

BASE 2.14.2 Documentation

BASE 2.14.2 Documentation

Version:

2.14.2 (build #5192)

Published

11/23/2009 09:27 AM

Table of Contents

I. Overview	1
1. Why use BASE	2
1.1.	2
2. BASE features	3
2.1. Supported array platforms and raw data formats	3
2.1.1. Vendor specific and custom printing array platforms	3
2.1.2. Available raw data types	4
3. Overview of BASE	5
3.1.	5
4. Resources	6
4.1. BASE project site	6
4.1.1. Download	6
4.1.2. Ticket system	6
4.1.3. Roadmap	6
4.1.4. Documentation	7
4.2. Core plug-in configurations	7
4.3. BASE plug-ins site	7
4.4. Demo server	7
4.5. Mailing lists	8
4.6. BASE-hacks	8
II. User documentation	9
5. Overview of user documentation	10
5.1. Working environment	10
5.2. Start working with BASE	10
5.2.1. Get going	11
6. Using the web client	13
6.1. Introduction	13
6.1.1. Logging in	13
6.1.2. Forgotten password	13
6.1.3. The home page	13
6.1.4. Using the menu bar	15
6.1.5. Getting help	15
6.2. Configuring your account	16
6.2.1. Contact information	16
6.2.2. Other information	16
6.2.3. Changing password	17
6.2.4. Preferences	17
6.3. Working with items	19
6.3.1. Create a new item	20
6.3.2. Edit an existing item	20
6.3.3. Delete items	21
6.3.4. Restore deleted items	21
6.3.5. Share items to other users	21
6.3.6. Change owner of items	23
6.3.7. Export items	24
6.4. Listing items	24
6.4.1. Ordering the list	25
6.4.2. Filtering the list	25
6.4.3. Configuring which columns to show	27
6.4.4. Presets	28
6.5. Trashcan	31
6.5.1. Delete items permanently	32
6.5.2. View dependencies of a trashed item	32
6.6. Item overview	33
6.6.1. Validation options	34

6.6.2. Fixing validation failures	35
7. Projects and the permission system	37
7.1. The permission system	37
7.1.1. Permission levels	37
7.1.2. Getting access to an item	38
7.1.3. Plug-in permissions	38
7.2. Projects	38
7.2.1. The active project	39
7.2.2. How to give other users access to your project	40
7.2.3. Working with the items in the project	41
8. File management	43
8.1. File system	43
8.1.1. Browse the file system	44
8.1.2. Disk space quota	44
8.2. Handling files	44
8.2.1. Upload a new file	44
8.2.2. Edit a file	46
8.2.3. Move files	47
8.2.4. Viewing and downloading files	48
8.2.5. Directories	49
8.3. File types	50
9. Jobs	51
9.1. Properties	51
10. Reporters	54
10.1. Introduction	54
10.2. Reporter types	54
10.3. Reporters	55
10.3.1. Import/update reporter from files	55
10.3.2. Manual management of reporters	55
10.3.3. Deleting reporters	58
10.4. Reporter lists	58
11. Annotations	60
11.1. Annotation Types	60
11.1.1. Properties	61
11.1.2. Options	62
11.1.3. Item types	63
11.1.4. Units	64
11.1.5. Categories	65
11.2. Annotation type categories	65
11.3. Best Practices and Tab2Mage Export functionality	65
11.4. Annotating items	65
11.4.1. Inheriting annotations from other items	66
11.4.2. Mass annotation import plug-in	67
12. Experimental platforms and data file types	69
12.1. Platforms	69
12.1.1. Platform variants	71
12.2. Data file types	71
13. Protocols and protocol types	74
13.1. Protocol types	74
13.2. Protocols	75
13.2.1. Protocol properties	75
13.2.2. Protocol parameters	75
14. Hardware	77
14.1. Hardware types	77
14.2. Hardware	77
15. Software	79
15.1. Software types	79
15.2. Softwares	79

16. Array LIMS	80
16.1. Plates	80
16.2. Array designs	80
16.2.1. Properties	80
16.2.2. Importing features to an array design	81
16.3. Array batches	81
16.3.1. Creating array batches	81
16.3.2. Properties	82
16.4. Array slides	82
16.4.1. Creating array slides	82
16.4.2. Properties	83
17. Biomaterial	85
17.1. Introduction	85
17.2. Biosources	85
17.2.1. Properties	85
17.3. Samples	86
17.3.1. Create sample	86
17.3.2. Properties	87
17.4. Extracts	89
17.4.1. Create extract	89
17.4.2. Properties	89
17.5. Labels	91
17.5.1. Properties	91
17.6. Labeled extracts	91
17.6.1. Creating labeled extracts	91
17.6.2. Properties	92
17.7. Bioplate	93
17.7.1. Properties	93
17.7.2. Biowell	94
17.8. Hybridizations	95
17.8.1. Creating hybridizations	95
17.8.2. Properties	96
18. Experiments and analysis	98
18.1. Scans and images	98
18.1.1. Scan properties	98
18.1.2. Images	98
18.2. Raw bioassays	99
18.2.1. Raw bioassay properties	99
18.2.2. Import raw data	100
18.2.3. Raw data types	101
18.2.4. Spot images	101
18.3. Experiments	103
18.3.1. Experiment properties	103
18.3.2. Experimental factors	104
18.3.3. Tab2Mage export	105
18.4. Analysing data within BASE	106
18.4.1. Transformations and bioassay sets	106
18.4.2. Filtering data	107
18.4.3. Normalizing data	107
18.4.4. Other analysis plug-ins	107
18.4.5. The plot tool	107
18.4.6. Experiment explorer	107
19. Import of data	108
19.1. General import procedure	108
19.1.1. Select plug-in and file format	108
19.1.2. Specify plug-in parameters	110
19.1.3. Add the import job to the job queue	111
19.2. Batch import of data	112

19.2.1. File format	112
19.2.2. Running the item batch importer	113
19.2.3. Comments on the item batch importers	114
20. Export of data	116
20.1. Select plug-in and configuration	116
20.2. Specify plug-in parameters	116
20.3. The table exporter plug-in	118
III. Admin documentation	120
21. Installation, setup, migration, and upgrade instructions	121
21.1. Upgrade instructions	121
21.2. Installing job agents	123
21.2.1. BASE application server side setup	123
21.2.2. Database server setup	124
21.2.3. Job agent client setup	124
21.2.4. Configuring the job agent	125
21.3. Installation instructions	125
21.4. Server configurations	129
21.4.1. Sending a broadcast message to logged in users	130
21.5. Migration instructions	131
22. Plug-ins	133
22.1. Installing plug-ins	133
22.1.1. Select installation method	133
22.1.2. Plug-in properties	134
22.1.3. Automatic installation of plug-ins	136
22.1.4. BASE version 1 plug-ins	137
22.2. Plug-in permissions	137
22.3. Job agents	139
22.4. Plug-in configurations	140
22.4.1. Configuring plug-in configurations	141
22.4.2. Importing and exporting plug-in configurations	143
22.4.3. The Test with file function	143
23. Extensions	148
23.1. Installing extensions	148
23.2. Installing the X-JSP compiler	149
23.3. Configuring the extensions system	149
23.3.1. Settings	150
23.3.2. Disable/enable extensions	151
24. Account administration	152
24.1. Users administration	152
24.1.1. Edit user	152
24.1.2. Default group and role membership	154
24.2. Groups administration	155
24.2.1. Edit group	155
24.3. Roles administration	156
24.3.1. Pre-defined system roles	156
24.3.2. Edit role	156
24.4. Disk space/quota	158
24.4.1. Edit quota	158
24.4.2. Disk usage	159
IV. Developer documentation	160
25. Developer overview of BASE	161
25.1. Fixed vs. dynamic database	162
25.2. Hibernate and the DbEngine	163
25.3. The Batch API	163
25.4. Data classes vs. item classes	164
25.5. The Query API	164
25.6. The Controller API	165
25.7. Plug-ins	165

25.8. Client applications	165
26. Plug-in developer	167
26.1. How to organize your plug-in project	167
26.1.1. Using Ant	167
26.1.2. With Eclipse	169
26.1.3. Make the plug-in compatible with the auto-installation wizard	169
26.2. The Plug-in API	170
26.2.1. The main plug-in interfaces	170
26.2.2. How the BASE core interacts with the plug-in when... ..	180
26.2.3. Using custom JSP pages for parameter input	182
26.3. Import plug-ins	184
26.3.1. Autodetect file formats	184
26.3.2. The AbstractFlatFileImporter superclass	185
26.4. Export plug-ins	190
26.4.1. Immediate download of exported data	190
26.4.2. The AbstractExporterPlugin class	192
26.5. Analysis plug-ins	193
26.5.1. The AbstractAnalysisPlugin class	196
26.5.2. The AnalysisFilterPlugin interface	196
26.6. Other plug-ins	197
26.6.1. Authentication plug-ins	197
26.6.2. Secondary file storage plug-ins	199
26.6.3. File unpacker plug-ins	200
26.6.4. File packer plug-ins	202
26.6.5. File validator and metadata reader plug-ins	203
26.6.6. Logging plug-ins	204
26.7. Enable support for aborting a running a plug-in	206
26.8. How BASE load plug-in classes	207
26.9. Example plug-ins (with download)	208
27. Extensions developer	209
27.1. Overview	209
27.1.1. Download code examples	209
27.1.2. Terminology	209
27.2. Hello world as an extension	210
27.3. Custom action factories	211
27.4. Custom images, JSP files, and other resources	215
27.4.1. Javascript and stylesheets	216
27.4.2. X-JSP files	216
27.5. Custom renderers and renderer factories	217
27.6. Extension points	218
27.7. Custom servlets	220
28. Web services	223
28.1. Available services	223
28.1.1. Services	223
28.2. Client development	224
28.2.1. Receiving files	224
28.3. Services development	226
28.3.1. Generate WSDL-files	226
28.4. Example web service client (with download)	227
29. API overview (how to use and code examples)	228
29.1. The Public API of BASE	228
29.1.1. What is backwards compatibility?	228
29.2. The database schema and the Data Layer API	229
29.2.1. Basic classes and interfaces	230
29.2.2. User authentication and access control	234
29.2.3. Hardware and software	237
29.2.4. Reporters	238
29.2.5. Quota and disk usage	239

29.2.6. Client, session and settings	240
29.2.7. Files and directories	242
29.2.8. Experimental platforms	244
29.2.9. Parameters	246
29.2.10. Annotations	248
29.2.11. Protocols	251
29.2.12. Plug-ins, jobs and job agents	252
29.2.13. Biomateria LIMS	256
29.2.14. Array LIMS - plates	258
29.2.15. Array LIMS - arrays	260
29.2.16. Hybridizations and raw data	262
29.2.17. Experiments and analysis	264
29.2.18. Other classes	268
29.3. The Core API	269
29.3.1. Using files to store data	269
29.3.2. Sending signals (to plug-ins)	274
29.4. The Query API	276
29.5. Analysis and the Dynamic and Batch API:s	276
29.6. Extensions API	276
29.6.1. The core part	276
29.6.2. The web client part	278
29.7. Other useful classes and methods	281
30. Write documentation	282
30.1. User, administrator and developer documentation with Docbook	282
30.1.1. Documentation layout	282
30.1.2. Getting started	282
30.1.3. Docbook tags to use	287
30.2. Create UML diagrams with MagicDraw	293
30.2.1. Organisation	293
30.2.2. Classes	294
30.2.3. Diagrams	299
30.3. Javadoc	299
30.3.1. Writing Javadoc	300
31. Core developer reference	302
31.1. Publishing a new release	302
31.2. Subversion / building BASE	302
31.3. Coding rules and guidelines	302
31.3.1. Development process and other important procedures	302
31.3.2. General coding style guidelines	303
31.3.3. API changes and backwards compatibility	303
31.3.4. Data-layer rules	304
31.3.5. Item-class rules	318
31.3.6. Batch-class rules	318
31.3.7. Test-class rules	318
31.4. Internals of the Core API	318
31.4.1. Authentication and sessions	319
31.4.2. Access permissions	319
31.4.3. Data validation	319
31.4.4. Transaction handling	319
31.4.5. Create/read/write/delete operations	319
31.4.6. Batch operations	319
31.4.7. Quota	319
31.4.8. Plugin execution / job queue	319
V. FAQ	320
32. Frequently Asked Questions with answers	321
32.1. Reporter related questions with answers	321
32.2. Array design related questions with answers	321
32.3. Biomaterial, Protocol, Hardware, Software related questions with answers	322

32.4. Data Files and Raw Data related questions with answers	323
32.5. Data Deposition to Public Repositories related questions with answers	324
32.6. Analysis related questions with answers	325
VI. Appendix	326
A. Menu guide	327
B. Core plug-ins shipped with BASE	331
B.1. Core analysis plug-ins	331
B.2. Core export plug-ins	332
B.3. Core import plug-ins	332
B.3.1. Core batch import plug-ins	334
B.4. Core intensity plug-ins	334
B.5. Uncategorized core plug-ins	335
C. base.config reference	336
D. extended-properties.xml reference	342
E. Platforms and raw-data-types.xml reference	346
E.1. Default platforms/variants installed with BASE	346
E.2. raw-data-types.xml reference	346
F. web.xml reference	350
G. jobagent.properties reference	351
H. jobagent.sh reference	354
I. migrate.properties reference	356
I.1. mysql-migration-queries.sql	356
J. Other configuration files	357
J.1. mysql-queries.xml and postgres-queries.xml	357
J.2. log4.properties	357
J.2.1. Migration logger	357
J.3. hibernate.cfg.xml	357
J.4. ehcache.xml	357
K. API changes that may affect backwards compatibility	358
K.1. BASE 2.13 release	358
K.2. BASE 2.12 release	359
K.3. BASE 2.11 release	360
K.4. BASE 2.10 release	360
K.5. BASE 2.9 release	360
K.6. BASE 2.7.1 release	360
K.7. BASE 2.7 release	361
K.8. BASE 2.6 release	361
K.9. BASE 2.5 release	362
K.10. BASE 2.4 release	363
K.11. BASE 2.3 release	364
K.12. BASE 2.2 release	364
L. Things to consider when updating an existing BASE installation	365
L.1. BASE 2.9 release	365
L.2. BASE 2.7.2 release	365
L.3. BASE 2.7 release	366
L.4. BASE 2.4.4 release	366

Part I. Overview

This document is not finished but the most important chapters are concluded. Until the overview is written there is only three pointers here; i) Early in your BASE experience you should get acquainted with the permission and projects system (Chapter 7, *Projects and the permission system* (page 37)) since this will be helpful when you get going, ii) see Chapter 4, *Resources* (page 6) for further BASE information resources, and iii) read this document.

Chapter 1. Why use BASE

1.1.

Chapter 2. BASE features

This chapter will explain the important features of BASE.

2.1. Supported array platforms and raw data formats

BASE supports many different vendor specific and custom printing microarray platforms and data formats, there are even users that use BASE for protein arrays. For 2 channel array platforms it is straightforward to customize 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 2 channel platform. For Affymetrix, BASE comes with a specific Affymetrix platform and Illumina can be supported by customizing 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 customize his local BASE installation for their specific need. What comes pre-configured when BASE is installed is indicated in the lists below.

2.1.1. 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 analyzed. 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.

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 plug-in¹ available for seamless integration of the Illumina array platform to BASE.

ImaGene

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

¹ <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.illumina>

Unlisted

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

2.1.2. 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

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-in`² that adds Illumina array platform support to BASE.

ImaGene

QuantArray Biotin

QuantArray Cy

SpotFinder

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

Chapter 3. Overview of BASE

3.1.

Chapter 4. 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.

4.1. 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.

4.1.1. 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.

4.1.2. 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.

4.1.3. 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 2.x** milestone is used to collected tickets that we think should be fixed but are less important or require too much work. Contributions are welcome.

4.1.4. 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.

4.2. Core plug-in configurations

In the section called **Plug-ins** on the home page, is a link to a page that lists the core plug-ins⁴. All plug-ins that are included in the installation of BASE are listed on this page, some together with one or several examples of suitable configuration files. These configuration files can be downloaded and imported with the plug-in configuration importer in BASE.

4.3. BASE plug-ins site

Plug-ins which are not included in the installation of BASE, have their own site, called BASE plug-ins web site⁵ 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.

4.4. 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/>⁶

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).

⁴ http://base.thep.lu.se/chrome/site/doc/historical/admin/plugin_configuration/coreplugins.html

⁵ <http://baseplugins.thep.lu.se/>

⁶ <http://base2.thep.lu.se/demo/>

4.5. Mailing lists

BASE project has three different mailing lists available for subscription. Visit the mailing list page⁷ 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.

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

4.6. 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 site⁸ you will find a list of modified packages and information about the changes performed.

⁷ <http://base.thep.lu.se/wiki/MailingLists>

⁸ <http://dev.thep.lu.se/basehacks>

Part II. User documentation

Chapter 5. 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.

5.1. 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 6.1, "Introduction" (page 13) , 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 6.2, "Configuring your account"(page 16) , 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 6.3, "Working with items"(page 19) and Section 6.4, "Listing items" (page 24) , 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.

5.2. 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 7, *Projects and the permission system* (page 37) .

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 8, *File management* (page 43) gives you information about the server's file system and how to upload the files.

Chapter 19, *Import of data* (page 108) explains how the import is done and Chapter 20, *Export of data* (page 116) 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.

5.2.1. Get going

This description will guide you from the initiating tasks of creating the first account to running an analysis plug-in. Most of the steps below ends with a reference to somewhere in the documentation where more information can be found.

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 24, *Account administration* (page 152) .

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 19, *Import of data* (page 108)

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 `active project` . This should be done before any other items are created. Section 7.2, “Projects” (page 38) tell you more how working in project 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`) . More information see Section 18.2, “Raw bioassays” (page 99)
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 18.3, “Experiments” (page 103)
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 18.4, “Analysing data within BASE”(page 106) 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 6. Using the web client

6.1. Introduction

6.1.1. 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:8080/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. There is a checkbox which allows you to **encrypt the password** before it is sent to the BASE server. It is checked by default, and it is a good idea to leave it checked unless you have problems logging in. If you are sure you are entering the correct login and password, but still cannot log in, try unchecking the encryption option. If the checkbox is not visible, which happens if the server is using an external authentication server, the password is not encrypted.

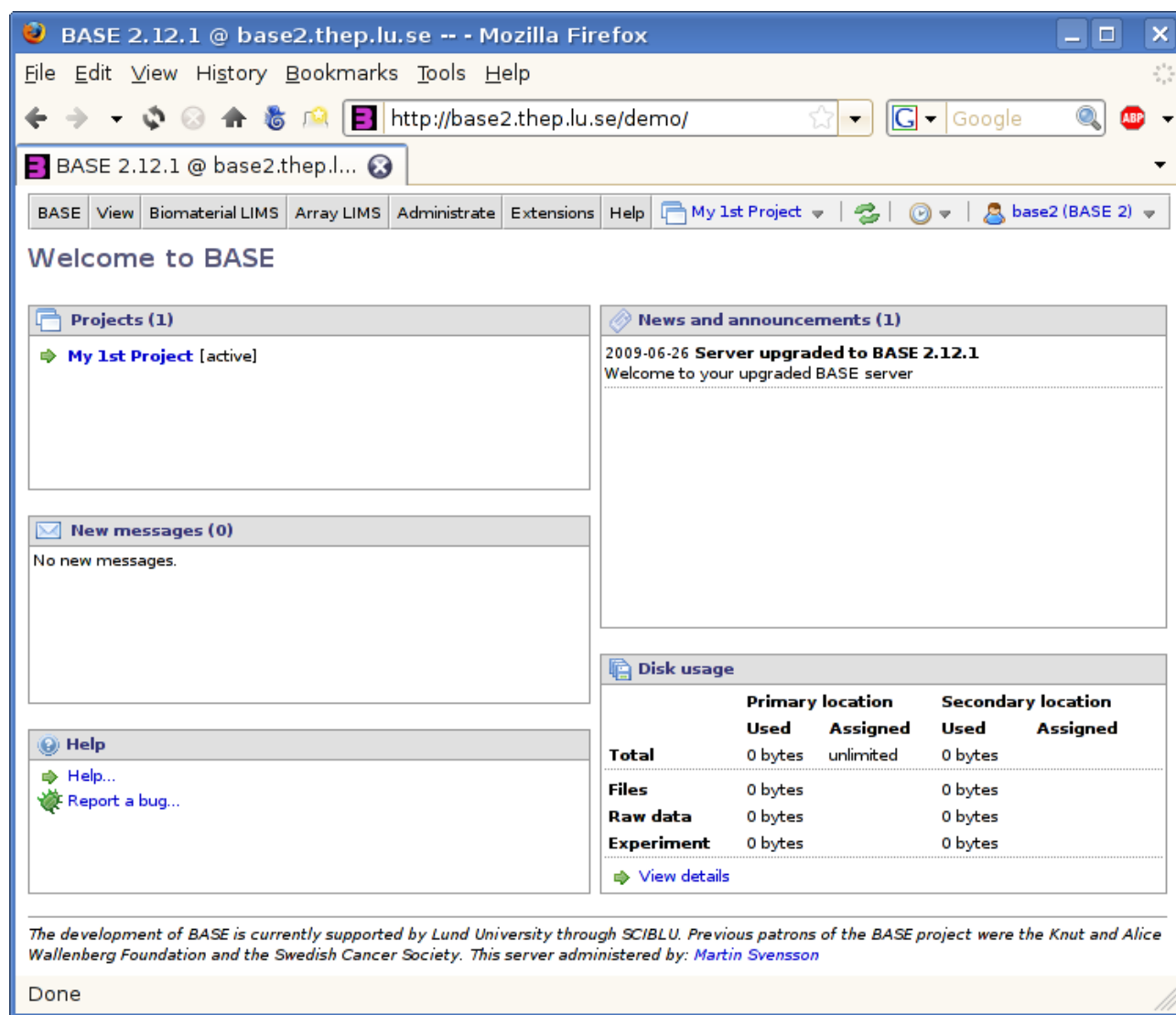
6.1.2. Forgotten password

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.

6.1.3. 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

Figure 6.1. The home page

New messages

Messages are 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 7, *Projects and the permission system* (page 37).

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.

6.1.4. 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:



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 19) for configuration information.



Projects

A list of all projects you are a member of. Selecting a project in the list will make that project the active project. The list can display a maximum of 25 projects. If you are a member of more projects, the last menu entry will take you to the complete list of projects.

Tip

The sort order in the menu 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.




Logged in user

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

6.1.5. 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 4, *Resources* (page 6).

6.2. Configuring your account

6.2.1. 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).

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.

6.2.2. 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.

6.2.3. 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.

6.2.4. Preferences

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

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.
- **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-black-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
- ss: two-digit second

Decimals

The number of decimals to display for numeric values. The default is 2.

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

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

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.

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.

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 it's 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.

6.3. Working with items

No matter what you are doing in BASE some things works 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 6.4, “Listing items” (page 24) for more information.

Single-item view

Displays information about a single item. Sometimes it is very little, sometimes 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.

6.3.1. 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 Appearance tab”(page 17) 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.

6.3.2. 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.

6.3.3. 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 6.5, “Trashcan” (page 31).
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.

6.3.4. 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 another section (Section 6.5, “Trashcan” (page 31)).

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.

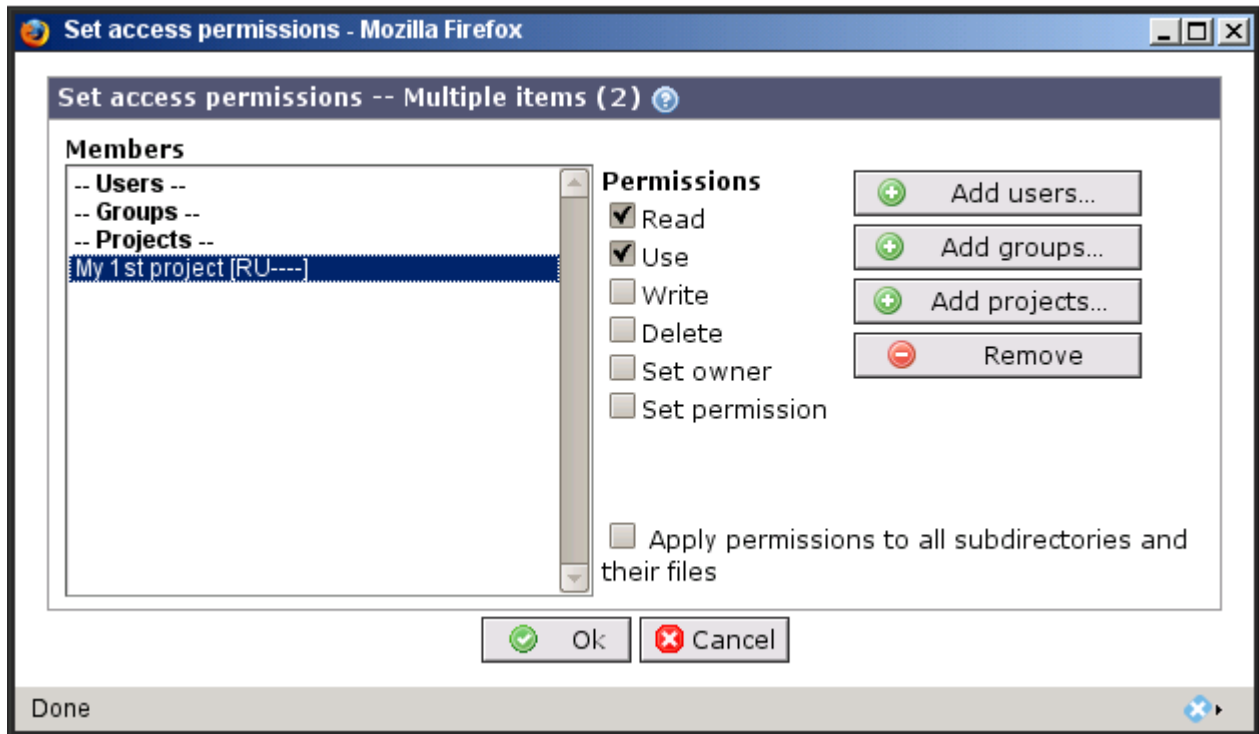
6.3.5. Share items to other users

Sharing data with other users is an important feature of BASE, which allows you cooperate in teams. If you follow the instructions in Chapter 7, *Projects and the permission system*(page 37) 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 to 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 6.2. 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.

Permissions

When you select a user, group or project in the list, the checkboxes will display the current permission. 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.

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 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.

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.

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.

6.3.6. Change owner of items

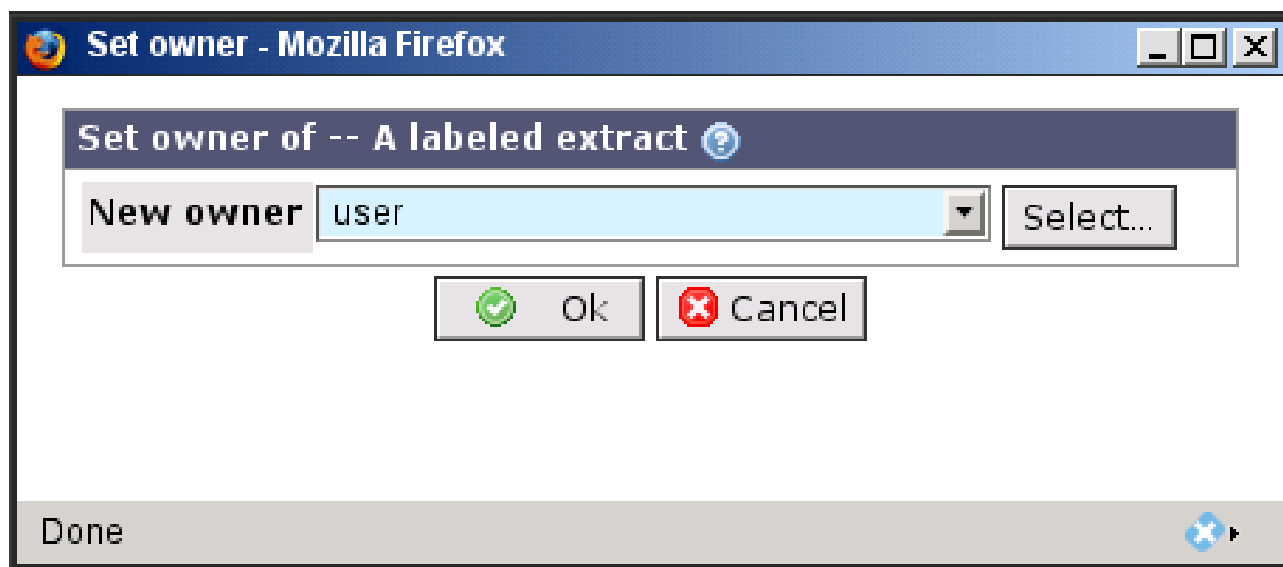
Sometimes it may be necessary to change the owner of an item. This can be done by everyone with *Set owner* permission on the item. For a user to have the rights to change owner of an item, the item must either be owned by or shared with *Set owner* permission to the user. See Section 6.3.5, "Share items to other users" (page 21).

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 6.3. Select a new owner**Note**

The original owner may not have access permissions to the items any longer. If that is desired, the new owner must share the items to the original owner.

6.3.7. Export items

This has a chapter of it's own. See Chapter 20, *Export of data* (page 116).

6.4. 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 (type, name and description) since these are the only properties that are common among all items.
- The list cannot be filtered (except by item type) or sorted. 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 6.4. A typical list page

Experiments

<div> New... Delete Restore Share... Take ownership... Columns... Export... </div>					❶
<div> </div> 1 (3 hits, 30 per page)					❷
<div> <div>- view / presets - </div> <div> <div>Name ▾</div> <div>Raw data type</div> <div>Description</div> <div>Actions</div> </div> </div>					
<div> <div></div> <div> <div></div> <div></div> <div></div> </div> </div>					
<div> <div>1</div> <div><input type="checkbox"/></div> <div>Affy experiment</div> <div>Affymetrix</div> <div></div> <div>Analyze</div> </div>					
<div> <div>2</div> <div><input checked="" type="checkbox"/></div> <div>Experiment A</div> <div>GenePix</div> <div></div> <div>Analyze</div> </div>					
<div> <div>3</div> <div><input type="checkbox"/></div> <div>Experiment B</div> <div>GenePix</div> <div></div> <div>Analyze</div> </div>					
<div> </div> 1 (3 hits, 30 per page)					❸

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.

6.4.1. 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.

6.4.2. 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.

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

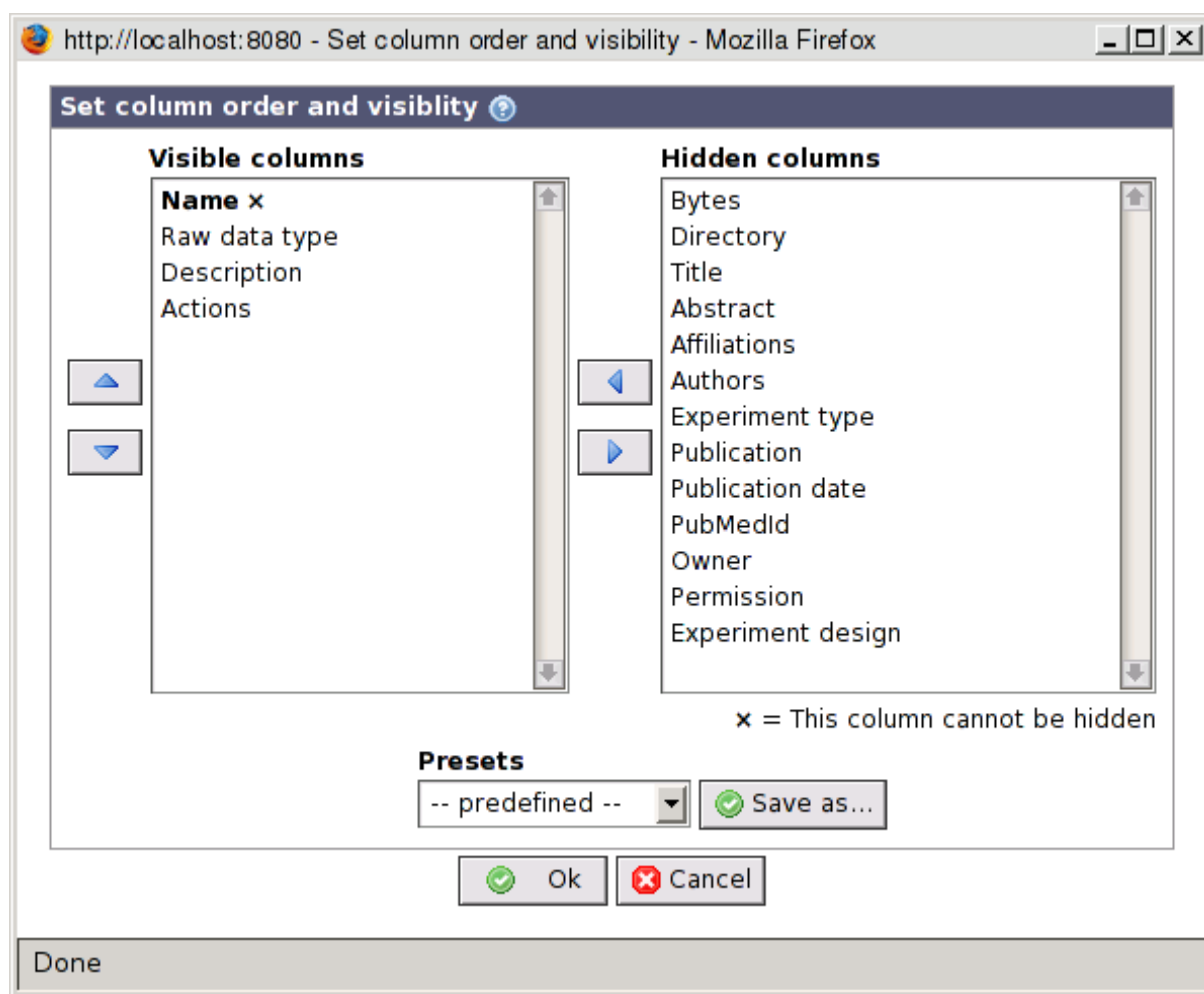
Some units contains hard-to-type characters. For example, the greek letter µ in µg, and mℓ and mℓ for areas and volumes. In all those cases it is also possible to use ug, m2 and m3, respectively.

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*.

6.4.3. 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 6.5. 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.

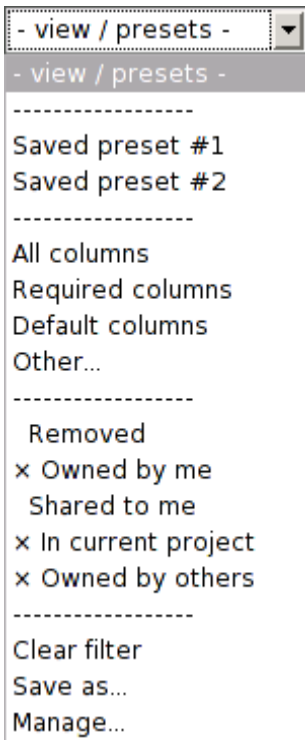
Presets

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 does not affect filters, sort order, etc.

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

6.4.4. Presets

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

Figure 6.6. The view / presets dropdown

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 6.4.3, “Configuring which columns to show” (page 27).

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 6.7. Save preset as

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

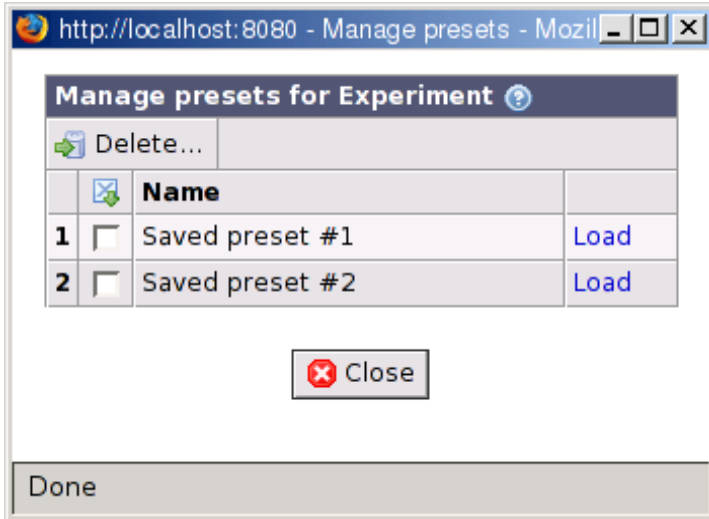
This options is only available for users which has the `SHARE_TO_EVERYONE` permission. If checked the preset is visible to all users.

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 6.8. 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.

6.5. 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 6.3.4, "Restore deleted items"(page 21) and Section 6.3.2, "Edit an existing item" (page 20) . If an item is restored, it will of course disappear from the trashcan.

6.5.1. 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 6.5.2, “View dependencies of a trashed item” (page 32) .

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.

6.5.2. 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 6.9. 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

1

2

☐
☐

Labeled extract A.ref

Labeled extract A.ref (dye-swap)

Labeled extract

Labeled extract

1. This icon indicates that the item cannot be deleted permanently cause of some dependencies, see #3 (page 32).
2. Common properties for all removable items.
3. A list of other items that are using the current item.

32

6.6. Item overview

With the **Item overview** function you can get an overview of all hybridizations, 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 6.10. The item overview

Experiments ▶ Experiment A

The screenshot shows the 'Overview' tab of the web client. The left pane displays a tree structure of 'Experiment A' with sub-items like 'Raw bioassays (4)', 'Scan: Scan A.00h', 'Hybridization: Hybridization A.00h', 'Labeled extracts (2)', 'Sample: Sample A.00h' (selected), and 'Experimental factors (3)'. The right pane is divided into three sections: 1) Item metadata (Path, Sample, Errors (children), Warnings (children), Description), 2) Annotations & protocol parameters (Time: 0 h), and 3) Failure details (1 error(s), 1 warning(s)). The failure details section lists: 1. Missing value for parameter: Temperature (Sample: Sample A.00h) and 2. Missing biosource (Sample: Sample A.ref).

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.

