        Hql.innerJoin("roles", Item.ROLE.getAlias())
    );
    query.restrictPermanent(
        Restrictions.eq(
            Hql.alias(Item.ROLE.getAlias()),
            Hql.entity(this)
        )
    );
    return query;
}

// User.java
/**
    Get a query that returns the roles where this user is a
    member. The query excludes roles that the logged in user doesn't have
    permission to read.
    @see Role#getQuery()
*/
public ItemQuery<Role> getRoles()
{
    ItemQuery<Role> query = Role.getQuery();
    query.joinPermanent(
        Hql.innerJoin("users", Item.USER.getAlias())
    );
    query.restrictPermanent(
        Restrictions.eq(
            Hql.alias(Item.USER.getAlias()),
            Hql.entity(this)
        )
    );
    return query;
}
```

Note that we have a query method in both classes, but the association can only be changed from the Role. We recommend that modifier methods are put in one of the classes only. The last example is the many-to-many relation between projects and users which is a map to the permission for the user in the project.

```
// Project.java
/**
    Grant a user permissions to this project. Use an empty set
    or null to remove the user from this project.

    @param user The user
    @param permissions The permissions to grant, or null to revoke all permissions
    @throws PermissionDeniedException If the logged in user doesn't have
        write permission for the project
    @throws InvalidDataException If the user is null
    @see Permission
*/
```

```

public void setPermissions(User user, Set<Permission> permissions)
    throws PermissionDeniedException, InvalidDataException
{
    checkPermission(Permission.WRITE);
    if (user == null) throw new InvalidUseOfNullException("user");
    if (permissions == null || permissions.isEmpty())
    {
        getData().getUsers().remove(user.getData());
    }
    else
    {
        getData().getUsers().put(user.getData(), Permission.grant(permissions));
    }
}

/**
 * Get the permissions for a user in this project.
 * @param user The user for which we want to get the permission
 * @return A set containing the granted permissions, or an
 *         empty set if no permissions have been granted
 * @throws InvalidDataException If the user is null
 * @see Permission
 */
public Set<Permission> getPermissions(User user)
    throws InvalidDataException
{
    if (user == null) throw new InvalidUseOfNullException("user");
    return Permission.fromInt(getData().getUsers().get(user.getData()));
}

/**
 * Get a query that returns the users that
 * are members of this project. This query excludes users that the logged
 * in user doesn't have permission to read.
 * @see User#getQuery()
 */
public ItemQuery<User> getUsers()
{
    ItemQuery<User> query = User.getQuery();
    query.joinPermanent(
        Hql.innerJoin("projects", Item.PROJECT.getAlias())
    );
    query.restrictPermanent(
        Restrictions.eq(
            Hql.alias(Item.PROJECT.getAlias()),
            Hql.entity(this)
        )
    );
    return query;
}

// User.java
/**
 * Get a query that returns the projects where this user is a
 * member. The query excludes projects that the logged in user doesn't have
 * permission to read. The query doesn't include projects where this user is
 * the owner.
 * @see Project#getQuery()
 */
public ItemQuery<Project> getProjects()
{
    ItemQuery<Project> query = Project.getQuery();
    query.joinPermanent(
        Hql.innerJoin("users", Item.USER.getAlias())
    );
    query.restrictPermanent(
        Restrictions.eq(
            Hql.index(Item.USER.getAlias(), null),
            Hql.entity(this)
        )
    );
    return query;
}

```



```
}
```

As you can see from these examples, the code is very different depending on the type of association. We don't give any more examples here, but if you are unsure you should look in the source code to get more inspiration.

30.3.6. Batch-class rules

TODO

30.3.7. Test-class rules

TODO

Part V. FAQ

Chapter 31. Frequently Asked Questions with answers

This chapter presents a list of solutions to common problems and tasks in BASE. The information is collected from the mailing lists, private communication, and from issues frequently encountered in BASE introduction courses. If you have BASE solutions that should be added to this chapter please contact us through the usual communication channels, see Chapter 3, *Resources* (page 9) for contact information.

31.1. Reporter related questions with answers

Q: My favourite database is not used for annotating reporters. Can I add my database to BASE and if so, how should it proceed?

A: Yes, you can add resources to annotate reporters. You will need to upgrade BASE and you may have to contact your system administrator for doing so.

In order to change, remove or add annotation fields attached to reporters, you will need modify the `extended-properties.xml` file and run a BASE update. Please refer to section Section 20.2, "Installation instructions" (page 167) for information about both processes. Once done with the upgrade, you'll have to define a new reporter import plug-in. Instructions can be found in Chapter 21, *Plug-ins and extensions* (page 178).

Q: I have made a mistake while loading my reporters. How can I delete them all in one go ?

A: The reporter import plug-in can be executed in *delete* mode. Run the plug-in again and select the same file you used for the import. Select the **Mode=delete** option. In the **Error handling** section select the **Reporter is used=skip** option. This will delete all reporters that was created in the previous import.

Q: I get a message "Error: Unable to import root bioassay. Item not found: Reporter[externalId=AFFX-2315060]" when I try to create a root bioassayset.

A: BASE requires all reporters (probesets in Affymetrix speak) to be stored in the database before they can be used. The reporter information is typically imported from a reporter annotation file but in some cases the reporter annotation file supplied by Affymetrix fails to describe all reporters (probesets) on a chip. BASE will refuse to store data related to such chips until the missing reporters are added to the database. Hence the rejection of the new root bioassayset.

The resolution is straightforward, simply import the probeset information from the CDF file associated with the array design. The catch is that normal BASE user credential is not enough to perform the import therefore someone with proper credential (the BASE server administrator is one of them) must perform the import. Follow the instructions at Section 9.2.1, "Import/update reporter from files" (page 66) to import reporters. Make sure to select plug-in option to ignore already existing reporters when starting the import otherwise the existing reporter annotations will be changed. The goal here is to add missing reporters to allow BASE work with your data. The CDF file does not contain any annotation information and cannot be used to annotate reporters.

31.2. Array design related questions with answers

Q: What is the best way to create an array design in BASE when starting from a GAL file?

A: This requires some work but here is the procedure to remember:

A gal file tells where **Reporters** have been spotted on an array. So a GAL file can be used to do 2 things

1. Define the features of an array design for a non-Affymetrix platform using the **Reporter Map importer plug-in**.

To do so, after having created an new array design, go to the single-item view by of the newly created array design. Click on the **Import** the button. If you do not see it, it means that you have not enough privileges (contact the administrator).

This starts the plug-in configuration wizard. Select the **auto detect** option and in the next step your GAL file.

Now, there is the risk that no file format has been defined for GAL files. This must be done by an administrator or other user with proper privileges. See Section 21.2, "Plug-in configurations" (page 186) for information about this.

Once done (and if everything went fine), you can see from the Array Design list view that the **Has features** entry has been modified and is set to 'Yes (n)' where n indicates the number of spots (features) for this array.

Note

Features can also be loaded from a Genepix GPR file with the same procedure.

2. Define the **Reporters** present on the array design using the **Reporter importer plug-in**.

To do so, Go to View Reporters and click on **Import**. This starts a **Reporter Importer plug-in**

More information about importing Reporters can be found in Chapter 9, *Reporters* (page 65)

Q: I am confused. What is the difference between **Reporter map importer**, **Print map importer** and **Reporter importer**?

A: The reporter map and print map importer are used to import features to an array design. The latter one must be used when your array design is connected to PCR plates and supports two file formats: Biorobotics TAM and Molecularware MWBR. See http://www.flychip.org.uk/protocols/robotic_spotting/fileformats.php for more information about those file formats. If you are only using commercial platforms or if you do not use plates in the array LIMS, you have no need for the print map importer and should use the reporter map importer instead.

The reporter importer is used to load reporter annotations into BASE.

31.3. Biomaterial, Protocol, Hardware, Software related questions with answers

Q: I have just created a new item but I can not see it. Am I doing something wrong?

A: Try clearing the filter. To do so: use the **view / presets** dropdown and select the **clear filter** entry. This will remove all characters in the search boxes and all preselection of item in the drop down lists. If this does not solve your problem, then check if the **view / presets** has the **owned by me** entry selected.

Q: I can only see XX columns in the list view but I know I have a lot more information. Is there a way I can customise the column display?

- A: Yes, you can display many more columns. See Section 5.4.3, “Configuring which columns to show” (page 30).
- Q: Is it possible to sort the values in a column in the list view?
- A: Of course it is. See Section 5.4.1, “Ordering the list” (page 28).
- Q: Is it possible to sort the annotation types from **Annotation & parameters** tab in the single-item view?
- A: No. This is not possible at the moment. The annotations are always sorted by the name of the annotation type.
- Q: I have to create pools of samples in my experiment due to scarcity of the biological material. Can I represent those pooled samples in BASE?
- A: Yes, you can. From the sample list select a number of samples by marking their checkboxes. Then click on the **Pool** button. For more information see Section 16.2.1, “Create sample” (page 111). Pooling can also be applied to extracts.
- Q: I need to create a new item subtype but the **New...** button is grey and does not work. Why?
- A: Your privileges are not high enough and you have not been granted permission to create subtypes. Contact your BASE administrator. For more information about permissions, please refer to Chapter 22, *Account administration* (page 194).
- Q: I have created an Annotation Type **Temperature** and shared it to everyone but when I want to use it for annotating a sample, I can not find it! How is that?
- A: The most likely explanation is that this particular annotation type has been declared as a protocol parameter. This means that it will only be displayed in BASE if you have used a sample creation protocol which uses that parameter.
- Q: I have carried out an experiment using both Affymetrix and Agilent arrays but I can not select more than one raw data type in BASE. What should I do?
- A: In this particular case and because you are using 2 different raw data file formats, you will have to split your experiment in 2. One experiment for those samples processed using Affymetrix platform and another one using Agilent platform. You do not necessarily have to provide all information about the samples again but simply create new raw bioassay data which can be grouped in a new experiment.

31.4. Data Files and Raw Data related questions with answers

- Q: It seems that BASE does not support the data files generated by my brand new scanner. Is it possible to add it to BASE?
- A: Yes it is possible to extend BASE so that it can support your system. You will need to define a new raw data type for BASE by modifying the `raw-datatypes.xml` configuration file.
- Then, you will have to run the `updatedb.sh` to make the new raw data type available to the system. See Section 20.1, “Upgrade instructions” (page 165).
- Finally, you will have to configure a raw data import plug-in in order to be able to create raw-bioassays. See Section 21.2, “Plug-in configurations” (page 186) and Section 17.2.3, “Import raw data” (page 140) for further information.
- Q: Are Affymetrix CDT and CAB files supported by BASE?

A: There is no support for CDT or CAB. Currently only CDF and CEL files are supported by the Affymetrix plug-ins. Annotation files (.csv) are used for uploading probeset (reporter in BASE language) information. The issue of supporting CDT and CAB files is an import and a plug-in issue. There are two ways to solve this:

1. Write code that treats the files in a proper way and submit the solution to the developer team (preferred route).
2. Submit a ticket through <http://base.thep.lu.se> explaining what you'd like to see with respect to CDT and CAB files.

Note

To include CDT and CAB support to BASE, the file formats must be open, that is we must be able to read them without proprietary non-distributable code.

Q: Are Illumina data files supported by BASE?

A: Yes, but not by default. There is an Illumina package¹ that provides Illumina support to BASE. The package is straightforward to install, visit the package site for more information.

31.5. Analysis related questions with answers

Q: Is it possible to use the formula filter to filter for `null` values (or `non-null` values)?

A: Yes, use an expression like: `ch(0) != null`. This will match all values, except `null` values.

Q: OK, I have uploaded 40 CEL files in BASE but are there any tool to perform normalisation on Affymetrix raw data?

A: Yes, there is. BASE team has created a plug-in based on RMAExpress methods from Bolstad and Irizarry² so you can normalise Affymetrix data sets of reasonable size (not 1000 CEL files at a time though even though this might depend on your set-up...) The plug-in is not included in a standard BASE installation, but can be downloaded from the BASE plug-ins web site³.

Q: I am trying to import raw bioassays using the import button in the experiment properties view but BASE claims that *Could not find any plugins that you have permission to use*. I know there are import plug-ins available to me since I have successfully imported data before, why does the import fail?

A: All raw bioassays in the experiment are already imported. In this case the BASE server cannot detect anything to import and returns the somewhat confusing message. Simply add the non-imported raw bioassays to the experiment and try again.

Part VI. Appendix

Appendix A. Core plug-ins shipped with BASE

Here is a categorized list of all plug-ins installed with a pristine BASE installation. Some plug-ins must be configured before use. The requirements are listed below and configuration samples are given for plug-ins that supports/requires configurations. Use the right-click menu of the mouse to download these XML files for further import into BASE (see Section 21.2.2, “Importing and exporting plug-in configurations” (page 189)).

Contributed plug-ins are available at <http://baseplugins.thep.lu.se>¹. These plug-ins are either developed outside the core team or require external non-Java compilers and tools. These packages are excluded from the BASE package to make the installation process somewhat simpler.

A.1. Core analysis plug-ins

BASE 1 plug-in executor

Simulates the plug-in runner from Base 1.2. Must be configured before use. The recommended approach is to use the plug-in configuration file from the BASE 1.2 installation. See Section 21.1.3, “BASE version 1 plug-ins” (page 183) for more information.

External program executor

Export data from BASE and execute an external program for analysis. Afterwards, data can be imported back to BASE. A configuration is needed to run this plug-in. The plug-in is very flexible and can handle several export/import data formats (which can be extended by adding special export/import plug-ins). Here is a list of the built-in exporter/importer implementations.

- *BASEFile exporter for the ExternalProgramExecutor*: Exporter implementation that export data in BASE file format.
- *BASEfile importer for the ExternalProgramExecutor*: Importer implementation that can import data in BASEfile format.
- *BFS exporter for the ExternalProgramExecutor*: Exporter implementation that export data in BFS file format.
- *BFS importer for the ExternalProgramExecutor*: Importer implementation that can import data in BFS file format.
- *File-only importer for the ExternalProgramExecutor*: Importer implementation that simply upload all created files to BASE.

Note

This plug-in can in theory handle everything (and more) that the *BASE 1 plug-in executor* can, except that it doesn't do any translation of field names used in BASE 1.

Formula extra value calculator

Calculates extra values for a bioassay set using a user-defined formula. No configuration is needed.

Formula filter

Filters a bioassay set using a user-defined formula. No configuration is needed.

Formula intensity transformer

Creates a new bioassay set with transformed intensity values using a user-defined formula. No configuration is needed.

¹ <http://baseplugins.thep.lu.se>

Manual derived bioassay creator, Manual transformation

Allows a user to manually register an external analysis procedure that has happened outside of BASE and to register the parameters used and the generated output files. On plug-in create a derived bioassay set and the other a transformation/bioassay set. Both plug-ins need a configuration to register possible parameters and output files.

Normalisation: Lowess

Normalisation using LOWESS algorithm. No configuration is needed.

Normalisation: Median ratio

Normalisation based on median ratio. No configuration is needed.

A.2. Core export plug-ins

Unless otherwise noted, none of the export plug-ins need a configuration.

BASEfile exporter

Exports bioassay set data to serial or matrix BASEfile format.

BFS exporter

Exports bioassay set data to BFS format.

GAL exporter

Exports the features of an array design to a GAL file.

Help texts exporter

Exports help texts to an XML file.

Packed file exporter

Exports files and directories as an archive-file. A configuration is needed to specify compression format. Support for the following formats are included in BASE:

- *BZipped TAR archive*: Collects the selected files/directories into a TAR file and compress it with BZIP2.
- *GZipped TAR archive*: Collects the selected files/directories into a TAR file and compress it with GZIP.
- *TAR archive*: Collects the selected files/directories into an uncompressed TAR file.
- *ZIP archive*: Collects the selected files/directories into a ZIP file.

See Section 25.6.4, “File packer plug-ins” (page 258) for information about implementing support for other file formats.

Plate mapping exporter

Exports plate mappings.

Plugin configuration exporter

Exports plug-in configurations to an XML file.

Table exporter

Exports data from table listings in the web-interface to a TAB-separated text file or XML file.

A.3. Core import plug-ins

There are many import plug-ins in BASE. Their use are in most cases seamless and the user does not need to be aware of detailed plug-in usage. However, there is a set of batch import plug-ins

that are targeted for importing multiple items into BASE. These batch importers require some user knowledge for proper and efficient use of them. The batch plug-ins are listed in the Section A.3.1, “Core batch import plug-ins” (page 417) sub-section below together with pointers to further reading on how to use the plug-ins.

Affymetrix CDF probeset importer

This plug-in is used to import probesets (reporters in BASE language) from an Affymetrix CDF file. It can be used in import mode from the reporter list view and from the array design view and in verification mode from the array design view. The plug-in can only set the name and ID of the reporters, since the CDF file doesn't contains any annotation information. Probesets already in BASE will not be affected by the import. No configuration is needed.

Annotation importer

Imports annotation to any annotatable item in BASE. Configurations are supported but not required.

GTF reporter importer

Imports reporter information from GTF (Gene transfer format) files. Configurations are supported but not required. BASE has pre-installed configurations that uses the `gene_id` or `transcript_id` as reporter id.

GTF reporter map importer

Imports array design features from GTF (Gene transfer format) files. A configuration is needed. BASE has pre-installed configurations that uses the `gene_id` or `transcript_id` as reporter id.

Help texts importer

Imports help texts from an XML file into BASE. No configuration is needed.

Illumina raw data importer

This plug-in is used to import raw data from Illumina BeadStudio data files. No configuration is needed.

Plate importer

Imports plates from a simple flat file. A configuration is needed. BASE has pre-installed configurations for 96- and 384-well plates.

Plate mapping importer

Imports plate mappings exported by the *Plate mapping exporter*. No configuration is needed.

Plugin configuration importer

Imports plug-in configurations from an XML file. No configuration is needed.

Print map importer

Imports array design features from TAM or MwBR files. This plug-in require that the array design is connected with plates. No configuration is needed.

Raw data importer

Imports raw data from a text file. A configuration is needed. BASE has pre-installed configurations for GenePix and Cufflinks data files.

Reporter importer

Import reporter (probeset) information from a file. A configuration is not needed, but is recommended. BASE has pre-installed configurations for several different file types.

Since BASE 2.0, available configurations:

Reporter map importer

Imports array design features from text files. This plug-in can be used without connection to plates. A configuration is needed. BASE has pre-installed configurations for some file formats.

A.3.1. Core batch import plug-ins

The batch import plug-ins all work similarly and their usage is described in Section 18.2, “Batch import of data” (page 155). All batch importers have support for configurations but can also be used without.

Array batch importer

Imports and updates array batches in a batch.

Array design importer

Imports and updates array designs in a batch.

Array slide importer

Imports and updates array slides in a batch.

Bioplate importer

Imports and updates bioplates in a batch. Note that this import can't be used to put biomaterial on the bioplates.

Biosource importer

Imports and updates biosources in a batch.

Derived bioassay importer

Imports and updates derived bioassays in a batch.

Extract importer

Imports and updates extracts in a batch.

Physical bioassay importer

Imports and updates physical bioassays in a batch.

Raw bioassay importer

Imports and updates raw bioassays in a batch.

Sample importer

Imports and updates samples in a batch.

A.4. Core intensity plug-ins

Formula intensity calculator

Calculate intensities from raw data using a user-defined formula. No configuration is needed, but formulas must be defined using View Formulas.

A.5. Uncategorized core plug-ins

Spot images creator

Converts a full-size image into JPEG images for each spot. No configuration is needed.

TAR file unpacker

Unpacks a tar file on the BASE file system. It also supports TAR files compressed with GZIP or BZIP algorithms. This plug-in can be used to automatically unpack files during upload. No configuration is needed.

ZIP file unpacker

Unpacks ZIP and JAR files on the BASE's file system. This plug-in can be used to automatically unpack files during upload. No configuration is needed.

Appendix B. base.config reference

The `base.config` file is the main configuration file for BASE. It is located in the `<basedir>/www/WEB-INF/classes` directory. Most configuration properties have sensible defaults or are only used for advanced features. However, a few are required and may need to be specified or changed:

- `db.dialect`, `db.driver`, `db.url`, `db.username`, `db.password`: Settings for connecting to the database.
- `userfiles`: Setting that specify where uploaded files are stored on the BASE server.
- `plugins.dir`: Settings that specify where plug-ins and extensions are installed.

Database driver section

This section has parameters needed for the database connection, such as the database dialect, username and password.

`db.dialect`

The Hibernate dialect to use when generating SQL commands to the database. Use:

- `org.hibernate.dialect.MySQL5InnoDBDialect` for MySQL
 - `org.hibernate.dialect.PostgreSQLDialect` for PostgreSQL
- Other dialects may work but are not supported.

`db.driver`

The JDBC driver to use when connecting to the database. Use:

- `com.mysql.jdbc.Driver` for MySQL
 - `org.postgresql.Driver` for PostgreSQL
- Other JDBC drivers may work but are not supported.

`db.url`

The connection URL that locates the BASE database. The exact syntax of the string depends on the JDBC driver. Here are two examples which leaves all other settings to their defaults:

- `jdbc:mysql://localhost/basedb` for MySQL
- `jdbc:postgresql:basedb` for PostgreSQL

You can get more information about the parameters that are supported on the connection URL by reading the documentation from MySQL¹ and PostgreSQL².

Note

For MySQL we recommend that you set the character encoding to UTF-8 and enable the server-side cursors feature. The latter option will reduce memory usage since the JDBC driver does not have to load all data into memory. The value below should go into one line `jdbc:mysql://localhost/basedb?characterEncoding=UTF-8&useCursorFetch=true&defaultFetchSize=100&useServerPrepStmts=true`