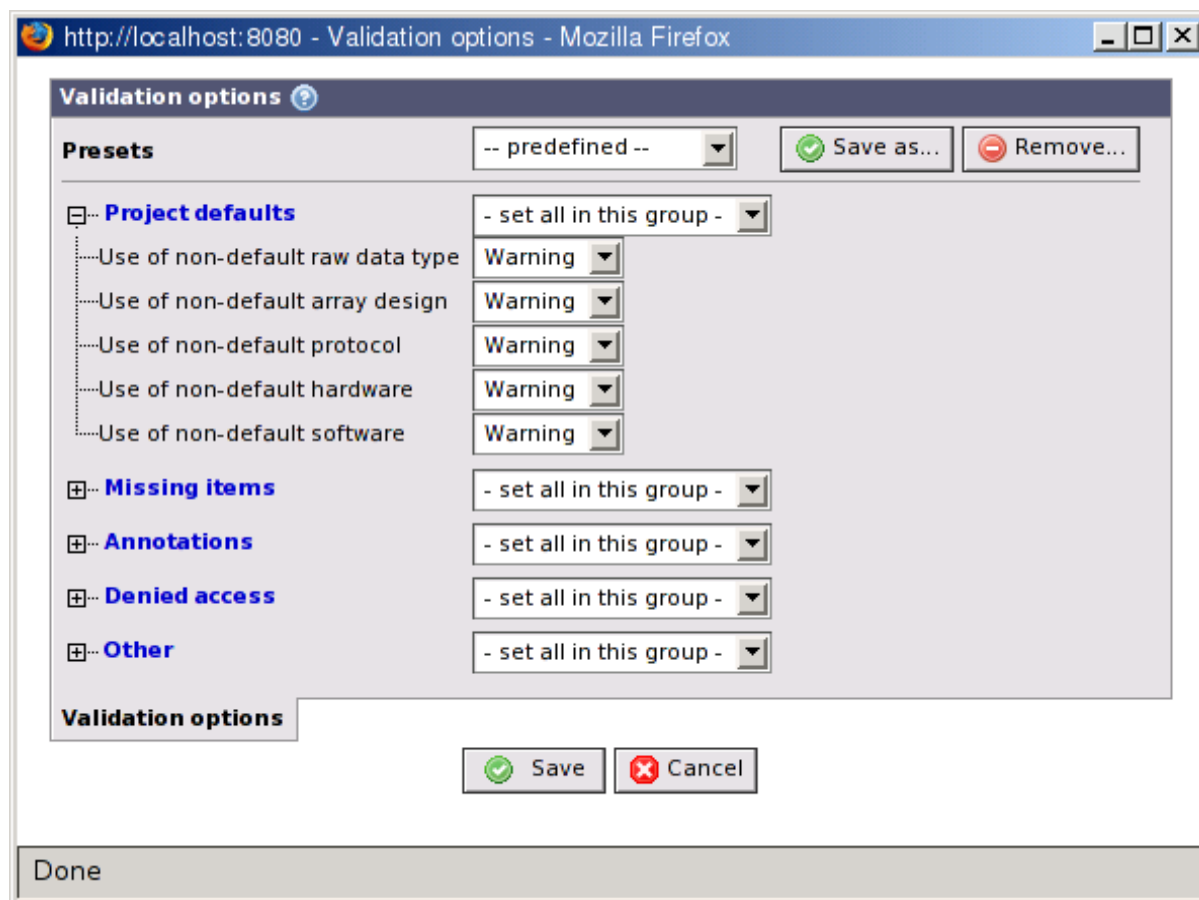
No links?

If you do not have permission to change things no links will be shown.

6.6.1. Validation options

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

Figure 6.11. Validation options



The validation procedure is highly configurable and you can select what you want to ignore, or something 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 7.2, “Projects” (page 38)). If no project is active or if the active project does not have default values these options are ignored.

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.

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.

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.

Other

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

- *Array deign mismatch*: Checks if the array design specified for a raw bioassay is the same array design specified for the hybridization.
- *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.
- *Incorrect number of labeled extracts*: Checks if the number of labeled extracts match the number of channels for the experiment. This is only checked when the root item is an experiment.
- *Non-unique name*: Checks if two items of the same type have the same name. A unique name if important when exporting data in Tab2Mage format.
- *Circular reference to pooled item*: If you have used pooling, checks that no circular references have been created.

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

6.6.2. 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.

Chapter 7. Projects and the permission system

7.1. 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 an 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.

7.1.1. 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 6.3.6, “Change owner of items” (page 23)), 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

An 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.

7.1.2. 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 can prevents an 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.
4. Check if you have been granted access to the item by the sharing system (cf. Section 6.3.5, “Share items to other users” (page 21)). 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.
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.

7.1.3. 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 has. 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 can give the plug-in the same permissions as listed above. There is one additional twist to the plug-in permission system. A permission can be granted regardless of if the user that invoked the plug-in had the permission or not, or a permission can be granted only if the user also has the permission. The first case makes it possible to develop a plug-in that allows users to do things that they normally do not have permission to do. The second case is the same as not using the plug-in permission system, except that unspecified permissions are always denied when the plug-in permission system is used.

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 22.2, “Plug-in permissions” (page 137) for more information.

7.2. Projects

Projects are an important part of 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 added to the active project so 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 6.4.4, “Presets” (page 28)).
- 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.

7.2.1. 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 included 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 drop-down menu can display a maximum of 25 projects. If you are a member of more projects, the last menu entry will take you to the complete list of projects.

Tip

The sort order in the menu 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.

Warning

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

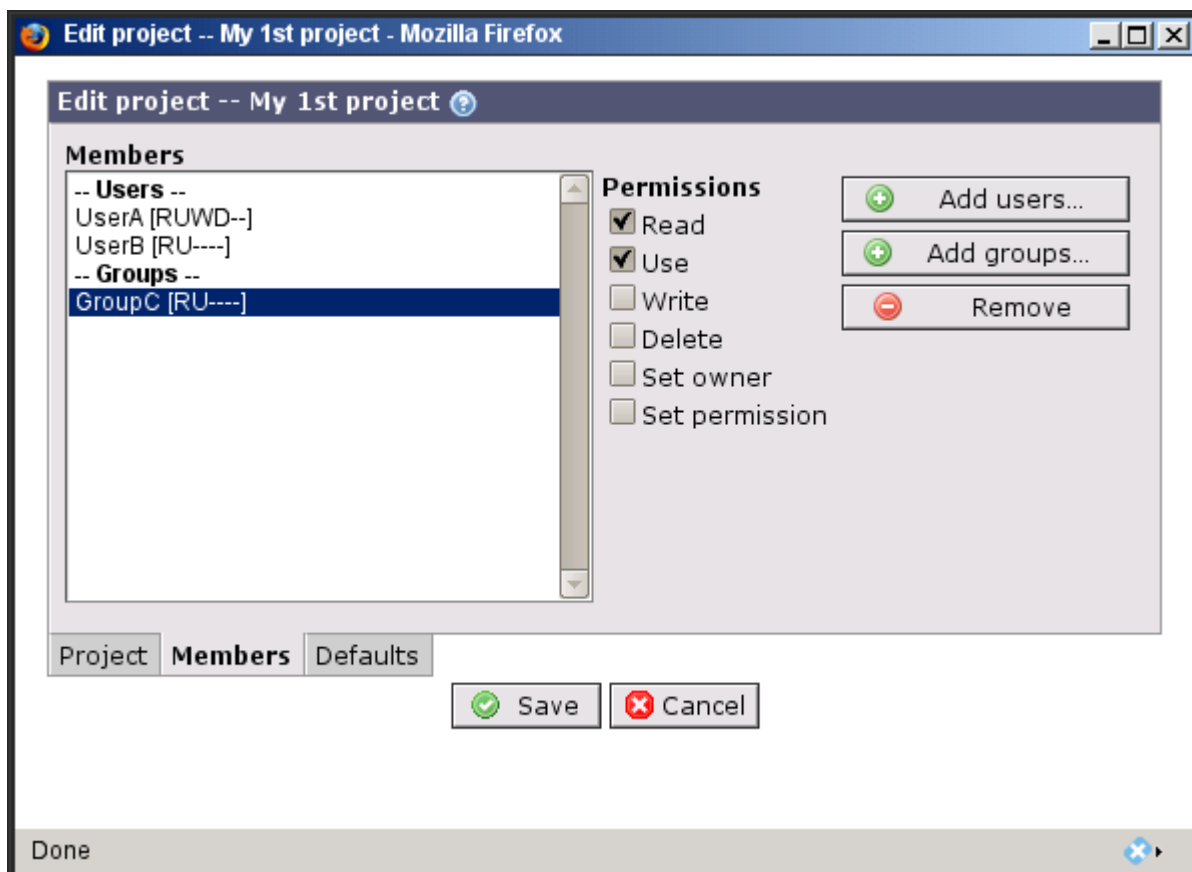
When a project is active all new items you create are automatically shared to this project. By default the permission are set to *read*, *use*, *write* and *delete*. It is possible to change the default permission level by modifying the settings for the project. Simply open the edit-view page for the project and select the permissions you want and save. From now on, all new items will be shared with the specified permissions.

7.2.2. 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 7.1. 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 an 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.

Note

The above note is not always true since the permission to an item in the project also depends on the permission that was set when adding the item to the project. The default permission is *delete* and the above note holds true. If the item's permission is manually changed to for example, *use*, no project member can get higher permission to the item.

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 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.

7.2.3. 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 7.2.



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 (name, description and owner) since these are the only properties that are common among all items.
- The list cannot be filtered (except by item type) or 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.

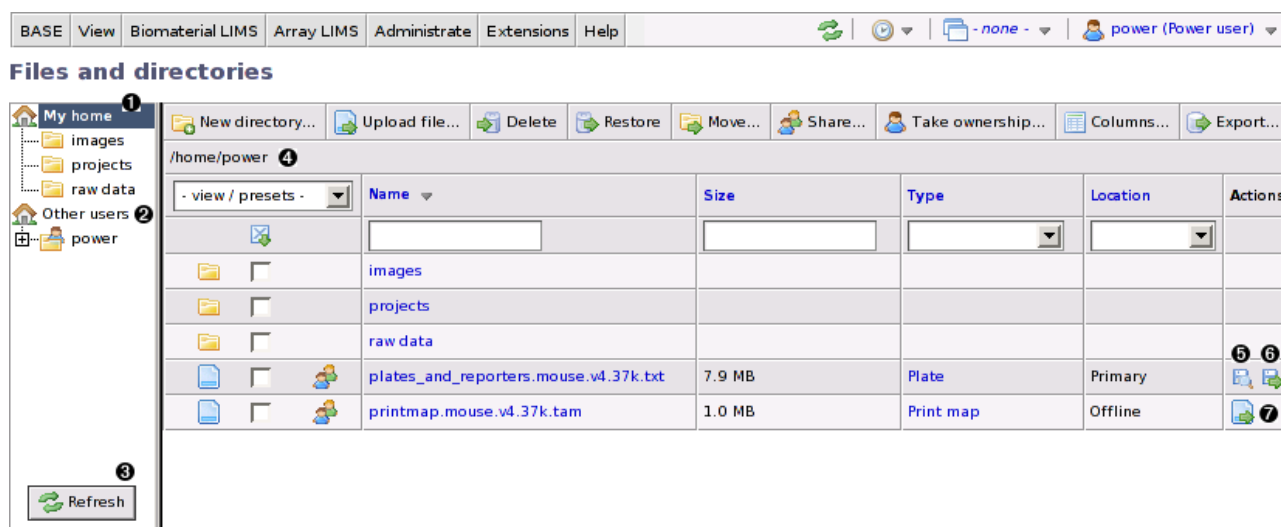
If you have more than one old project, repeat the above procedure.

Chapter 8. File management

8.1. 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 8.1. 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. More about this can be read in the section called "Home directory - 'My home'"(page 44). 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 (no plus sign = no sub directories).

2. Other users

The other users' home directories that the current user has permission to read are listed here, including his/her own.

3. Refresh button

This button, which is located at the bottom of the navigation tree, refreshes/updates the directory tree. It can for example be used to get a recently created directory to appear in the tree.

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 it 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.

8.1.1. Browse the file system

Browsing the BASE file system is done from the navigation bar by clicking on a directory in the tree to view it's contents. Both sub directories and files in the selected directory are showed. A directory with sub directories can be expand with a click on the plus-sign.

Browse a directory

A directory can only be open from the navigation tree and never from the list. A click on a directory's name in the list will open the directory's edit window.

Navigation tree

The navigation tree contains of folders/directories that the logged in user has permission to read. At the top is the current user's home directory and under it is a folder with all accessible home directories.

The tree can be updated with the refresh-button at the bottom of the panel (the tree is not self-refreshing).

Home directory - 'My home'

To make it easier for the logged in user to find his/her directory without having to scroll through the whole list of home directories, his or hers home directory is located at the top of the navigation tree under the My home folder.

No home directory

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

8.1.2. 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 6.1.3, "The home page" (page 13).

See Section 24.4, "Disk space/quota" (page 158) for more information about the quota system.

8.2. Handling files

8.2.1. 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 8.2. Upload new file

Upload new file ?

Directory /home/user

File [Browse...](#)

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 information

File

[Upload](#) [Cancel](#)

Done

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.

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 it's 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 it's normal format.

Type

This is the file-type that the uploaded file should get. The file-types to choose between from the drop-down list are described in Section 8.3, “File types”(page 50). Select – **none** –if the file should not be associated with any file type.

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 26.6.3, “File unpacker plug-ins” (page 200) 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 8.1, “The file page”(page 43). 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.

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

8.2.2. 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 8.2.3, “Move files” (page 47).

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.

8.2.3. 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.

8.2.4. 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.

8.2.5. 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.

Edit a directory

The window to edit a directory's properties is opened either by clicking on the directory's name in the list or when creating a new directory.

Properties

Path

This property is read-only in the edit window but can be changed by moving the directory, described in Section 8.2.3, "Move files" (page 47).

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 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.

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.

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.

8.3. File types

A file can be associated with one of the file types that exists in BASE. File types make it possible to filter the files depending of what kind of file it is. Here is a list of file types that are defined in BASE. [Administrative](#) [Types](#) [File types](#)

File types

Image

Indicates that the file is an image file.

Plate

The associated file is a file with information about plate/plates.

Plate mapping

Files with information about plate mapping.

Print map

Print map files

Protocol

Files with protocol information

Raw data

Raw data files

Reporter

Files with information about reporters

Reporter map

Files of this type contain information about how the reporters are mapped.

Spot images

A zip-file containing generated spot images in JPEG format.

Chapter 9. Jobs

Job is a configured task performed by a plug-in.

Creating/configuring a job is made from a context, which is supported by the plug-in you want to use. If no particularly context is needed, the job configuration is started from the plug-in's single item view.

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 estimated execution time.

9.1. Properties

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 use, those parameters are also listed here.

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.

[**Description**]

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

Type

The type of job, which is depending on the plug-in that is used. It can be one of these five:

- Export
- Import
- Analysis
- Intensity
- Other

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* - Job is waiting in the job queue.
- *Executing* - Job is being executed.
- *Done* - Indicates that the job is finished.

- *Error* - The job was aborted cause of some error.
- *Preparing* - Doing some preparing tasks befor executing the job.
- *Aborting* - Job has received an abort signal and tries to abort the work.

Status message

A message, given by the plug-in, with more information about the status.

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 registerd in the database.

Started

Date and time when execution of the job started.

Ended

Date and time when the job stopped running. Either cause of 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 24.3.2, "Edit role" (page 156).

User/Owner

The user who created/configured the job.

Experiment

Name of the experiment which the job was configured within.

Plugin

The plug-in to use in the job.

Configuration

Name of the plug-in configuration that is used.

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

Refresh

Update the page with the information. This button is available as long as the job has not finished. This button is only for the impatient since the page will automatically refresh every ten seconds anyway.

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 eecuting 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

Retry a failed job. 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.

Close

Close the window.

Chapter 10. Reporters

10.1. Introduction

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.

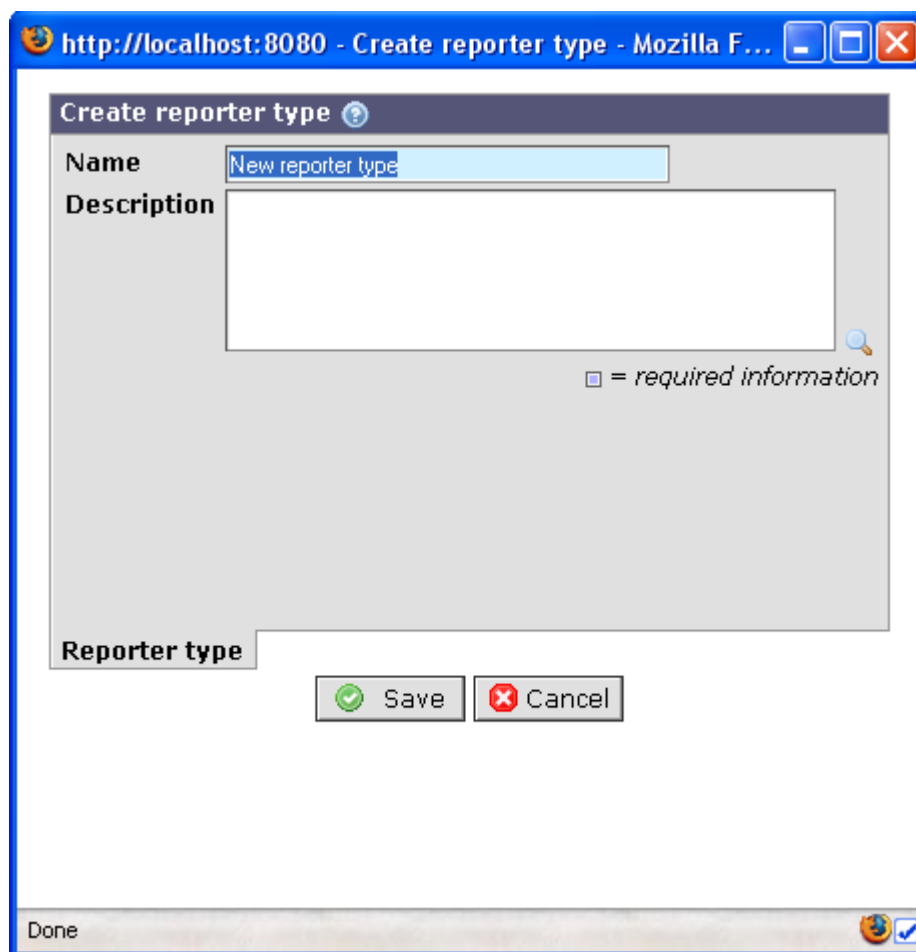
In BASE, reporters also refer to Affymetrix Probeset ID but reporters can be used to describe genes, transcripts, exons or any other sequence entity of biological relevance.

10.2. 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 [Administrate > Types > Reporter types](#).

Figure 10.1. Reporter type properties



The screenshot shows a web browser window with the address bar displaying 'http://localhost:8080 - Create reporter type - Mozilla F...'. The main content area is a form titled 'Create reporter type' with a help icon. The form has two input fields: 'Name' with the text 'New reporter type' and 'Description' which is empty. A legend indicates that a square icon represents 'required information'. At the bottom of the form, there is a 'Reporter type' label and two buttons: 'Save' (with a green checkmark icon) and 'Cancel' (with a red X icon). The browser's status bar at the bottom shows 'Done' and a small 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.

Description

A description of the reporter type.

10.3. Reporters

Go to View Reporter to view and manage the reporters.

10.3.1. 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 19, *Import of data*(page 108) for more information about importing and Section 22.4, “Plug-in configurations” (page 140) for more information about configuring file formats.

Exchanging plug-in configurations

As for any other plug-in, configuration parameters can be saved as an XML file and exchanged with another BASE instance, thereby reducing configuration burden (provided the two instances have identical `extended-properties.xml` files). See Section 4.2, “Core plug-in configurations” (page 7) for information about available configuration files.

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. The CDF file contains no annotation information and consequently annotation information is removed if the CDF used to upgrade annotations.

10.3.2. 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 6.3, “Working with items” (page 19).

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

Reporter properties

Figure 10.2. Reporter properties

http://localhost:8080 - Edit reporter -- (clone PWHLC2-24) myosin ...

Edit reporter -- (clone PWHLC2-24) myosin light chain 2

Name: (clone PWHLC2-24) myosin light chain 2

External ID: AI005197

Type: - none -

Gene symbol:

Description:

□ = required information

Reporter | Extended properties

Save Cancel

Done

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 use 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.

Extended properties

Figure 10.3. Extended reporter properties

The screenshot shows a web browser window titled "http://localhost:8080 - Edit reporter -- (clone PWHLC2-24) myosin lig...". The main content area is titled "Edit reporter -- (clone PWHLC2-24) myosin light chain 2". It contains a form with the following fields:

- Species**: A text input field.
- Cluster ID**: A text input field.
- Length**: A text input field.
- Sequence**: A large text area.
- Vector**: A text input field with a magnifying glass icon to its right.
- Tissue**: A text input field with a magnifying glass icon to its right.
- Library**: A text input field with a magnifying glass icon to its right.

At the bottom of the form, there are two tabs: "Reporter" and "Extended properties" (which is currently selected). Below the tabs are two buttons: "Save" (with a green checkmark icon) and "Cancel" (with a red X icon). The status bar at the very bottom says "Done" and has a small icon on the right.

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 21.3, "Installation instructions" (page 125) and Appendix D, *extended-properties.xml reference* (page 342) 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.

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

10.3.3. 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 6.5, “Trashcan” (page 31)). 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 select at most 99 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.

10.4. 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 two ways to do so:

1. Use the **New reporter list** button on the View Reporters page.
2. Use the **New** button on the View Reporter lists page.

Figure 10.4. The Create reporter list called from the reporters page.

The screenshot shows a web browser window titled 'http://localhost:8080 - Create reporter list - Mozilla Firefox'. The main content area is a form titled 'Create reporter list' with a help icon. The form contains the following fields and controls:

- Name:** A text input field containing 'New reporter list'.
- External ID:** An empty text input field.
- Which reporters?:** Three radio button options: 'Selected items', 'Current page', and 'All pages'. The 'All pages' option is selected.
- Description:** A large empty text area.
- Reporter list:** A tabbed interface with one visible tab labeled 'Reporter list'.
- Buttons:** 'Save' (with a green checkmark icon) and 'Cancel' (with a red X icon).
- Legend:** A small square icon followed by the text '= required information'.

The bottom status bar of the browser shows 'Done' and a small icon.

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.

Note

This option is only available when creating a reporter list from the View Reporters page, not when editing an existing list or creating it from the View Reporter lists page.

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.

Description

A description of the reporter list.

Chapter 11. Annotations

11.1. 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 *BioAssaysets*

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 8 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** 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 four different tabs.

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

11.1.1. Properties

Figure 11.1. Annotation type properties

The screenshot shows a web browser window with the address bar displaying 'http://localhost:8080 - Edit annotation type -- Temperature - Mozilla Fir'. The main content area is a form titled 'Edit annotation type -- Temperature'. The form contains the following fields and controls:

- Value type:** A dropdown menu set to 'Float'.
- Name:** A text input field containing 'Temperature'.
- External ID:** An empty text input field.
- Multiplicity:** A text input field containing '1', with a note '0 or empty = unlimited' to its right.
- Default value:** An empty text input field.
- Required for MIAME:** A checkbox that is checked.
- Protocol parameter:** A checkbox that is checked.
- Description:** A large, empty text area.
- Legend:** A small blue square icon followed by the text '= required information'.
- Tabs:** A row of five tabs: 'Annotation type' (selected), 'Options', 'Item types', 'Units', and 'Categories'.
- Buttons:** Two buttons at the bottom: 'Save' (with a green checkmark icon) and 'Cancel' (with a red X icon).
- Footer:** A 'Done' button in a separate bar at the very bottom of the browser window.

Name

The name of the annotation type. See Section 18.3.3, “Tab2Mage export” (page 105) if you are going to use this annotation type when exporting Tab2Mage files.

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.

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 13.2.2, “Protocol parameters” (page 75) for more information.

Description

A short textual description of the to clarify the usage.

11.1.2. Options

Figure 11.2. Annotation type options

The screenshot shows a web browser window with the address bar displaying 'http://localhost:8080 - Edit annotation type -- Temperature - Mozilla Fir'. The main content area is a dialog box titled 'Edit annotation type -- Temperature'. Inside the dialog, there are several sections: 'Interface' with three radio buttons (selected: text box, unselected: selection list, unselected: radiobuttons/checkboxes), 'Min value' with a text input containing '0' and the text 'empty = no limit', 'Max value' with a text input containing '100' and the text 'empty = no limit', 'Input box width' with a text input containing '40', and 'Values' with a large text area. Below the text area is the text 'One enumeration value per line'. At the bottom of the dialog, there are tabs for 'Annotation type', 'Options' (selected), 'Item types', 'Units', and 'Categories'. Below the tabs are 'Save' and 'Cancel' buttons. At the very bottom of the browser window, there is a 'Done' button.

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.

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

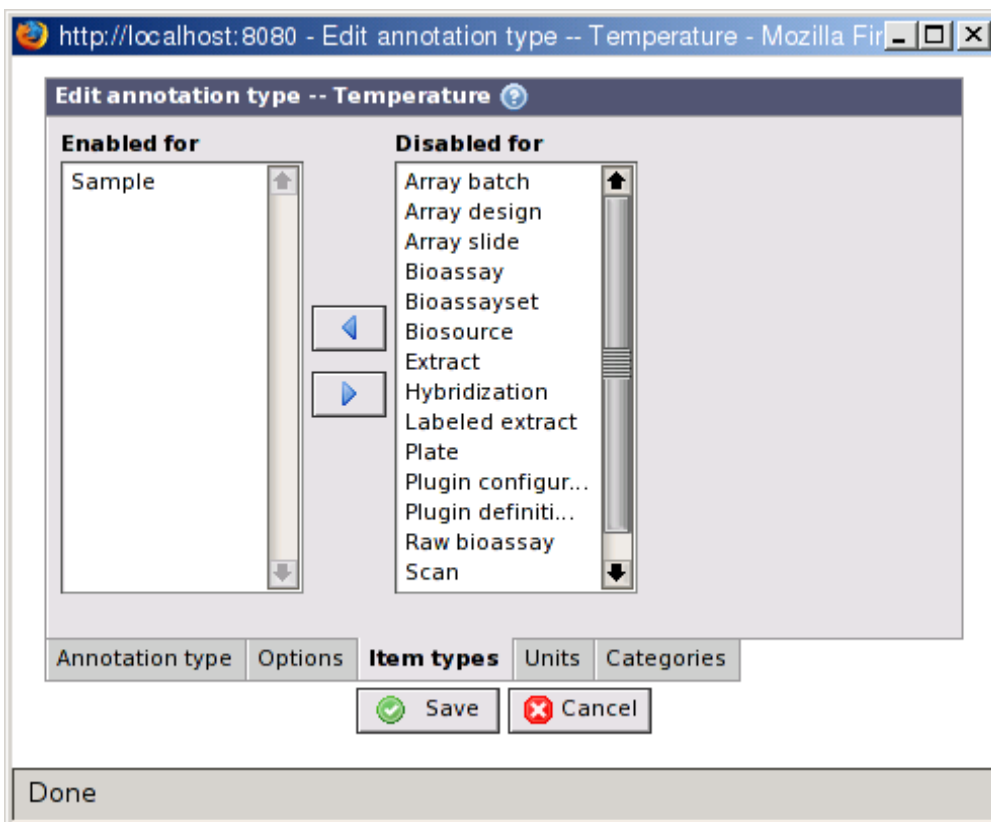
A suggested display width and height of the element used for input. These values are ignored in the current implementation.

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).

11.1.3. Item types

Figure 11.3. 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.

11.1.4. Units

Figure 11.4. 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.

11.1.5. 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.

11.2. Annotation type categories

Annotation Type Categories allow grouping of related Annotation Types, based on users requirements.

For managing categories go to [Administrate Types](#) [Annotation Type categories](#).

Example 11.1. Annotation category examples

- A category **Hematology Descriptors** could be created to group together annotation types such as **Lymphocyte count** and **Hematocrite**
- A category **Plasma Clinical Descriptors** could be created to group together annotation types such as **ALT activity (UI/mL)** and **LDH activity (UI/mL)**

11.3. Best Practices and Tab2Mage Export functionality

See Section 18.3.3, “Tab2Mage export” (page 105).

11.4. 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 11.5. Annotating a sample

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.

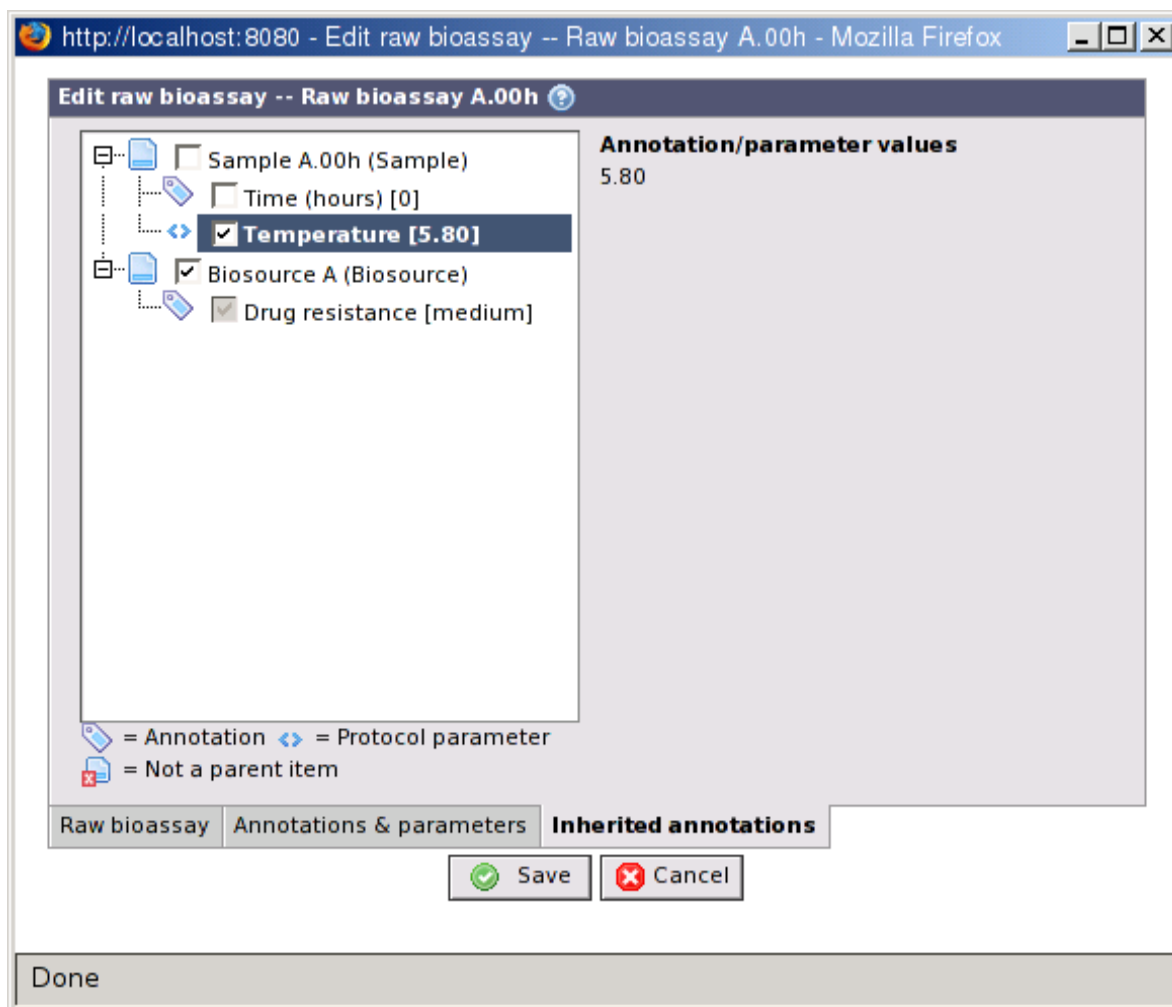
Annotation types marked with angle brackets (**<>**) are protocol parameters.

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.

11.4.1. 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 18.3.2, “Experimental factors” (page 104) for more information about experimental factors.

Figure 11.6. 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 6.6, “Item overview” (page 33).

11.4.2. 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. 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 12. Experimental platforms and data file types

12.1. 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 two platforms; A generic platform and the Affymetrix platform. Other platforms, such as Illumina, are available as non-core plug-in packages, see Section 4.3, “BASE plug-ins site” (page 7). An administrator may define additional platforms and file types.

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

Figure 12.1. Platform properties

http://localhost:8080 - Create platform - Mozilla Firefox

Create platform ?

Name

External ID

File-only ☒ no ☐ yes

Raw data type

Channels

Description

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

Platform

Done

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.

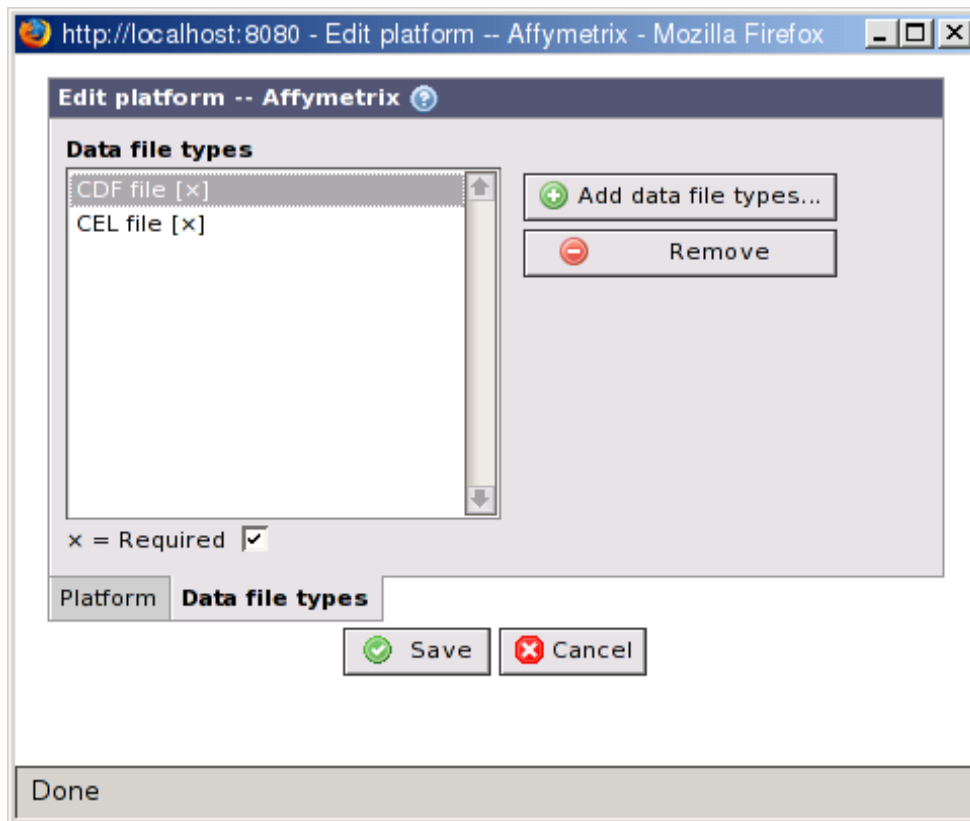
Channels

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 12.2. 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

This checkbox will modify the required flag for the selected file types.

Note

The required flag is not a hard requirement. It is only used for generating warnings when validating an experiment.

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.

12.1.1. 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.

12.2. 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 plug-ins. A data file type may specify which plug-in to use for validation and metadata extraction. Currently, BASE ships with plug-ins for CEL and CDF files.

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

Figure 12.3. Data file type properties

http://localhost:8080 - Create data file type - Mozilla Firefox

Create data file type ?

Name New data file type

External ID

Item type Array design

File extension

Generic file type - none -

Validator

Metadata reader

Description

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

Data file type

Save Cancel

Done

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).

Validator

The name of the Java class that can be used to validate if a given file is a valid file of this type. The class must implement the `DataFileValidator` interface. See Section 26.6.5, “File validator and metadata reader plug-ins” (page 203).

Metadata reader

The name of the Java class that can be used to extract metadata from a file of this type. The class must implement the `DataFileMetadataReader` interface. See Section 26.6.5, “File validator and metadata reader plug-ins” (page 203).

Description

A description of the file type.

Chapter 13. Protocols and protocol types

Information about laboratory standard operating procedure and protocols can be tracked in BASE using two structures, the use of which is detailed in the following sections.

13.1. Protocol types

Protocol Type allows classification of protocols based on their usage and purpose in the laboratory workflow. By default, BASE creates the 8 main Protocol types and those correspond to the main protocol families identified by MIAME requirements and applied in a canonical DNA microarray experiment meant for surveying gene expression. These 8 protocol types are namely **Printing**, **Sampling**, **Pooling**, **Extraction**, **Labeling**, **Hybridization**, **Scanning** and **Feature extraction**.

New applications of DNA microarray technology, for instance DNA binding site identification, imposes the creation of new protocol type in addition to those built-in in BASE.

Follow MIAME recommendations

It is advised to define the protocol type **Name** so that it is compatible with the MIAME requirements and recommendations issues by microarray data repositories.

You can manage the protocol types by going to [Administrate Types Protocol Type](#).

Figure 13.1. Protocol type properties

http://localhost:8080 - Edit protocol type -- Extraction - Mc

Edit protocol type -- Extraction

Name Extraction

Description Protocols used for creating extracts.

[] = required information

Protocol type

Save Cancel

Done

Name

The name of the protocol type.

Description

A description of the protocol type.

13.2. Protocols

In BASE, protocols can be created by two routes. Either from the single-item view of a protocol type or from the list view of protocols.

13.2.1. Protocol properties

Figure 13.2. Protocol properties

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.

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.

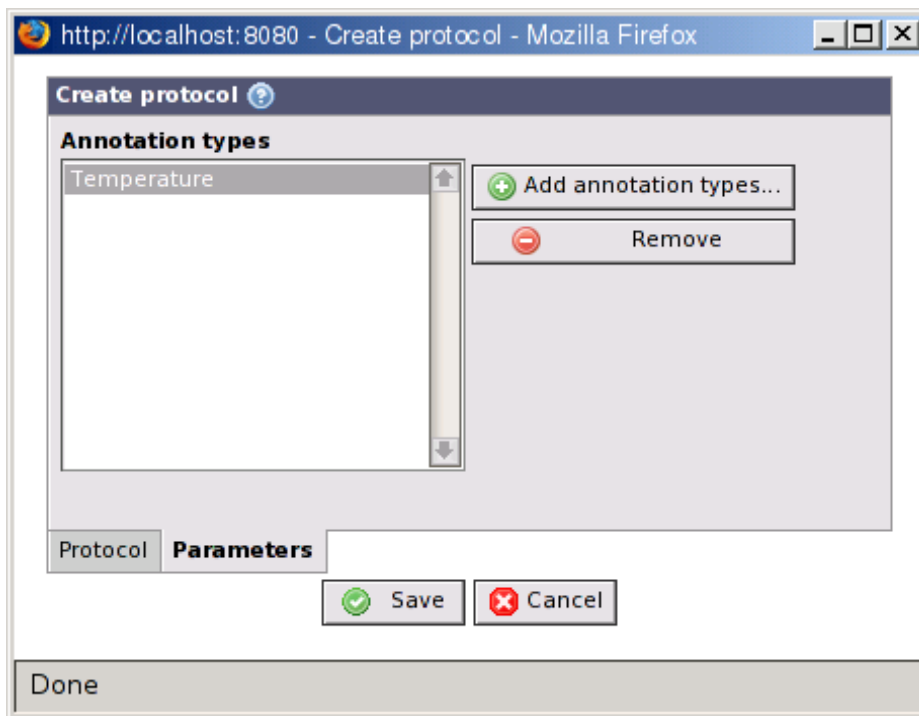
Description

A description of the protocol.

13.2.2. 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 11, *Annotations* (page 60).

Figure 13.3. 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.

Chapter 14. Hardware

Information about laboratory equipment can be tracked in BASE using two structures, the use of which is detailed in the following sections.

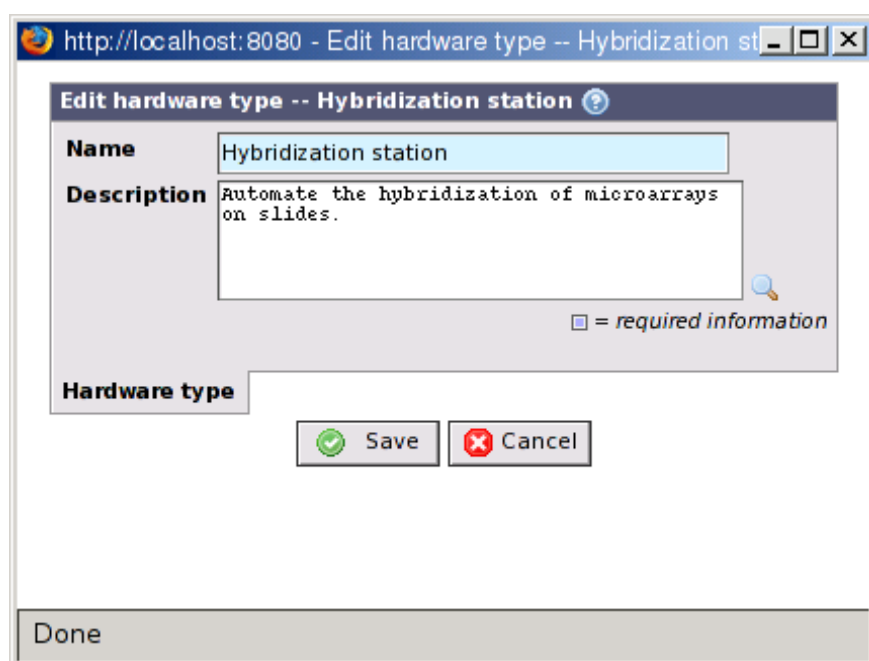
14.1. Hardware types

Hardware Type allows classification of hardware based on their usage and purpose in the laboratory workflow. By default, BASE creates the 3 main hardware types and those correspond to the main hardware families identified by MIAME requirements¹. These 3 hardware types are **Print Robot**, **Scanner**, and **Hybridization station**.