`db.dynamic.catalog`

The name of the catalog where the dynamic database used to store analysed data is located. If not specified the same catalog as the regular database is used. The exact meaning of catalog depends on the actual database. For MySQL the catalog is the name of the database so this

¹ <http://dev.mysql.com/doc/refman/5.1/en/connector-j-reference-configuration-properties.html>

² <http://jdbc.postgresql.org/documentation/81/connect.html>

value is simply the name of the dynamic database. PostgreSQL does not support connecting to multiple databases with the same connection so this should have the same value as the database in the `db.url` setting.

db.dynamic.schema

The name of the schema where the dynamic database used to store analysed data is located. MySQL does not have schemas so this value should be left empty. PostgreSQL supports schemas and we recommend that the dynamic part is created in it's own schema to avoid mixing the dynamic tables with the regular ones.

db.username

The username to connect to the database. The user should have full permission to both the regular and the dynamic database.

db.password

The password for the user.

db.batch-size

The batch size to use when inserting/updating items with the Batch API. A higher value requires more memory, a lower value degrades performance since the number of database connections increases. The default value is 50.

db.queries

The location of an XML file which contains database-specific queries overriding those that does not work from the `/common-queries.xml` file. Use:

- `/mysql-queries.xml` for MySQL
- `/postgres-queries.xml` for PostgreSQL

See also Section H.1, “mysql-queries.xml and postgres-queries.xml” (page 442).

db.extended-properties

The location of an XML file describing the extended properties for extendable item types, ie. the reporters. The default value is `/extended-properties.xml`. See Appendix C, *extended-properties.xml reference* (page 426) for more information about extended properties.

db.raw-data-types

The location of an XML file describing all raw data types and their properties. The default value is `/raw-data-types.xml`. See Appendix D, *Platforms and raw-data-types.xml reference* (page 430) for more information about raw data types.

db.cleanup.interval

Interval in hours between database cleanups. Set this to 0 to disable (recommended for job agents). The default value is 24. The cleanup will remove entries in the database that have been orphaned due to other information that has been removed. For example, change history entries, any-to-any links and permission keys.

Authentication section

This section describes parameters that are needed if you are going to use external authentication. If you let BASE handle this you will not have to bother about these settings. See Section 25.6.1, “Authentication plug-ins” (page 253) for more information about external authentication.

auth.driver

The class name of the plug-in that acts as the authentication driver. BASE ships with a simple plug-in that checks if a user has a valid email account on a POP3 server. It is not enabled by default. The class name of that plug-in is `net.sf.basedb.core.authentication.POP3Authenticator`. If no class is specified internal authentication is used.

auth.jarpath

The path to the JAR file containing the class specified by the `auth.driver` setting. If empty, it is assumed that class is on the class-path. Eg. in the `WEB-INF/lib` directory.

auth.init

Initialisation parameters sent to the plug-in when calling the `init()` method. The syntax and meaning of this string depends on the plug-in. For the `POP3Authenticator` this is simply the name or IP-address of the mail server.

auth.synchronize

If this setting is 1 or **TRUE**, BASE will synchronize the extra information (name, address, email, etc.) sent by the authentication driver when a user logs in to BASE. This setting is ignored if the driver does not support extra information.

auth.cachepasswords

If passwords should be cached by BASE or not. If the passwords are cached a user may login to BASE even if the external authentication server is down. The cached passwords are only used if the external authentication does not answer properly.

auth.daystocache

How many days a cached password is valid if caching is enabled. A value of 0 caches the passwords forever.

Internal job queue section

This section contains setting that control the internal job queue. The internal job queue is a simple queue that executes jobs more or less in the order they were added to the queue. To make sure long-running jobs do not block the queue, there are four slots that uses the expected execution time to decide if a job should be allowed to execute or not.

jobqueue.internal.enabled

If 0 or **FALSE** the internal job queue will be disabled.

jobqueue.internal.runallplugins

If 1 or **TRUE** the internal job queue will ignore the `useInternalJobQueue` flag specified on plug-ins. If 0 or **FALSE** the internal job queue will only execute plug-ins which has `useInternalJobQueue=true`

jobqueue.internal.signalreceiver.class

A class implementing the `SignalReceiver` interface. The class must have a public no-argument constructor. If no value is specified the default setting is: `net.sf.basedb.core.signal.LocalSignalReceiver`.

Change to `net.sf.basedb.core.signal.SocketSignalReceiver` if the internal job queue must be able to receive signals from outside this JVM.

jobqueue.internal.signalreceiver.init

Initialisation string sent to `SignalReceiver.init()`. The syntax and meaning of the string depends on the actual implementation that is used. Please see the Javadoc for more information.

jobqueue.internal.checkinterval

The number of seconds between checks to the database for jobs that are waiting for execution.

jobqueue.internal.shortest.threads,**jobqueue.internal.short.threads,****jobqueue.internal.medium.threads, jobqueue.internal.long.threads**

Maximum number of threads to reserve for jobs with a given expected execution time. A job with a short execution time may use a thread from one of the slots with longer execution time. When all threads are in use, new jobs will have to wait until an executing job has finished.

`jobqueue.internal.shortest.threadpriority,` `jobqueue.internal.short.threadpriority,`
`jobqueue.internal.medium.threadpriority, jobqueue.internal.long.threadpriority`

The priority to give to jobs. The priority is a value between 1 and 10. See <http://download.oracle.com/javase/6/docs/api/java/lang/Thread.html> for more information about thread priorities.

Job agent section

This section contains settings that BASE uses when communicating with external job agents. See Section 20.3, “Installing job agents” (page 171) for more information about job agents.

`agent.maxage`

Number of seconds to keep job agent information in the internal cache. The information includes, CPU and memory usage and the status of executing jobs. This setting controls how long the information is kept in the cache before a new request is made to the job agent. The default value is 60 seconds.

`agent.connection.timeout`

The timeout in milliseconds to wait for a response from a job agent when sending a request to it. The default timeout is 1000 milliseconds. This should be more than enough if the job agent is on the internal network, but may have to be increased if it is located somewhere else.

Secondary storage controller

This section contains settings for the secondary storage controller. See Section 25.6.2, “Secondary file storage plugins” (page 255) for more information about secondary storage.

`secondary.storage.driver`

The class name of the plug-in that acts as the secondary storage controller. BASE ships with a simple plug-in that just moves files to another directory, but it is not enabled by default. The class name of that plug-in is `net.sf.basedb.core.InternalStorageController`. If no class is specified the secondary storage feature is disabled.

`secondary.storage.init`

Initialisation parameters sent to the plug-in when calling the `init()` method. The syntax and meaning of this string depends on the plug-in. For the internal controller this is simply the path to the secondary directory.

`secondary.storage.interval`

Interval in seconds between each execution of the secondary storage controller plug-in. If this property is not specified, `secondary.storage.time` should be set, or the secondary storage feature will be disabled.

`secondary.storage.time`

Time-point values specifying the time(s) of day that the secondary storage controller should be executed. If present, this setting overrides the `secondary.storage.interval` setting. Time-point values are given as comma-separated list of two-digit, 24-based hour and two-digit minute values. For example: `03:10,09:00,23:59`.

Change history logging section

This section contains settings for logging the change history of items.

`changelog.factory`

The factory class that controls the entire logging system. The factory has control of what should be logged, where it should be logged, etc. BASE ships with one factory implementation `DbLogManagerFactory` which logs changes into tables in the database. The server admin may choose

a different implementation provided that it implements the `LogManagerFactory` interface. See Section 25.6.5, “Logging plug-ins” (page 259). If no factory is specified, logging is disabled.

`changelog.show-in-web`

A boolean value that specifies if the **Change history** tab should be visible in the web interface or not. The change history tab will show log information that has been stored in the database and it doesn't make sense to show this tab unless the `DbLogManagerFactory` is used.

Note

By default, only users that are members of the **Administrator** role have permission to view the change history. To give other users the same permission, add the **Change history** permission to the appropriate role(s).

`changelog.dblogger.detailed-properties`

A boolean value that specifies the amount of information that should be logged. If set, the log will contain information about which properties that was modified on each item, otherwise only the type of change (create, update, delete) is logged.

SMTP server section

This section contains settings for the SMTP server used for outgoing mail. This is optional, and if not configured outgoing mail will be disabled.

`mail.server.host`

The host name of the SMTP server to use for outgoing mail. If not configured mailing functions will be disabled.

`mail.server.port`

The port the SMTP server is listening on. If not configured a default port is used. Eg. 25 for regular mail server, 465 for SSL mail server.

`mail.server.ssl`

A boolean value that specifies if the SMTP server is using SSL or not.

`mail.server.tls`

A boolean value that specifies if the SMTP server is using TLS or not.

`mail.from.email`

The email address that will be used as the sender of outgoing emails. If not configured mailing functions will be disabled.

`mail.from.name`

The name that will be used as the sender of outgoing emails. If not configured, a default name is automatically generated using the host name of the BASE server.

Plug-ins and extensions

`plugins.dir`

The path to the directory where jar-files for external plug-ins and extensions are located. All new plug-ins and extensions found in this directory, can be selected for installation, see Section 21.1, “Managing plug-ins and extensions” (page 178).

`plugins.autounload`

Enable this setting to let BASE detect if a plug-in JAR file is changed and automatically load and use the new code instead of the old code. This setting is useful for plug-in developers since they don't have to restart the web server each time the plug-in is recompiled.

- **true, yes, 1** to enable

- **false**, **no**, **0** to disable (default if no value is specified)

Note that extensions doesn't support this feature. Use the installation wizard to update an extension.

extensions.disabled

A boolean flag that, if set, disables all external extensions. Plug-ins or core extensions will never be disabled.

Other settings

userfiles

The path to the directory where uploaded and generated files should be stored. This is the primary file storage. See the section called “Secondary storage controller”(page 421) for information about how to configure the secondary storage. Files are not stored in the same directory structure or with the same names as in the BASE file system. The internal structure may contain sub-directories.

permission.timeout

Number of minutes to cache a logged in user's permissions before reloading them. The default value is 10. This setting affect how quickly a changed permission propagate to a logged in user. Permissions are always reloaded when a user logs in.

cache.timeout

Number of minutes to keep user sessions in the internal cache before the user is automatically logged out. The timeout is counted from the last access made from the user.

cache.static.disabled

If the static cache should be enabled or disabled. It is enabled by default. Disabling the static cache may reduce performance in some cases. The static cache is used to cache processed information, for example images, so that the database doesn't have to be queried on every request.

cache.static.max-age

The maximum age in days of files in the static cache. Files that hasn't been accessed (read or written) in the specified amount of time are deleted.

helptext.update

Defines if already existing helptexts in BASE should be overwritten when updating the program, Section 20.1, “Upgrade instructions” (page 165)

- **true** will overwrite existing helptexts.
- **false** will leave the existing helptexts in database unchanged and only insert new helptexts.

locale.language, locale.country, locale.variant

Configure the server to a specific locale. The language and country should be valid ISO codes as specified by the `java.util.Locale`³ documentation. The variant can be any value that is valid as part of a filename.

Note

Note that language codes are usually lower-case but country codes are upper case. Eg. `sv` is the language code for swedish, and `SE` is the country code.

This configuration can be used to provide translations to some parts of the web gui. The aim is to externalize all hard-coded gui elements from the code but it's a long way before this is a reality. The default text elements of the gui are shipped within the BASE jar files and doesn't have any locale-specific dependency. This means that unless a more specific translation is provided the

³ <http://download.oracle.com/javase/6/docs/api/java/util/Locale.html>

default texts are always used as a fallback. Most of the default texts are found in property files in the `/net/sf/basedb/clients/web/resources` directory inside the `base-webclient-3.x.jar` file. Translations should be located in the same relative path either inside their own JAR file or in the `WEB-INF/classes` directory. The file names should be extended with the language, country and variant separated with an underscore. For example, files with a swedish translation should be named `*_sv.properties`, and files with a swedish translation in Finland using the 'foo' variant should be named `*_sv_FI_foo.properties`.

Note

Note that it is valid to have empty values for language and/or country and still specify a variant. Underscores are NOT collapsed. For example, in a swedish translation using the 'foo' variant the files should be named `*_sv__foo.properties`.

Important

All files should be saved in UTF-8 format.

SSL section

This section is for global configuration of SSL (HTTPS) connection settings used when BASE need to access external file items. Note that users can re-configure SSL connections per-file basis by setting up File-server items, so there is usually no need to change anything in this section. If the majority of users on the BASE server is using a particular https file server for external files it may make sense to register the certificates globally.

When a https connection is made the server must present a valid certificate or the client (BASE) will refuse to connect to it. Typically, all certificates that have been signed by a recognised Certification Authority are considered valid. The major reason for configuring this section is to provide support for servers that use a self-signed certificate. Server-side certificate are stored in a *trust-store*. A default trust-store is shipped with the Java runtime installation and is found in `<java-home>/jre/lib/security/cacerts`. This file contains the certificates of all recognised certification authorities.

A https server may also require that the client has a valid certificate in order to accept connections from it. Typically, the owner of the server issues a certificate that the client must install in order to access the server. This type of certificate is stored in a *key-store*. By default, no key-store is setup.

If all you need is to support servers with self-signed certificates we recommend that those certificates are imported to the above mentioned file. No configuration changes are needed. If a key-store is needed, you must also configure the trust-store. Read the Java Secure Socket Extension Reference Guide⁴ for more information about Java security and SSL. Java ships with a certificate management tool that can be used to manage certificate files and a lot of other things. The keytool - Key and Certificate Management Tool⁵ document contains more information about this tool.

If you want to setup your own test environment with a https server that only accepts clients with a trusted certificate you can find some information about this on our wiki: <http://base.thep.lu.se/wiki/HttpsFiles>

`ssl.context.protocol`

The SSL protocol to use. The default value is TLS.

`ssl.context.provider`

A security provider implementation. If not specified a suitable default is selected.

`ssl.keystore.file`

The full path to a key-store file. If not specified no key-store is used.

⁴ <http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html>

⁵ <http://java.sun.com/javase/6/docs/technotes/tools/windows/keytool.html>

`ssl.keystore.password`

The password for unlocking the keys in the key-store. All keys must use the same password.

`ssl.keystore.type`

The file type of the key-store file. The default value is 'JKS'.

`ssl.keystore.provider`

The name of a provider implementation to use for reading the key-store file. If not specified a suitable default is used.

`ssl.keystore.algorithm`

The algorithm used in the key-store file. The default value is 'SunX509'.

`ssl.truststore.file`

The full path to a trust-store certificate file. If not specified the default depends on the key-store setting. If no key-store is configured, the default trust-store is used. If a key-store has been configured no trust-store is used.

`ssl.truststore.password`

The password for unlocking the certificates in the trust-store. All certificates must use the same password.

`ssl.truststore.type`

The file type of the trust-store file. The default value is 'JKS'.

`ssl.truststore.provider`

The name of a provider implementation to use for reading the trust-store file. If not specified a suitable default is used.

`ssl.truststore.algorithm`

The algorithm used in the trust-store file. The default value is 'PKIX'.

Appendix C. extended-properties.xml reference

What is extended-properties.xml?

The `extended-properties.xml` file is a configuration file for customizing some of the tables in the BASE database. It is located in the `<basedir>/www/WEB-INF/classes` directory. Only a limited number of tables support this feature, the most important one is the table for storing reporter information.

Tip

It is also possible to put additional extended property definitions in the `<basedir>/www/WEB-INF/classes/extended-properties` subdirectory. BASE will merge all `*.xml` it finds with the main `extended-properties.xml` file. The extra configuration files should have the same format as the main `extended-properties.xml` file. The extra files may contain extra columns for classes that are already defined in the main file, but existing columns can't be removed or re-defined. We recommend that you don't modify the default `extended-properties.xml` file at all (unless you want to remove some of the columns). This will make it easier when upgrading BASE since you don't have to worry about losing your own changes.

The default `extended-properties.xml` that ships with BASE is biased towards the BASE version 1.2 setup for 2-spotted microarray data. If you want your BASE installation to be configured differently we recommend that you do it before the first initialisation of the database. It is possible to change the configuration of an existing BASE installation but it may require manual updates to the database. Follow this procedure:

1. Shut down the BASE web server. If you have installed job agents you should shut down them as well.
2. Modify the `extended-properties.xml` file or create a new file in the `extended-properties` subdirectory. If you have installed job agents, make sure they all have the same version as the web server.
3. Run the `updatedb.sh` script. New columns will automatically be created, but the script can't delete columns that have been removed, or modify columns that have changed data type. You will have to do these kind of changes by manually executing SQL against your database. Check your database documentation for information about SQL syntax.

Create a parallel installation

You can always create a new temporary parallel installation to check what the table generated by installation script looks like. Compare the new table to the existing one and make sure they match.

4. Start up the BASE web server and job agents, if any, again.

Start with few columns

It is better to start with too few columns, since it is easier to add more columns than it is to remove columns that are not needed.

Sample extended properties setups

- After installing BASE the default `extended-properties.xml` is located in the `<basedir>/www/WEB-INF/classes` directory. This setup is biased towards the BASE version 1.2 setup for 2-spotted cDNA arrays. If you are migrating from BASE version 1.2 you *must* to use the default setup.

- A `minimalistic_extended-properties.xml` setup which doesn't define any extra columns at all. This file can be found in the `<basedir>/misc/config` directory, and should be used if it is not known what reporter data will be stored in the database. The addition of more columns later is straightforward.

Format of the extended-properties.xml file

The `extended-properties.xml` is an XML file. The following example will serve as a description of the format:

```
<?xml version="1.0" ?>
<!DOCTYPE extended-properties SYSTEM "extended-properties.dtd">
<extended-properties>
  <class name="ReporterData">
    <property
      name="extra1"
      column="extra1"
      title="Extra property"
      type="string"
      length="255"
      null="true"
      update="true"
      insert="true"
      averagemethod="max"
      description="An extra property for all reporters"
    >
    <link
      regexp=".*"
      url="http://www.myexternaldb.com/find?{value}"
    />
    </property>
  </class>
</extended-properties>
```

Each table that can be customized is represented by a `<class>` tag. The value of the `name` attribute is the name of the Java class that handles the information in that table. In the case of reporters the class name is `ReporterData`.

Each `<class>` tag may contain one or more `<property>` tags, each one describing a single column in the table. The possible attributes of the `<property>` tag are:

	extended-properties.xml reference	<ul style="list-style-type: none"> • float • double • boolean
Table C.1. Attributes for the <property> tag		<ul style="list-style-type: none"> • string • date • timestamp <p>Note that the given types are converted into the most appropriate database column type by Hibernate.</p>
length	no	If the column is a string type, this is the maximum length that can be stored in the database. If no value is given, 255 is assumed.
null	no	If the column should allow null values or not. Allowed values are <code>true</code> (default) and <code>false</code> .
insert	no	If values for this property should be inserted into the database or not. Allowed values are <code>true</code> (default) and <code>false</code> .
update	no	If values for this property should be updated in the database or not. Allowed values are <code>true</code> (default) and <code>false</code> .
averagemethod	no	<p>The method to use when calculating the average of a set of values. This attribute replaces the <code>averagable</code> attribute. The following values can be used:</p> <ul style="list-style-type: none"> • <code>none</code>: average values are not supported (default for non-numerical columns) • <code>arithmetic_mean</code>: calculate the arithmetic mean (default for numerical columns; not supported for non-numerical columns) • <code>geometric_mean</code>: calculate the geometric mean (not supported for non-numerical columns) • <code>quadratic_mean</code>: calculate the quadratic mean (not supported for non-numerical columns) • <code>min</code>: use the minimum value of the values in the set • <code>max</code>: use the maximum value of the values in the set

Each `<property>` tag may contain zero or more `<link>` tags that can be used by client application to provide clickable links to other databases. Each `<link>` has a `regexp` and an `url` attribute. If the regular expression matches the value a link will be created, otherwise not. The order of the `<link>` tags are important, since only the first one that matches is used. The `url` attribute may contain the string `{value}` which will be replaced by the actual value when the link is generated.

Note

If the link contains the character `&` it must be escaped as `&`. For example, to link to a UniGene entry:

```
<link
  regexp="\w+\.\d+"

  url="http://www.ncbi.nlm.nih.gov/entrez/
query.fcgi?db=unigene&amp;term={value} [ClusterID]"
/>
```

Appendix D. Platforms and raw-data-types.xml reference

Raw data can be stored either as files attached to items and/or in the database. The `Platform` item has information about this. For more information see Section 28.3.9, “Using files to store data” (page 330).

D.1. Default platforms and variants installed with BASE

Platform		Variants		Data file types		
Name	ID	Name	ID	Item	Name	ID
Generic	generic	-	-	Array design	Reporter map	generic.reportermap
					Print map	generic.printmap
				Raw bioassay	Generic raw data	generic.rawdata
Affymetrix	affymetrix	-	-	Array design	CDF file	affymetrix.cdf
				Raw bioassay	CEL file	affymetrix.cel
Sequencing	sequencing	Expression-like	sequencing.expression-like	Array design	GTF ref-seq file	refseq.gtf
				Raw bioassay	FPKM tracking file	sequencing.fpkm_tracking

D.2. raw-data-types.xml reference

A given platform either supports importing data to the database or it doesn't. If it supports import, it may be locked to specific raw data type or it may use any raw data type. Among the default platforms installed with BASE, the Affymetrix platform doesn't support importing data while the Generic platform supports importing to any raw data type.

Raw data types are defined in the `raw-data-types.xml` file. This file is located in the `<basedir>/www/WEB-INF/classes` directory and contains information about the database tables and columns to use for storing raw data. BASE ships with default raw data types for many different microarray platforms, including Genepix, Agilent and Illumina.

Tip

It is also possible to put additional raw data type definitions in the `<basedir>/www/WEB-INF/classes/raw-data-types` subdirectory. BASE will merge all `*.xml` it finds with the main `raw-data-types.xml` file. The extra configuration files should have the same format as the main `raw-data-types.xml` file. Duplicate raw data types are not supported and it is not possible to add extra columns to existing types using this approach.

If you want your BASE installation to be configured differently we recommend that you do it before the first initialisation of the database. It is possible to change the configuration of an existing BASE installation but it requires manual updates to the database. Following procedure covers how to update:

1. Shut down the BASE web server. If you have installed job agents you should shut down them as well.

2. Modify the `raw-data-types.xml` file or create a new file in the `raw-data-types` subdirectory. If you have installed job agents, make sure they all have the same version as the web server.
3. Run the `updatedb.sh` script. Tables for new raw data types and new columns for existing raw data types automatically be created, but the script can't delete tables or columns that have been removed, or modify columns that have changed datatype. You will have to do these kind of changes by manually executing SQL against your database. Check your database documentation for information about SQL syntax.

Create a parallel installation

You can always create a new temporary parallel installation to check what the table generated by installation script looks like. Compare the new table to the existing one and make sure they match.

4. Start up the BASE web server and job agents, if any, again.

Start with few columns

It is better to start with too few columns, since it is easier to add more columns than it is to remove columns that are not needed.

Format of the raw-data-types.xml file

The following example will serve as a description of the format used in `raw-data-types.xml`:

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="raw-data-types.xsl"?>
<!DOCTYPE raw-data-types SYSTEM "raw-data-types.dtd" >
<raw-data-types>
  <raw-data-type
    id="genepix"
    name="GenePix"
    channels="2"
    table="RawDataGenePix"
  >
    <property
      name="diameter"
      title="Spot diameter"
      description="The diameter of the spot in µm"
      column="diameter"
      type="float"
    />
    <property
      name="ch1FgMedian"
      title="Channel 1 foreground median"
      description="The median of the foreground intensity in channel 1"
      column="ch1_fg_median"
      type="float"
      channel="1"
    />
    <!-- skipped a lot of properties -->
    <intensity-formula
      name="mean"
      title="Mean FG - Mean BG"
      description="Subtract mean background from mean foreground"
    >
      <formula
        channel="1"
        expression="raw('ch1FgMean') - raw('ch1BgMean') "
      />
      <formula
        channel="2"
        expression="raw('ch2FgMean') - raw('ch2BgMean') "
      />
    </intensity-formula>
```

```
<!-- and a few more... --->
</raw-data-type>
</raw-data-types>
```

Each raw data type is represented by a `<raw-data-type>` tag. The following attributes can be used:

Table D.1. Attributes for the `<raw-data-type>` tag

Attribute	Required	Comment
id	yes	A unique ID of the raw data type. It should contain only letters, numbers and underscores and the first character must be a letter.
name	yes	A unique name of the raw data type. The name is usually used by client applications for display.
table	yes	The name of the database table to store data in. The table name must be unique and can only contain letters, numbers and underscores. The first character must be a letter.
channels	yes	The number of channels used by this raw data type. It must be a number > 0 .
description	no	An optional (longer) description of the raw data type.