In case those built-in types are not enough, it is possible to create new ones.

You can manage the hardware types by going to `Administrate > Types > Hardware types`.

Figure 14.1. Hardware type properties



Name

The name of the hardware type.

Tip

It is advised to define the name so that it is compatible with the MIAME requirements and recommendations issues by microarray data repositories.

Description

A description of the hardware type.

14.2. Hardware

In BASE, hardware can be created by two routes. Either from the single-item view of a hardware type or from the list view of hardware.

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

Figure 14.2. Hardware properties

The screenshot shows a web browser window with the address bar displaying 'http://localhost:8080 - Create hardware - Mozilla Firefox'. The main content area contains a form titled 'Create hardware' with a help icon. The form has four labeled input fields: 'Name' (containing 'New hardware'), 'Version' (empty), 'Type' (a dropdown menu showing 'Hybridization station'), and 'Description' (a large empty text area). To the right of the 'Description' field is a blue square icon with a magnifying glass, and below it is the text '= required information'. At the bottom of the form are two buttons: 'Save' (with a green checkmark icon) and 'Cancel' (with a red X icon). Below the form is a 'Done' button. The browser window has standard minimize, maximize, and close buttons in the top right corner.

Name

The name of the hardware.

Version

The version number or model of the hardware.

Type

The hardware type of the hardware. The list may evolve depending on additions by the server administrator.

Description

A description of the hardware.

Chapter 15. Software

15.1. Software types

There is only one **Software Type** (Feature Extraction) in BASE and it is not possible to create new types. This is a difference compared to the way hardware can be managed.

15.2. Softwares

In BASE, software can be created by 2 routes. Either from the single-item view of a software type or from the list view of software.

Figure 15.1. Software properties

The screenshot shows a web browser window with the address bar displaying 'http://localhost:8080 - Edit software -- GenePix Pro - Mozilla Firefox'. The main content area is titled 'Edit software -- GenePix Pro' and contains a form with the following fields:

- Name:** A text input field containing 'GenePix Pro'.
- Version:** A text input field containing '6.0'.
- Type:** A dropdown menu with 'Feature extraction' selected.
- Description:** A large text area for entering a description.

Below the form fields, there is a tab labeled 'Software' and 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). A 'Done' button is located at the bottom left of the browser window.

Name

The name of the software.

Version

The version number of the software.

Type

The software type of the software.

Description

A description of the software.

Chapter 16. 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 events making up a microarray experiment is begun.

BASE is engineered to support microarray printing facilities. The system therefore offers objects to describe plates, their geometries, and the events (*e.g.*, merging and printing) affecting them. The first section of this chapter deals with this functionality as well as plate management. Users buying arrays from commercial sources can ignore the plate management component and immediately jump to Section 16.2, “Array designs” (page 80).

16.1. Plates

TODO

16.2. 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.

Tip

It's highly recommended to have read Section 6.3, “Working with items” (page 19) before continuing with this chapter.

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

16.2.1. Properties

Array design

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.

Description

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

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

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

Annotations and inherited annotations

This allows you to input values associated to annotation types devised to refine array design description. Read more about annotations in Chapter 11, *Annotations* (page 60).

16.2.2. 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 19, *Import of data* (page 108). 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.

16.3. Array batches

16.3.1. Creating 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 6.4.3, "Configuring which columns to show" (page 27)

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.

16.3.2. Properties

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

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

See Chapter 11, *Annotations* (page 60) for information about annotating items and inherit annotations.

How many slides each batch has can be found in the **Slides** column, list view page.

16.4. Array slides

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

16.4.1. 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 the section called “Multiple slides wizard” (page 83).

16.4.2. Properties

Figure 16.1. Create new array slide

The screenshot shows a web browser window titled "http://localhost:8080 - Create array slide - Mozilla Firefox". Inside the browser is a form titled "Create array slide". The form contains the following fields and controls:

- Name:** A text input field containing "New array slide".
- Barcode:** An empty text input field.
- Destroyed:** A checkbox that is currently unchecked.
- Array batch:** A dropdown menu with a "Select..." button next to it.
- Index:** A text input field containing "1".
- Description:** A large text area for entering a description.
- Legend:** A small blue square icon followed by the text "= required information".
- Tabs:** Three tabs are visible: "Array slide" (selected), "Annotations", and "Inherited annotations".
- Buttons:** "Save" (with a green checkmark icon) and "Cancel" (with a red X icon).

At the bottom of the browser window, the status bar shows the word "Done".

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.

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, both showed in the picture Figure 16.2, "Create multiple array slide" (page 84) .

1. The first step reminds alot of the normal edit window of an array slide, but there are some differences:

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.

Press **Next** to continue to next step of the wizard.

Important

Be sure to get everything right before proceeding, because it is not possible to go back from step 2.

- In step 2 should the barcodes for the array slides be entered, for those which have one. The autogenerated names of the slides can also be changed if that is wanted, as long as no name-field is left empty.

Figure 16.2. Create multiple array slide

The figure consists of two side-by-side screenshots of a web application window titled 'Create array slides'.

The left screenshot shows the initial form with the following fields:

- Name:** New slides.
- Array batch:** A dropdown menu with a 'Select...' button.
- Quantity:** 10 (with a note '(1-999)').
- Start at:** 1
- Pad size:** (empty)
- Description:** A large text area.
- Array slides:** A section with a 'Next' button and a 'Cancel' button.

The right screenshot shows the 'Names and barcodes' table with 10 rows of pre-filled names and empty barcode fields:

	Name	Barcode
1	New slides.01	
2	New slides.02	
3	New slides.03	
4	New slides.04	
5	New slides.05	
6	New slides.06	
7	New slides.07	
8	New slides.08	
9	New slides.09	
10	New slides.10	

Below the table are 'Save' and 'Cancel' buttons.

Fill in the necessary information as exhaustively as possible.

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

See Chapter 11, *Annotations* (page 60) for information about annotating items and inherit annotations.

Chapter 17. Biomaterial

17.1. Introduction

The generic term biomaterial refers to any biological material used in an experiment. Biomaterial is subdivided in 4 components, biosource, sample, extract and labeled extract. 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 with the realization of a microarray hybridization. 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.
- Labeled extracts correspond to nucleic acid materials which have undergone a marking procedure using a fluorescent or radioactive compound for detection in a microarray assay.

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.

Tip

It is highly recommended that you have read Section 6.3, “Working with items” (page 19) before continuing with this chapter.

17.2. Biosources

17.2.1. Properties

Biosource

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

Name

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

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.

Figure 17.1. Biosource properties

Annotations

This allows BASE users to use annotation types to refine biosource description. More about annotating items can be read in Section 11.4, “Annotating items” (page 65) .

17.3. 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.

17.3.1. 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 a sample is to click on  in the sample column of the biosource list view. There is also a corresponding button, **Create sample...** in the toolbar when viewing a single biosource.

from the sample list page

Pooled samples can also be created by first selecting the parents from the list of samples and then press **Pool...** in the toolbar.

17.3.2. Properties

Sample

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.

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.

Figure 17.2. Sample properties

Parents

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

- If the parent is a biosource the radio-button is set to **No** . This will make the biosource select button active, which allows users to point to a biosource from which the sample originates from.
- When the parent is one or several other samples the radio-button is set to **Yes** . Upon selection, the biosource select button is deactivated and the samples box and button are activated. This allows users to specify one or more samples to be selected from a sample list view page.

Annotations & parameters

As seen in the biosource section, this tab allows users to further supply information about the sample provided they have defined or shared annotation types to annotate sample items.

To learn more about annotation types and how to define a value for a type, please refer to Chapter 11, *Annotations* (page 60)

Inherited annotations

This 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 11.4.1, "Inheriting annotations from other items" (page 66) .

17.4. 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.

17.4.1. 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 extracts column for the sample that should be a parent of the extract.

There is also a corresponding button, **Create extract...** in the toolbar when viewing a single sample.

from the extract list page

Pooled extract can also 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.

17.4.2. Properties

Extract

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.

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.

Figure 17.3. Extract tab

The screenshot shows a 'Create extract' dialog box with the following fields and controls:

- Name:** Text field containing 'New extract'.
- External ID:** Empty text field.
- Original quantity:** Empty text field with a unit of μg .
- Created:** Empty text field with a 'Calendar...' button.
- Registered:** Text field containing '2009-01-21'.
- Protocol:** Dropdown menu showing '- none -' with a 'Select...' button.
- Bioplate:** Dropdown menu showing '- none -' with a 'Select...' button.
- Biowell:** Dropdown menu showing '- none -' with a 'Select...' button.
- Description:** Large empty text area.

A legend at the bottom right of the form indicates that a blue square icon represents 'required information'. At the bottom of the dialog are 'Save' and 'Cancel' buttons. Below the dialog window is a 'Done' button.

Parents

This important tab allows users to select the extract origin. BASE distinguished between two cases which are controlled by the **Pooled** radio-button.

- If the parent is a sample the radio-button is set to **No**. The Sample select button is active and allows users to point to the sample from which the sample originates.
- The parent is another extract and the radio-button is set to **Yes**. Upon selection, the samples select button is deactivated and the extracts box and button are activated. This allows users to specify one or more extracts to be selected from an extract list view page.

Annotations & parameters

As seen in the biosource and sample sections, this tab allows users to supply further information about the extract, provided they have defined annotation types to annotation extract items or have such elements shared to them.

To learn more about annotation types, please refer to Chapter 11, *Annotations* (page 60)

Inherited Annotations

This tab contains a list of those annotations that are inherited from the extract parents. Information about working with inherited annotations can be found in Section 11.4.1, “Inheriting annotations from other items” (page 66) .

17.5. Labels

Before attempting to create labeled extracts, users should make sure that the appropriate label object is present in BASE. To browse the list of labels, go to Biomaterial LIMS [Labels](#)

17.5.1. Properties

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

Name

The name of the label(required).

Description

An explaining text or other information associated with the label.

17.6. Labeled extracts

Labeled extract items should be used to describe the event that transformed an extract material in a labeled extract material. Labeled extracts can be created from extract items or from one or more labeled extract items. When a labeled extract is created from several other labeled extracts, a pooling event is performed.

During the transformation from extracts to labeled 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 extract processing task. Also, it is good practice to provide protocol information.

17.6.1. Creating labeled extracts


Beside the regular way of using the **New...** button in Biomaterial LIMS [Labeled extracts](#) , a labeled extract can be created in one of following ways.

pooling selected labeled extracts

The toolbar at the list page of labeled extracts contains an addition **Pool...** button. This button allows users to create pooled labeled extracts by selecting the list of labeled extracts used to derived a new labeled extract and then click on the button.

This provides an easy and simple way to create pooled labeled extracts. The result of such process is the creation of a new labeled extract, in which, when navigating to the parent tab, shows that all the labeled extracts involved are already set and listed in the Labeled Extract box of the tab.

from the extract pages.

Following the laboratory workflow, a natural way to create a labeled extract from an extract is to click on the  from the labeled extract column of the extract list view. Corresponding button, **New labeled extract** is located on the single item view page for each extract. By creating a labeled extract from an extract page will automatically set the extract as a parent.

from single item view of a label

Click on the **New labeled extract...** to use the current label and create a new labeled extract with it pre-selected as **Label** .

17.6.2. Properties

Labeled extract

Name

The name of the labeled extract (required). BASE by default assigns names to labeled extract (by suffixing *lbe#* when creating a labeled extract from an existing extract or *New labeled extract* otherwise) but it is possible to edit it at will

Label

Used to specify which dye or marker was used in the labeling reaction (required).

External ID

An id used to recognize the labeled extract outside BASE.

Original quantity

The mass of labeled extract that was created.

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. day effect).

Registered

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

Protocol

The labeling protocol that was used to produce the labeled extract.

Bioplate

The bioplate where this labeled extract is located.

Biowell

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

Description

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

Figure 17.4. Labeled extract properties

The screenshot shows a web browser window titled "Create labeled extract - Mozilla Firefox". Inside is a form titled "Create labeled extract". The form has the following fields and controls:

- Name:** A text input field containing "New labeled extract".
- External ID:** An empty text input field.
- Label:** A dropdown menu showing "cy3".
- Original quantity:** A text input field followed by "(µg)".
- Created:** A text input field with a "Calendar..." button next to it.
- Registered:** A text input field containing "2009-01-21".
- Protocol:** A dropdown menu showing "- none -" with a "Select..." button.
- Bioplate:** A dropdown menu showing "- none -" with a "Select..." button.
- Biowell:** A dropdown menu showing "- none -" with a "Select..." button.
- Description:** A large text area.

At the bottom right of the form, there is a legend: a blue square icon followed by "= required information". Below the form are four tabs: "Labeled extract", "Parents", "Annotations & parameters", and "Inherited annotations". At the very bottom are "Save" and "Cancel" buttons.

Parents

This important tab allows users to select the labeled extract origin. BASE distinguished between two cases which are controlled by the **Pooled** radio-button.

- The parent is an extract

The radio-button is set to **No** . The Extract select button is active and allows users to point to one and only one extract from which the labeled extract originates from.

- The parent is another labeled extract

The radio-button has to be set to **Yes** . Upon selection, the extract select button is deactivated and the labeled extracts box and button are activated. This allows users to specify one or more extracts to be selected from the labeled extract list view page.

Annotations & parameters

As seen in the biosource and sample sections, this tab allows users to further supply information about the labeled extract provided they have defined annotation types to annotate labeled extract items or have such elements shared to them.

Important

In order to use this feature, annotation type must be declared and made available. To learn more about annotation types and how these are set, please refer to Chapter 11, *Annotations* (page 60) .

Inherited annotations

This tab contains a list of those annotations that are inherited from the parents of the labeled extract. Information about dealing with inherited annotations can be found in Section 11.4.1, “Inheriting annotations from other items” (page 66) .

17.7. Bioplate

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

Use Biomaterial LIMS [Bioplate](#) to get to the list of bioplates.

17.7.1. Properties

Name

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

Plate geometry

Information about the plate design defining the number of rows and columns on the bioplate. This field are 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.


Figure 17.5. Bioplate properties

Annotations

This allows BASE users to use annotation types to refine bioplate description. More about annotating items can be read in Section 11.4, “Annotating items” (page 65) .

17.7.2. Biowell

Biowells existence are managed through the bioplate they belong to. Creating a bioplate will automatically create the biowells on the plate, correspondingly deleting a plate will also remove the associated biowells. The only thing that can be changed for a biowell is the biomaterial it holds. Go

to the **Wells**-tab when viewing a bioplate and click on  in the biomaterial column for the specific biowell you want to change. Assigning a biomaterial to a biowell can also be done when editing a sample, extract or labeled extract item.

Properties

Bioplate

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

Coordinate

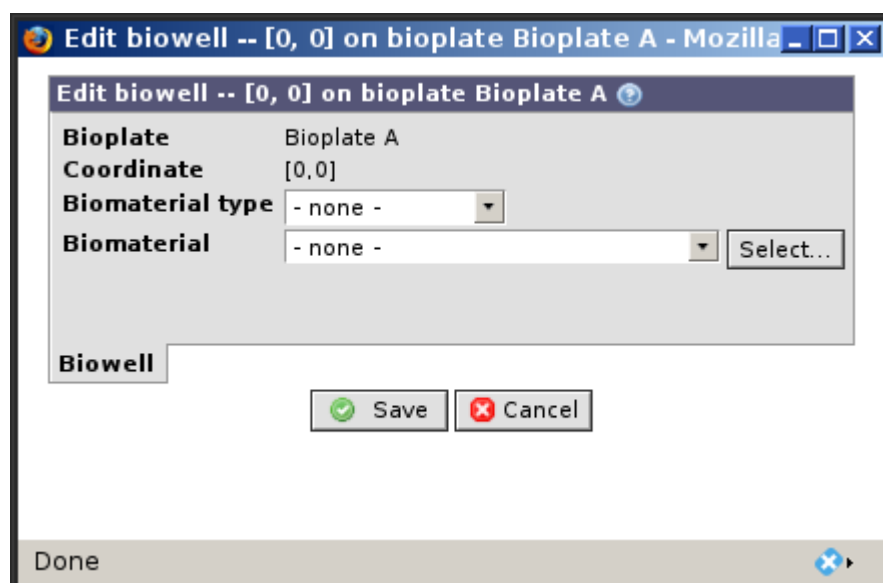
The biowell location on the bioplate in format [row,column]. This property is read-only.

Biomaterial type

The type of biomaterial stored in this biowell. This property must be selected before before a biomaterial can be selected.

Biomaterial

Name of the biomaterial in this biowell. Before changing this you must select the appropriate **Biomaterial type**.

Figure 17.6. Biowell properties

17.8. Hybridizations

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 Hybridizations to get to the hybridizations.

17.8.1. Creating hybridizations

In BASE, there are two possible routes to create an hybridization object except the common way with the **New...** button at hybridization list page.

from the labeled extract list view page

Select at least one labeled extract, to create a hybridization from, by ticking the selection boxes before the name field.

Click on the **New hybridization...** from the toolbar of labeled extract list view.

from a labeled extract single item page

When viewing a label extract in single item view, click on the **New hybridization...** button from the toolbar of the labeled extract item view.

17.8.2. Properties

Hybridization

Figure 17.7. Hybridization tab

Name

New hybridization is the BASE default name but it is strongly advise to provide a meaningful and unique name (required).

Arrays

The number of sub-arrays on the slide that was used in this hybridization. The default value is 1, but some platforms, for example Illumina, has slides with 6 or 8 arrays. When the array-slide property below is changed this value will be updated automatically to be consistent with the number of sub-arrays on the used array-design.

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 hybridization protocol that was used to do the hybridization.

Hardware

The hybridization-station that was used during the hybridization.

Array slide

The array slide that was used in the hybridization.

Note

Ideally, The Array Slides should have been created but for those users with permission to do, Array Slides could be generated at that point.

Description

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

Labeled extracts

This important tab allows users to select the labeled extracts applied to an array slide, and specify the amount of material used, expressed in microgram.

Use the **Add labeled extracts** button to add items and the **Remove** button to remove items. Select one or several labeled extracts in the list and write the used mass and sub-array index in the fields below.

Annotations & parameters

As seen in the biosource and sample sections, this tab allows users to supply further information about the hybridization provided annotation types have been defined or shared to annotate hybridization items.

Important

In order to use this feature, annotation type must be declared and made available. To learn more about annotation types and how these are set, please refer to Chapter 11, *Annotations* (page 60)

Inherited annotations

This tab contains a list of those annotations that are inherited from the labeled extracts. Information about dealing with inherited annotations can be found in Section 11.4.1, “Inheriting annotations from other items” (page 66) .

Chapter 18. Experiments and analysis

18.1. Scans and images

When you have done your hybridization and scanned it, you can register information about the scanning process as a **Scan** item in BASE. A scan item holds information about the scanning process, such as which scanner and what settings you have used. It can also hold information about the images that was produced as well as the actual image files.

Note

A scan does not have information about the spots or raw data. The process of analysing the images is considered a separate step. This information is held by raw bioassays.

18.1.1. Scan properties

A scan has the following properties:

Name

The name of the scan.

Hybridization

The hybridization this scan comes from.

Scanner

The scanner that was used (optional).

Protocol

The protocol used for scanning (optional). Scanning parameters may be registered as part of the protocol.

Description

A decription of the scan (optional).

A scan can have annotations. Read more about this in Chapter 11, *Annotations* (page 60).

18.1.2. Images

If you want you can upload the images from the scan to BASE. This is optional, but the images are needed if you want to create spot images later on.

To upload images for a scan you must first go to the single-item view for the scan. On this page you will find the **Images** tab which will take you to a list view of the images from this scan. New images are created in the same way as any other item in BASE, by clicking the **New...** button. You may add as many images as you like.

An image has the following properties:

Name

The name of the image. The name is automatically generated from the name of the scan, but you may enter a different name.

File

The file containing the actual image data (optional). Use the **Select** button to select an existing file or upload a new one.

Format

The image format (optional). You can select between *TIFF*, *JPEG* and *Unknown*.

Preview

If the image is a preview image or the full scan.

Description

A description of the image (optional).

18.2. Raw bioassays

A **Raw bioassay** is the representation of the result of analysing one or more images from a **Scan**. This typically generates a raw data file with lots of measurements for the spots on the hybridization.

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 list view.
2. Upload the file(s) with the raw data and attach it/them to the raw bioassay.
3. The used platform may require that data is imported to the database. See Chapter 19, *Import of data* (page 108). 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 from one hybridization must be in a single file. Support for other file formats may be added through plug-ins.

18.2.1. Raw bioassay properties

Figure 18.1. Raw bioassay properties

Create raw bioassay ?

Name

Array index

Platform

Raw data type

Array design

Protocol

Scan

Software

Description

☐ = required information

Raw bioassay

Done

Name

The name of the raw bioassay.

Array index

The index of the sub-array on the hybridization this raw bioassay's data is linked with. The default value is 1. With some platforms, for example Illumina, which has slides with 6 or 8 arrays the value should be changed to reflect the correct sub-array. This information is important to link the raw bioassay with the correct biomaterial entries.

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 18.2.3, "Raw data types" (page 101).

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.

Scan

The scan this raw bioassay is related to (optional). Changing this property will also update the value in **Array design**, but only if the selected scan is connected to an array design and the current user has permission to view it.

Software

The software used to analyse the image or images (optional).

Protocol

The protocol used when analysing the image(s) (optional). Software parameters may be registered as part of the protocol.

Description

A description of the raw bioassay (optional).

A raw bioassay can have annotations. Read more about this in Chapter 11, *Annotations* (page 60).

18.2.2. 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 19, *Import of data* (page 108) for more information.

File-only platforms

File-only platforms, such as Affymetrix, is handled differently and data is not imported into the database. See the section called “File-only platforms” (page 101).

18.2.3. Raw data types

A raw data type defines the types of measured values that can be stored for individual spots 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 scanner and software used to analyse the images from a hybridization. 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 E, *Platforms and raw-data-types.xml reference* (page 346) for more information.

File-only platforms

BASE 2.5 introduced a generic way to keep the data in files instead of having to import it to the database. In older BASE versions this ability was limited to the Affymetrix platform. The reason for keeping the data in files is that the number of spots tend to grow, which may result in bad performance if the database should be used. 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.

18.2.4. Spot images

If you have uploaded the image or images from the scan you may create spot images. Spot images allows you to view the image of each spot separately in the analysis. For this to work the raw data must contain the 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 18.2. Create spot images
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.

18.3. 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, hybridizations, 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.

18.3.1. Experiment properties

Figure 18.3. Create experiment

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

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.

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.

18.3.2. 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.

¹ <http://www.ncbi.nlm.nih.gov/entrez/query/static/overview.html>

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 11.4.1, “Inheriting annotations from other items” (page 66) 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.

18.3.3. Tab2Mage export

Tab2Mage format² is a tab-delimited format veted by EBI's ArrayExpress³ repository for submission microarray data. Tab2Mage format has been chosen by BASE to provide an easy way for data deposition to public repository when submitting a manuscript and publishing experimental data.

BASE has been engineered to closely map the MIAME concepts and a number of BASE entities can be mapped directly to Tab2Mage elements. However, since MIAME is focused on microarray processing workflow, information about the biological sample is down to the user. To accommodate the annotation needs of users, BASE provides a mechanism that allows annotation customization to meet user specific requirements. BASE allows to create annotation type for quantitative annotation and qualitative annotation

BASE can export an experiment to Tab2Mage format thanks to a dedicated export plug-in. For the plug-in to work, it is important to understand that information recorded in BASE should be formatted following a small number of rules. Failing to do so may impair the possibility of exporting to ArrayExpress.

Note

The Tab2Mage export plug-in has not yet been included in the main distribution. Hopefully, it will appear in the next (2.4) release.

Biomaterial annotations

Tab2Mage specifications only allow *BioSource* items to be annotated with *BioMaterialCharacteristics*.

Warning

All BASE Annotation Types used to annotate at the level of *Sample* and *Extract* items will be lost during the export in Tab2Mage format in order to comply with the ArrayExpress Tab2Mage parser.

Note

In the context of data exchange between BASE instances, the export function can be altered to allow attachment of annotations to items other than biosources, therefore avoiding loss of information.

² <http://tab2mage.sourceforge.net/>

³ <http://www.ebi.ac.uk/microarray/>

Annotation units

To associate units to BASE annotation types and remain compatible with Tab2Mage specifications, users need to adhere to the following convention:

`annotation_type_name(unit_name)` as in `body mass(kg)` or `concentration(mg/ml)`

Warning

Only one unit can be specified at any one time for any given annotation type. In order to enable Tab2Mage support, it might be necessary to declare several related Annotation Type in order to report similar kind of information but expressed in a different unit. Specifying Age for instance is a good example on how to deal with such cases: One should create several related annotation types e.g. `Age(week)`, `Age(year)` or `Age(month)` as those variations maybe be necessary when reporting the age of a mouse or the age of a human volunteer.

Protocol parameters

In order to ensure MIAME compliance, Tab2Mage specifications cater for reporting parameters attached to protocols and all parameters attached to a protocol should be declared in the protocol section of a Tab2Mage file.

In BASE terms, Tab2Mage elements such as *BioMaterialCharacteristics*, *Parameter* or *Factor-Values* are all annotation types. But, it is necessary to flag those annotations types meant to be used as protocol parameters as such so that they can identified by the Tab2Mage exporter and handled appropriately.

Warning

It is not possible to use the same annotation type *Temperature* for reporting a patient body temperature (which is a *Biomaterial Characteristic*) and hybridization temperature (which is a protocol parameter). Hence it will be necessary to declare 2 distinct annotation types:

- Annotation type to be used as *BioSource characteristics*: `body temperature (degree_C)`
- Annotation type to be used as *protocol parameter*: `hybridization temperature (degree_C)`

Experimental factors

It is a MIAME requirement to identify *Experimental Variables* when submitting data to Array-Express (provided the study is an intervention study). Therefore, BASE users willing to use the Tab2Mage export function will have to declare *Experimental Factors* using the the **Experimental Factor** tab available when editing experiments. See Section 18.3.2, “Experimental factors” (page 104) for more information.

Values for the experimental factors are take from annotations. The annotation must exist at the raw bioassay level, which probably means that you have to inherit the annotation from some other item, for example, a biosource or a sample. It is also possible to use a protocol parameter as experimental factor. See Chapter 11, *Annotations* (page 60) for more information about annotations.

18.4. Analysing data within BASE

TODO

18.4.1. Transformations and bioassay sets

TODO

The root bioassay set

TODO

Overview plots

TODO

18.4.2. Filtering data

TODO

Formulas

TODO

18.4.3. Normalizing data

TODO

18.4.4. Other analysis plug-ins

TODO

18.4.5. The plot tool

TODO

Scatter plots

TODO

Histogram plots

TODO

Filtering plots

TODO

Save plots

TODO

18.4.6. Experiment explorer

TODO

Reporter view

TODO

Reporter search

TODO

Chapter 19. 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 extracts. 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, the import plug-ins need some information about how to find headers and data lines in files. BASE ships with a number of export plug-ins as a part of the core plug-ins package, cf. Section B.3, “Core import plug-ins” (page 332). The core plug-in section links to configuration examples for some of the plugins. Go to [Administrate Plugins 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.

Missing/unavailable 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.

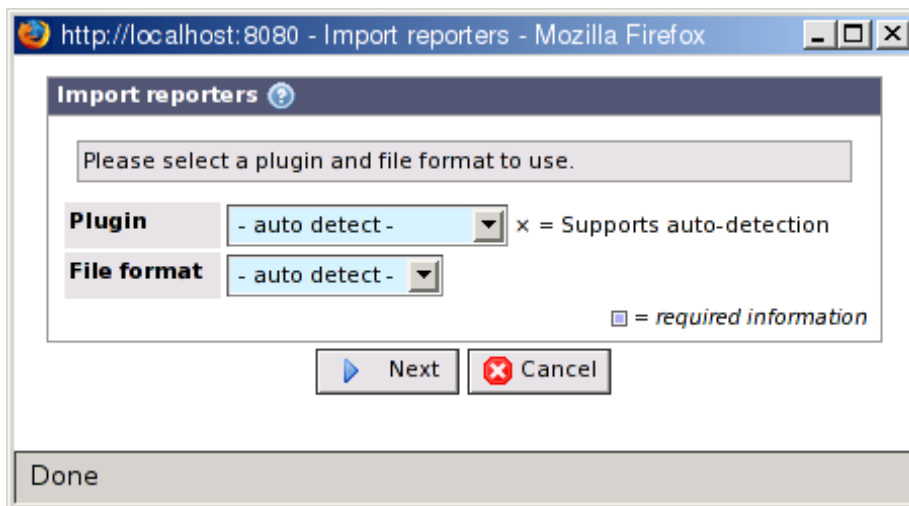
19.1. General import procedure

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 select 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.

19.1.1. 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 19.1. Select plug-in and file format**Plugin**

A list of all plug-ins that are available in the current context. The list only includes plug-ins that the logged in user has permission to use. If you select a plug-in a short description of about it is displayed below the lists. More information about the plug-ins can be found under the menu choice **Administrate Plugins Definitions**

File format

A list of different file formats configurations supported by the selected plug-in. **Administrate Plugins Configurations.**

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 both the plug-in and file format lists 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 option either for both plug-ins and file formats or only for file formats to use this feature. Continue to the next step by clicking on the **Next** button.

You must now select a file to import from.

Figure 19.2. Select file for auto detection
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 8, *File management* (page 43).

Recently used

A list of files you have recently used for auto detection.

Click on the **Next** button to start the auto detection.

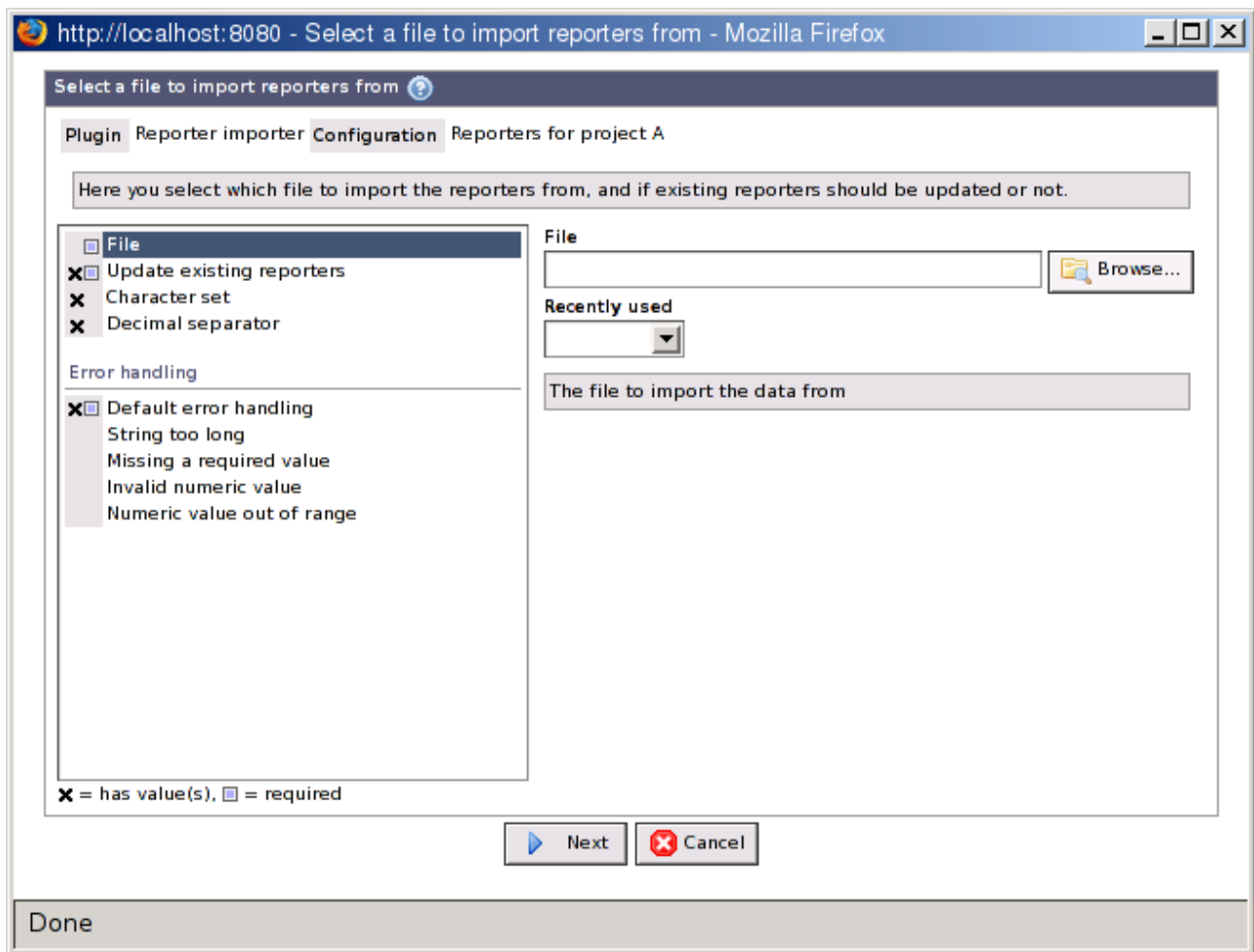
If the auto detection finds a exactly one plug-in and file format the next step is to configure any additional parameters needed by the plug-in. This is the same step as if you had selected the same plug-in and file format in the first step. If no plug-in can be found an error message is displayed.

More then one compatible plug-in/file format

If more than one matching plug-in or file format is used 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.

19.1.2. 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 you can enter additional parameters for the plug-in is displayed.

Figure 19.3. 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 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.

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 while or if the line with the error should be skipped.

Continue to the next step by clicking the **Next** button.

19.1.3. Add the import job to the job queue

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.

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.

19.2. 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 available (Section B.3.1, “Core batch import plug-ins” (page 334)). These importers allows 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 22.4, “Plug-in configurations” (page 140).

The batch importers either creates new items or updates 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 labeled extracts of a hybridization or pooled samples are two examples of items that refer to several other items; a hybridization may contain several labeled 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 array index a labeled 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 already existing annotation importer plug-in (Section 11.4.2, “Mass annotation import plug-in” (page 67)). However, the annotation importer and batch item importers have similar behaviour and functionality to minimize the learning cost for users.

The importer only works one item type 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.

19.2.1. File format

For proper and efficient use of the batch importers users need to understand how the files to be imported should be formatted. For users who wishes to get a hands-on experience there is an OpenOf-

file spreadsheet with sample sheets that work with the batch importers¹ available for download. This file can be used to import a set of data from the biosource level down to hybridizations with proper associations and properties simply by using the batch importers.

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.

19.2.2. 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 19.1, “General import procedure” (page 108) and the setting of column mapping parameters is assisted with the **Test with file** function described in Section 22.4.3, “The **Test with file** function” (page 143). 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.

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

¹ http://base.thep.lu.se/attachment/wiki/DocBookSupport/batchimport_sample.ods?format=raw

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 .

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 22.4.3, “The **Test with file** function” (page 143) to set the column mapping parameters.

When creating pooled items, the pooled property is used to tell the plug-in that an item is pooled. Pooled in BASE language really means that the item parent is of the same type as the item itself. If an item is not pooled then the parent is of another type following a predefined hierarchy in BASE. In ascending order the BASE ordering of *parent - child - grandchild - ...* item relation is *biosource - sample - extract - labeled extract*.

The values accepted for pooled are `empty (' ')`, `0`, `1`, `no`, `yes`, `false`, and `true`. Any other string is interpreted as the item is pooled. Sometimes all items in a file to be imported are pooled but there is no column that marks the pooled status. This can be resolved by setting the pooled mapping to a constant string `'1'` which make all items to be treated as pooled in the import (no backslash `'\'` character, compare with column header mapping strings that contain backslash characters like `'\pool column\'`).

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 ignored the line.

19.2.3. 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 11.4.2, “Mass annotation import plug-in” (page 67) 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 20. 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 B.2, “Core export plug-ins” (page 332). Go to [Administrate Plugins 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.

Missing/unavailable 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.

20.1. Select plug-in and configuration

This dialog is very similar to the dialog for selecting an import plug-in. See Figure 19.1, “Select plug-in and file format” (page 109) 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

Select the plug-in to use. Only plug-ins that supports the current context and that the logged in user has permission to use are available in the list.

Configuration

Select the configuration that should be used together with the plug-in. Not all plug-ins supports configuration and this option is only visible if the selected plug-in has this support.

Click on **Next** to show the configuration of parameters for the job.

20.2. 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

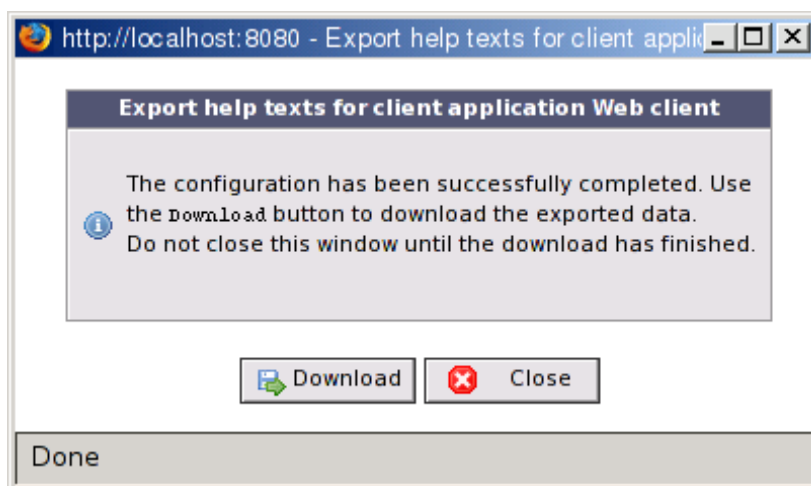
The path and file name in the BASE file system where the exported data should be saved. Some plug-ins support immediate download to the local file system if you leave the file parameter empty. For saving the exported data within the BASE file system, it's recommended to use the **Browse...** button to 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 20.1. 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.