Following the `<raw-data-type>` tag is one or more `<property>` tags. Each one defines a column in the database that is designed to hold data values of a particular type. The following attributes can be used on this tag:

Table D.2. Attributes for the `<property>` tag

Attribute	Required	Comment
*		All attributes defined by the <code><property></code> tag in <code>extended-properties.xml</code> . See Table C.1, “Attributes for the <code><property></code> tag” (page 428).
channels	no	The channel number the property belongs to. Allowed values are 0 to the number of channels specified for the raw data type. If the property doesn't belong to any channels set the value to 0 or leave it unspecified.

Following the `<property>` tags comes 0 or more `<intensity-formula>` tags. Each one defines mathematical formulas that can be used to calculate the intensity values from the raw data. In the Genepix case, there are several formulas which differs in the way background is subtracted from foreground intensity values. For other raw data types, the intensity formula may just copy one of the raw data values.

The intensity formulas are installed as `Formula` items in the database. This means that you can manually add, change or remove intensity formulas directly from the web interface. The intensity formulas in the `raw-data-types.xml` file are only used at installation time.

The `<intensity-formula>` tag has the following attributes:

Table D.3. Attributes for the `<intensity-formula>` tag

Attribute	Required	Comment
name	yes	A unique name for the formula. This is only used during installation.
title	yes	The title of the formula. This is used by client applications for display.
description	no	An optional, longer, description of the formula.

The `<intensity-formula>` must contain one `<formula>` tag for each channel of the raw data type. The attributes of this tag are:

Table D.4. Attributes for the `<formula>` tag

Attribute	Required	Comment
channel	yes	The channel number. One tag for each channel must be specified. No duplicates are allowed.
expression	yes	<p>The mathematical expression used to calculate the intensities. The expression is parsed with the <code>Jep</code> parser. It supports the common mathematical operations such as <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, some mathematical function like, <code>log2()</code>, <code>ln()</code>, <code>sqrt()</code>, etc. See the API documentation for <code>Jep</code> for more information. You can also use two special function developed specifically for this case:</p> <ul style="list-style-type: none"> <code>raw(name)</code>: Get the value from the raw data property with the given name, for example: <code>raw('ch1FgMedian')</code>. <code>mean(name)</code>: Get the mean value of the raw data property with the given name, for example: <code>mean('ch1BgMean')</code>. The mean is calculated from all raw data spots in the raw bioassay.

Appendix E. web.xml reference

The `web.xml` file is one step up from the main configuration directory. It is located in the `<basedir>/www/WEB-INF` directory. This configuration file contains settings that are related to the web application only. Most settings in this file should not be changed because they are vital for the functionality of BASE.

`<error-page>`

If an error occurs during a page request, the execution is forwarded to the specified JSP which will display information about the error.

`<context-param>: max-url-length`

This setting is here to resolve a potential problem with too long generated URL:s. This may happen when BASE needs to open a pop-up window and a user has selected a lot of items (*e.g.*, several hundred). Typically the generated URL contains all selected ID:s. Some web servers have limitations on the length of an URL (*e.g.*, Apache has a default max of 8190 bytes). If the generated URL is longer than this setting, BASE will re-write the request to make the URL shorter and supply the rest of the parameters as part of a POST request instead. This functionality can be disabled by setting this value to 0. For more information see <http://base.thep.lu.se/ticket/1032>.

`<servlet>: BASE`

A servlet that starts BASE when Tomcat starts, and stops BASE when Tomcat stops. Do not modify.

`<servlet>: view/download`

File view/download servlet. It is possible to change the default MIME type for use with files of unknown type.

`<servlet>: upload`

Servlet for handling file uploads. Do not modify.

`<servlet>: spotimage`

Servlet for displaying spot images. Do not modify.

`<servlet>: plotter`

Servlet for the plot tool in the analysis section. You may specify max and default values for the width and height for the generated images. The supported image formats are "png" and "jpeg".

`<servlet>: eeplotter`

Servlet for the plot tool in the experiment explorer section. It can use the same configuration properties for size and image format as the plotter servlet.

`<servlet>: news-feed`

Servlet for generating a RSS feed for the news on the front page. Comment out this servlet if you do not want to use the RSS feed.

`<servlet>: AxisServlet/AxisRESTServlet`

Servlet handling web service requests. If you are not planning to access your BASE installation using web services these servlets may be disabled.

`<servlet>: ExtensionsServlet`

Servlet for handling startup/shutdown of the extensions system as well as requests to extension servlets. Do not modify. Do not disable even if extensions are not used.

`<servlet>: xjsp`

Experimental servlet for compiling *.xjsp files used by extensions. The servlet redirects the compilation of *.xjsp files to a compiler that includes the extension supplied JAR file(s) in the class path. Can be disabled if no extensions use this feature. See also Section 21.1.4, "Installing the X-JSP compiler" (page 183) for more information about how to enable this feature.

`<servlet>: compile`

Experimental servlet for compiling all JSP files. This is mostly useful for developers who want to make sure that no compilation error exists in any JSP file. Can also be used to pre-compile all JSP files to avoid delays during browsing. This servlet is disabled by default.

`<filter>: characterEncoding`

A filter that sets the character encoding for the JSP generated HTML. We recommend leaving this at the default UTF-8 encoding, this default should work with most language in all modern browsers.

Appendix F. jobagent.properties reference

The `jobagent.properties` file is the main configuration file for job agents. It is located in the `<basedir>/www/WEB-INF/classes` directory.

BASE settings

This section describes the configuration parameters that are used by the job agent to get access to the BASE server.

`agent.user`

Required. The BASE user account used by the job agent to log on to the BASE server. The user account must have sufficient privileges to access jobs and job agents. The *Job agent* role is a predefined role with all permissions a job agent needs. There is also a predefined user account with the user name *jobagent*. This account is disabled by default and has to be enabled and given a password before it can be used.

`agent.password`

Required. The password for the job agent user account.

`agent.id`

Required. A unique ID that identifies this job agent among other job agents. If multiple job agents are installed each job agent should have its own unique ID.

`agent.name`

Optional. The name of the job agent. If not specified the ID is used. The name is only used when registering the job agent with the BASE server.

`agent.description`

Optional. A description of the job agent. This is only used when registering the job agent.

Job agent server settings

This section describes the configuration parameters that affect the job agent server itself.

`agent.port`

Optional. The port the job agent listens to for control requests. Control requests are used for starting, stopping, pausing and getting status information from the job agent. It is also used by the **jobagent.sh** script to control the local job agent. The default value is 47822.

`agent.remotecontrol`

Optional. A comma-separated list of IP addresses or names of computers that are allowed to send control requests to the job agent. If no value is specified, only the local host is allowed to connect. It is recommended that the web server is added to the list if the job agent is not running on the same server as the web server.

`agent.allowremote.stop`

Optional. If the **stop** command should be accepted from remote hosts specified in the `agent.remotecontrol` setting. If `false`, which is the default value, only the local host is allowed to stop the job agent.

Note

Once the job agent has been stopped it cannot be started by remote control. You must use the **jobagent.sh** script for this.

agent.allowremote.pause

Optional. If the **pause** command should be accepted from remote hosts specified in the `agent.remotecontrol` setting. If `false`, only the local host is allowed to pause the job agent. The default value is `true`.

agent.allowremote.start

Optional, valid only when job agent is paused. If the **start** command should be accepted from remote hosts specified in the `agent.remotecontrol` setting. If `false`, only the local host is allowed to start the job agent when it is paused. The default value is `true`.

Custom request handlers

agent.request-handler.*

Optional. One or more entries for custom remote control handlers. The `*` should be replaced with the name of the protocol and the value should be the name of a class implementing the `CustomRequestHandler` interface. Requests can then be sent to the agent's remote control port on the form: `foo://custom-data.....`

Job execution settings

This section describes the configuration parameters that affect the execution of jobs.

agent.executor.class

The name of the Java class that handles the actual execution of jobs. The default implementation for a job agent ships three implementations:

- `net.sf.basedb.clients.jobagent.executors.ProcessJobExecutor` : Executes the job in an external process. This is the recommended executor and is the default choice if no value has been specified. With this executor, a misbehaving plugin does not affect the job agent or other jobs. The drawback is that since a new virtual machine has to be started, more memory is required and the start up time can be long.
- `net.sf.basedb.clients.jobagent.executors.ThreadJobExecutor` : Executes the job in a separate thread. This is only recommended for plugins that are trusted and safe. A misbehaving plugin can affect the job agent and other jobs, but the start up time is short and less memory is used.
- `net.sf.basedb.clients.jobagent.executors.DummyJobExecutor`: Does not execute the job. It only marks the job as being executed, and after waiting some time, as finished successfully. Use it for debugging the job agent.

It is possible to create your own implementation of a job executor. Create a class that implements the `net.sf.basedb.clients.jobagent.JobExecutor` interface.

agent.executor.process.java

Optional. The path to the Java executable used by the `ProcessJobExecutor` . If not specified the `JAVA_HOME` environment variable will be checked. As a last resort **java** is used without path information to let the operating system find the default installation.

agent.executor.process.options

Optional. Additional command line options to the Java executable. Do not add memory options (`-Mx` or `-Ms`), it will be added automatically by the executor. This setting is used by the `ProcessJobExecutor` only.

agent.executor.dummy.wait

Optional. Number of seconds the `DummyJobExecutor` should wait before returning from the "job execution". The executor first sets the progress to 50% then waits the specified number of seconds before setting the job to completed. If no value is specified it returns immediately.

agent.checkinterval

Optional. Number of seconds between querying the database for jobs that are waiting for execution. The default value is 30 seconds.