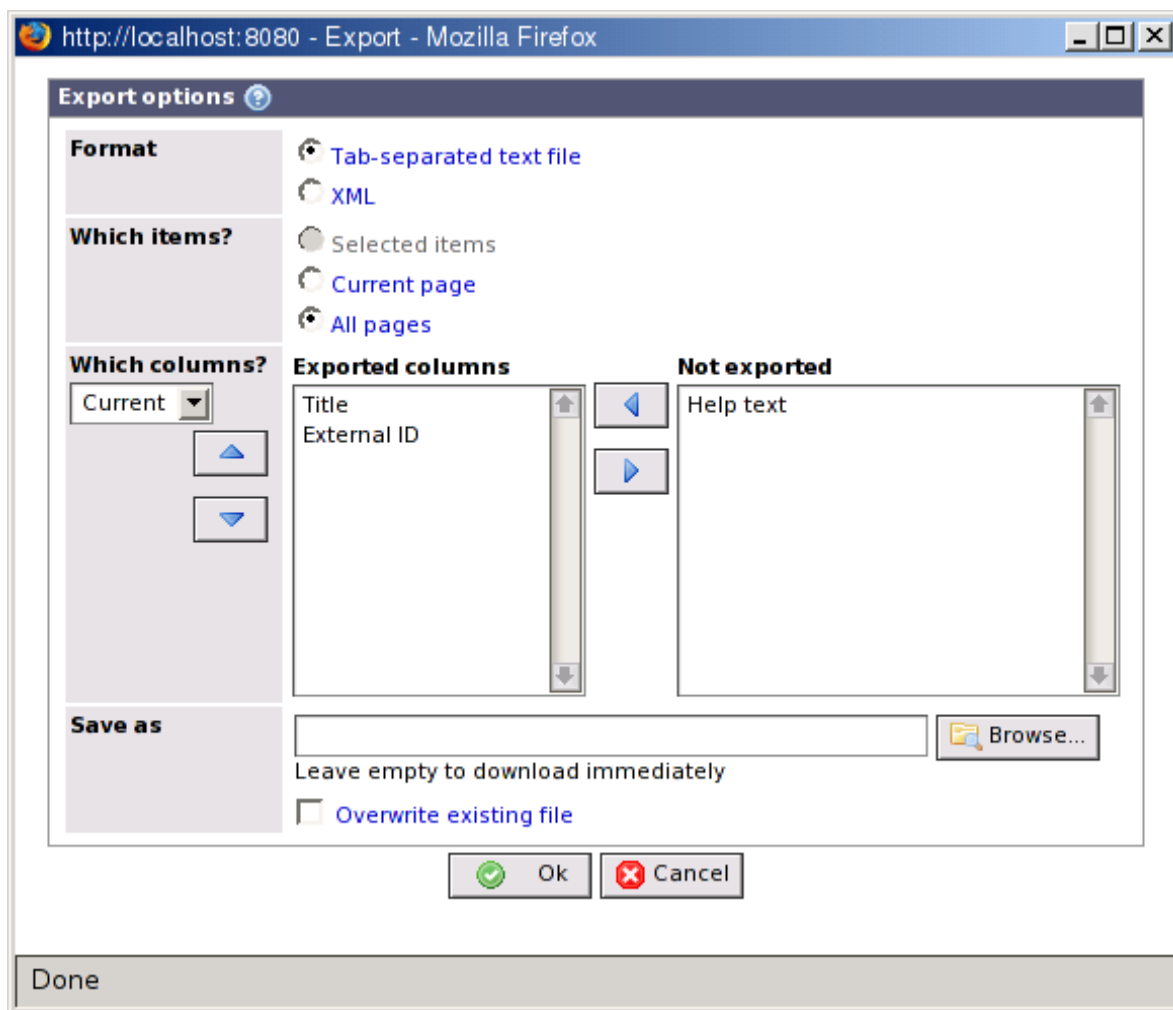
20.3. The table exporter plug-in

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 taken 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 20.2. The table exporter configuration dialog



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 decides which items that should be included in the exported data.

- **Selected items:** Only selected items will be exported. This option 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.

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 21. Installation, setup, migration, and upgrade instructions

Note

These instructions apply only to the BASE release which this document is a part of.

This chapter is divided into four parts. First, the process of upgrading a BASE server is described. Followed by set up of job agents. (For these two first parts it is assumed that there is a running server.) Then, the first time installation instructions follows, and the chapter is concluded with information on how to migrate data from a BASE 1.2.17 server to a current BASE server.

The first time installation is only to be performed once, optionally followed by a migration. The migration can only be done to a pristine (empty) BASE version 2 server with one exception; The migration can be restarted if it fails during the RawBioAssaysData transfer. For all other failure points the migration has to be restarted with an empty database. Migration from several BASE version 1 installations to one BASE server is not supported.

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.

21.1. Upgrade instructions

Important information for upgrading to the current release

This section list some important information that may or may not apply when upgrading from the *previous* BASE release to the current release (eg. 2.8.x to 2.9.x). If you are upgrading from a BASE installation that is older (eg. 2.7.x to 2.9.x) you should also read Appendix L, *Things to consider when updating an existing BASE installation* (page 365).

BASE 2.9 must use a database that supports UTF-8

If you are upgrading from BASE 2.8 or lower and your existing database is not using UTF-8 you must convert the database to UTF-8 *before you execute the* `./updatedb.sh` script.

BASE 2.9 includes a utility that can convert an existing MySQL database. After installing the BASE 2.9 files, but *before* running the `./updatedb.sh` script, execute the following on the command line:

```
cd <base-dir>/bin
./onetimefix.sh utf8 -x
```

The `-x` option makes the script update the database immediately. You can leave this option out to have it generate a SQL script file (`convert-to-utf8.sql`) instead. The script will by default not try to convert tables that it already thinks are using UTF-8. If the script for some reason is incorrect when detecting this, you can use the option `-f` to force conversion of all tables.

The conversion utility only works with MySQL. PostgreSQL users should instead use a backup and restore using as described in the PostgreSQL manual². Eg. dump the existing BASE database, create a new database that uses UTF8 and restore the backup into the new database.

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

¹ <http://tomcat.apache.org/>

² <http://www.postgresql.org/docs/8.1/static/backup.html>

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 21.4.1, “Sending a broadcast message to logged in users” (page 130).

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 4.1.1, “Download” (page 6) 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 Section 31.2, “Subversion / building BASE” (page 302) 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.

Remove Tomcat cache

As Tomcat user, remove cached files and directories. Do something like

```
cd /usr/share/apache-tomcat-6.0/  
rm -rf work/Catalina
```

Start Tomcat

Start the Tomcat server: **sudo /etc/init.d/tomcat6.0 start**

Done! Upgrade of BASE is finished.

21.2. 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-in. 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.

21.2.1. 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.

Note

Prior to BASE 2.5 the internal job queue had to be disabled completely. This is no longer the case since it is possible to enable/disable the internal job queue separately for each plug-in.

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 124) below.

21.2.2. 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 126). 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 124) below in job agent server setup. You can use the same database user and password as in the regular database setup.

21.2.3. 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 127), 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 124) 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.
- See the Appendix C, *base.config reference* (page 336) 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 124) above.
- **agent.id:** An ID that must be unique for each job agent accessing the BASE application.

- **agent.remotecontrol**: The name/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 G, *jobagent.properties reference* (page 351) 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 H, *jobagent.sh reference* (page 354) for more information about what you can do with the job agent command line interface.

21.2.4. Configuring the job agent

Before the job agent starts executing jobs for you it must be configured. The configuration is done through the BASE web interface. See Section 22.3, “Job agents” (page 139)

Configure the plug-ins the job agent should handle

- Go to the **Administrate** → **Plugins** → **Job agents** menu.
- Select the job agent and click on the **Edit...** button.
- On the **Plugins** tab you can specify which plug-ins the job agent should handle. Note that if you have installed external plug-ins on the web server, those plug-ins must be installed on the job agent as well. It is possible to specify different paths to the JAR file for each job agent.

Grant users access to the job agent

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.

21.3. Installation instructions

Java

Download and install Java SDK 1.6 (aka Java 6), available from <http://java.sun.com/>.

Tomcat

Download and install Apache Tomcat 6.0.20 or later, available from <http://tomcat.apache.org>. It is a good idea to specify the maximum allowed memory that Tomcat can use. The default setting is usually not large enough. If you are using Tomcat 6.0.18 or higher you also need to disable strict parsing of JSP files.

Unless you have manually downloaded and installed JAI (Java Advanced Imaging) native acceleration libraries (see <https://jai.dev.java.net/>) it is also a good idea to disable the native acceleration of JAI.

All of the above is done by setting Java startup options for Tomcat in the `CATALINA_OPTS` environment variable. Basically add the next line (as a single line) close to the top of the `catalina.sh` script that comes with Tomcat (directory `bin`):

```
CATALINA_OPTS="-Xmx500m  
-Dorg.apache.jasper.compiler.Parser.STRICT_QUOTE_ESCAPING=false
```

```
-Dcom.sun.media.jai.disableMediaLib=true"
```

For more information about Tomcat options see <http://tomcat.apache.org/tomcat-6.0-doc/index.html>.

Set up SQL database

BASE utilize Hibernate³ for object persistence to a relational database. Hibernate supports many database engines, but so far we only work with MySQL⁴ and PostgreSQL⁵.

MySQL

Download and install MySQL (tested with version 5.0), 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 8.2 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.

Note

PostgreSQL versions prior to 8.2 have a non-optimal solution for locking rows in certain situations. This may cause two seemingly independent transactions to lock if they just reference a common database row. This may happen, for example, when importing raw data that have references to the same reporters. The problem has been solved in PostgreSQL 8.2.

BASE (download and unpacking)

Download BASE⁶ 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 site⁷).

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 document⁸. When you return back 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

³ <http://www.hibernate.org/>

⁴ <http://www.mysql.com>

⁵ <http://www.postgresql.org/>

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

⁷ <http://base.thep.lu.se/wiki/DownloadPage>

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.

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 above.

See the Appendix C, *base.config reference* (page 336) 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 properties⁹ 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 hostname) 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.

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://www.apache.org/>, and a apache-tomcat connector, available from <http://jakarta.apache.org/tomcat/connectors-doc/index.html>. So, we got you there;-) To be honest, this step is not really well documented since we previously used SuSE 9.3 on our demo/test server, and `apache/tomcat/mod_jk` comes pre-installed. The current server does not use the `apache/tomcat` connector. What you need to do is something like this

- Get that Tomcat server running in stand-alone mode.
- Get the Apache 2 server running.
- Install `mod_jk`. Note, different version are used for apache 1.3 and 2. In SuSE 9.3 this step is done by installing `mod_jk-ap20`.
- Create a `workers.properties` file in the Tomcat base directory (commonly copied from a template).

⁹ <http://base.thep.lu.se/chrome/site/doc/historical/admin/extended-properties.html>

- Create a `jk.conf` file in the `apache conf` directory (commonly copied from a template), and make sure that `jk` is added to the modules to be loaded when apache starts.
- In `jk.conf` add the lines below and change paths appropriately.

```
# The following lines makes apache aware of the location of
# the /base2 context
Alias /base2 "/srv/www/tomcat6/base/webapps/base2"
<Directory "/srv/www/tomcat6/base/webapps/base2">
Options Indexes FollowSymLinks
allow from all
</Directory>
# The following lines mounts all base2 jsp files to Tomcat
JkMount /base2 ajp13
JkMount /base2/* ajp13
# The following lines prohibits users from directly accessing WEB-INF
<Location "/base2/WEB-INF/">
AllowOverride None
deny from all
</Location>
```

You must restart the Apache and the Tomcat server after above steps.

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 24, *Account administration* (page 152).

If you are planning to perform a migration of data from BASE version 1.2.x please perform the steps in Section 21.5, “Migration instructions” (page 131) before doing anything else with your new BASE installation.

21.4. 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.

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 none-registered 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 an 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>¹⁰. 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.

21.4.1. 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.

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.

¹⁰ <http://base.thep.lu.se/chrome/site/doc/html/index.html>

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.

21.5. Migration instructions

The disclaimer section

We have made extensive testing of the migration from BASE version 1.2 (BASE1.2) to BASE version 2 (BASE2). To our knowledge the migration works but we cannot guarantee perfect functionality. The migration tool is supplied as is and usage is done at your risk. We are committed to solve migration problems at the level of transferring data from BASE1.2 to BASE2, but cannot resolve data loss issues in a running BASE2 server due to imperfect migration. When you start the migration tool you are required to pass parameter "*disclaimer_understood*" to the `migrate_from_1.2.sh` script. Remember to check that the migration performed well before you decide to delete your 1.2 installation.

Since BASE 2.6 we no longer actively test or maintain the migration program. We still accept bug reports and will try to fix critical bugs, but only if we get help with testing the fixes before they are released. Supplying a patch for a bug will increase the chance of getting it fixed.

Verify that your new BASE installation is up and running before attempting migration. Preferably try to log in using the root user through the web interface.

Make sure your BASE1.2 runs with the latest (*pristine*) schema version, *i.e.*, the migration will only support an unmodified BASE1.2 installation. If you have an out of date schema version, please upgrade to the latest schema using BASE1.2 tools before migrating. If you have made local changes to the BASE1.2 schema you need to patch the BASE2 schema as well as make proper changes to the migration program. If there are added columns to the reporter table in your BASE1.2 database you need to transfer the additional information after migration (even if you modified the BASE2 `extended-properties.xml` file).

The behaviour of migration is controlled through `migration.properties` file (see Appendix I, *migrate.properties reference* (page 356)), but you should know what you do when you change parameters in this file.

Migration can be restarted in one specific case only; If the migration fails during the RawBioAssay-Data transfer simply restart the migration and the script will ask you to verify that a restart should be attempted. For all other failure points the migration has to be restarted with an empty database.

Migration is performed with the following steps:

1. To transfer files from BASE1.2 (default migration behaviour), you must have file system access to BASE1.2 files, *i.e.*, the `/BASE 1.2/data` directory containing directories `rawdata`, `uploads`, `protocols`, ...
2. Change settings in the file `<base-dir>/dist/www/WEB-INF/classes/migrate.properties`. The available options are commented:
 - Modify `db1.*` parameters to match your BASE1.2 installation.
 - Set `userfiles` (note, this is not the same parameter as in `base.config`) to point to the directory containing the BASE1.2 files (defined in a previous step above).
 - Set the `pluginDir` to point to the directory where your BASE1.2 plug-ins are installed. The default is `/usr/local/base/plugins`.
 - Modify `root.password`.
 - Change the `deletefiles` setting wisely. If you set `deletefiles=yes` all BASE1.2 files will be physically removed when copied to the BASE2 server. Leave this options as `deletefiles=no` unless you are absolutely sure of what you are doing.

3. Run migration utility:

```
cd /path_to_base/dist/bin
./migrate_from_1.2.sh disclaimer_understood
```

If the migration fails during the RawBioAssayData transfer you can restart the migration at the point of failure. Simply restart the migration and the script will ask you to verify that a restart should be attempted. For all other failure points the migration has to be restarted with an empty database.

4. *Optional, depends on your BASE1.2 reporter table.* Additional columns (as compared with a pristine database schema) in the reporter table in the BASE1.2 schema are not transferred during migration. You have to perform the transfer manually after migration. Simply export the reporter information from BASE1.2, and import the data into the new BASE installation. In BASE1.2: i) View the reporters, ii) Use `Get as tab-separated text` to create a tab separated file (right click and save). In BASE2, to import the file follow the instructions in Section 10.3.1, “Import/update reporter from files” (page 55).
5. Migration done! Happy BASEing.

Chapter 22. Plug-ins

BASE can get extended functionality by the use of plug-ins. In fact, most 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. See Chapter 4, *Resources* (page 6) for more information about the core plug-ins and 3rd-party plug-ins.

22.1. Installing plug-ins

The first step is to install the plug-in on the web server. To make these instructions easier to read we assume the plug-in comes as a single JAR file and that it does not depend on any other JAR files.

We recommend that you install the plug-in JAR file outside the web server's Classpath. Do not put the plug-in in the `<base-dir>/www/WEB-INF/lib` directory or any other directory where the web server keeps it's classes. This makes BASE use it's own class loader that support unloading of classes as well. This means that you can install new plug-ins and update existing ones without restarting the web server.

We recommend that you create a separate directory for plug-ins and install all of them there. You may use a sub-directory for each plug-in if you like, for example: `<base-dir>/plugins/<name-of-plugin>`

Plug-ins that use other JAR files

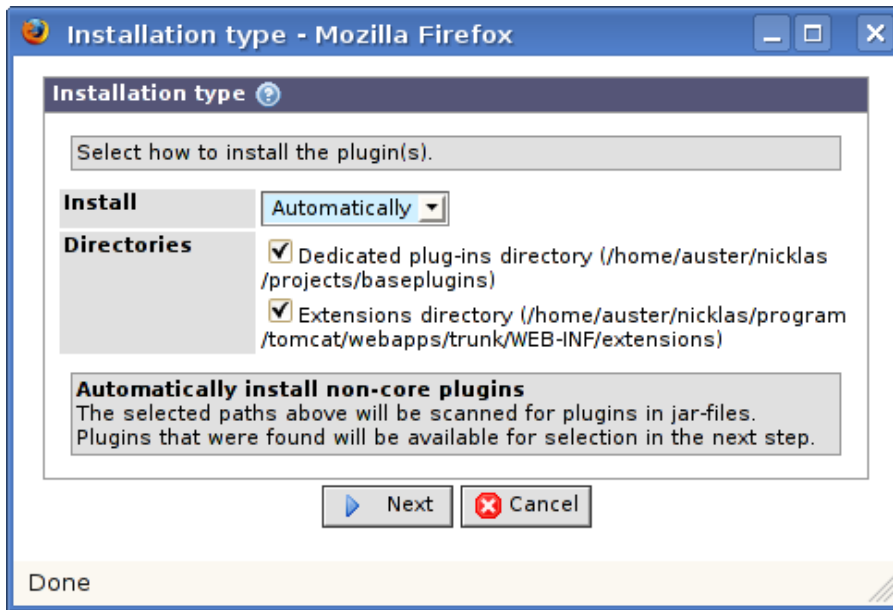
Some plug-ins may depend on other JAR files. Normally, these JAR files should be put in the same directory as the plug-in JAR file. This may, however, vary from plug-in to plug-in so always check the plug-in documentation first.

When the plug-in is installed on the server you must register it with BASE. To register the plug-in with BASE go to **Administrate** **Plugins** **Definitions** and click on the **New...** button. In the pop-up window that appears you should first decide if you want to do a manual or an (almost) automatic installation. An automatic installation scans the server disks for plug-ins and lets you select which ones to install or update from a list in the web interface. In a manual installation you are required to enter the path and class name of the plug-in yourself.

22.1.1. Select installation method

This window appears, like described above, when automatic installation/registration of plug-ins is available.

Figure 22.1. Select installation type



Install

How to register the plug-in(s) with BASE.

Automatically

BASE will scan the selected directories for JAR files containing plug-ins. If it finds any, you will have the option to select which ones to install from a list. The next step will be Section 22.1.3, “Automatic installation of plug-ins” (page 136)

Manually

Class name and JAR path have to be entered manually by the administrator. Only one plug-in can be registered at a time in this way. Next step will be Section 22.1.2, “Plug-in properties” (page 134)

Directories (only when *Automatically* is selected)

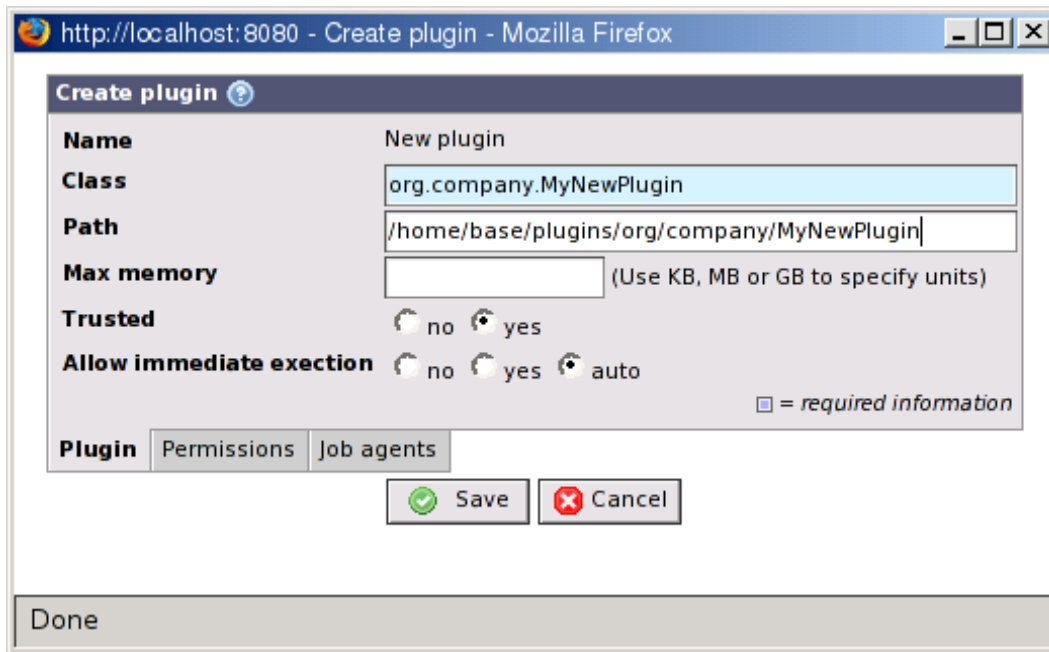
Select the checkboxes were BASE should look for new plug-ins. For security reasons the only options are the dedicated plug-ins directory that is configured in the `base.config` directory and the extensions that directory that is normally used for extensions to the web client.

Note

The automatic installation doesn't allow you to set all properties that can be set in a manual installation. If you for example, need to share the plug-ins to other users or assign them to job agents this must be done after the auto-installation has been completed.

22.1.2. Plug-in properties

This section describes the manual installation of a plug-in.

Figure 22.2. 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 Java class name of the plug-in.

Path

The path to the JAR file on the web server. 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 22.3, "Job agents" (page 139) later in this chapter.

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 117).

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.

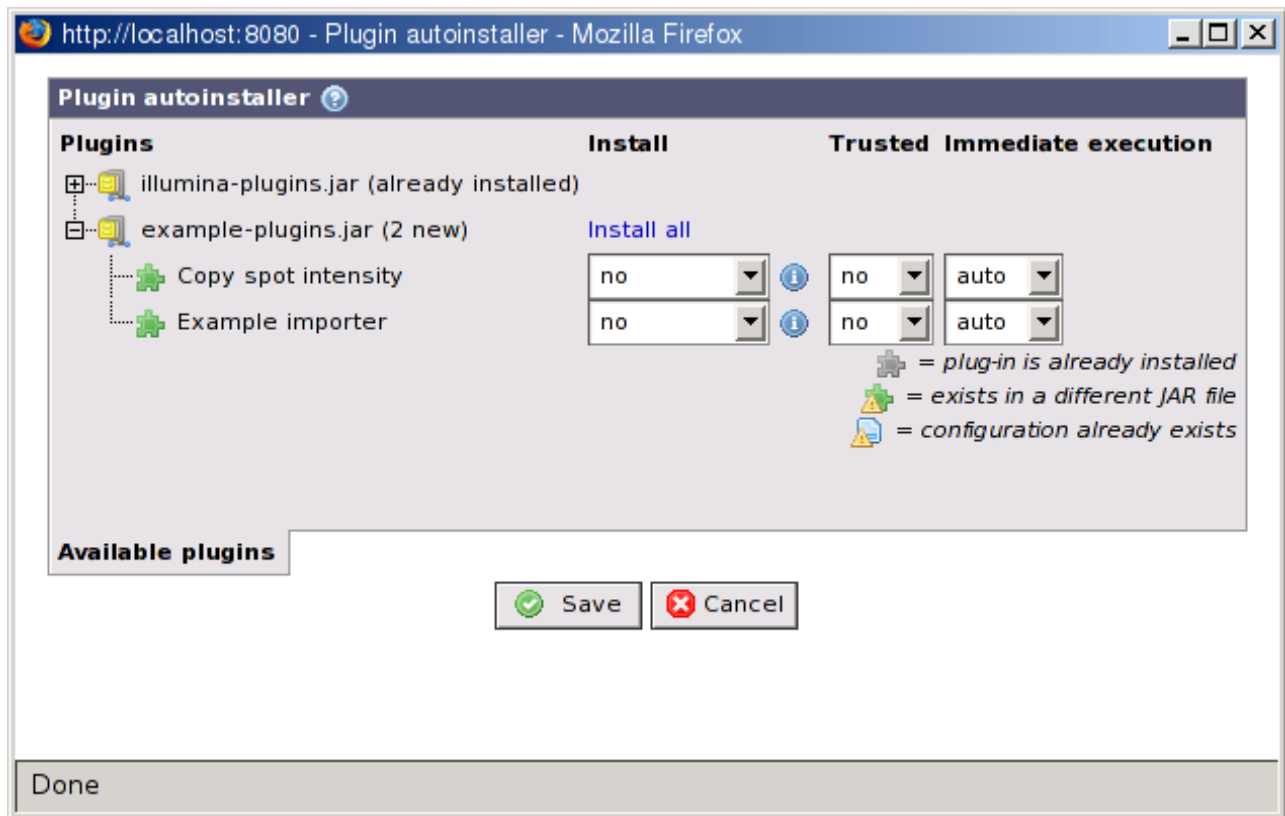
- **Auto:** BASE will allow export plug-ins to execute immediately, and deny all other types of plug-ins. This alternative is only available when registering a new plug-in.

Click on **Save** to finish the registration or on **Cancel** to abort.

22.1.3. Automatic installation of plug-ins

This section describes the automatic installation of plug-ins using a wizard.

Figure 22.3. Auto install plug-ins



This window lists all the plug-ins that were found in JAR files, after scanning the selected directories from the previous step. There are a few options to set for each plug-in before they can be registered.

The page has, for each plug-in, a row that is divided into four columns with information or settings. These are explained more in details below. There is an icon in the **Install** column that displays summarized information about the plug-in when the mouse pointer is held over it. Most of this information is from the plug-in's *About* property but some, like 'Works with', 'Class' and 'Jar' is from the installation file included in the JAR file. A plug-in can also be distributed with one or many configurations that can be selected for import together with plug-in registration. The configurations are by default hidden but can easily be viewed by expanding the configuration tree for a plug-in.

Plugins

The plug-in's name, from the plug-in class. Names of the configurations that comes with a plug-in are also displayed in this column but in a tree under the plug-in they belong to.

Install

A selection list for each plug-in found when scanning the plug-in directory. There are at least two option for each plug-in in this drop-down list. The default option *no* will not register the plug-in to BASE while *yes* will register the plug-in when proceeding. If the plug-in also have configurations the *yes* option is replaced with *plugin only* and *plugin + configurations*. If the last option is selected all configurations for the specific plug-in will be imported.

Each listed configuration can individually be defined if it should be imported or not. This is done by selecting *yes* or *no*.

Note

Plug-ins that already are registered in BASE will be disabled in the list, unless the plug-in comes from a JAR file in a different path than the one registered in BASE, or if the version does not consist with the one registered in BASE. In these cases the drop-down list is enabled for selection but the plug-in name is marked with an exclamation mark and the plug-in will be re-registered and updated if any of the two install options are selected.

Note

If a configuration in the list has an identical name as a configuration in BASE and the configurations are for the same plug-in, there will be a warning in the list beside the install-select box. The already existing configuration will not be overwritten if the new plug-in configuration is set to be imported, but there will be two configurations with the same name and for the same plug-in.

Trusted

If the plug-in is trusted enough to be executed in an unprotected environment. See Section 22.1.2, “Plug-in properties” (page 134)

Immediate execution

If the plug-in is allowed to bypass the job queue and be executed immediately. See Section 22.1.2, “Plug-in properties” (page 134)

Click on **Save** to finish the registration or on **Cancel** to abort.

22.1.4. 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. See Section 22.4, “Plug-in configurations” (page 140). 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 the *.base file describing the BASE version 1 plug-in and enter the path and file name to the location of the executable.
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 bioassayset 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.

22.2. 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 plugin 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 22.4. 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 157).

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 plugin 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

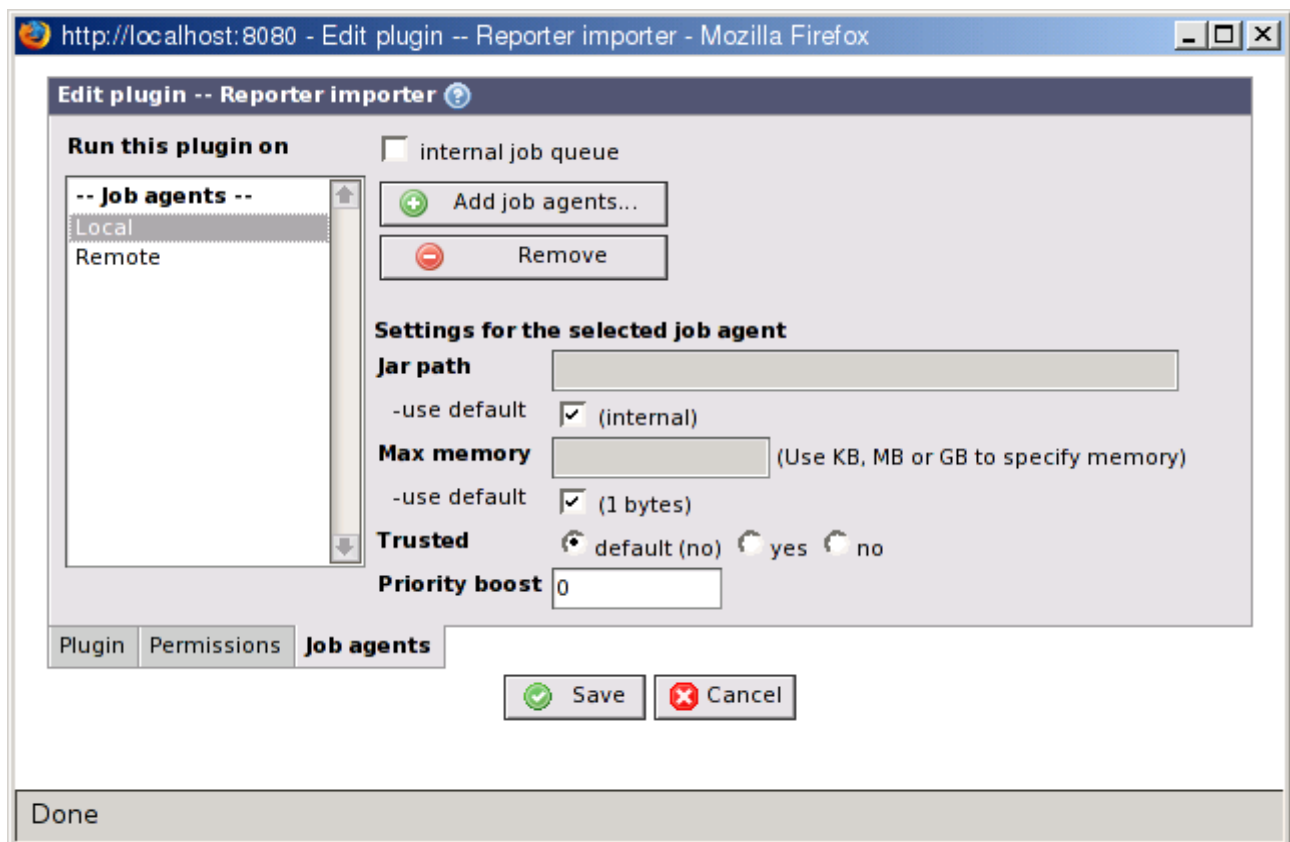
To make it easier for the server administrator to assign permissions, the plug-in developer can let the plug-in include a list of permissions that are needed. Plug-in developers are advised to only include the minimal set of permissions that are required for the plug-in to function. Click on the **Use requested permissions** button to give the plug-in the requested permissions.

22.3. Job agents

Job agents are used for executing plug-ins on external computers. Using job agents will free up resources on the BASE application server, and allow the server to concentrate on serving web pages. Job agents are optional and must be installed separately. See Section 21.2, “Installing job agents” (page 123) for more information about setting up job agents. This section assumes that at least one job agent is setup to serve your BASE installation.

A job agent will not execute a plug-in unless the administrator has configured the job agent to do so. 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 job agent route is described in Section 21.2, “Installing job agents” (page 123).

Figure 22.5. 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 plug-in 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 its own settings of these properties. If you leave the values unspecified the job agent will use the default values specified on the **Plugin** tab.

Jar path

The path on the external server to the JAR file containing the plug-in code. If not specified the same path as on the web server is used.

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 its 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 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.

22.4. 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 Base1PluginExecutor 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 Administrative Plugins Configurations page.

Click on the **New...** button to create a new configuration.

Figure 22.6. 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.

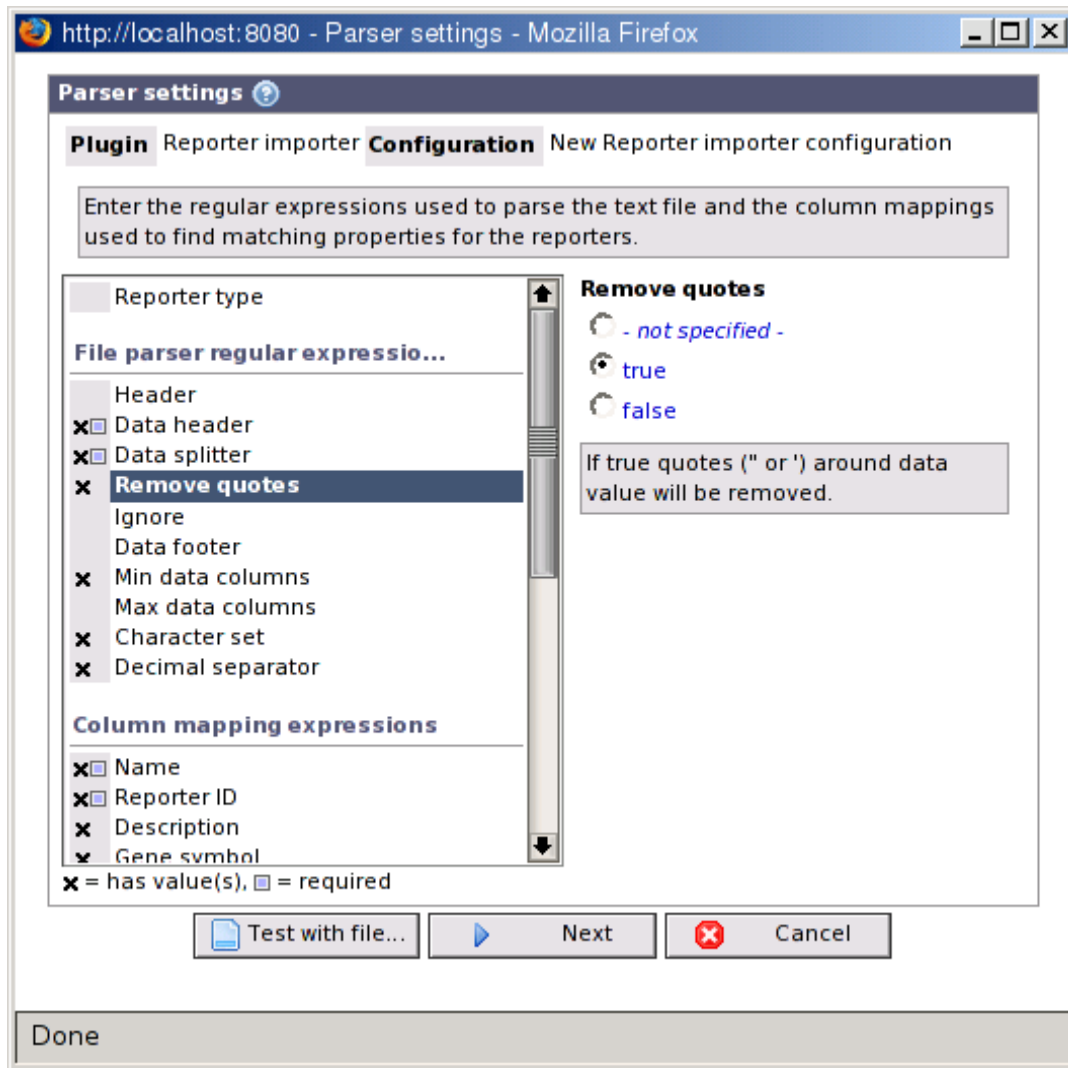
Use the **Save** button to save the configuration or the **Save and configure** button to save and then start the configuration wizard.

22.4.1. 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 22.7. 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.

22.4.2. 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. The BASE website also has a page where you can download additional configurations¹ not included in the main distribution.

Both the import and the export is started from the plug-in configuration list view: [Administrate Plugins](#) [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.

22.4.3. 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 22.7, “The plug-in configuration wizard”(page 142) 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 26.3, “Import plug-ins”(page 184) in Chapter 26, *Plug-in developer*(page 167) Clicking on the **Test with file** button opens the following dialog:

¹ http://base.thep.lu.se/chrome/site/doc/historical/admin/plugin_configuration/coreplugins.html

Figure 22.8. The test with file function

http://localhost:8080 - Test with file - Mozilla Firefox

Test with file

File to test:

Lines to parse: Character set:

Header regexp: Min data columns:

Data splitter regexp: Max data columns:

Ignore regexp: Remove quotes: ☒

Data header regexp:

- 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 22.9. Mapping columns from a file

Property	Mapping expression	File columns
Name	\Name\	
Reporter ID	=col('ID')	
Description		
Gene symbol		
Score		
Reporter type		
Species		
Cluster ID		

- *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 22.1, "Mapping expression examples" (page 147).

Table 22.1. 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 23. Extensions

The BASE web client has an extension system that can be used to add functionality to BASE. The extension system should not be confused with plug-ins (Chapter 22, *Plug-ins* (page 133)). Extensions are additions to the user interface, for example, additional menus, toolbar buttons, etc. Extensions are for the web interface only, they can not be used on job agents, or by plug-ins. Plug-ins, on the other hand, are used to perform some kind of work, for example, importing, exporting or analysing data, and are not restricted to be used from the web client.

More reading

- Chapter 27, *Extensions developer* (page 209).
- Section 29.6, “Extensions API” (page 276).

23.1. Installing extensions

The first step is to install the actual extension code on the web server. Extensions are always packaged as XML or JAR files. To install an extension put the XML or JAR file in the `<base-dir>/www/WEB-INF/extensions` folder. This is the only place where extensions can be installed.

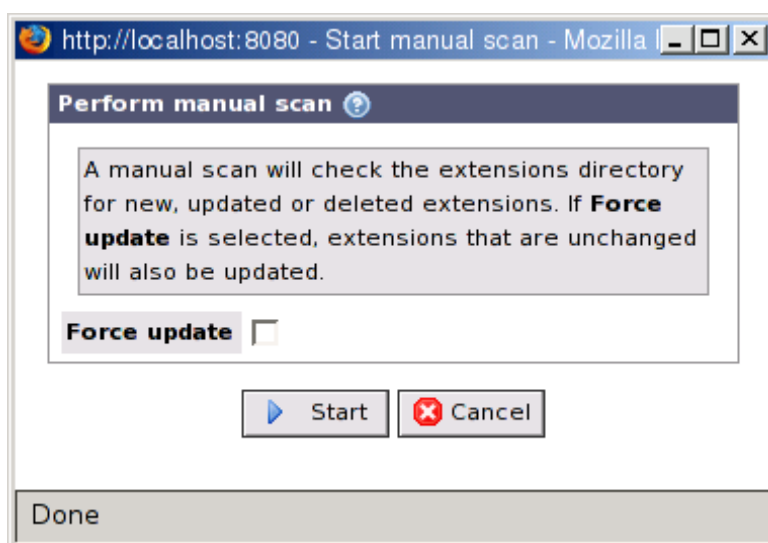
Make sure the extensions folder is writable by Tomcat

The extension 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 extensions system, 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.

If you have enabled automatic installation you just have to wait and the extensions will be installed and registered automatically. Otherwise, you have to do a manual scan, using the following instructions.

Go to Extensions Manual scan...