Slots and priorities

The job agent does not execute an arbitrary number of jobs simultaneously. This would sooner or later break the server. A *slot manager* is used to assign jobs to a pre-configured number of slots.

agent.slotmanager.class

The name of the Java class that handles slot assignment to jobs. The standard job agent ships with three different implementations:

- `net.sf.basedb.clients.jobagent.slotmanager.InternalSlotManager` : This is the default slot manager. It uses a simple system with four different slots. Each slot is reserved for jobs that are estimated to be finished in a certain amount of time. The exception is that a quick job may use a slot with longer expected time since that will not block the slot very long. See the table below for default settings.
- `net.sf.basedb.clients.jobagent.slotmanager.MasterSlotManager` : This is an extension to the internal slot manager that also accepts requests for slot assignments from other job agents. The other job agent(s) should be using the `RemoteSlotManager`. This makes it possible for a number of job agents to share a common pool of slots to avoid bottlenecks, for example, at the database level.
- `net.sf.basedb.clients.jobagent.slotmanager.RemoteSlotManager` : The remote slot manager connects with another job agent (running with a `MasterSlotManager`) and asks it for a slot. When this slot manager is used you need to specify the ip-address/name and port of the job agent running the master slot manager.

It is possible to create your own implementation of a slot manager. Create a class that implements the `net.sf.basedb.clients.jobagent.slotmanager.SlotManager` interface.

agent.slotmanager.remote.server

The ip-address or name of a job agent running as the master slot manager. This setting is needed by the `RemoteSlotManager`.

agent.slotmanager.remote.port

The remote control port number of the job agent running as the master slot manager. Make sure that the master job agent is accepting connection from this job agent. This setting is needed by the `RemoteSlotManager`.

This table lists slot settings for the internal and master slot managers. The remote slot manager will get slots from another job agent. A thread priority is associated with each slot. The priority is a value between 1 and 10 as defined by the `java.lang.Thread`¹ class. The priorities are not handled by the slot managers, but by the job agent core and apply to all job agents, no matter which slot manager that is selected.

Property	Default value	Estimated execution time
<code>agent.shortest.slots</code>	1	< 1 minute
<code>agent.shortest.priority</code>	4	
<code>agent.short.slots</code>	1	< 10 minutes
<code>agent.short.priority</code>	4	
<code>agent.medium.slots</code>	2	< 1 hour
<code>agent.medium.priority</code>	3	

¹ <http://java.sun.com/javase/6/docs/api/java/lang/Thread.html>

Property	Default value	Estimated execution time
agent.long.slots	2	> 1 hour
agent.long.priority	3	

Appendix G. jobagent.sh reference

The `jobagent.sh` (or `jobagent.bat` on Windows) is a command-line utility for controlling the job agent. The syntax is:

```
./jobagent.sh [options] command
```

The options are optional, but a **command** must always be given. The script is located in the `<base-dir>/bin` directory and you must change to that directory to be able to use the script.

Options

-c

The path to the configuration file to use, for example:

```
./jobagent.sh -c other.config start
```

The default value is `jobagent.properties`. The classpath is not searched which means that it doesn't find the configuration file in `<base-dir>/www/WEB-INF/classes/` unless you specify the path to it. See Appendix F, *jobagent.properties reference* (page 436) for more information about job agent configuration files.

Commands

register

Register the job agent with the BASE server. If the job agent already exists this command does nothing.

unregister

Unregister/delete the job agent from the BASE server. If the job agent does not exist this command does nothing.

start

Start the job agent. As soon as it is up and running it will check the database for jobs that are waiting to be executed.

pause

Pause the job agent. The job agent will continue running but does not check the database for jobs. To start it again use the **start** command.

stop

Stop the job agent. To start it again use the **start** command.

info

Get information about the job agent. This will output a string in the form:

```
Status:Running
Cpu:15
Total memory:8254955520
Used memory:8002252800
Job:42
Job.42.slot:SHORT
```

Status can be either Running or Paused. There is some information about the current CPU and memory usage, but this information may not be available on all platforms. For each job that is

currently running, the ID is given. A second entry gives information about the slot the job uses for execution. In the future, the **info** command may output more information.

status

Similar to the **info** command but with less output. The output is either `Running`, `Pauses` or `Stopped`. In case of an unexpected error, an error message may be displayed instead.

help

Display usage information.

Appendix H. Other configuration files

H.1. mysql-queries.xml and postgres-queries.xml

TODO

H.2. log4j.properties

TODO

H.3. hibernate.cfg.xml

TODO

H.4. ehcache.xml

TODO

Appendix I. API changes that may affect backwards compatibility

In this document we list all changes to code in the *Public API* that may be backwards incompatible with existing client applications and or plug-ins. See Section 28.1, “The Public API of BASE” (page 287) for more information about what we mean with the *Public API* and backwards compatible.

I.1. BASE 3.0 release

There are a lot incompatible changes between BASE 3 and any of the BASE 2.x versions. We do not list those changes here since we do not expect existing plug-ins, extensions or other client application to work with BASE 3 without modification. See Chapter 23, *Migrating code from BASE 2 to BASE 3* (page 213) for more information.

I.2. All BASE 2.x releases

The list of changes made in the various BASE 2.x releases can be found in the BASE 2.17 documentation¹.

¹ <http://base.thep.lu.se/chrome/site/2.17/html/appendix/appendix.incompatible.html>

Appendix J. Things to consider when updating an existing BASE installation

This document is a list of things that may have to be considered when updating a BASE installation to a newer version. The Section 20.1, “Upgrade instructions” (page 165) section only include the most recent information that is needed for updating the previous BASE version to the current version.

J.1. All BASE 2.x releases

We only support updating to BASE 3 from BASE 2.17. If you have an older BASE version and wish to update to BASE 3, you first have to upgrade to BASE 2.17. BASE 2.17 can be downloaded from the BASE download page¹. Documentation for BASE 2.17 is available as part of the download and at <http://base.thep.lu.se/chrome/site/2.17/html/index.html>.

¹ <http://base.thep.lu.se/wiki/DownloadPage>

Appendix K. File formats

K.1. The BFS (BASE File Set) format

The BASE File Set (BFS) format is a collection of file formats that can be used together to transport all kinds data. The major use is to send spot data to a plug-in for analysis and then to import the analyzed results. We have tried to keep the format generic and extendable so it is not unlikely that the BFS format can be used for other applications in the future.

K.1.1. The basics of BFS

The idea is to use simple, plain-text files with data organised into rows and columns. A single type of file may not be able to hold all kinds of data, so to begin with we have defined three types of files:

- Metadata files: Holds information about the data that is found in the other files in the file set.
- Annotation files: Column-based files that holds one record per line. The first line is a header line. The remaining lines are data lines identified by a unique positive ID value in the first column.
- Data files: Pure matrix data files without header lines or ID columns. Data is usually identified by matching it line-by-line with data in annotation files, or with information in the metadata file.

Character encoding

All files are text-based and should use the UTF-8 character encoding. A newline (`\n`) is used as a record separator and a tab (`\t`) is used a column separator. Data that contains newline or tab characters need to be escaped. A backslash (`\`) is used to indicate the start of an escaped sequence. This means that the backslash character must also be escaped. Since some editors includes a carriage return (`\r`) in line breaks, we should also escape carriage return.

Table K.1. Escaped characters in the BFS format

Character	Escape sequence
<backslash>	<code>\\</code>
<newline>	<code>\n</code>
<carriage return>	<code>\r</code>
<tab>	<code>\t</code>

It is recommended that parsers are forgiving and if an invalid escape sequence is found, eg. a backslash followed by anything else than `\`, `n`, `r` or `t`, the input is taken literally. Strict parsers may throw exceptions or log warning messages.

Numerical values

Numeric values should use dot (`.`) as decimal point. Scientific notation is accepted. Null, NaN, Infinity, and other special values should all be represented by empty string values. It is recommended that parsers are forgiving and treat invalid numerical data as empty values.

Comments and white-space

Lines starting with `#` are comment lines and should be ignored. Empty lines should also be ignored. A line that contains only white-space is considered as empty. White-space=spaces, tabs and other characters that matches `\s` in regular expressions.

Note

This can only be used in metadata files. Annotation files and data files doesn't allow comments or empty lines.

Metadata files

A BASE File Set usually contains one metadata file. This file contains information about the other files that make up the file set. The metadata file can also hold information that is specific to a use case.

A metadata file always starts with the beginning-of-file (BOF) marker `BFSformat`, optionally followed by a tab and a value indicating the subtype of the file. This must be the first line of the file. Comments or empty lines are not allowed before the beginning-of-file marker.

All data in a metadata file must be inside a section. A section is started by surrounding a value in brackets on a line by it's own, for example, `[my section]`. There is no restriction on the name of the section as long as it is escaped using the normal rules. Note that there is no need to escape brackets in the name. For example, `[[a\\b]]` is a valid section with the name `[a\b]`. Trailing white-space after the closing bracket is ignored.

Multiple sections may have the same name, and the order of the sections is usually of no concern. However, this may be restricted in specific cases if there is need to, for example, require unique section names or enforce a specific order. Parsers are recommended to provide access to sections by name and by ordinal number, starting at 0 and writers are recommended to write sections in the order they are added.

Each section contains data in the form of tab-separated key-value pairs. Keys may not start with `#` or `[` since this would interfere with comments and sections. Otherwise, the normal escape rules should be used for both keys and values. Values are allowed to use non-escaped tab characers, which makes it possible to use vector-type values.

A key doesn't have to be unique within a section, but specific use cases may require this globally or on section-per-section basis. The order of the keys are usually not important, except if the use case requires it. Parser implementations are recommended to provide access to keys by name and by ordinal number, starting at 0. Generic writers implementations are recommended to write keys and values in the order they are added to each section.

If the file set includes more files than the metadata file, those files should be listed in the `[files]` section. Keys should be unique, but there are no other restrictions. The value is the file name without path information. The files are expected to be located in the same container as the current metadata file. A container could for example be a folder in the file system, a zip-file, or any other logical item that group files. Metadata about the files and file content is not part of the generic BFS specification. This is left to specific use cases.

Note

Files doesn't have to be other BFS file types. It can be any type of files, like pdf files, images, etc.

Example K.1. Example BFS metadata file

```
BFSformat subtype
# The 'BFSformat' must be on the the first line, subtype is optional
# A comment line starts with '#'. Empty lines are ignored

# A section is started by enclosing the section name in brackets
# Section entries are key/value pairs separated by tab
# Vector-type values are allowed. Duplicate keys may or may
# not be allowed depending on the use case.
[settings]
key-1 value1
key-2 value2a value2b

# The 'files' section points to additional files in the file set
# Keys should be unique
[files]
report report.txt
table tabla-data.txt
plot plotted-data.png
```


Annotation files

The first line is a header line containing the column names for each column. The first column is required and must always be `ID`. Other columns are optional, but must have unique names. Column names are separated with tabs and are encoded using the normal rules. All other lines are data lines. Each line must have *exactly the same number of columns* as the header line. Comment lines and empty lines are not supported, but a column may have an empty value.

The ID column holds a unique identifier used internally by BASE. A given ID should only be used once and may not be repeated later in the file. The ID is a numeric positive integer value. Zero, negative or empty values are not allowed. There is no special ordering (unless a specific use-case require this). Note that the ID values are not indexes. They don't have to start at 1 and there may be "holes" in the range of values used. Some use-cases may use ID values with some specific meaning, other use-cases may simply enumerate the rows using a counter.

Data files

A data file is a matrix containing one data value for each row-column element. Data starts on the first line. There is no header line. All data lines *must have the same number of columns*. The number of rows and columns and their order are defined by other, use-case specific, information in the metadata file or in annotation file(s). Comment lines and empty lines are not supported, but a column may hold an empty value.

K.1.2. Using BFS for spotdata to and from external plug-ins

The use case is to use BFS to transport data to and from an external analysis plug-in. The general outline is:

1. Export bioassay set data to BFS.
2. Execute the external plug-in which process the data and generates a new BFS.
3. Import the transformed data to BASE.

The export will generate at least two files. One metadata file and one data file. It is also possible to export reporter and assay annotations if the plug-in needs it. Note that reporter and assay annotation files are always needed if new spot data is going to be imported so in most cases at least four files will be created.

The metadata file

There are two subtypes:

- **serial:** One data file is required for each assay. The columns in the data files represents different spot data values, eg. first column = Ch 1, second column = Ch 2, etc.
- **matrix:** One data file is required for each spot data value. The columns in the data files represents assays.

For both subtypes the `[files]` section is used to name the files holding data and annotations. The following entries should be used:

- `rdata`: The filename of the file containing reporter annotations
- `pdata`: The filename of the file containing assay annotations
- `sdata1`, `sdata2`, ..., `sdataN`: N entries, numbered from 1 to N, with the filenames of the files containing spot data. If the serial subtype is used there should be one file for each assay in the bioassayset. If the matrix subtype is used, there should be one file for each entry in the `[sdata]` section.

Other files may be included if they use `x-` as a prefix.

Example:

```
BFSformat serial
[files]
rdata reporters.txt
pdata assays.txt
sdata1 Assay 1.txt
sdata2 Assay 2.txt
x-custom custom.txt
```

The `[sdata]` section contains information about the spot data that is found in the `sdataX` files. The key of each entry is the name or title of the data that is exported. The value describes the data type and can be either `text`, `float` or `int`.

The order in this section is important. If the matrix subtype is used, the entries in this section must match the `sdataX` entries in the `[files]` section. Eg. the data that corresponds to the first entry in this section is found in the `sdata1` file. The number of entries in this section must be the same as the number of `sdataX` entries in the `[files]` section.

If the serial subtype is used the entries in this section must match the column order in each of the `sdataX` files. Eg. the data that corresponds to the first entry in this section is found in the first column in all `sdataX` files. The number of entries in this section must match the number of columns in the `sdataX` files.

Example:

```
[sdata]
Ch 1 float
Ch 2 float
Weight float
Flag int
```

The `[parameters]` section contains extra parameters needed by the plug-in. Keys and values are defined by the plug-in and/or job configuration. Duplicate keys are not allowed, and order is not important. Multiple values for the same parameter are separated with a tab character.

Example:

```
[parameters]
beta 0.5
length 100
vector 10 10.3 23
median true
```

Reporter and assay annotations

The file used for reporter annotations is given by the `rdata` entry in the `[files]` section. This file is optional when exporting but required when importing. The only required column is the `ID` column, which holds the internal spot position values. All `sdataX` files must have the same number of rows as this file (not counting the header line) and data should be sorted in the same order. Additional columns may be included in the export.

Note that the same underlying reporter may be assigned to more than one position. If the plug-in needs to operate on merged-per-reporter data the export should include either the internal or external reporter id in an additional column so that the plug-in can use this information to determine what should be merged. The exporter has no support for exporting merged data.

The file used for assay annotations is given by the `pdata` entry in the `[files]` section. This file is optional when exporting but required when importing. The only required column is the `ID` column,

which holds the internal bioassay id values. If the matrix subtype is used the columns in the `sdataX` files must be in the same order as the assays appear in this file. The number of columns in the data files must be the same as the number of rows in this file (not counting the header line).

If the serial subtype is used, the `sdata1` file has data for the assay that is described in the first line in this file, the `sdata2` file has data for the second assay, etc. The number of data files must match the number of lines in this file.

Data files

Data files contains data in matrix format. More than one data file may be required. The organisation of the data depends on the BFS subtype. In both subtypes the number and order of the rows must match the number and order of rows in the reporter annotations file.

If the matrix subtype is used, the columns in the data files corresponds to assays. The number of columns and their order must match the lines in the assay annotations file. The number of data files and their content is defined by the entries in the `[sdata]` section.

If the serial subtype is used, the the number of columns and their order must match the entries in the `[sdata]` section. Each data file has data from one assay. The number of `sdata` files in the `[files]` section must match the number of lines in the assay annotations file.

Importing spot data

The above information is mostly true for both export and import, but there are a few additional things that a plug-in should know about when generating data that is going to be imported. The most important thing is that both reporter and assay annotation files are required for importing spot data. If the program only generates extra files the `[sdata]` section should not be included and no data or annoatation files are need. All files are specified in the `[files]` section in the same way as for the export. File entries starting with `x-` will be uploaded to BASE and linked with the new bioassay set.

Note

The importer currently supports importing spot data intensity values and extra files. Position/reporter mapping and child/parent assay mapping may remain the same or they may be changed. The importer can also upload additional files generated by the plug-in, for example plots. The importer has no support for importing extra values, reporter lists or annotations.

In the metadata file, a `[settings]` section may be included to control certain aspects of the import. The following entries can be used:

- `new-data-cube`: If this is set, the data is imported into a new data cube. A new data cube is needed whenever the position/reporter mappings has changed or when parent assays has been merged. This setting requires that the reporter annotations file contains information about the new mapping. It needs to include either `Internal ID` or `External ID` columns so that the importer can map the new position to the correct reporter. The reporter must already exist in the database. The position values have no relation to the position values in the old bioassay set. We recommend that a plug-in simply starts enumerates the lines starting at 1.
- `multi-assay-parents`: If this is set, a child assay may have more than one parent assay (for example, due to a merge). A new data cube is needed and this setting is ignored unless `new-data-cube` is also set. This setting requires that the assay annotations file has a `Parent ID` column which holds a comma-separated list with the ID:s of the parent assays.
- `transform`: If not specified, the child spot data is assumed to use the same intensity transform as the parent data. To force a specific a specific intensity transform for the child bioassay set include this setting and choose one fo the values: none, log2, log10.

In the metadata file, the precense of an `[sdata]` section indicates that spot data should be imported. If this section is not present only extra files are uploaded to BASE and they are attached to the

transformation instead of a child bioassay set. If the `[sdata]` section is present it must include one entry for each channel with names like, `Ch 1`, `Ch 2`, and so on. The value is always `float`. All other entries in this section are ignored.

In the reporter annotations file, the `ID` column should hold the position values. Values must be positive integers and duplicates are not allowed. The order of the values doesn't matter. If importing data to a new data cube the reporter annotations file also needs either `Internal ID` or `External ID` columns.

In the assay annotations file, the `ID` column usually holds the internal assay id of the parent assay. The exception is if the `multi-assay-parents` options has been enabled. In this case the id values have no special meaning, but the `Parent ID` column must have a comma-separated list with id values instead.

The assay annotations file may optionally have a `Name` column. If present, the values in this columns are used as names on the child assays. Otherwise, they are given default names (usually the same name as the parent assay).

K.2. The BASEfile format

K.2.1. To be done