Figure 23.1. Manual scan

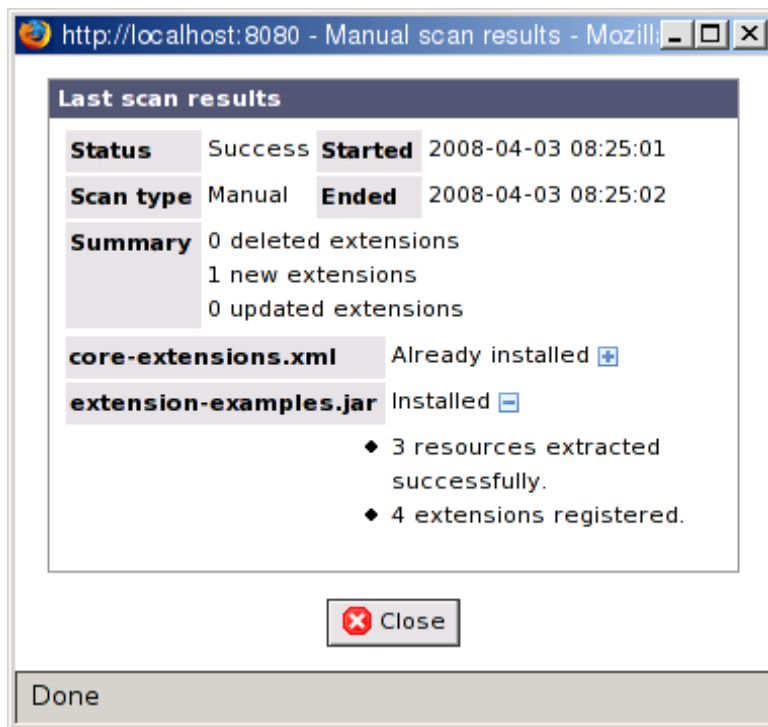


Performs a manual scan for new, updated or deleted extensions. If **Force update** is checked, extensions that have not been modified will also be re-registered. Leave this option unchecked unless there is any problem with the extension system.

Click on **Start** to start the scan.

If everything goes well you should get a report of what happened. The new XML or JAR file is hopefully listed as **Installed**. Click on the + icons to show more details.

Figure 23.2. Scan results



23.2. 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 `WEB-INF/extensions` 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 (`$CATALINA_HOME/lib`) since this is the only place where Tomcat look for compilers. The installation is easy. Simply copy `<base-dir>/bin/jar/base2-xjsp-compiler.jar` to `$CATALINA_HOME/lib` and restart Tomcat.

X-JSP is experimental

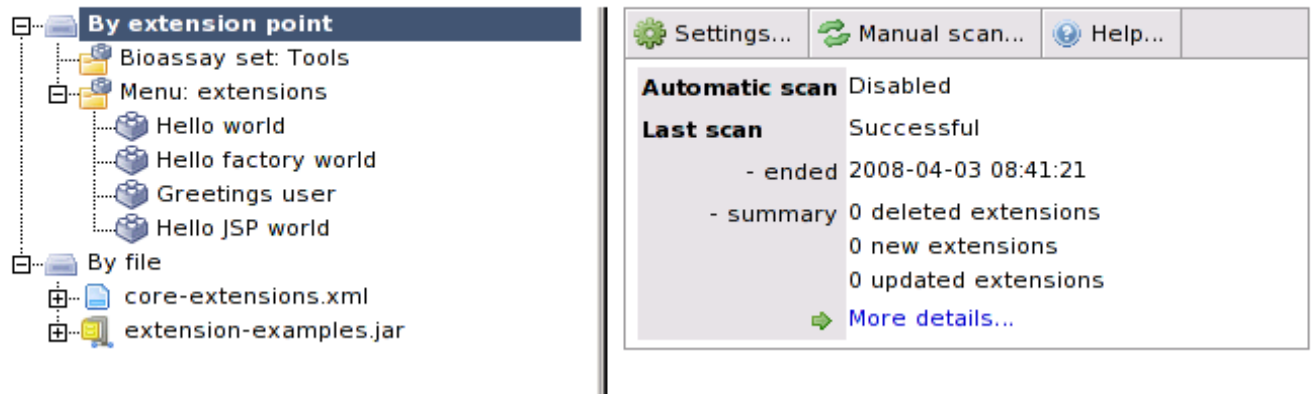
This is an experimental feature that depends on functionality in Tomcat. It may or may not work with future versions of Tomcat. The compiler will most likely not work with other servlet containers.

23.3. Configuring the extensions system

Go to Extensions Installed extensions to display an overview of all installed extensions.

Figure 23.3. Installed extensions

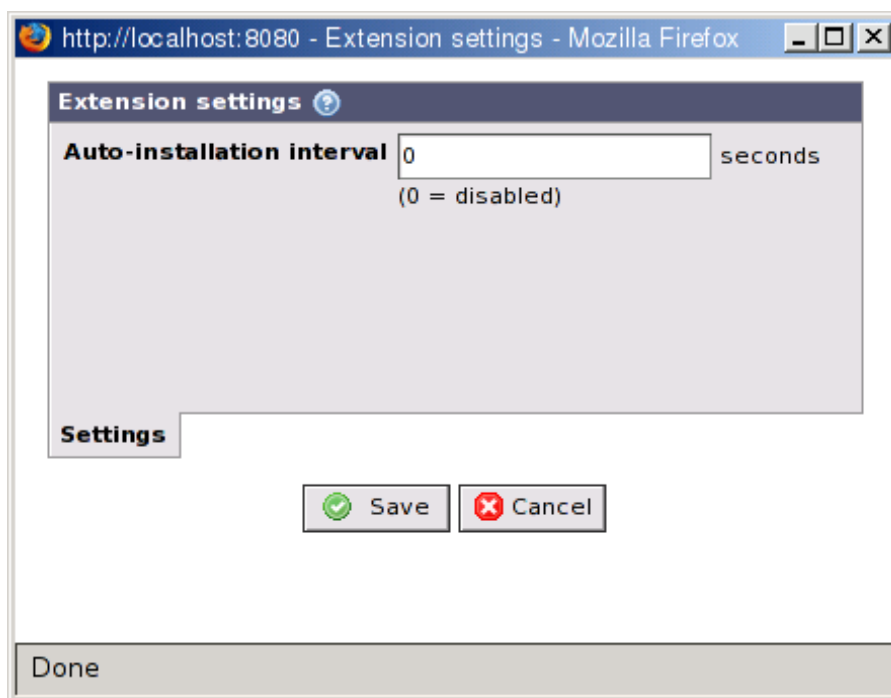
Installed extensions



The left-hand side of the screen shows you a tree with all installed 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.

23.3.1. Settings

Click on the **Settings...** button to display a popup dialog that allows you to change some global settings.

Figure 23.4. Extension settings

Auto-installation interval

The number of seconds between each check by the automatic installer. Enter 0 to disable automatic installation. The automatic installation performs the same steps as a manual scan with the **Force update** option unchecked. The default setting is to have the automatic installation disabled and we don't recommend enabling it in a production environment.

Click on **Save** to save the settings.

23.3.2. Disable/enable extensions

It is possible to disable specific extensions and/or entire extension points without uninstalling the XML or JAR file. When you click on an extension or extension point 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 will also have a **Disable** button. Click on that button to disable the extension or all extensions for an extension point. The button will change to **Enable** which lets you enable the extension (point) again.

Chapter 24. Account administration

Read Chapter 7, *Projects and the permission system* (page 37)

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.

24.1. Users administration

The user list is accessed with **Administrate Users** and from here are the users' account and contact information managed.

24.1.1. 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

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

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 then 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

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 D, *extended-properties.xml reference* (page 342)

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

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.

24.1.2. 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.

24.2. 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.

24.2.1. Edit group

The pop-up window where a group can be edited has two tabs, **Group** and **Members**.

Properties

Name

The name of the group.

Default

Mark this checkbox to let BASE automatically add new users as members to this group.

[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

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.

24.3. 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.

24.3.1. 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 lins 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.

24.3.2. 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

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

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

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.

24.4. 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 24.1.1, “Edit user” (page 152) and Section 24.2.1, “Edit group” (page 155) 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`.

24.4.1. Edit quota

The edit window has two tabs, one with information about the quota and one where the limits are defined.

Properties

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

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**.

24.4.2. 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.

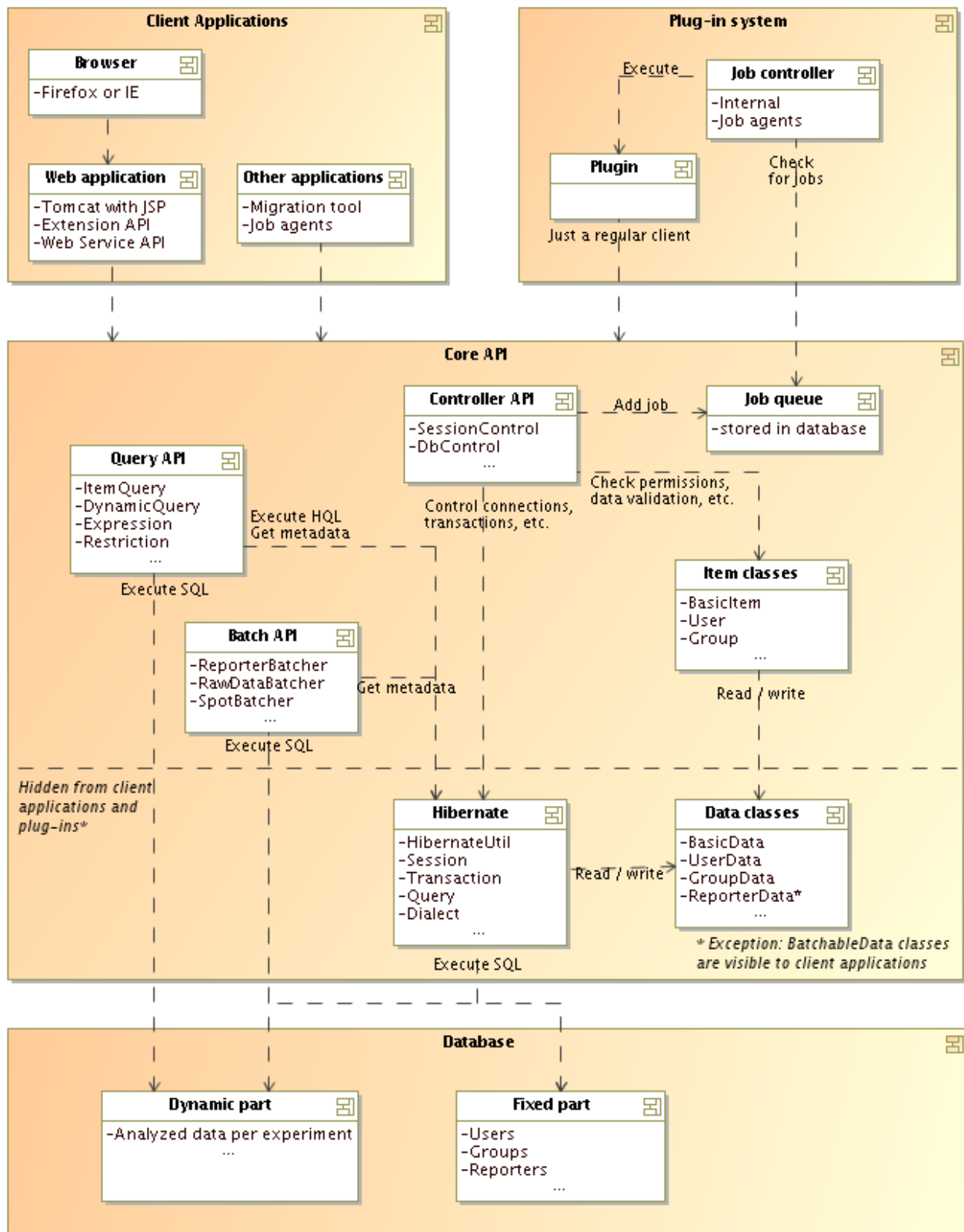
Per group

Gives an overview of the disk usage per group, with the same functionality as the per user overview.

Part IV. Developer documentation

Chapter 25. 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 25.1. Overview of the BASE application

25.1. 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, extracts and labeled 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 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 29.5, “Analysis and the Dynamic and Batch API:s” (page 276)

25.2. 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 31.3.4, “Data-layer rules” (page 304)
- www.hibernate.org²

25.3. 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 29.5, “Analysis and the Dynamic and Batch API:s” (page 276)
- Section 31.3.6, “Batch-class rules” (page 318)
- Section 31.4.6, “Batch operations” (page 319)

25.4. 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 31.3.4, “Data-layer rules” (page 304)
- Section 31.3.5, “Item-class rules” (page 318)
- Section 31.3.6, “Batch-class rules” (page 318)
- Section 31.4.2, “Access permissions” (page 319)
- Section 31.4.3, “Data validation” (page 319)
- Section 31.4.6, “Batch operations” (page 319)

25.5. 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 29.4, “The Query API” (page 276)

25.6. The Controller API

The Controller API is the very heart of the Base 2 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 31.4, “Internals of the Core API” (page 318)

25.7. 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 26, *Plug-in developer* (page 167)
- Section 31.4.8, “Plugin execution / job queue” (page 319)

25.8. Client applications

Client applications are application that use the BASE Core API. The current web application is built with Java Server Pages (JSP). It is supported by several application servers but we have only tested it

with Tomcat. Other client applications are the external job agents that executes plug-ins on separate servers, and the migration tool that migrates data from a BASE 1.2.x installation to BASE 2.

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 28, *Web services* (page 223)
- Chapter 27, *Extensions developer* (page 209)
- The BASE plug-ins site³ also has examples of extensions and web services implementations.

Chapter 26. Plug-in developer

26.1. How to organize your plug-in project

26.1.1. 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 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/  
pluginname/bin/  
pluginname/lib/  
pluginname/src/org/company/  
pluginname/META-INF/
```

The *bin/* directory is empty to start with. It will contain the compiled code. In the *lib/* directory you should put *BASE2Core.jar* and other library files your plug-in depends on. 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 best functionality.

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 26.1. 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>
  </path>

  <!-- main target -->
  <target
    name="build.plugin"
    description="Compiles the plug-in and put in jar"
  >
    <javac
      encoding="ISO-8859-1"
      srcdir="${src}"
      destdir="${bin}"
      classpathref="classpath">
    </javac>
    <jar
      jarfile="${plugin.name}.jar"
      basedir="bin"
      manifest="META-INF/MANIFEST.MF"
    >
      <!--Include this to add required files for auto registration wizard-->
      <metainf file="META-INF/base-plugins.xml"></metainf>
      <metainf file="META-INF/base-configurations.xml"></metainf>
    </jar>

  </target>
</project>
```

If your plug-in depends on other JAR files than the `BASE2Core.jar` you must create a file called `MANIFEST.MF` in the project `META-INF` directory. 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: OtherJar.jar ASecondJar.jar
```

See also Section 26.8, “How BASE load plug-in classes”(page 207) for more information regarding class loading when a plug-in depends on a external JAR files.

If your plug-in should support registration with the auto-installation wizard it is a good idea to add the `metainf` tags. This will add the two files `META-INF/base-plugins.xml` and `META-INF/base-configurations.xml` to the `META-INF` directory in the JAR file. The two files contains information about your plug-in and BASE can automatically find and extract information from those. See Section 26.1.3, “Make the plug-in compatible with the auto-installation wizard”(page 169) 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 including the dependent JAR files (if any). Place all files together in the same directory. For more information read Section 22.1, “Installing plug-ins” (page 133).

26.1.2. With Eclipse

If somebody is willing to add information to this chapter please send us a note or some written text to put here. Otherwise, this chapter will be removed.

26.1.3. 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 22.1.3, “Automatic installation of plug-ins” (page 136).

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/base-plugins.xml`. This file contains some information about the plug-in(s). Here is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plugins SYSTEM "base-plugins.dtd" >
<plugins jarname="jarfile.jar">
  <pluginclass classname="se.lu.thep.PluginClass">
    <minbaseversion>2.4</minbaseversion>
    <hasconfigurations/>
  </pluginclass>
  .
  .
  .
</plugins>
```

The first two lines should be the same in all `base-plugins.xml` files. The rest of the tags are described in following list.

`<plugins>`

This is the root element in the XML-file and must have the attribute `jarname` set. The value must be the same as name of the JAR file the plug-in is located in or the auto-installation wizard will fail with an error.

`<pluginclass>`

This tag defines information about a plug-in in the JAR file. The `classname` attribute needs to have the full classname of the plug-in's main class. There can be one or several of this element inside the `<plugins>` tag, which means that there should be one element for each plug-in in the JAR file.

`<minbaseversion>`

A required child element in `<pluginclass>`. This defines from which version of BASE the plug-in can be used. The plug-in will not be included in auto installer if the BASE-version is lower then the value of this tag. The format of the value in this tag should be (numeric) *majorversion.minorversion*. It is not possible to check the bugfix or revision numbers.

`<hasconfigurations>`

A required child element in `<pluginclass>`. This should tell if there are plug-in configurations shipped with the plug-in. Setting the value to *yes* will indicate that there are available configurations in the JAR file. This can be left as an empty tag if no configurations are available for import.

Configurations shipped with a JAR file should be exported with the plug-in configuration exporter in BASE. The exported file must be given the name `base-configurations.xml` and be placed in `META-INF/`.

26.2. The Plug-in API

26.2.1. The main plug-in interfaces

The Base2 core 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 About getAbout();
```

Return information about the plug-in, i.e. the name, version, and a short description about what the plug-in does. The `About` object also has fields for naming the author and various other contact information. The returned information is copied by the core at installation time into the database. The only required information is the name of the plug-in. All other fields may have null values.

Example 26.2. A typical implementation stores this information in a static field

```
private static final About about = new AboutImpl(
    (
        "Spot images creator",
        "Converts a full-size scanned image into smaller preview jpg " +
        "images for each individual spot.",
        "2.0",
        "2006, Department of Theoretical Physics, Lund University",
        null,
        "base@thep.lu.se",
        "http://base.thep.lu.se"
    )
);

public About getAbout()
{
    return about;
}
```

```
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 26.3. 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 22.2, “Plug-in permissions” (page 137) 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 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 26.4. 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 can 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 can be null, but if it is not we recommend all plug-ins to use it. However, it should be used sparingly, since each call to set the progress results in a database update. If the execution involves several thousands of items it is a bad idea to update the progress after processing each one of them. 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 the `Response.setDone()` or the `Response.setError()` 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 26.5. Here is a skeleton that we recommend each plug-in to use in its implementation of 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 26.6. 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 `SpotImageCreator` 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, kind of hardcoded, approach can also be used for other plug-ins, but then it usually requires modification of the client application as well.

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 corresponds 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 26.7. 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.

Note

The contract of this method was changed in in BASE 2.4 to allow warning and error level message. Prior to BASE 2.4 all messages were treated as error message. We recommend that existing plug-ins are updated to throw exception to indicate error-level messages since the default is to not show warning messages to users.

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 26.8. 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 26.9. 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 26.10. 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
            response.setDone("Job configuration complete", Job.ExecutionTime.SHORT);
            // TODO - check file size to make a better estimate of execution time
        }
    }
    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 plugin 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.

26.2.2. 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()`, `Plugin.requiresConfiguration()` and `Plugin.getAbout()` 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 22.4.1, “Configuring plug-in configurations” (page 141)). 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 181).
 - `Response.setContinue()`: The parameters are correct but the plug-in wants more parameters. The process continues with step 3 (page 181) 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 181). 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 21.2, “Installing job agents” (page 123). 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 or the `Response.setError()` to report errors.
4. In both cases the `Plugin.done()` method is called and the plug-in instance is discarded.

26.2.3. 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` 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 you should reload the page 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();
```

26.3. Import plug-ins

A plugin becomes an import plugin simply by returning `Plugin.MainType.IMPORT` from the `Plugin.getMainType()` method.

26.3.1. 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 a regexp.

The `AbstractFlatFileImporter` 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);
    }
}
```

Note that the input stream doesn't have to be a text file. 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.

```
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.

Try casting to `ImportInputStream`

As of BASE 2.9 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));
```

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.

26.3.2. 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 `Plugin.getAbout()` method. See the section called “The `net.sf.basedb.core.plugin.Plugin` interface” (page 170) for more information.
- Implement the `InteractivePlugin` methods. See the section called “The `net.sf.basedb.core.plugin.InteractivePlugin` interface” (page 173) 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 getConfigurePluginParameters(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.

26.4. 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.

26.4.1. 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 182).

- 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.

26.4.2. 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 `AbstractExporterPlugin` 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.

26.5. 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()` must include: `[Item.BIOASSAYSET, Type.ITEM]` since this is the main place where the web client looks for analysis plug-ins. 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.

```

private static final Set<GuiContext> guiContexts =
    Collections.singleton(new GuiContext(Item.BIOASSAYSET, GuiContext.Type.ITEM));

public Set<GuiContext> getGuiContexts()
{
    return guiContexts;
}

```

If the plugin 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 plugin 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 getConfigureJob(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;
}
```

```

    );
}
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 29.5, “Analysis and the Dynamic and Batch API:s”(page 276) for more examples of using the analysis API.

26.5.1. The AbstractAnalysisPlugin class

This class is an abstract base class. It is a useful class for most analysis plug-ins 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()`

26.5.2. 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.

26.6. Other plug-ins

26.6.1. Authentication plug-ins

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 to 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:

- 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*.

Note

Prior to BASE 2.4 it was hardcoded to add the new user to the *Users* role only.

- 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.

Note

The **Encrypt password** option that is available on the login page does not work with external authentication. The simple reason is that the password is encrypted with a one-way algorithm making it impossible to call `Authenticator.authenticate()`.

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.

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, 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` instead.
- `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 337).

`auth.driver`

The class name of the authentication plug-in.

`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.

26.6.2. 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 C, *base.config reference* (page 336).

`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.

26.6.3. File unpacker plug-ins

The BASE web client has integrated support for unpacking of compressed files. See Section 8.2.1, “Upload a new file” (page 44). 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 except `Plugin.getAbout()`. 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,
                 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`.
- 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 `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 182).

- 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.

26.6.4. 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).

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.

26.6.5. File validator and metadata reader plug-ins

See also

- Section 29.3.1, “Using files to store data” (page 269)
- Section 29.2.8, “Experimental platforms” (page 244)

In those cases where files are used to store data instead of importing it to the database, BASE can use plug-ins to check that the supplied files are valid and also to extract metadata from the files. For example, the `net.sf.basedb.core.filehandler.CelFileHandler` 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.

The validator and metadata reader plug-ins are not regular plug-ins (ie. they don't have to implement the `Plugin` interface). This means that you don't have to mess with configuration or job parameters.

Validator plug-ins must implement the `net.sf.basedb.core.filehandler.DataFileHandler` and `net.sf.basedb.core.filehandler.DataValidator` interfaces. Metadata reader plug-ins should implement the `net.sf.basedb.core.filehandler.DataFileHandler` and `net.sf.basedb.core.filehandler.DataFileMetadataReader` interfaces.

Note

Meta data extraction can only be done if the file has first been validated. We recommend that metadata reader plug-ins also takes the role as validator plug-ins. This will make BASE re-use the same object instance and the file doesn't have to be parsed twice.

Always extend the `net.sf.basedb.core.filehandler.AbstractDataFileHandler` class

We consider the mentioned interface to be part of the public API only from the caller side, not from the implementor side. Thus, we may add methods to those interfaces in the future without prior notice. The `AbstractDataFileHandler` will provide default implementations of the new methods in order to not break existing plug-ins.

Methods in the `DataFileHandler` interface

```
public void setFile(FileSetMember member);
```

Sets the file that is going to be validated or used for metadata extraction. If the same plug-in can be used for validating more than one type of file, this method will be called one time for each file that is present in the file set.

```
public void setItem(FileStoreEnabled item);
```

Sets the item that the files belong to. This method is only called once.

Methods in the `DataFileValidator` interface

```
public void validate(DbControl dc)

throws InvalidDataException, InvalidRelationException;
```

Validate the file. The file is valid if this method returns successfully. If the file is not valid an `InvalidDataException` should be thrown. Note that BASE will still accept the file, but will

indicate the failure with a flag and also keep the message of the exception in the database to remind the user of the failure.

The `InvalidRelationException` should be used to indicate a partial success/partial failure, where the file as such is a valid file, but in relation to other files it is not. For example, we may assign a valid CEL file to a raw bioassay, but the chip type doesn't match the chip type of the CDF file of the related array design. This exception will also allow metadata to be extracted from the file.

Methods in the `DataFileMetadataReader` interface

```
public void extractMetadata(DbControl dc);
```

Extract metadata from the file. It is up to the plug-in to decide what to extract and how to store it. The `CelFileHandler` will, for example, extract headers and the number of spots from the file and store it with the raw bioassay.

```
public void resetMetadata(DbControl dc);
```

Remove all metadata that the plug-in usually can extract. This method is called if a file is unlinked from an item or if the validation fails. It is important that the plug-in cleans up everything so that data from a previous file doesn't remain in the database.

Methods in the `AbstractDataFileHandler` class

```
public FileStoreEnabled getItem();
```

Get the item that was previously added to `setItem()`

```
public FileSetMember getMember(String dataFileTypeId);
```

Get a file that was previously added to `setFile()`. The `dataFileTypeId` is the external ID of the `DataFileType`.

26.6.6. 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.

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 339)). 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.

```
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.

26.7. Enable support for aborting 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.

Here is a general outline for 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");
    }
}
```

Another signal handler implementation is the `ProgressReporterSignalHandler`. See that javadoc for information about how to use it. For more information about the signalling system as a whole, see Section 29.3.2, “Sending signals (to plug-ins)” (page 274).

26.8. How BASE load plug-in classes

We recommend that plug-in JAR files are installed outside the web server's classpath. If you are using Tomcat this means that you should not install the plug-in in the `<base-dir>/www/WEB-INF/lib` directory or any other directory where the web server keeps its classes. The rest of the information in this section only applies to plug-ins that have been installed following this restriction.

If the above recommendation has been followed BASE will use its own classloader to load the plug-in classes. This has 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 unload the old code from the web interface first.
- Plug-ins may use its 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 its own JAR file, BASE core classes, Java system classes and from JAR files listed in the plug-in's `MANIFEST.MF` file. See Section 26.1, “How to organize your plug-in project” (page 167).
- 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(Plugin.class, null, sc, null, null);

// Not ok; fails with ClassCastException
MyOtherPlugin other = def.newInstance(MyOtherPlugin.class, null, sc, null, null);
```

The reason that the second call fails is that 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 first call always 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.

A third 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-5.5-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.

26.9. 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 code¹ from the BASE plug-ins website.

¹ <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.examples.plugins>

Chapter 27. Extensions developer

27.1. Overview

The BASE web client 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 **Extensions Installed extensions** 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 23, *Extensions* (page 148).

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 needed when the extension needs to execute custom-made code or use custom resources such as icons, css stylesheets, or JSP files.

More reading

- Chapter 23, *Extensions* (page 148).
- Section 29.6, “Extensions API” (page 276).

27.1.1. Download code examples

The code examples in this chapter can be downloaded from The BASE plug-ins site¹.

27.1.2. Terminology

Extension point

An extensions point is a place in the BASE web client interface where it is possible to extend the GUI 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`. The extension point also defines an `Action` class that all extensions must "implement".

Extension

An extensions is a custom addition to the BASE web client interface. 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

¹ <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.examples.extensions>

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.

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.

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.

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.

27.2. 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 `/WEB-INF/extensions` directory. The filename must end with `.xml`. If you have enabled automatic installation just wait a few seconds and the extension will be installed automatically. Otherwise you may have to do a manual scan: (Extensions Manual scan...).

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 Sun².

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.

27.3. 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;
```

² <http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html>

```
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>
```



```
</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 needs to be thread-safe if the factories re-use the same objects.

27.4. Custom images, JSP files, and other resources

Some times your extension may need other resources. It can for 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 `WEB-INF/extensions` 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 27.4.2, "X-JSP files" (page 216) for more information.

27.4.1. 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.

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.

27.4.2. 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 `WEB-INF/extensions` 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 23.2, "Installing the X-JSP compiler" (page 149).

27.5. 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(InvokationContext 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);
        }
    }
}
```

27.6. 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
      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 Sun³.

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

³ <http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html>

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.

It is finally time to write the JSP code that actually uses the extension point. It is usually not very complicated. Here is an exemple 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);
}
%>
```

27.7. 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. Since BASE 2.13 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 (since BASE 2.10) 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.helloservletworld"
  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.

Chapter 28. 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.

28.1. 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.

28.1.1. 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.

`BioAssaySetService`, `BioAssaySetClient`

Services related to bioassay sets. Get access to the data in a bioassay set that has been previously exported to a file.

`RawBioAssayService`, `RawBioAssayClient`

Services related to raw bioassays. Find out which raw data files that are present and download them.

`ArrayDesignService`, `ArrayDesignClient`

Services related to array design. Find out which data files that are present and download them.

`AnnotationTypeService`, `AnnotationTypeClient`

Services related to annotation types. Find out which annotation types that can be used for different types of items.

28.2. Client development

How to develop client applications for the web services in BASE is, of course, 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, false);

// 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`.

28.2.1. 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 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
```

¹ <http://baseplugins.thep.lu.se/wiki/net.sf.basedb.examples.webservices>

² http://ws.apache.org/axis2/1_0/mtom-guide.html

```

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 programlisting, 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.

```

// 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();
}

```

28.3. 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 28.1. 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>

```

28.3.1. 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 serviceName="SessionService"/>
```

28.4. 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 29. API overview (how to use and code examples)

29.1. The Public API of BASE

Not all public classes and methods in the `BASE2Core.jar` 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

This will maximize the chance that your plug-in 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 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.

See Appendix K, *API changes that may affect backwards compatibility* (page 358) to read more about changes that have been introduced by each release.

29.1.1. 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

¹ <http://base.thep.lu.se/chrome/site/doc/api/index.html>

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.

Source code compatibility

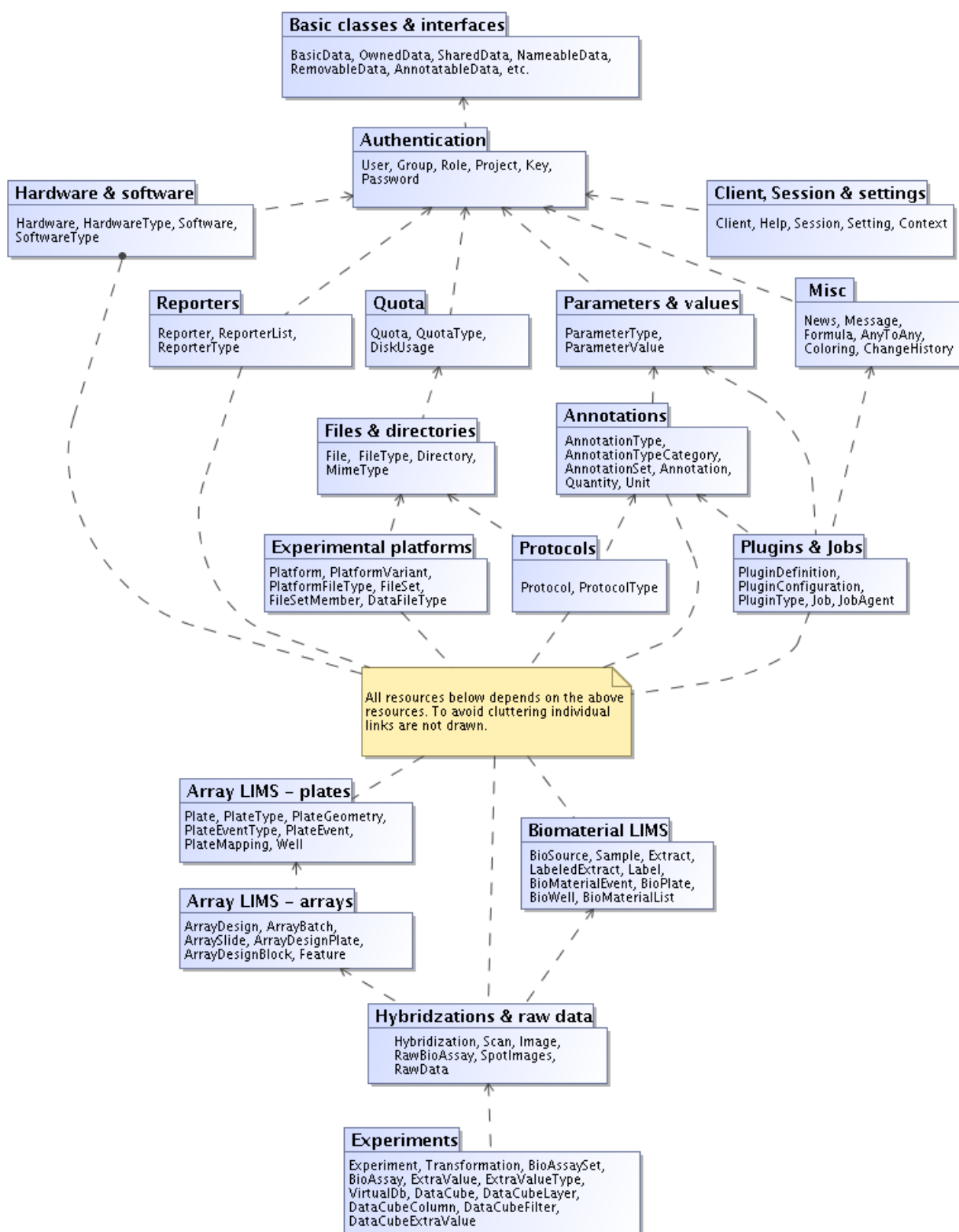
This is not an important matter and is not always possible to achieve. In most cases, the problems are easy to fix. Example:

- Adding a class may break a plug-in or client that import classes with `. *` if the same class name exists in another package.

29.2. The database schema and 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.

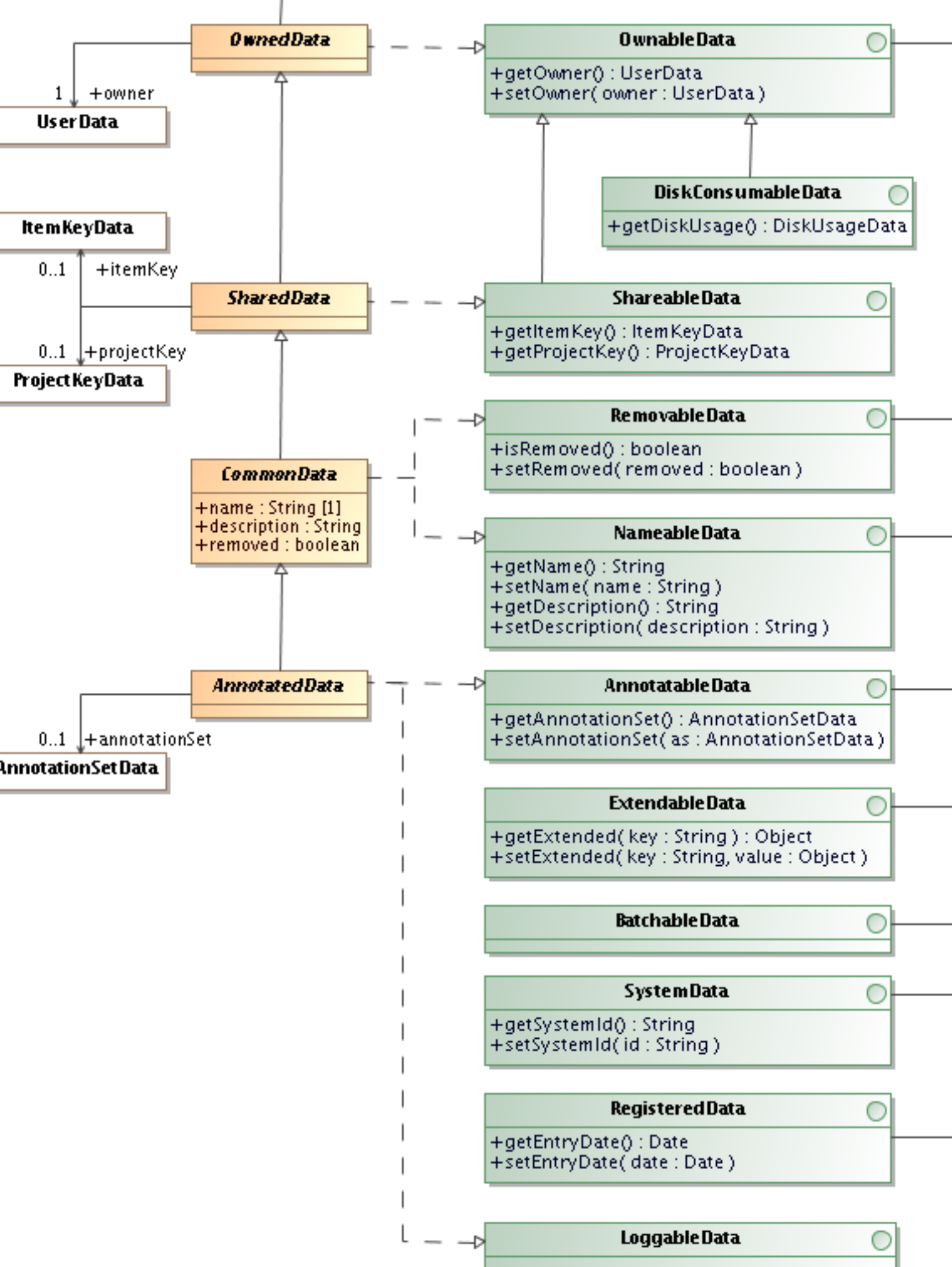
Figure 29.1. Data layer overview



29.2.1. 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

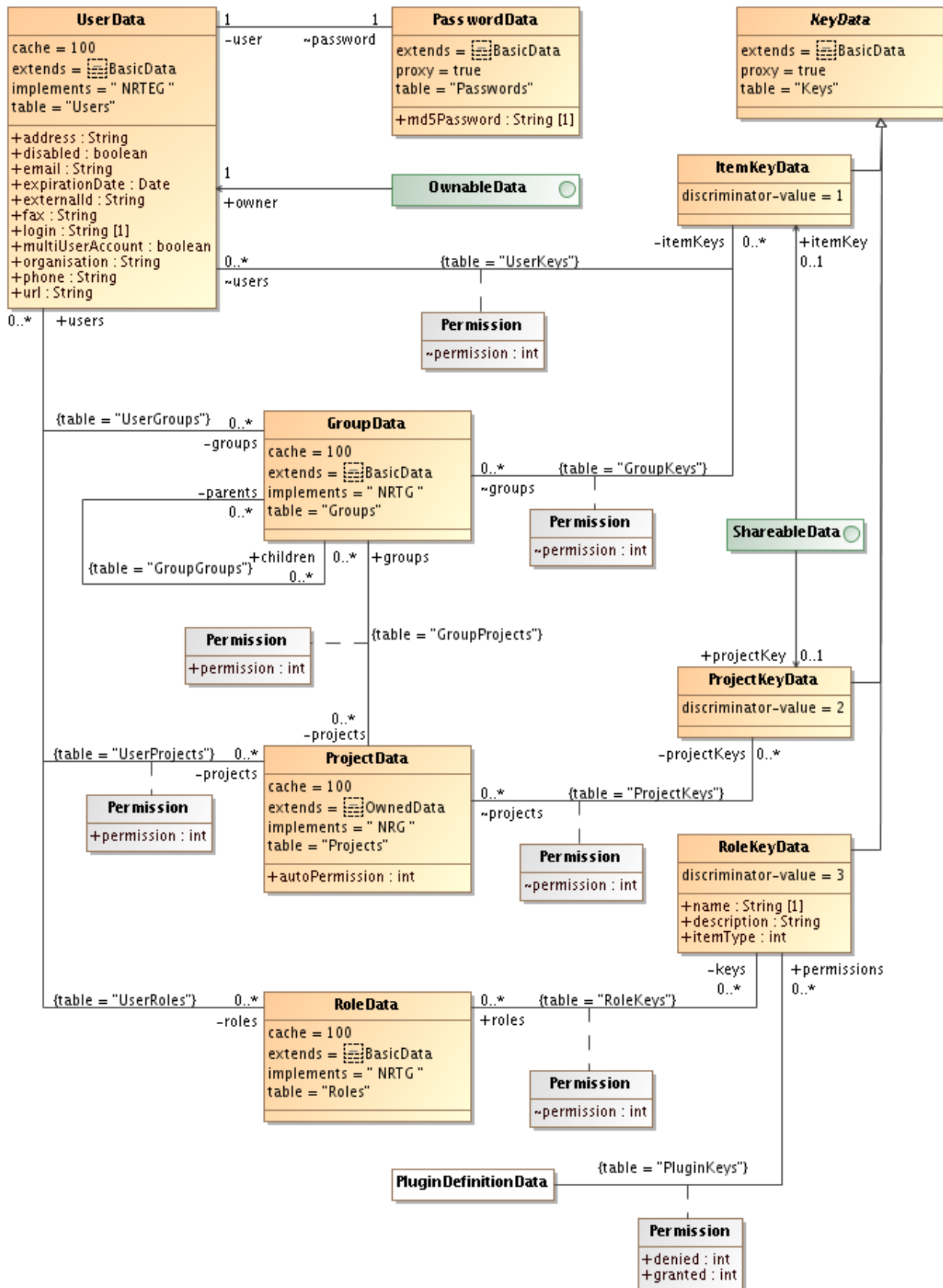
This is a tagging interface that indicates that the `DbLogManagerFactory` logging implementation should log changes made to items that implements it.

29.2.2. 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.

UML diagram

Figure 29.3. 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 and projects

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. 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.

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 added to that project with the permission given by the `autoPermission` property.

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

Value	Permission
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.

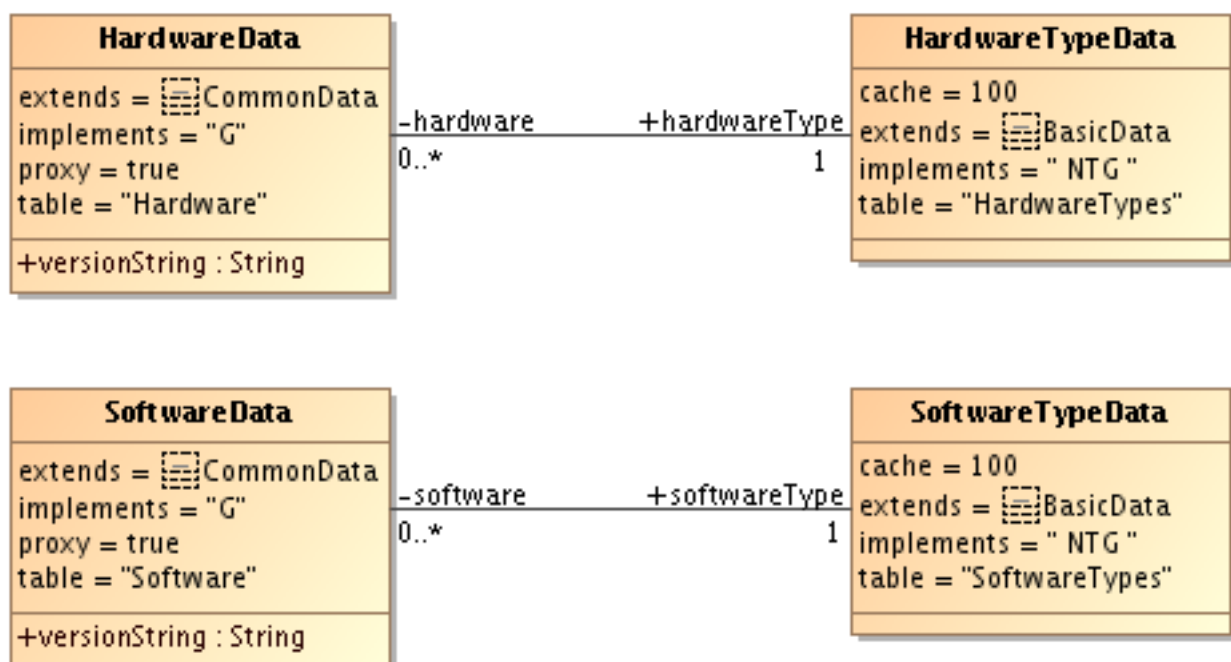
The RESTRICTED_WRITE permission is in most cases the same as the WRITE permission. So far the RESTRICTED_WRITE permission is only given to users to their own UserData object so they can change their address and other contact information, but not quota, expiration date and other administrative information.

29.2.3. Hardware and software

This section gives an overview of hardware and software in BASE.

UML diagram

Figure 29.4. Hardware and software



Hardware and software

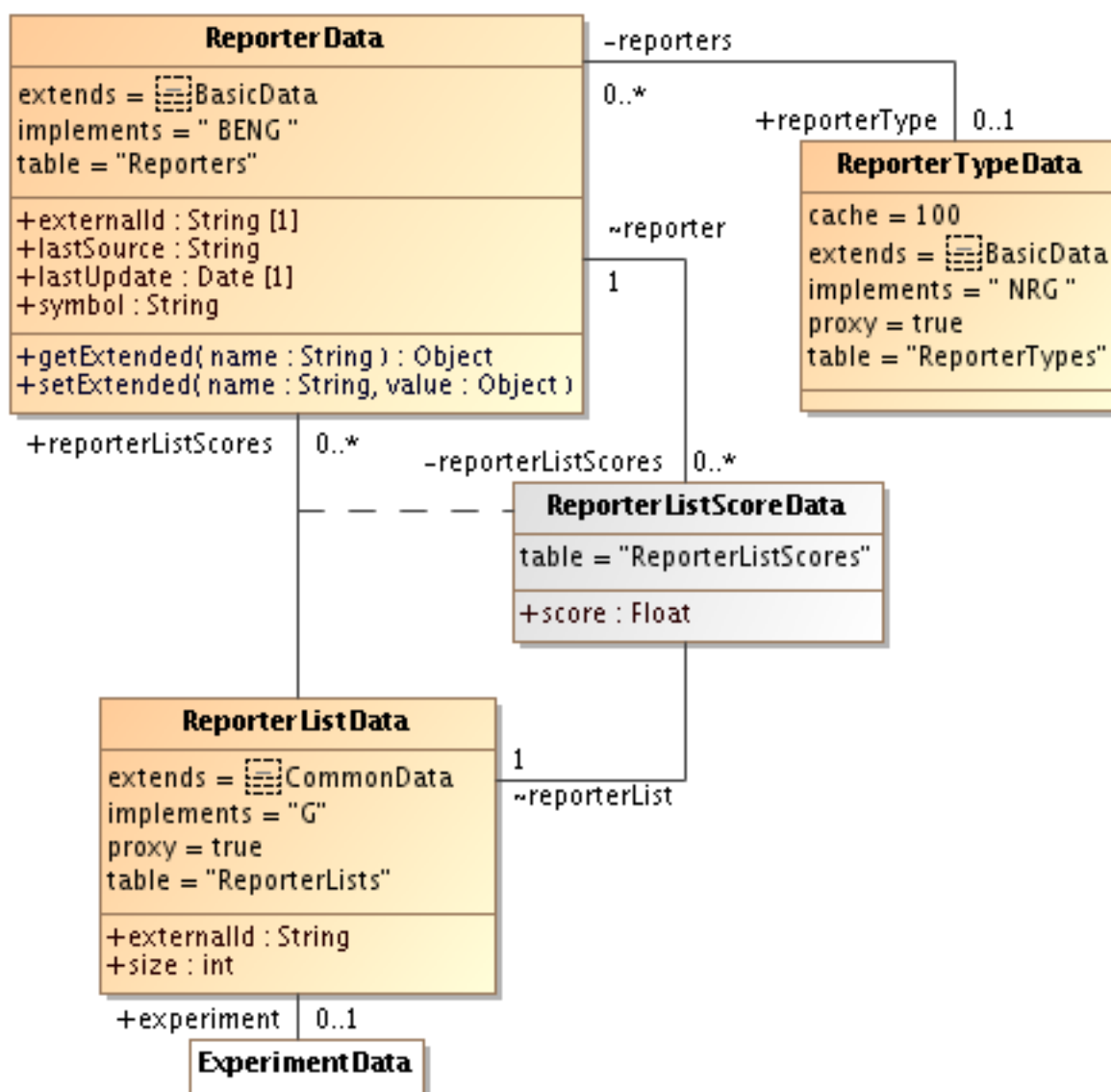
BASE is pre-installed with a set of hardware and software types. They are typically used to filter the registered hardware and software depending on what a user is doing. For example, when adding raw data to BASE a user can select a scanner. The GUI will display the hardware that has been registered as *scanner* hardware types. Other hardware types are *hybridization station* and *print robot*. An administrator may register more hardware and software types.

29.2.4. Reporters

This section gives an overview of hardware and software in BASE.

UML diagram

Figure 29.5. 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 D, *extended-properties.xml reference* (page 342) 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

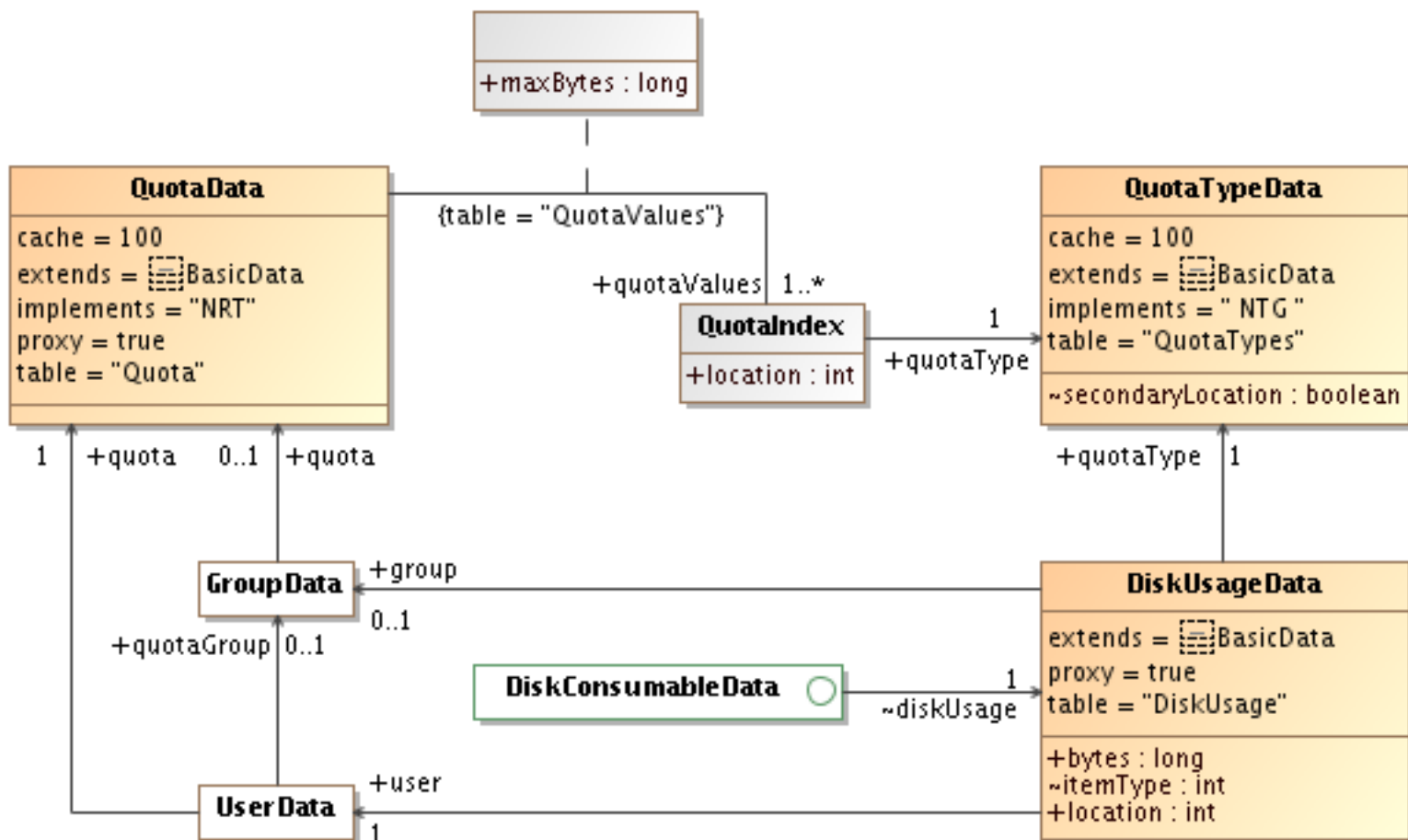
Reporter lists can be used to group reporters that are somehow related to each other. This could for example be a list of interesting reporters found in the analysis of an experiment. Each reporter in the list may optionally be assigned a score. The meaning of the score value is not interpreted by BASE.

29.2.5. Quota and disk usage

This section gives an overview of quota system in BASE and how the disk usage is kept track of.

UML diagram

Figure 29.6. 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

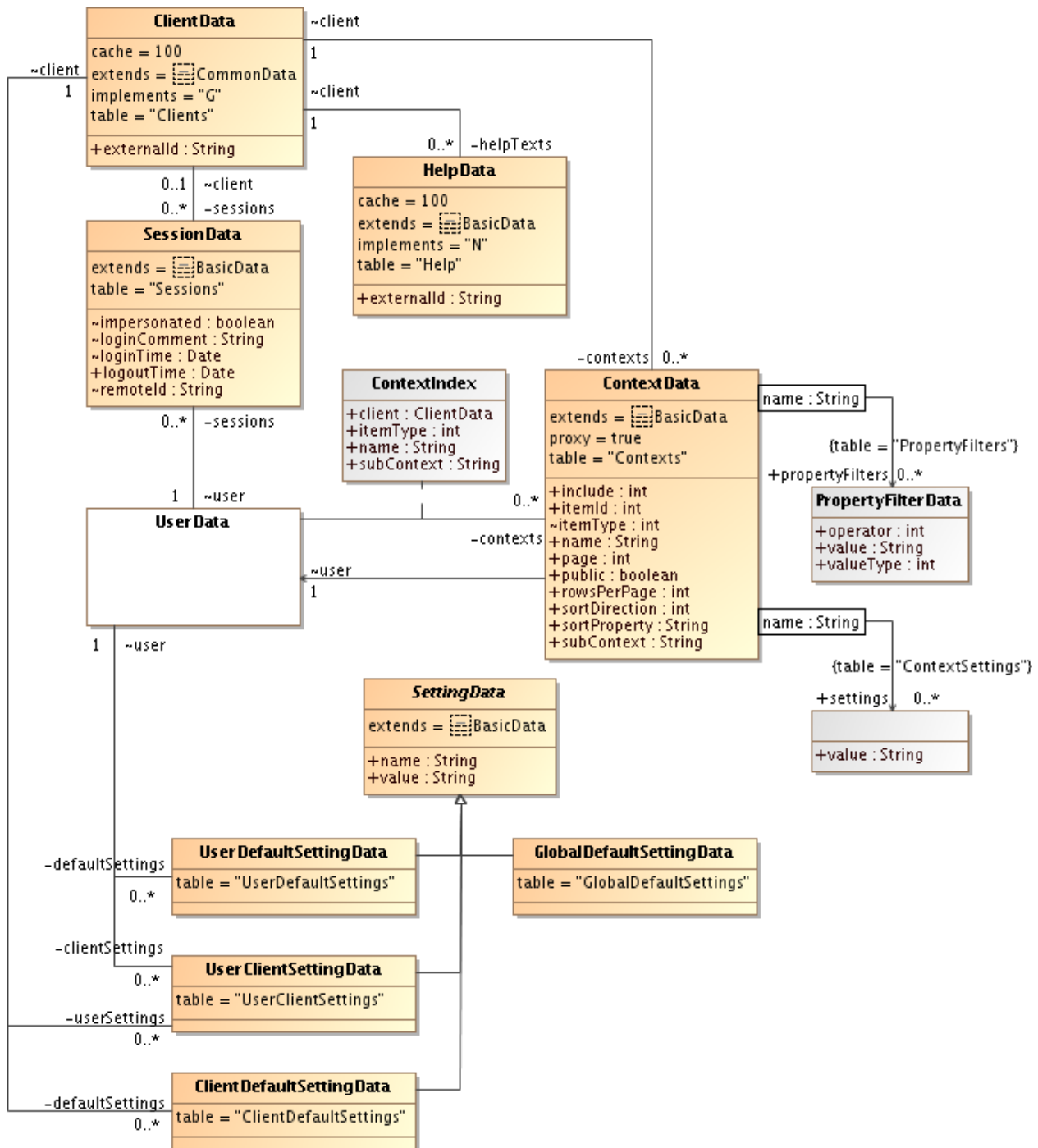
A `DiskConsumableData` (for example a file) item is automatically linked to a `DiskUsageData` item. This holds information about the number of bytes, the location and quota type the item uses. It also holds information about which user and group (optional) that should be charged for the disk usage. The user is always the owner of the item.

29.2.6. Client, session and settings

This section gives an overview of hardware and software in BASE.

UML diagram

Figure 29.7. 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.

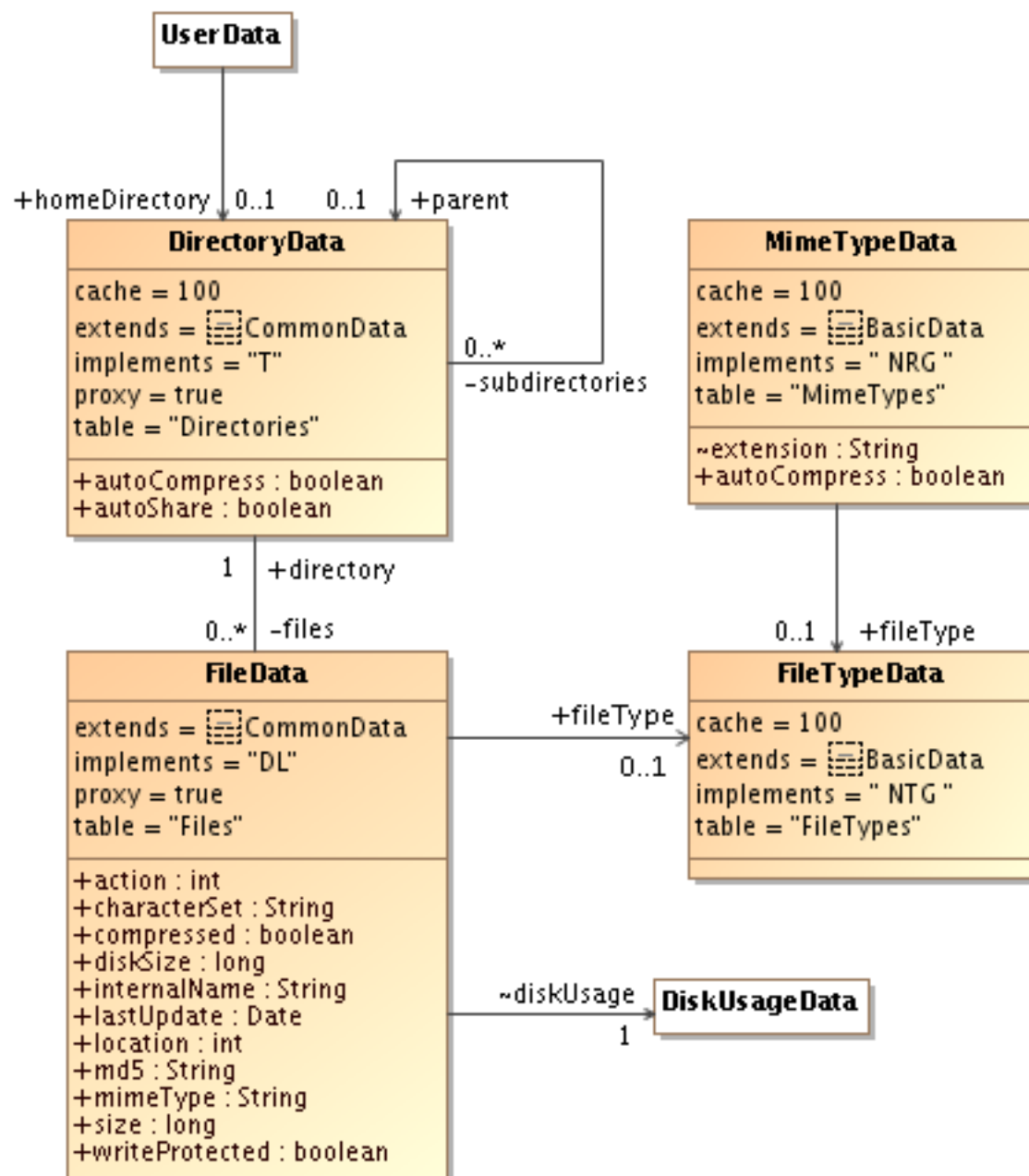
Context-sensitive settings are only accessible if a client application has been registered. The settings may be named to make it possible to store several presets and to quickly switch between them. In any case, BASE maintains a current default setting with an empty name. An administrator may mark a named setting as public to allow other users to use it.

29.2.7. Files and directories

This section covers the details of the BASE file system.

UML diagram

Figure 29.8. Files and directories



Description

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 `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.

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 339) and Section 26.6.2, “Secondary file storage plugins” (page 199) 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 `diskSize` 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 `FileTypeData` class holds information about file types. It is used only to make it easier for users to organise their files.

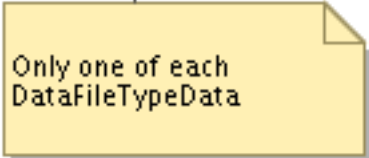
The `MimeTypeData` is used to register mime types and map them to file extensions. The information is only used to lookup values when needed. Given the filename we can set the `File.mimeType` and `File.fileType` properties. The MIME type is also used to decide if a file should be stored in a compressed format or not. The extension of a MIME type must be unique. Extensions should be registered without a dot, ie `html`, not `.html`.

29.2.8. Experimental platforms

This section gives an overview of experimental platforms and how they are used to enable data storage in files instead of in the database.

See also

- Section 29.3.1, “Using files to store data” (page 269)
- Section E.1, “Default platforms/variants installed with BASE” (page 346)
- Section 26.6.5, “File validator and metadata reader plug-ins” (page 203)



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 it's 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 `FileType` 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.

FileStoreEnabled items and data files

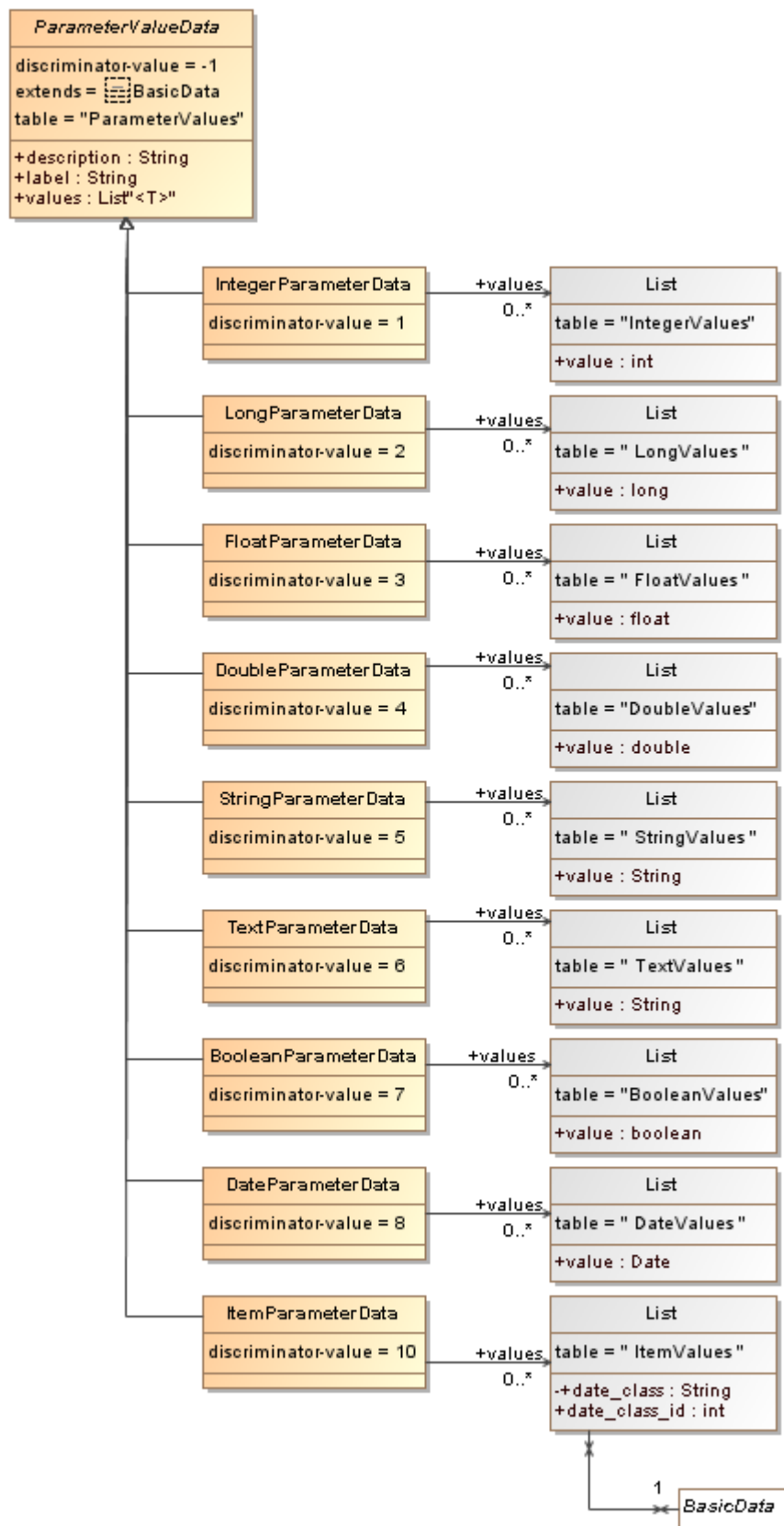
An item must implement the `FileStoreEnabledData` interface to be able to store data in files instead of in the database. The interface creates a link to a `FileSetData` object, which can hold several `FileSetMemberData` items. Each member points to specific `FileData` item. A file set can only store one file of each `DataFileTypeData`.

29.2.9. 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.

UML diagram

Figure 29.10. Parameters



Parameters

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. 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();
```

29.2.10. 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. 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 $m\leq$ or $^{\circ}\text{C}$. 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 $>5\text{mg}$ 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

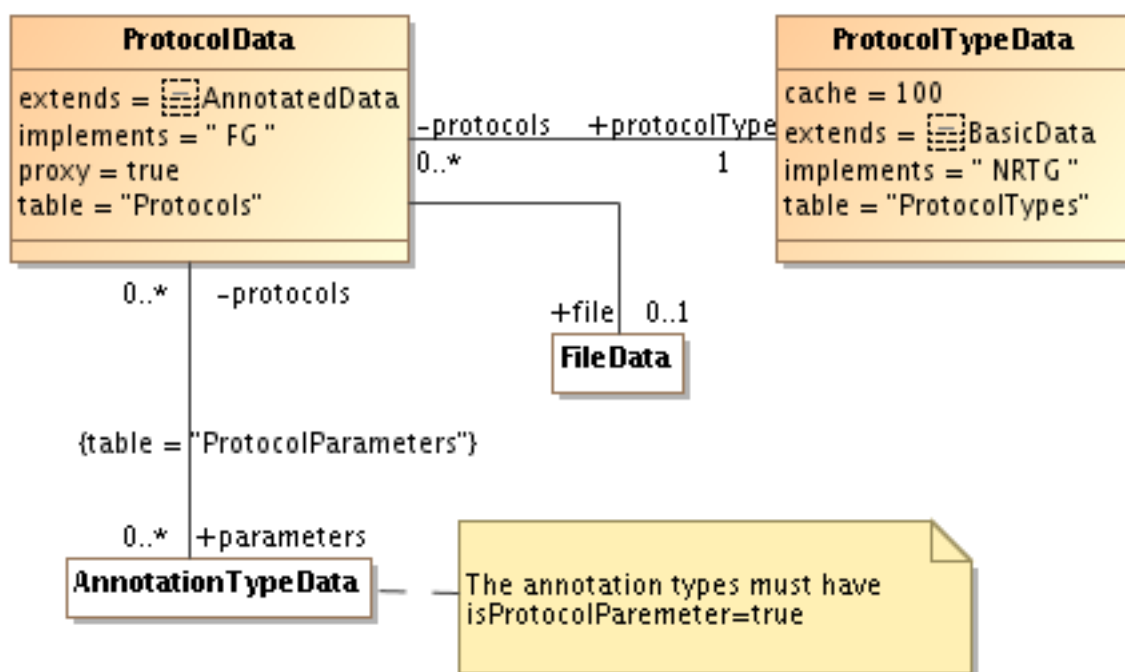
The `AnnotationTypeCategoryData` class defines categories that are used to group annotation types that are related to each other. This information is mainly useful for client applications when displaying forms for annotating items, that wish to provide a clearer interface when there are many (say 50+) annotations type for an item. An annotation type can belong to more than one category.

29.2.11. Protocols

This section gives an overview of how protocols that describe various processes, such as sampling, extraction and scanning, are used in BASE.

UML diagram

Figure 29.12. Protocols



Protocols

A protocol is something that defines a procedure or recipe for some kind of action, such as sampling, extraction and scanning. 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.

29.2.12. Plug-ins, jobs and job agents

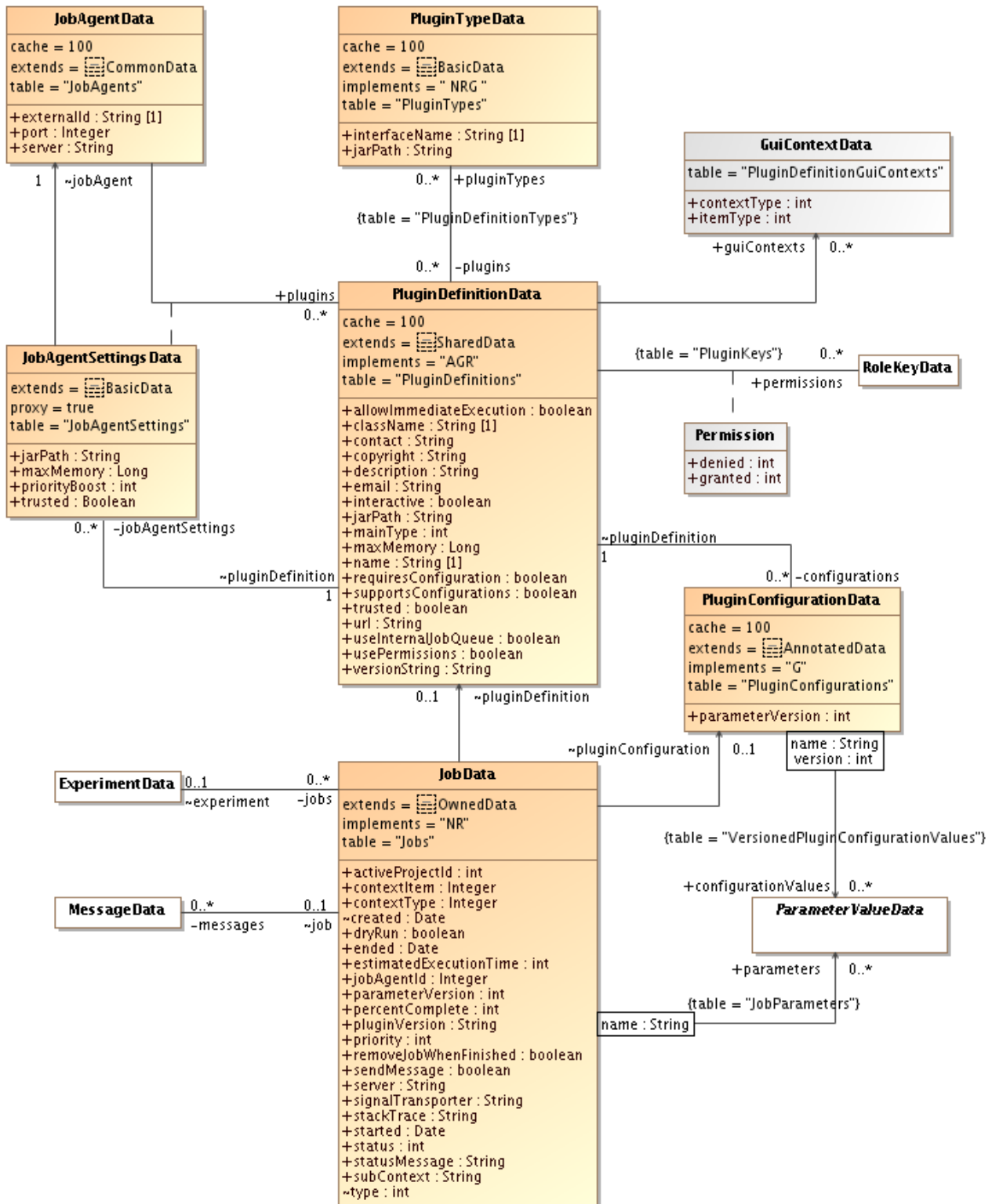
This section gives an overview of plug-ins, jobs and job agents.

See also

- Section 22.1, “Installing plug-ins” (page 133)
- Section 21.2, “Installing job agents” (page 123)

UML diagram

Figure 29.13. 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.

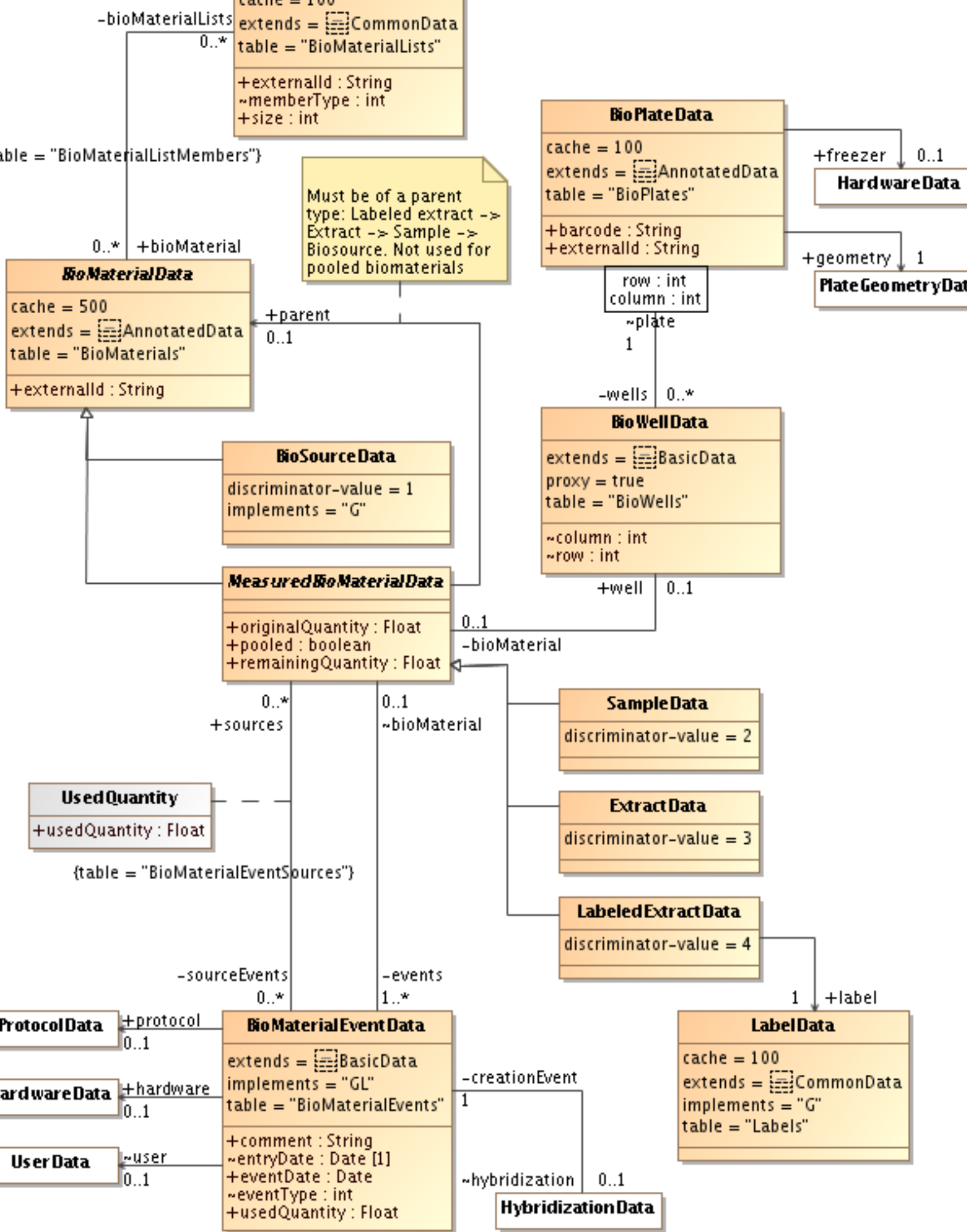
- **UNCONFIGURED** (`status = 0`): The job is not yet ready to be executed.
- **WAITING** (`status = 1`): The job is waiting to be executed.
- **PREPARING** (`status = 5`): The job is about to be executed but hasn't started yet.
- **EXECUTING** (`status = 2`): The job is currently executing.
- **DONE** (`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.



Biomaterial LIMS

There are four types of biomaterials: `BioSourceData`, `SampleData`, `ExtractData` and `LabeledExtractData`. All four types 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 three 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 parent type. `BioSourceData` is the parent type of samples, `SampleData` is the parent type of extracts, and `ExtractData` is the parent type of labeled extracts. In this case the `pooled` property is `false` and the parent is specified in the `parent` property. If the parent is not a `BioSourceData` this information is duplicated, with the addition of an optional `usedQuantityValue`, 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 one or more items of the same type, i.e pooling. In this case the `pooled` property is `true` and 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.

Biomaterial events

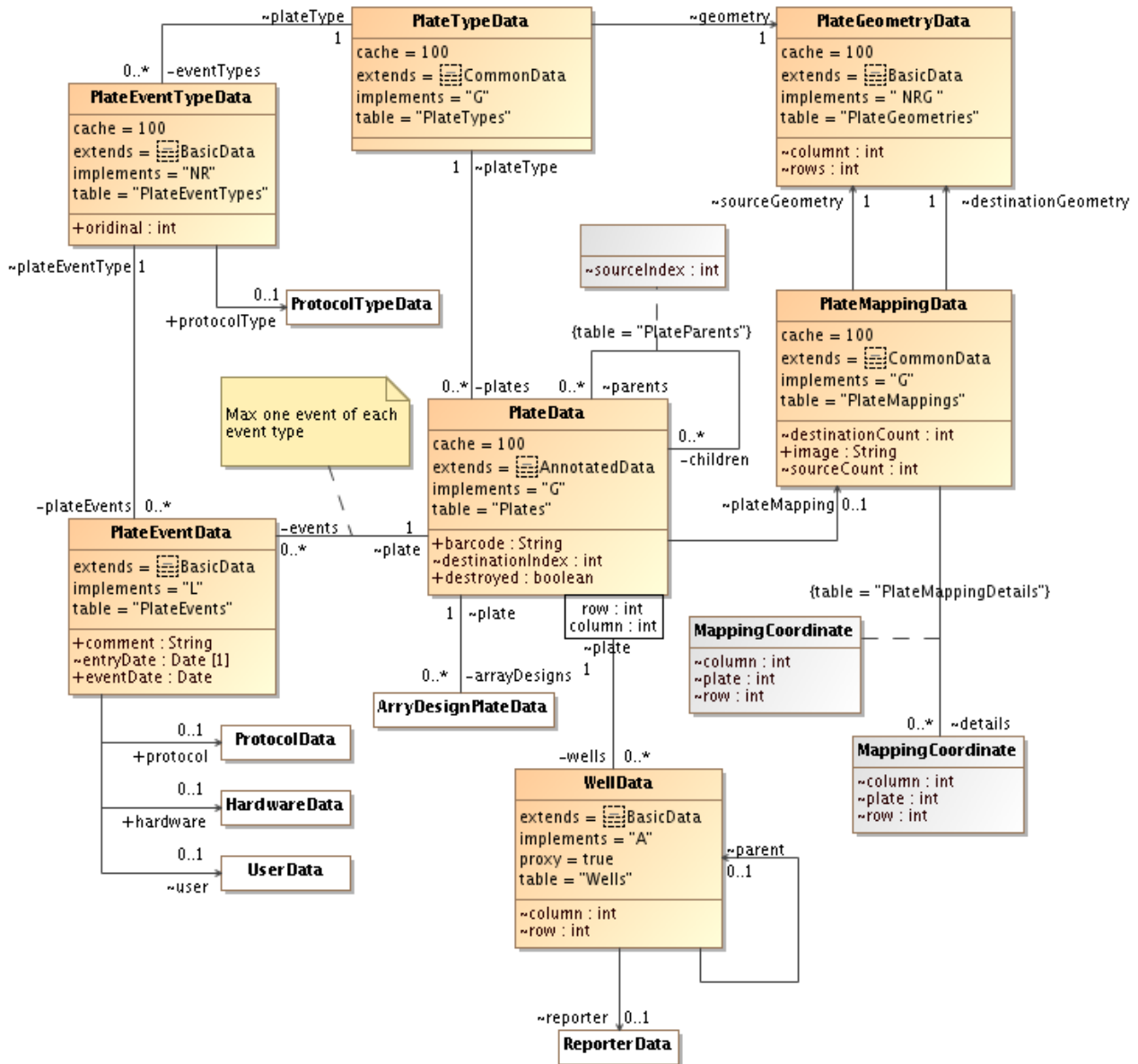
An event represents something that happened to one or more biomaterials, for example the creation of another biomaterial. The `BioMaterialEventData` holds information about entry and event dates, protocols used, the user who is responsible, etc. There are three types of events represented by the `eventType` 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. If the biomaterial is a pooled biomaterial all sources must be of the same type. Otherwise there can only be one source of the parent type. These rules are maintained by the core.
2. *Hybridization event*: This event represents the creation of a hybridization. This event type is needed because we want to keep track of quantities for labeled extracts. This event has a hybridization as a product instead of a biomaterial. The `sources` collection can only contain labeled extracts.
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.

29.2.14. Array LIMS - plates

UML diagram

Figure 29.15. 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.

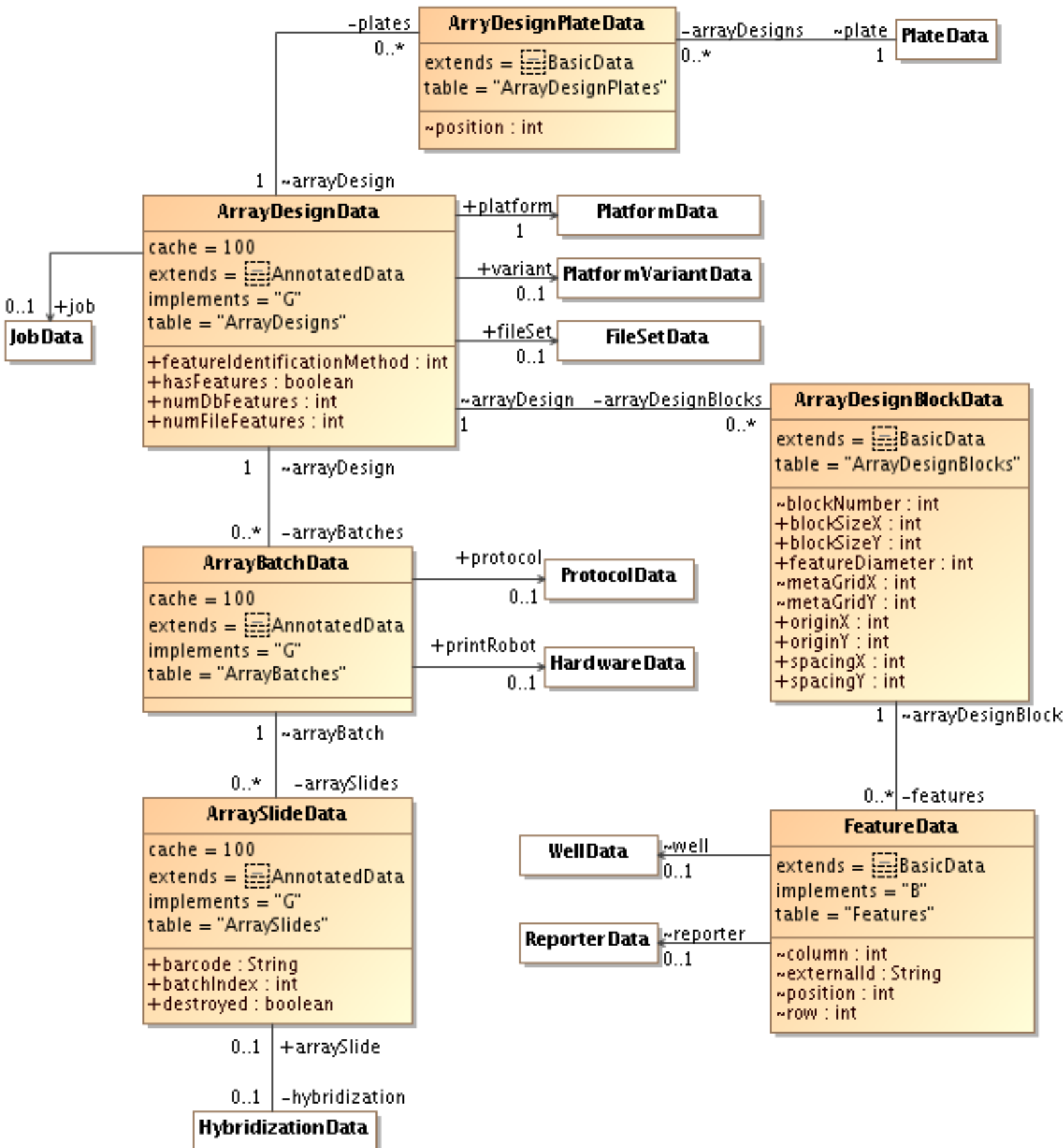
Plate mappings

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.

29.2.15. Array LIMS - arrays

UML diagram LIMS - arrays



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.

The scanning of the hybridized slide is registered as separate scan events. One or more images can optionally be attached to each scan. The images are not used by BASE.

Raw data

A `RawBioAssayData` object represents the raw data that is produced by analysing the image(s) from a single scan. 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.

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.

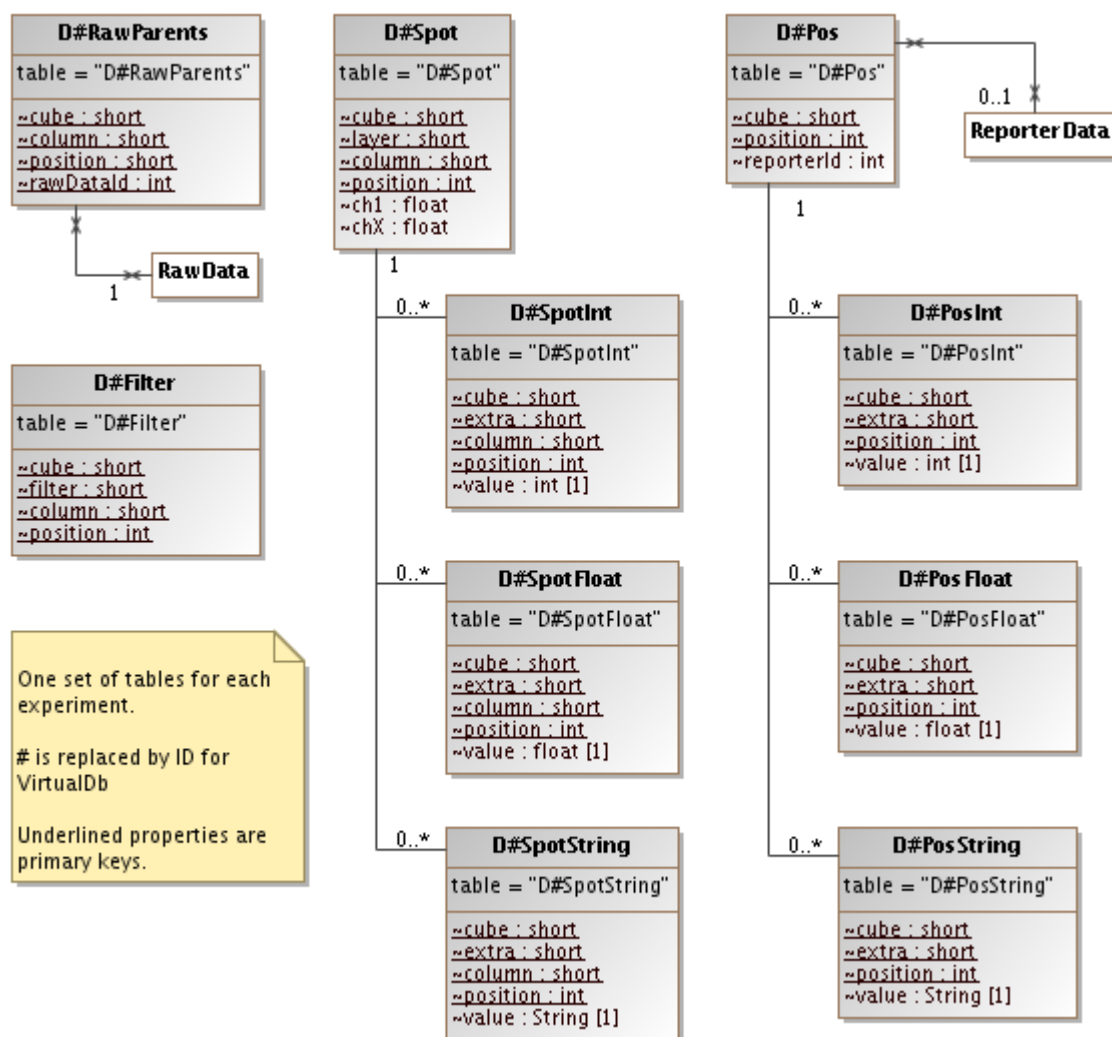
A child bioassay set may use the same data cube as it's parent bioassay set if all of the following conditions are true:

- 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 29.19. 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

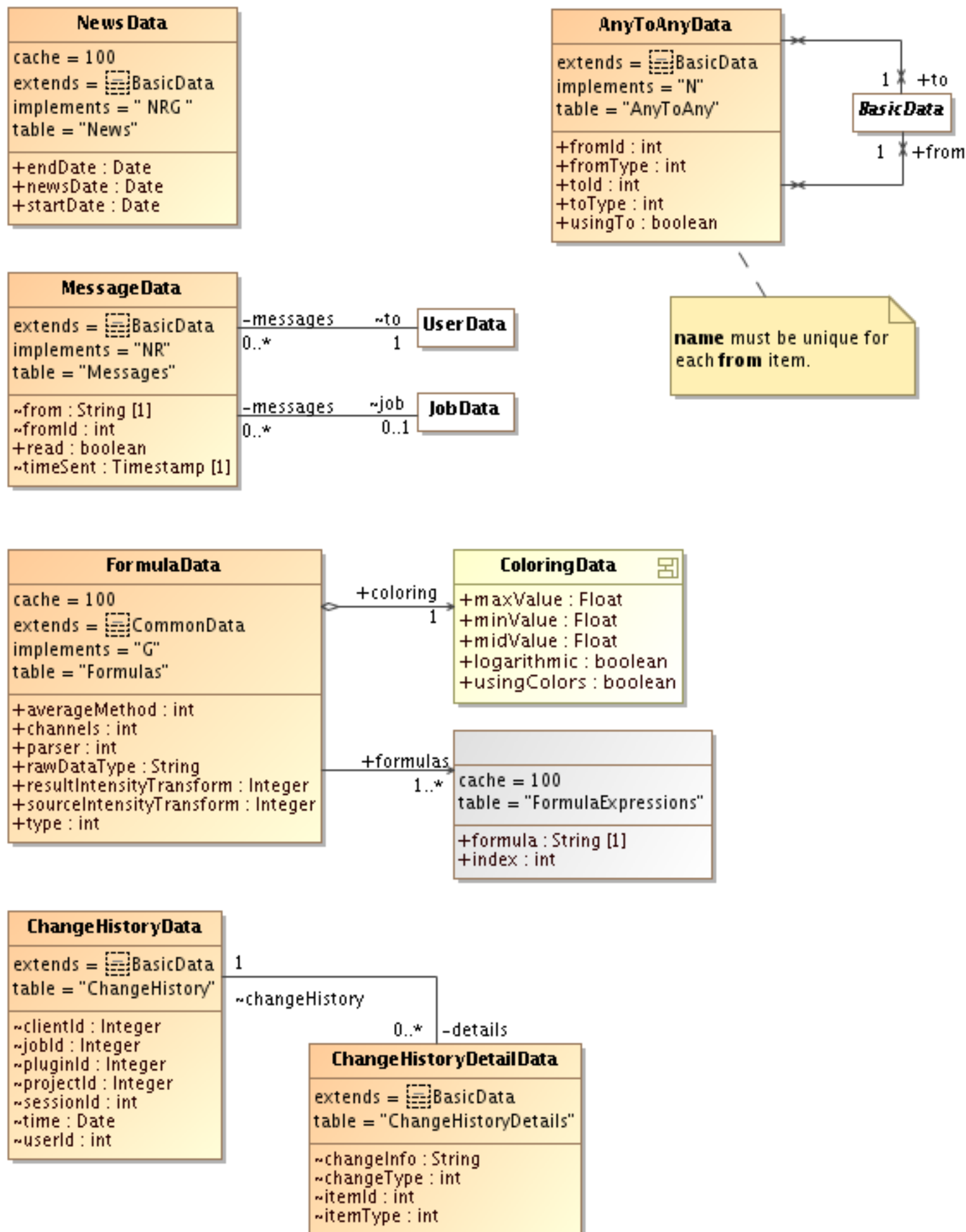
This table stores the reporter id for each position in a cube. Extra values attached to the position are kept in separate tables, one for each type of value (D#PosInt, D#PosFloat and D#PosString).

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.



29.3. The Core API

This section gives an overview of various parts of the core API.

29.3.1. Using files to store data

BASE 2.5 introduced the possibility to use files to store data instead of importing it into the database. Files can be attached to any item that implements the `FileStoreEnabled` interface. Currently this is `RawBioAssay` and `ArrayDesign`. 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. We would have liked to enforce that (raw) data is always present in files, but this will not be backwards compatible with older installations, so 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.

One major change from earlier BASE versions is that the registration of raw data types has changed. The `raw-data-types.xml` file should only be used for raw data types that are stored in the database. The `storage` tag has been deprecated and BASE will refuse to start if it finds a raw data type definitions with `storage="file"`.

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 29.2.8, "Experimental platforms" (page 244)
- Section 26.6.5, "File validator and metadata reader plug-ins" (page 203)
- Section E.1, "Default platforms/variants installed with BASE" (page 346)
- Section K.9, "BASE 2.5 release" (page 362) in Appendix K, *API changes that may affect backwards compatibility* (page 358)

This is rather large set of classes and methods. The ultimate goal is to be able to create links between a `RawBioAssay / ArrayDesign` 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 `DataFileType.getQuery(FileStoreEnabled)` can be used for this. Internally, the method uses the result from `FileStoreEnabled.getPlatform()` and `FileStoreEnabled.getVariant()` methods to restrict the query to only return file types for a given platform and/or variant. If the item doesn't have a platform or variant the query will return all file types that are associated with the given item type. In any case, we get 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.

It might also be interesting to know the currently selected file for each file type and if the platform has set the required flag for a particular file type. 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();

// Get list of DataFileTypes used by the platform
ItemQuery<DataFileType> query =
    DataFileType.getQuery(item);
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
    PlatformFileType pft = platform == null ?
        null : platform.getFileType(type, variant);
    boolean isRequired = pft == null ?
        false : pft.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. The only limitation is that it can only contain one file for each file type. Call `FileSet.setMember()` to store a file in the file set. If a file already exists for the given file type it is replaced, otherwise a new entry is created. 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, true);
```

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 done by the core when the `FileSet.validate()` is called. The process is partly pluggable since each `DataFileType` can name a class that should do the validation and/or metadata extraction.

Note

The `FileSet.validate()` only validates the files where the file types have specified plugins that can do the validation and metadata extraction. The 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 checks the `DataFileType` of all members in the file set and creates `DataFileValidator` and `DataFileMetadataReader` objects. Only one instance of each class is created. If the file set contains members which has the same validator or metadata reader, they will all share the same instance.
2. Each validator/metadata reader class is initialised with calls to `DataFileHandler.setItem()` and `DataFileHandler.setFile()`.
3. Each validator is called. The result of the validation is saved for each file and can be retrieved by `FileSetMember.isValid()` and `FileSetMember.getErrorMessage()`.
4. Each metadata reader is called, unless the metadata reader is the same class as the validator and the validation failed. If the metadata reader is a different class, it is called even if the validation failed.

Only one instance of each validator class is created

The validation/metadata extraction is not done until all files have been added to the fileset. If the same validator/meta data reader is used for more than one file, the same instance is reused. I.e. the `setFile()` is called one time for each file/file type pair. The `validate()` and `extractMetadata()` methods are only called once.

All validators and meta data extractors should extend the `AbstractDataFileHandler` class. The reason is that we may want to add more methods to the `DataFileHandler` interface in the future. The `AbstractDataFileHandler` will be used to provide default implementations for backwards compatibility.

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 ...
}
```

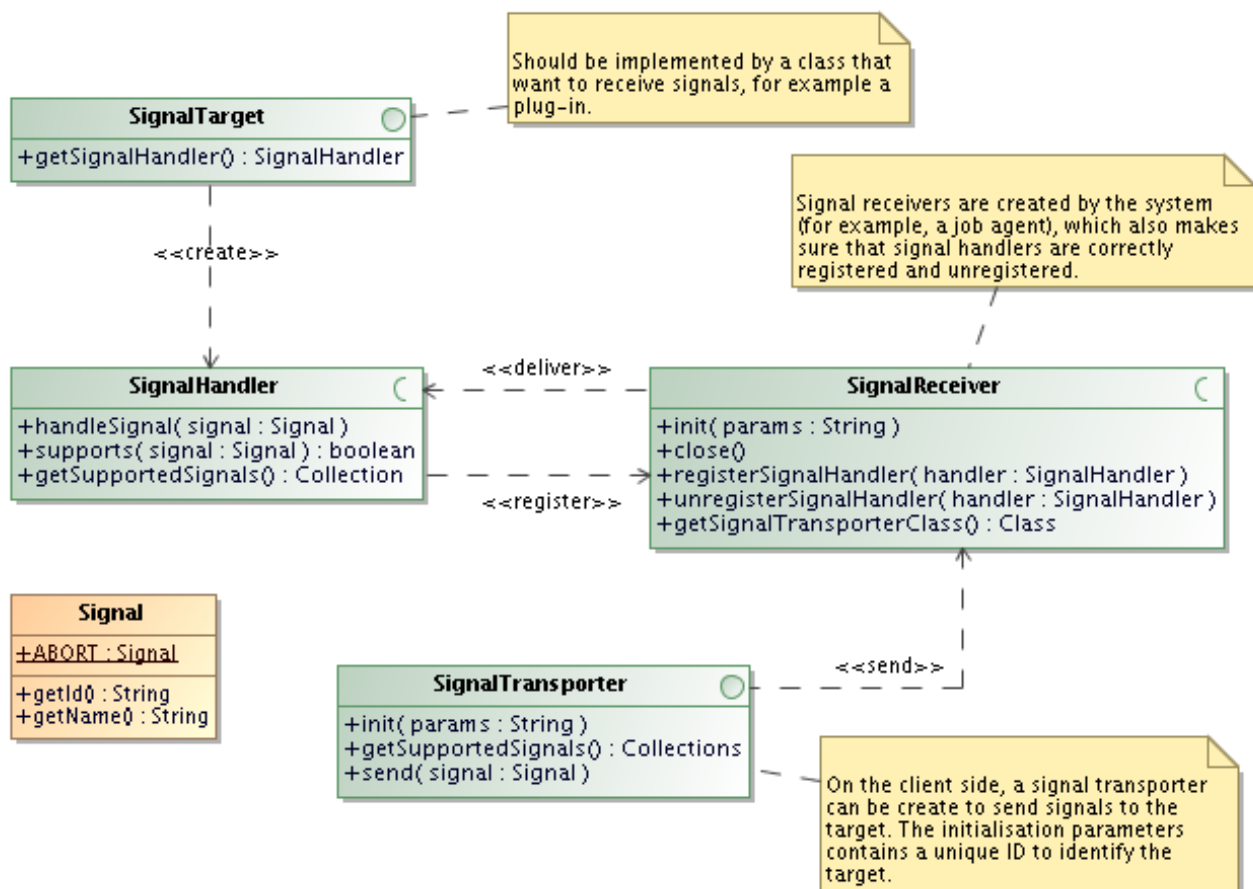
29.3.2. Sending signals (to plug-ins)

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 26.7, “Enable support for aborting a running a plug-in” (page 206).

Diagram of classes and methods

Figure 29.22. The signalling system



The signalling system is rather simple. An object that wish to receive signals must implement the `SignalTarget`. It's 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 it must know that it should check `Thread.interrupted()` at regular intervals from the main 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 delievered 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 338) 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 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 sees 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 its turn trigger some action in the signal target which soon will abort what it is doing and exit.

29.4. The Query API

This documentation is only available in the old format. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/query/index.html>

29.5. Analysis and the Dynamic and Batch APIs

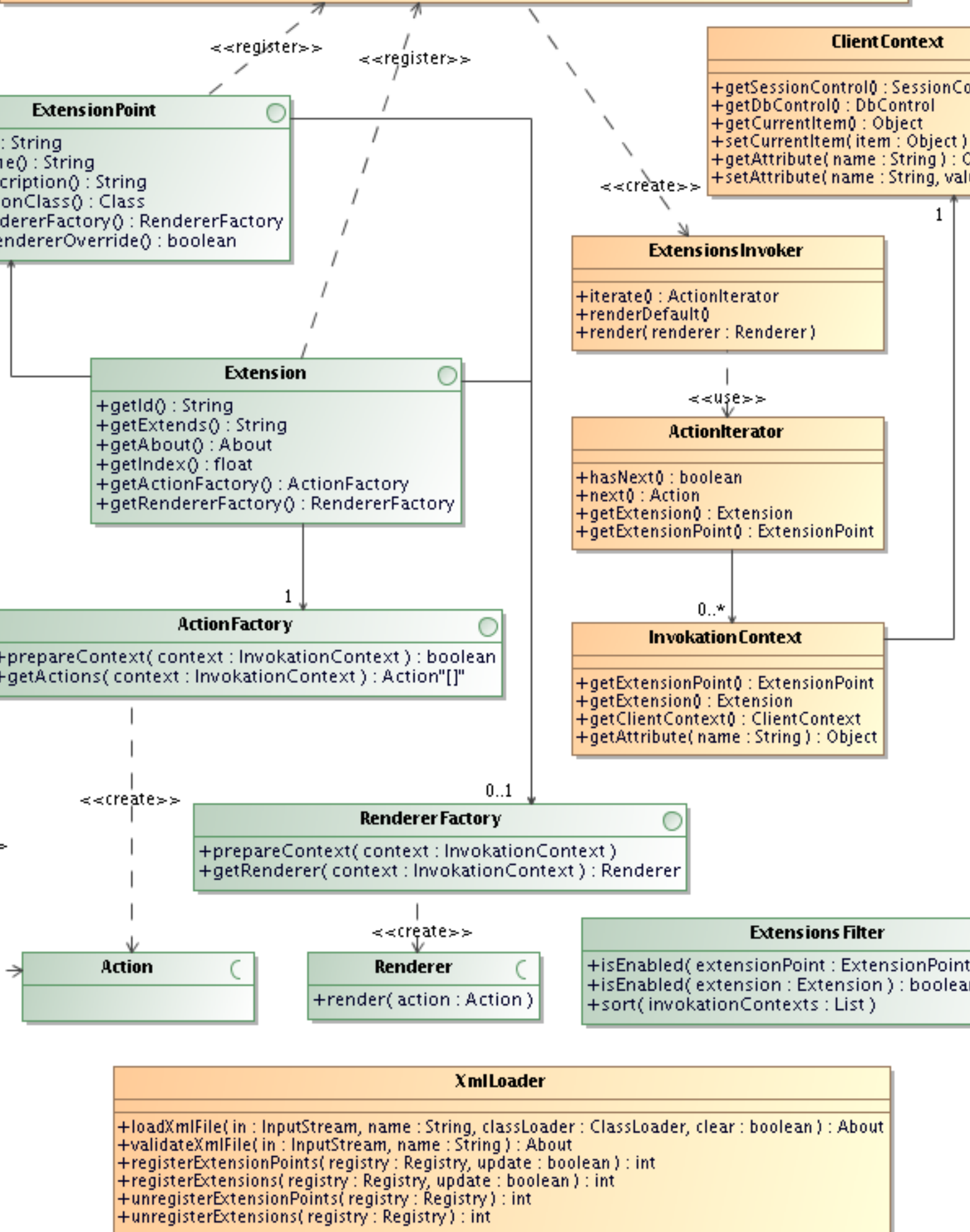
This documentation is only available in the old format. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/dynamic/index.html>

29.6. Extensions API

29.6.1. 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 its sub-packages, and consists of three sub-parts:

- A set of interface definitions which forms the core of the Extensions API. The interfaces defines, for example, what an `Extension` is and what an `ActionFactory` should do.
- A `Registry` that is used to keep track of installed extensions. The registry also provides functionality for invoking and using the extensions.
- Utility classes that are useful when implementing a client application that can be extendable. The most useful example is the `XmlLoader` which can read extension definitions from XML files and create the proper factories, etc.



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 29.1. 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.

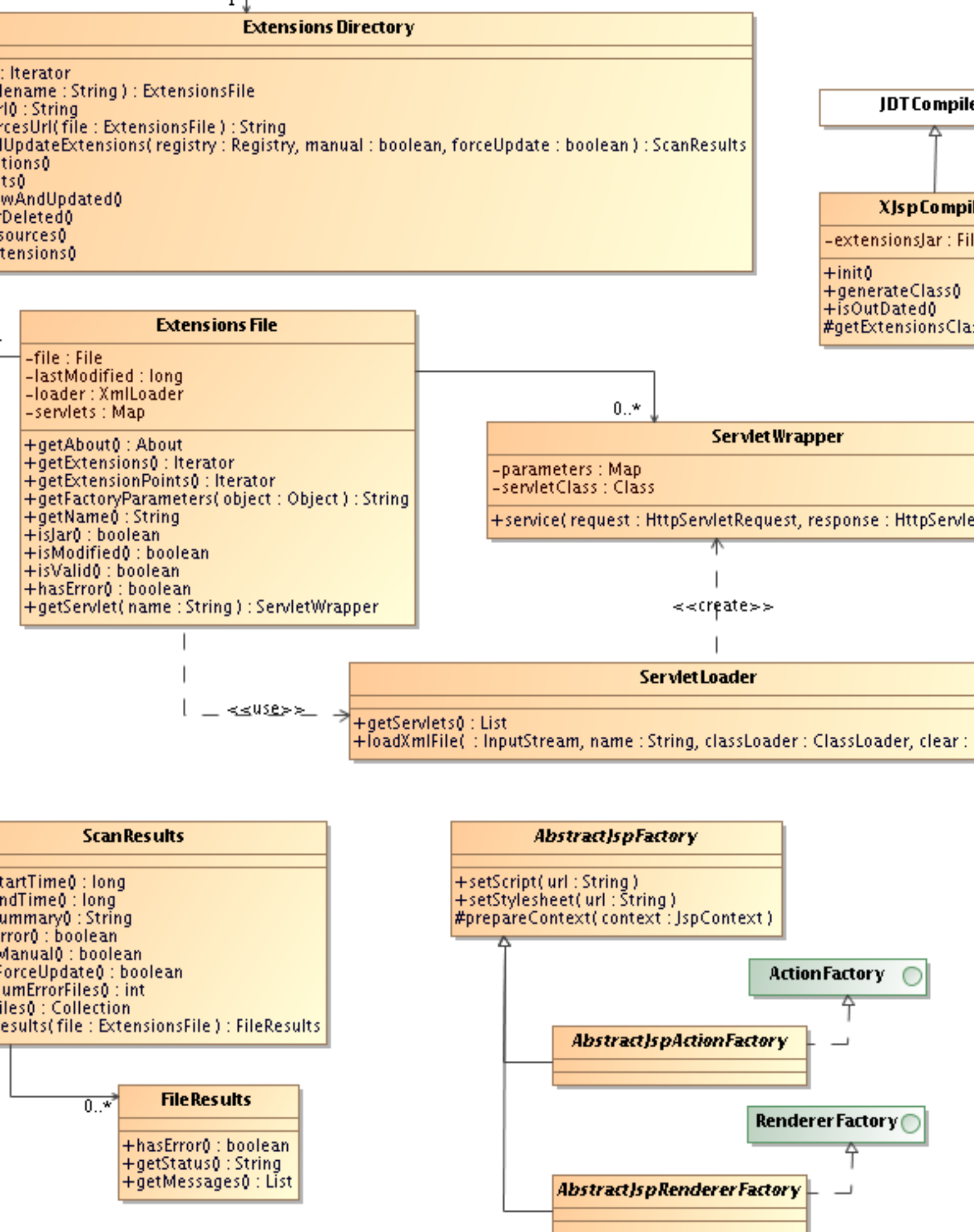
29.6.2. The web client part

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 classes used to administrate the extension system. Here is for example the `ExtensionsControl` class which is the master controller for the web client extensions. It:

- Keeps track of installed extensions and which JAR or XML file they are installed from.
- Can, manually or automatically, find and install new or updated extensions and uninstall deleted extensions.
- Adds permission control to the extension system, so that only an administrator is allowed to change settings, enable/disable extensions, etc.

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.

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.



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, which is equivalent to doing a manual scan with the force update setting to false. This means that the extension system is up and running as soon as the first user logs in to BASE.
- 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 `ExtensionsFile` and `ServletWrapper` and then invoke the custom servlet. More information can be found in Section 27.7, “Custom servlets” (page 220).

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 scan, 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. This feature is experimental and requires installing an extra JAR into Tomcat's lib directory. See Section 23.2, “Installing the X-JSP compiler” (page 149) for more information.

29.7. Other useful classes and methods

TODO

Chapter 30. Write documentation

30.1. User, administrator and developer documentation with Docbook

This chapter is for those who intent 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.

30.1.1. 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.

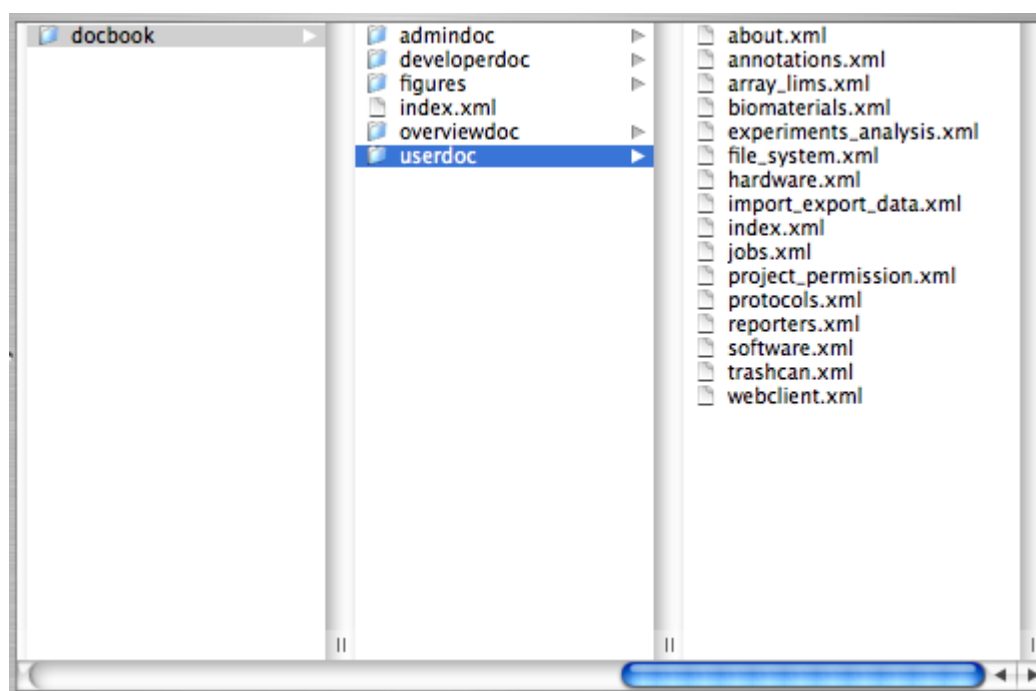
30.1.2. Getting started

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 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.

¹ <http://www.docbook.org/tdg/en/html/>

Figure 30.1. The organization of documentation files

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 30.1. 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="folder name to put the chapter's files in"?>
  <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 attribute `dir` set to the value that the chapter's output folder should have as a name.

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.

Example 30.2. Include a chapter

```
<part id="userdoc">
  <title>User documentation</title>
  <include file="about.xml"/>
  <include file="webclient.xml"/>
  <include file="project_permission.xml"/>
  <include file="trashcan.xml"/>
  <include file="file_system.xml"/>
  <include file="jobs.xml"/>
  <include file="reporters.xml"/>
  <include file="annotations.xml"/>
  <include file="protocols.xml"/>
  <include file="hardware.xml"/>
  <include file="example_chapter.xml"/>
  <include file="software.xml"/>
  <include file="array_lims.xml"/>
  <include file="biomaterials.xml"/>
  <include file="experiments_analysis.xml"/>
  <include file="import_export_data.xml"/>
</part>
```

Order of chapters

The chapters will come in the same order as they are included in the index file.

Now it's only to go ahead with the documentation writing.

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 4, *Resources* (page 6)), 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 using the upper level's `id` and add the section's title or a part of the title. For `<sect2>` it should be like this;

```
chapter_title.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
```

Mark help texts

This documentation is also use to create the help texts that show up in the 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 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 30.3. How to use the help text tag

```
<sect1>
  <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
```

Generate output

Requirements

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.

Compile - generate output files

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.

30.1.3. 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	<chapter>	See Example 30.1, "Example of a chapter" (page 284)
Title	<title>	Example 30.1, "Example of a chapter" (page 284) shows how this can be implemented

² <http://www.docbook.org/tdg/en/html/docbook.html>

Define	Element to use	Comments
Paragraph	<para>	Used almost everywhere around real text.
Top-level subsection	<sect1>	See the section called “ The id attribute ” (page 285)
Second level section	<sect2>	See the section called “ The id attribute ” (page 285)
Third level section	<sect3>	See the section called “ The id attribute ” (page 285)
Fourth level section	<sect4>	
Fifth level section	<sect5>	

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	<classname>	The name of a (Java) class. <code>docapi-attribute</code> can be used to link the class to it's Javadoc (only HTML-version). This is done by setting the attribute to the package name of the class, like <code>net.sf.basedb.core</code>
Interface name	<interfacename>	The name of an (Java) interface. Has a <code>docapi-attribute</code> which has the same functionality as in <code>classname</code> above.
User input	<userinput>	Text that is entered by a user.
Variable name	<varname>	The name of a variable in a program.
Constant	<constant>	The name of a variable in a program.
Method definition	<methodsynopsis>	See Example 30.4, “ Method with no arguments and a return value ” (page 289)
Modifier of a method	<modifier>	See Example 30.4, “ Method with no arguments and a return value ” (page 289)
Classification of return value	<type>	See Example 30.4, “ Method with no arguments and a return value ” (page 289)
Method name	<methodname>	See Example 30.4, “ Method with no arguments and a return value ” (page 289)
No parameter/type	<void>	See Example 30.4, “ Method with no arguments and a return value ” (page 289)

Define	Element to use	Comment
Define a parameter	<code><methodparam></code>	See Example 30.5, “ Method with arguments and no return value ” (page 289)
Parameter type	<code><type></code>	See Example 30.5, “ Method with arguments and no return value ” (page 289)
Parameter name	<code><parameter></code>	See Example 30.5, “ Method with arguments and no return value ” (page 289)

Follow one of the examples below to insert a method definition in the document.

Example 30.4. Method with no arguments and a return value

```
<methodsynopsis language="java">
  <modifier>public</modifier>
  <type>Plugin.MainType</type>
  <methodname>getMainType</methodname>
  <void />
</methodsynopsis>
```

Example 30.5. 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>
```

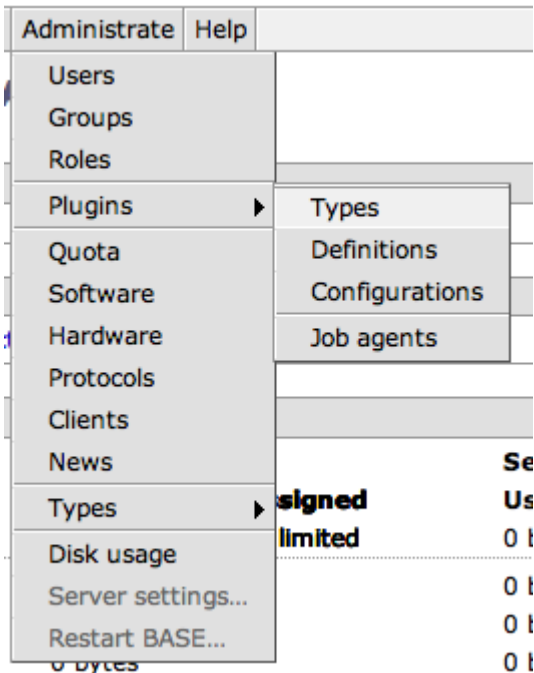
Gui elements

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	<code><guibutton></code>	
Label	<code><guilabel></code>	
Menu choice	<code><menuchoice></code>	
Menu	<code><guimenu></code>	
Submenu	<code><guisubmenu></code>	
Menu item	<code><guimenuitem></code>	
Icon	<code><guiicon></code>	

Example 30.6, “Describe a menu choice” (page 290) shows how the menu choice in figure Figure 30.2, “Menu choice” (page 290) can be described.

Figure 30.2. Menu choice



Example 30.6. Describe a menu choice

```
<menuchoice>
  <guimenu>Administrate</guimenu>
  <guisubmenu>Plugins</guisubmenu>
  <guimenuitem>Types</guimenuitem>
</menuchoice>
```

In the text it will look like this: Administrate Plugins Types

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 30.7. Screen-shot in the documentation

Figure 6.1, “The home page” (page 14) is implemented with the following code

```
<figure id="docbook.figures.menuchoice">
  <title>The home page</title>
  <screenshot>
    <mediaobject>
      <imageobject>
        <imagedata
          scalefit="1"
          width="100%"
          fileref="figures/homapage.png" format="PNG"
        />
      </imageobject>
    </mediaobject>
  </screenshot>
</figure>
```

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 6.2.4, "Preferences" (page 17), 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 markup inside the `<programlisting>` tag very well. You should avoid that. By default, java program examples include line numbering, but not xml examples. 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 30.8. Example in the documentation

This shows how Example 26.2, “A typical implementation stores this information in a static field” (page 170) is written in the corresponding XML-file.

```
<example id="net.sf.basedb.core.plugin.Plugin.getAbout">
  <title>A typical implementation stores this information
    in a static field</title>
  <programlisting language="java">
private static final About about = new AboutImpl
(
  "Spot images creator",
  "Converts a full-size scanned image into smaller preview jpg " +
  "images for each individual spot.",
  "2.0",
  "2006, Department of Theoretical Physics, Lund University",
  null,
  "base@thep.lu.se",
  "http://base.thep.lu.se"
);

public About getAbout()
{
  return about;
}
</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	<warning>	
Notification text	<note>	
A tip	<tip>	
Important text	<important>	
Something to be cautious about	<caution>	

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	<itemizedlist>	
Term definition list	<variablelist>	
Ordered list	<orderedlist>	
List item	<listitem>	

The example below shows how to create a list for term definition in the text.

Example 30.9. 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>
```

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 30.10. 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.

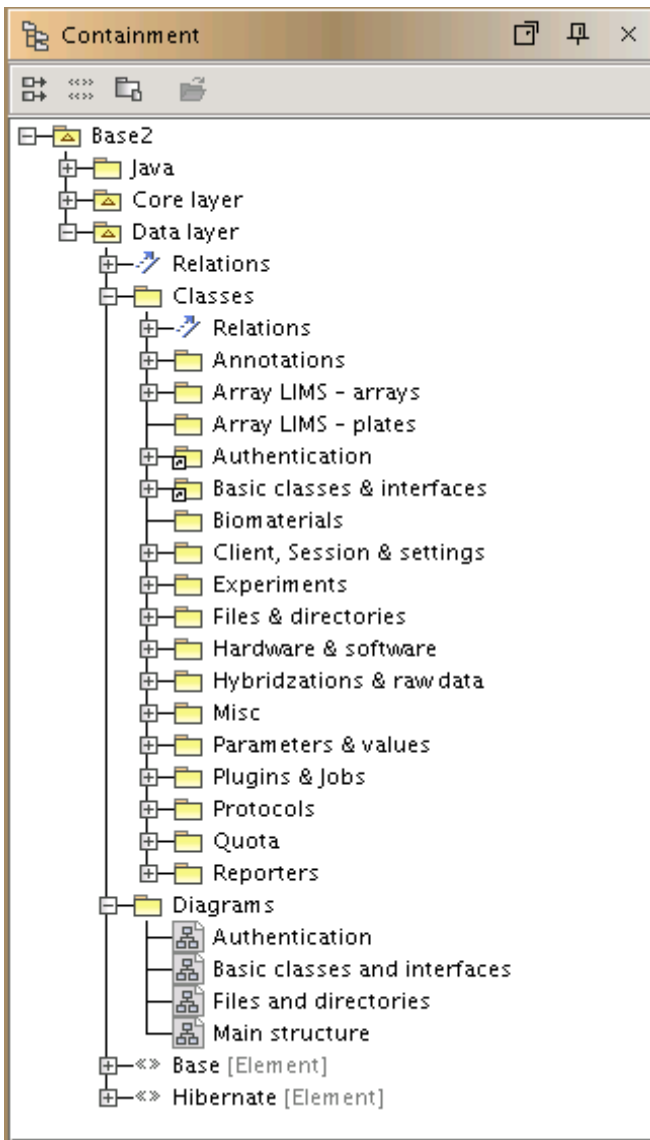
30.2. Create UML diagrams with MagicDraw

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/>

30.2.1. Organisation

All classes and diagrams are in a single UML file. It can be found at `<base-dir>/doc/src/uml/base.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 30.3. MagicDraw organisation

30.2.2. 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 modified symbols (+, ~ and -) to indicate if a property is updatable or not.

Setting tagged values

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. 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.

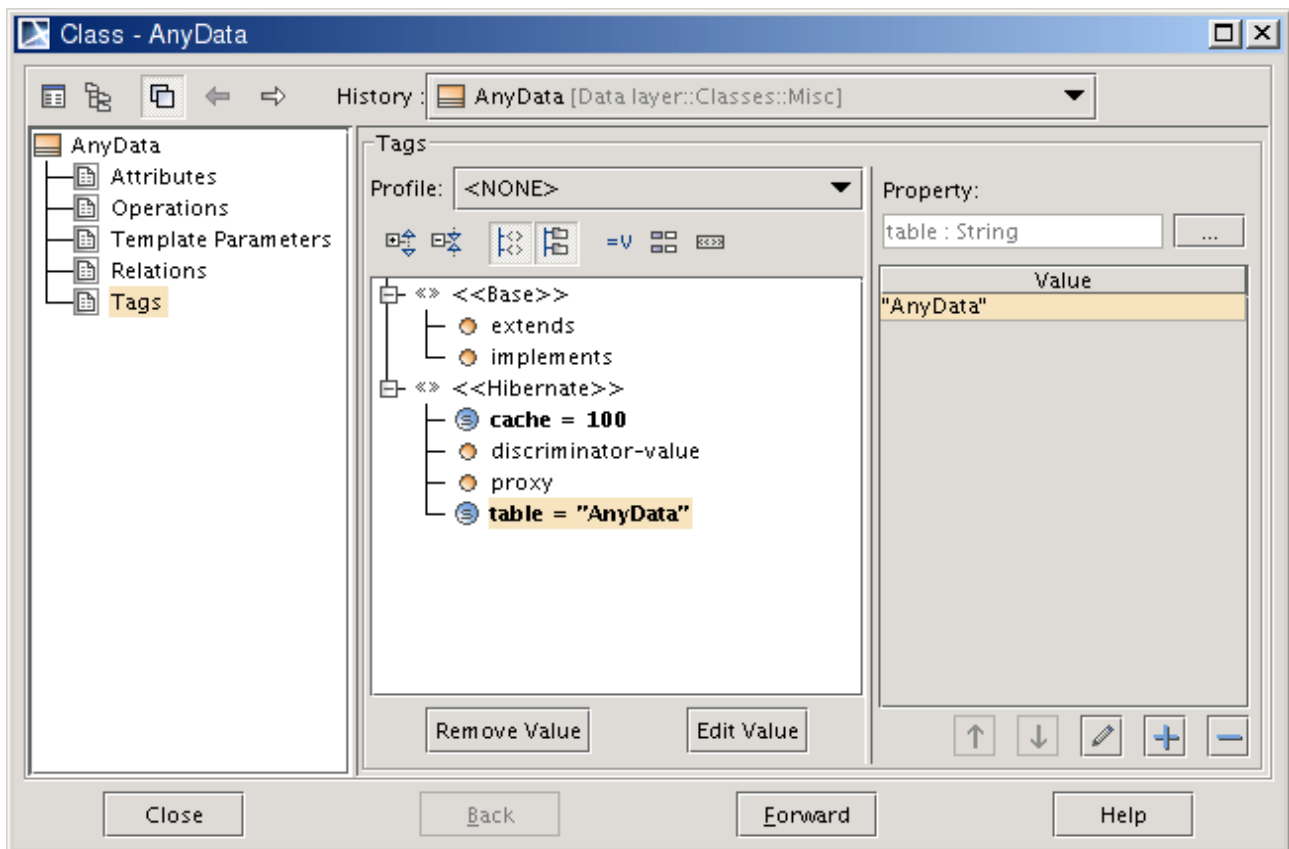
extends

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

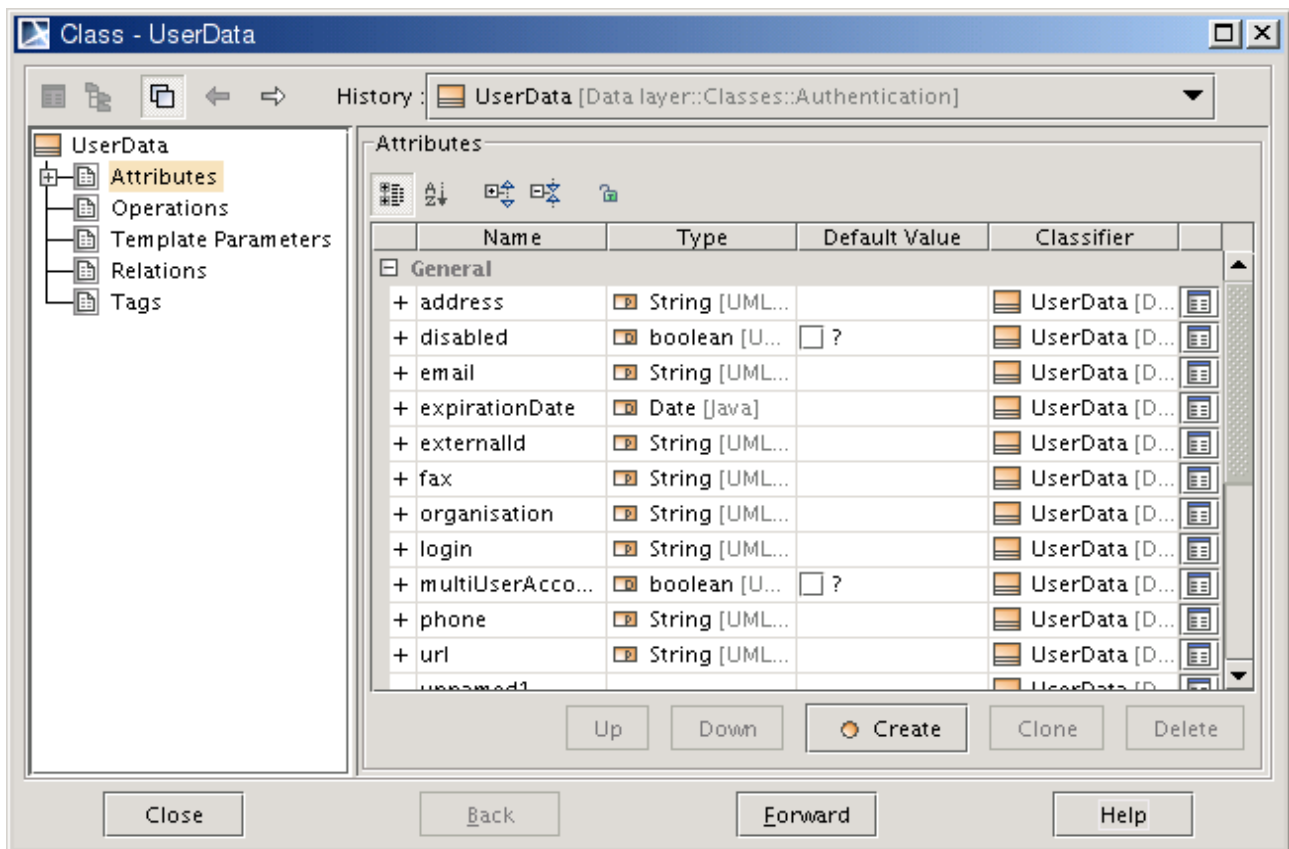
Figure 30.4. Setting tagged values

Specifying simple properties for a class

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.
- 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 30.5. Class attributes

Associations to other classes

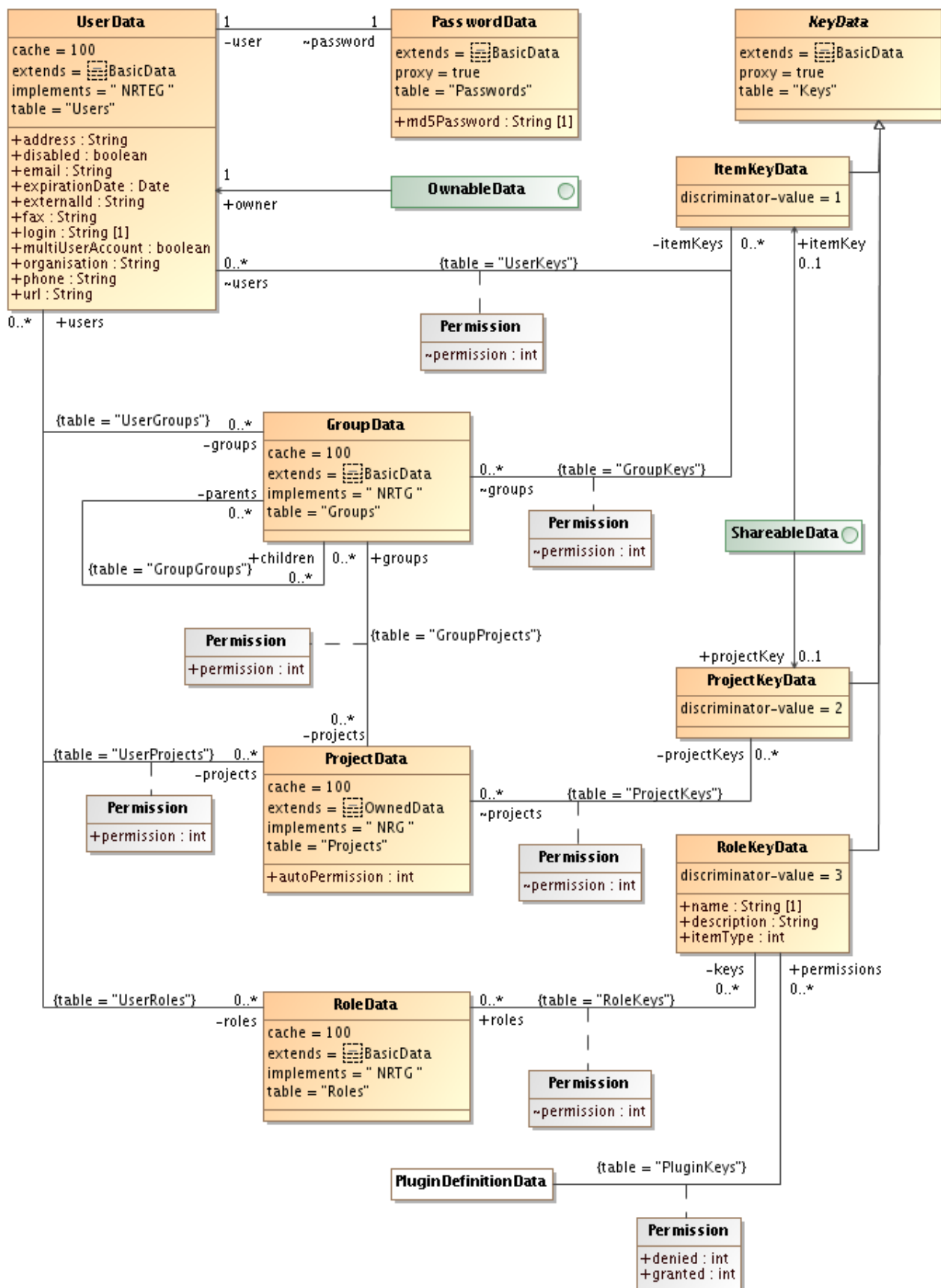
Associations to other classes are easiest created from a diagram view by drawing an **Association** link between the two classes. The ends should be given a name, multiplicity and visibility should be selected. For the visibility we use the same options as for attributes, but with a slightly different 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 30.6. Authentication UML diagram

Core layer classes

TODO

30.2.3. 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.

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.

30.3. 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 its 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 29.1, “The Public API of BASE” (page 228).

30.3.1. Writing Javadoc

This section only covers Javadoc comments, how to write proper none-Javadoc comments are described in Section 31.3.2, “General coding style guidelines” (page 303)

General information about Javadoc and how it is written in a proper way can be found at <http://java.sun.com/j2se/javadoc/writingdoccomments/index.html>³. 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 only should be shown in the internal documentation and not in the public, 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.

```
@version
```

From which version of Base the class is available.

```
@see
```

Optional. Links to some related subjects.

³ <http://java.sun.com/j2se/javadoc/writingdoccomments/index.html>

```
@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 31. Core developer reference

31.1. Publishing a new release

This documentation is available on the BASE wiki¹.

31.2. Subversion / building BASE

This documentation is only available in the old format. See <http://base.thep.lu.se/chrome/site/doc/historical/development/build.html>

31.3. Coding rules and guidelines

31.3.1. 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 29.1, “The Public API of BASE” (page 228) and Section 31.3.3, “API changes and backwards compatibility” (page 303).
- 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>

- 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 31.3.4, “Data-layer rules” (page 304).
- 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 K, *API changes that may affect backwards compatibility* (page 358) document if you have introduced any incompatible changes.

31.3.2. General coding style guidelines

This documentation is only available in the old format. See <http://base.thep.lu.se/chrome/site/doc/historical/development/coding/generic.html>

31.3.3. 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.

Do not forget to log changes!

Any change that may affect backwards compatibility must be logged in Appendix K, *API changes that may affect backwards compatibility* (page 358).

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.

Does the changes affect the Public API?

See Section 29.1, “The Public API of BASE” (page 228) and the javadoc² for information about the public API.

Changes made to the non-public API does not have to follow the same rules.

Contract compatibility

TODO

Binary compatibility

TODO

Internal data structure compatibility

TODO

Source code compatibility

TODO

31.3.4. 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.

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)
{
    this.address = address;
}

private int row;
/**
 * @hibernate.property column="`row`" type="int"
 */
```

² <http://base.thep.lu.se/chrome/site/doc/api/index.html>

```
public int getRow()
{
    return row;
}
public void setRow(int row)
{
    this.row = row;
}
```

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
{
    ...
}
```

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 29.2.1, “Basic classes and interfaces” (page 230).

Define a public no-argument constructor

Always define a public a 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 `get` 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 the section called “Many-to-many and one-to-many mappings” (page 312).

```
// 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 user documentation: 4.1.1. Implement a no-argument constructor³

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 hascodes are contradicting when the hashcode 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 it's 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 user documentation: 4.3. Implementing equals() and hashCode()⁴

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 user documentation: 4.1.3. Prefer non-final classes⁵
- Hibernate user documentation: 19.1.3. Single-ended association proxies⁶

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` 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.

It is required that one thinks a bit of the usage of a class 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 `LabelData` 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 user documentation: 19.2. The Second Level Cache⁷

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

Starting with Hibernate 3.1 there is a new 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.

Stateless sessions are used by batchable 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` 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 31.1. 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 user documentation: 19.1.3. Single-ended association proxies⁸

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
 * @version 2.0
 * @hibernate.class table="`Anys`" lazy="false" batch-size="10"
 * @base.modified $Date: 2007-08-17 09:18:29 +0200 (Fri, 17 Aug 2007) $
 */
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).

Specify a value for 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 user documentation: 5.1.3. class⁹

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 user documentation: 5.1.9. property¹⁰
- Hibernate user documentation: 5.2.2. Basic value types¹¹

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 31.1, "Choosing cache and proxy settings" (page 308) 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 user documentation: 5.1.10. many-to-one¹²

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 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 `get` method must always return the same object that Hibernate passed to the `set` 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 required 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 `GroupData` object. The methods in the `RoleData` 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 GroupData grp WHERE grp.users ...
SELECT ... FROM UserData usr WHERE usr.groups ...
```

Without the inverse 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.

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 user documentation: Chapter 6. Collection Mapping¹³
- Hibernate user documentation: Chapter 21. Example: Parent/Child¹⁴

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 hybridizations and arrayslides.

```
// HybridizationData.java
private ArraySlideData arrayslide;
/**
    Get the array slide
    @hibernate.many-to-one column="`arrayslide_id`" not-null="false" unique="true"
*/
public ArraySlideData getArraySlide()
{
    return arrayslide;
}
public void setArraySlide(ArraySlideData arrayslide)
{
    arrayslide.setHybridization(this);
    this.arrayslide = arrayslide;
}
```

```
// ArraySlideData.java
private HybridizationData hybridization;
/**
 * Get the hybridization
 * @hibernate.one-to-one property-ref="arraySlide"
 */
public HybridizationData getHybridization()
{
    return hybridization;
}
void setHybridization(HybridizationData hybridization)
{
    this.hybridization = hybridization;
}
```

As you can see, we use the many-to-one mapping on with a `unique="true"` option for the hybridization. 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. Also, note that the we can only change the link with the `HybridizationData.setArraySlide()` method, and that this method also updates the other end.

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 user documentation: 5.1.11. one-to-one¹⁵

Documentation

The data layer code needs documentation. A simple approach is used for Javadoc documentation

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
 * @version 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 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 30.2, “Create UML diagrams with MagicDraw” (page 293).

31.3.5. Item-class rules

This documentation is only available in the old format. See <http://base.thep.lu.se/chrome/site/doc/historical/development/coding/item/index.html>

31.3.6. Batch-class rules

TODO

31.3.7. Test-class rules

TODO

31.4. Internals of the Core API

This documentation is only available in the old format. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/core/index.html>

31.4.1. Authentication and sessions

This documentation is only available in the old format. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/core/authentication.html>

31.4.2. Access permissions

This documentation is only available in the old format. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/core/accesspermissions.html>

31.4.3. Data validation

TODO

31.4.4. Transaction handling

TODO

31.4.5. Create/read/write/delete operations

This documentation is only available in the old format. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/core/itemhandling.html>

31.4.6. Batch operations

This documentation is only available in the old format. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/core/batchprocessing.html>

31.4.7. Quota

TODO

31.4.8. Plugin execution / job queue

This documentation is only available in the old format. See <http://base.thep.lu.se/chrome/site/doc/historical/development/overview/core/plugins.html>

Part V. FAQ

Chapter 32. 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 4, *Resources* (page 6) for contact information.

32.1. 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 21.3, "Installation instructions" (page 125) 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 22, *Plug-ins* (page 133).

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 just 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 10.3.1, "Import/update reporter from files" (page 55) 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.

32.2. Array design related questions with answers

Q: What it 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 22.4, "Plug-in configurations" (page 140) 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 10, *Reporters* (page 54)

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.

32.3. 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 6.4.3, “Configuring which columns to show” (page 27).
- Q: Is it possible to sort the values in a column in the list view?
- A: Of course it is. See Section 6.4.1, “Ordering the list” (page 25).
- 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 17.3.1, “Create sample” (page 86). Pooling can also be applied to extracts and labeled extracts.
- Q: I would like to add a software type in BASE but there is no button for doing so. Is it possible?
- A: No, this is not possible, since there is only one place software is used, namely to register the feature extraction of a scanned image.
- Q: I need to create a new hardware type 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 hardware types. Contact your BASE administrator. For more information about permissions, please refer to Chapter 24, *Account administration* (page 152).
- 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.

32.4. 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 21.1, “Upgrade instructions” (page 121).

Finally, you will have to configure a raw data import plug-in in order to be able to create raw-bioassays. See Section 22.4, “Plug-in configurations” (page 140) and Section 18.2.2, “Import raw data” (page 100) 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.

Q: Unzipped files never inherit file type specified during upload, why?

A: Unfortunately this is an “unfixable” defect. The API for unpacking files is a kind of plug-in interface (FileUnpacker) that has been made public. The “mistake” is that the API does not include any information from the file upload dialog except the directory to upload to (not even the file name is included). Introducing new method or parameters will break the API and we prefer to not do that.

As a workaround to this problem the extension of each file is checked against the list of MIME types (Administrative Types Mime Types). If a MIME type with the same extension is found and has been connected to a file type, this file type is transferred to the file. This will obviously only work for non-generic file extensions, *eg.*, .gpr for GenePix raw data files, .cel for Affymetrix CEL files, etc.

32.5. Data Deposition to Public Repositories related questions with answers

Q: I am asked by reviewers to deposit my microarray data in a public repository. How can BASE help me?

A: The BASE development team are working on a plug-in that produces a Tab2Mage file accepted by ArrayExpress. For the plug-in to work properly, a series of rules need to be followed, please refer to Section 18.3.3, “Tab2Mage export” (page 105) for additional information.

More information about the Tab2Mage format can be found at <http://tab2mage.sourceforge.net>. To send the submission to array express, use the ArrayExpress FTP site at <ftp://ftp.ebi.ac.uk> pwd. Log in with username and password **aexpress**.

Warning

The current export plug-in does not support Tab2Mage normalised and transformed files, so some additional work might be required to be fully compliant, please refer to Tab2Mage help notes for creating these final gene expression files and update the Tab2Mage files.

- Q: Repositories want me to be MIAME compliant but how do I know that?
- A:
- Make sure to format your annotation types following the rules detailed in Section 18.3.3, “Tab2Mage export” (page 105).
 - Before exporting, it is probably a good idea to run the **Experiment Overview** detailed in Section 6.6, “Item overview” (page 33). By selecting stringent criteria from the interface, the tool will detect all missing information that could be requested by repositories.
 - If the experiment overview does not report any error any more, you can run the Tab2Mage export plug-in suitable for ArrayExpress.
- Q: I have exported in Tab2Mage file, does it mean I am MIAME compliant?
- A: No, not necessarily! Tab2Mage exporter complies with Tab2Mage specifications so you will be Tab2Mage compliant. However, MIAME compliance depends very much on the kind of annotation you have supplied. Please refer to the previous question for more information about how to check for MIAME compliance in BASE using the Experiment overview function.
- Q: I have deleted the data files from BASE file system since I have imported them in tables. So I do not have data files to send to ArrayExpress anymore.
- A: You can export the data from the tables again. Go to the **Raw data** tab for each of the raw bioassays you need to export. Use the **Export** button to export the data. See Section 20.3, “The table exporter plug-in” (page 118). for more information.
- Q: I have created pooled samples in BASE. Can I export in Tab2Mage format?
- A: No, sorry, not for the moment. The Tab2Mage exporter does not support pooling events. We are working on adding this features in future version of the plug-in.

32.6. Analysis related questions with answers

- Q: Is it possible to use the formula filter to filter for `null` values (or `non-null` values)?
- A: It is possible to trick the system to filter out `null` values but not `non-null` values. Use an expression like: `ch(1) == 0 || ch(1) != 0`. 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. Menu guide

This appendix covers all menu items in BASE, including submenus. You should use it as a map to easily find your way through the menus. Some menu items are referring to other section of the documentation, that holds more information. The list of menu items is organized after three different types,

- Configurations
- Items
- Miscellaneous

Configurations

Settings		
Configure	Menu choice	Comment
Appearance of web client	BASE Preferences...	Section 6.2.4, “Preferences” (page 17)
Contact information (your own)	BASE Contact information...	Section 6.2.1, “Contact information” (page 16)
Most recently used items	The tab Most recent under BASE Preferences...	the section called “The Recent items tab” (page 19)
Other information (your own)	BASE Other information...	Useful information about yourself.
Password (your own)	BASE Change password...	Section 6.2.3, “Changing password” (page 17)
Plugin messages	The tab Plugins under BASE Preferences...	the section called “The Plugins tab” (page 19)
Server settings	Administrative Server settings...	Currently no documentation.

Miscellaneous

Miscellaneous		
Action	Menu choice	Comment
About the program	Help About...	Useful information about the program.
BASE project information	Help BASE project site...	The home page of the BASE project
Disk usage overview	Administrative Disk usage	Disk usage of users, groups etc. Currently no other documentation.
Exit from the program	BASE Logout...	You are logged out and redirected to the login page.
License information	Help License...	The GNU GPL (General Public License)
Load your current permissions	BASE Reload permissions	This updates the logged in user's permissions. See Section 7.1, “The permission system” (page 37)

Miscellaneous		
Action	Menu choice	Comment
Report program defects	Help Report a bug...	This is where you should turn if there are any defects in the program
Restart server/program	Administrative Restart BASE...	Restarts the program. Requires everyone, that were logged in, to identify themselves again with username and password.

Items

Items (A-H, I-P, Q-Z)			
Want to view...		Menu path	Comment
A-H	All items own by yourself	View All items	Gives a complete overview of all your items.
	Annotation types	Administrative Types Annotation types	Chapter 11, <i>Annotations</i> (page 60)
	Array batches	Array LIMS Array batches	Chapter 16, <i>Array LIMS</i> (page 80)
	Array designs	Array LIMS Array designs	Chapter 16, <i>Array LIMS</i> (page 80)
	Array slides	Array LIMS Array slides	Chapter 16, <i>Array LIMS</i> (page 80)
	Biomaterial lists	Biomaterial LIMS Biomaterial lists	Chapter 17, <i>Biomaterial</i> (page 85)
	Bioplates	Biomaterial LIMS Bioplates	Chapter 17, <i>Biomaterial</i> (page 85)
	Biosources	Biomaterial LIMS Biosources	Chapter 17, <i>Biomaterial</i> (page 85)
	Categories of annotation types	Administrative Types Annotation type categories	Chapter 11, <i>Annotations</i> (page 60)
	Clients	Administrative Clients	
	Data file types	Administrative Platforms Data file types	Section 12.2, "Data file types" (page 71)
	Experimental platforms	Administrative Platforms Experimental platforms	Section 12.1, "Platforms" (page 69)
	Experiments	View Experiments	Section 18.3, "Experiments" (page 103)
	Extra value types	Administrative Types Extra value types	
	Extracts	Biomaterial LIMS Extracts	Chapter 17, <i>Biomaterial</i> (page 85)
	Files and directories	View Files	Chapter 8, <i>File management</i> (page 43)
	File types	Administrative Types File types	Section 8.3, "File types" (page 50)
	Formulas	View Formulas	
	Groups	Administrative Groups	Section 24.2, "Groups administration" (page 155)
	Hardware	Administrative Hardware	Chapter 14, <i>Hardware</i> (page 77)

Items (A-H, I-P, Q-Z)			
Want to view...		Menu path	Comment
Help-texts	Hardware types	Administrate Types Hardware types	
	Administrate Clients, use the Help text tab when viewing a client.		
Hybridizations	View Hybridizations	Chapter 17, <i>Biomaterial</i> (page 85)	
I-P	Jobs	View Jobs	
	Labeled extracts	Biomaterial LIMS Labeled extracts	Chapter 17, <i>Biomaterial</i> (page 85)
	Labels	Biomaterial LIMS Labels	
	MIME types	Administrate Types MIME types	
	Messages	View Messages	
	News	Administrate News	
	Plates	Array LIMS Plates	Chapter 16, <i>Array LIMS</i> (page 80)
	Plate geometries	Array LIMS Plate geometries	Chapter 16, <i>Array LIMS</i> (page 80)
	Plate mappings	Array LIMS Plate mappings	Chapter 16, <i>Array LIMS</i> (page 80)
	Plate types	Array LIMS Plate types	Chapter 16, <i>Array LIMS</i> (page 80)
	Plugins	Administrate Plugins Definitions	Chapter 22, <i>Plug-ins</i> (page 133)
	Plugin configurations	Administrate Plugins Configurations	Section 22.4, “Plug-in configurations” (page 140)
	Plugin types	Administrate Plugins Types	Chapter 22, <i>Plug-ins</i> (page 133)
	Projects	View Projects	Chapter 7, <i>Projects and the permission system</i> (page 37)
	Protocols	Administrate Protocols	
	Protocol types	Administrate Types Protocol types	
	Quota	Administrate Quota	
	Quota types	Administrate Types Quota types	Section 24.4, “Disk space/quota” (page 158)
	Raw bioassays	View Raw bioassays	Section 18.2, “Raw bioassays” (page 99)
	Removed items	View Trashcan	Section 6.5, “Trashcan” (page 31)
Q-Z	Reporter lists	View Reporter lists	
	Reporter types	Administrate Types Reporter types	
	Reporters	View Reporters	
	Samples	Biomaterial LIMS Samples	
	Scans	View Scans	Section 18.1, “Scans and images” (page 98)
	Sessions	View Sessions	

Items (A-H, I-P, Q-Z)			
Want to view...		Menu path	Comment
	Software	Administrate Software	Chapter 15, <i>Software</i> (page 79)
	Software types	Administrate Types Software types	
	Users	Administrate Users	Chapter 24, <i>Account administration</i> (page 152)

Appendix B. Core plug-ins shipped with BASE

A categorized list of all plug-ins installed with a pristine BASE installation. Some plug-ins must be configured before use, 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 22.4.2, “Importing and exporting plug-in configurations” (page 143)).

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.

B.1. Core analysis plug-ins

Base1PluginExecuter

Simulates the plug-in runner from Base 1.2.

Since BASE 2.2, no configuration needed for the Base1PluginExecuter but may be required for the underlying BASE version 1.2 plug-ins.

JEP extra value calculator

Calculates extra values for a bioassay set.

Since BASE 2.1, no configuration needed.

JEP filter plug-in

Bioassay set filter. Expressions parsed with JEP.

Since BASE 2.0, no configuration needed.

JEP intensity transformer

Transforms the intensities of a bioassayset.

Since BASE 2.1, no configuration needed.

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.

Since BASE 2.12. Configuration is needed to register possible parameters and output files.

Normalisation: Lowess

Normalisation using LOWESS algorithm.

Since BASE 2.0, no configuration needed.

Normalisation: Median ratio

Normalisation based on median ratio.

Since BASE 2.0, no configuration needed.

¹ <http://baseplugins.thep.lu.se>

B.2. Core export plug-ins

BASEfile exporter

Exports bioassay set data to serial or matrix BASEfile format.

Since BASE 2.12, no configuration needed.

GAL exporter

Exports the features of an array design to a GAL file.

Since BASE 2.7, no configuration needed.

Help texts exporter

Exports help texts to an XML file.

Since BASE 2.0, no configuration needed.

Packed file exporter

Exports files and directories as an archive-file.

Since BASE 2.4, no configuration needed.

Plate mapping exporter

Exports plate mappings.

Since BASE 2.0, no configuration needed.

Plugin configuration exporter

Exports plug-in configurations to an XML file.

Since BASE 2.1, no configuration needed.

Table exporter

Exports table listings in the web-interface.

Since BASE 2.0, no configuration needed.

B.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 B.3.1, “Core batch import plug-ins” (page 334) 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.

Since BASE 2.4, no configuration needed.

Annotation importer

Imports annotation to any annotatable item in BASE.

Since BASE 2.4, no configuration needed.

Help texts importer

Imports help texts from an XML file into BASE.

Since BASE 2.0, no configuration needed.

Illumina raw data importer

This plug-in is used to import raw data from Illumina data files.

Since BASE 2.4, no configuration needed.

Plate importer

Imports plates from a simple flat file.

Since BASE 2.0, available configurations: 384 wells-plate² and 96 wells-plate³

Plate mapping importer

Imports plate mappings.

BASE 2.0, no configuration needed.

Plugin configuration importer

Imports plug-in configurations from an XML file.

Since BASE 2.1, no configuration needed.

Print map importer

Imports array designs from a print map.

Since BASE 2.0, no configuration needed.

Raw data importer

Imports raw data from a text file.

Since BASE 2.0, available configurations: cy3/cy5 GenePix⁴ and cy5/cy3 GenePix⁵

Reporter importer

Import reporter (probeset) information from a file.

Since BASE 2.0, available configurations:

GenePix related 96 wells⁶, 384 wells⁷, and GenePix⁸.

Affymetrix related, these samples import a small amount of information: HG-U133_Plus_2 and MG-U74Av2⁹ and HG-U133A¹⁰. The samples may work on other Affymetrix chips, the listed ones are tested and known to work.

Reporter map importer

Imports GenePix features from a gpr-file.

Since BASE 2.0, GenePix¹¹ sample configuration available.

² http://base.thep.lu.se/attachment/wiki/DocBookSupport/plate_importer_384wells.xml?format=raw

³ http://base.thep.lu.se/attachment/wiki/DocBookSupport/plate_importer_96wells.xml?format=raw

⁴ http://base.thep.lu.se/attachment/wiki/DocBookSupport/raw_data_importer_genepix-cy3_cy5.xml?format=raw

⁵ http://base.thep.lu.se/attachment/wiki/DocBookSupport/raw_data_importer_genepix-cy5_cy3.xml?format=raw

⁶ http://base.thep.lu.se/attachment/wiki/DocBookSupport/reporter_importer_96wells.xml?format=raw

⁷ http://base.thep.lu.se/attachment/wiki/DocBookSupport/reporter_importer_384wells.xml?format=raw

⁸ http://base.thep.lu.se/attachment/wiki/DocBookSupport/reporter_importer_genepix.xml?format=raw

⁹ http://base.thep.lu.se/attachment/wiki/DocBookSupport/reporter_importer_affymetrix.xml?format=raw

¹⁰ http://base.thep.lu.se/attachment/wiki/DocBookSupport/reporter_importer_affymetrix2.xml?format=raw

B.3.1. Core batch import plug-ins

The batch import plug-ins all work similarly and their usage is described in Chapter 19, *Import of data* (page 108).

Array batch importer

Imports and updates array batches in a batch.

Since BASE 2.8, no configuration required.

Array design importer

Imports and updates array designs in a batch.

Since BASE 2.8, no configuration required.

Array slide importer

Imports and updates array slides in a batch.

Since BASE 2.8, no configuration required.

Biosource importer

Imports and updates biosources in a batch.

Since BASE 2.8, no configuration required.

Extract importer

Imports and updates extracts in a batch.

Since BASE 2.8, no configuration required.

Hybridization importers

Imports and updates hybridizations in a batch.

Since BASE 2.8, no configuration required.

Labeled extract importer

Imports and updates labeled extracts in a batch.

Since BASE 2.8, no configuration required.

Raw bioassay importer

Imports and updates raw bioassays in a batch.

Since BASE 2.8, no configuration required.

Sample importer

Imports and updates samples in a batch.

Since BASE 2.8, no configuration required.

Scan importer

Imports and updates scans in a batch.

Since BASE 2.8, no configuration required.

B.4. Core intensity plug-ins

Formula intensity calculator

Calculate intensities from raw data.

Since BASE 2.0, no configuration needed.

B.5. Uncategorized core plug-ins

Spot images creator

Converts a full-size image into JPEG images for each spot.

Since BASE 2.0, no configuration needed.

TAR file unpacker

Unpacks a tar file on the BASE file system. It also supports TAR files compressed with GZIP or BZIP algorithms.

Since BASE 2.1, no configuration needed.

ZIP file unpacker

Unpacks zip and jar file on the BASE's file system.

Since BASE 2.1, no configuration needed.

Appendix C. 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.

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.MySQLInnoDBDialect` 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/base2` for MySQL
- `jdbc:postgresql:base2` 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/base2?characterEncoding=UTF-8&useCursorFetch=true&defaultFetchSize=100&useServerPrepStmts=true`

Some MySQL versions (for example, 5.0.27) contains a bug that causes some queries to fail if `useCursorFetch=true`. If you experience this problem, you have to set it to false. For more information see <http://base.thep.lu.se/ticket/509>.

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 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.

¹ <http://dev.mysql.com/doc/refman/5.0/en/connector-j-reference-configuration-properties.html>

² <http://jdbc.postgresql.org/documentation/81/connect.html>

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 J.1, “mysql-queries.xml and postgres-queries.xml” (page 357).

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 D, *extended-properties.xml reference* (page 342) 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 E, *Platforms and raw-data-types.xml reference* (page 346) 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.

export.max.items

The maximum number of items the export function should try to load in a single query. This setting exists because MySQL prior to 5.0.2 does not support scrollable result sets, but loads all data into memory. This will result in out of memory exception if exporting too many items at the same time. PostgreSQL does not have this problem. Use:

- 0 for PostgreSQL
- 0 for MySQL version 5.0.2 or above. Requires that `useCursorFetch=true` is specified in the connection url and that `defaultFetchSize=xxx` is set to a value > 0.
- As large as possible value for other MySQL versions. A low value results in more queries and slower performance when exporting data.

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 26.6.1, “Authentication plug-ins” (page 197) 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.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://java.sun.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 21.2, “Installing job agents” (page 123) 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 26.6.2, “Secondary file storage plugins” (page 199) 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. It must be a value greater than zero 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 26.6.6, “Logging plug-ins” (page 204). 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.

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 339) 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 21.1, “Upgrade instructions” (page 121)

- **true** will overwrite existing helptexts.
- **false** will leave the existing helptexts in database unchanged and only insert new helptexts.

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)

plugins.dir

The path to the directory where jar-files for external plugins should be located if they should be used with the auto-installer. All new plugins found in this directory, or in any of its sub-directories, can be selected for installation, see Section 22.1, “Installing plug-ins” (page 133). The plugging auto-installer will not be available if this property isn't defined.

Another benefit is that all plug-ins inside this directory are registered with relative paths. This means that there is a lot less hassle when using job agents to run plug-ins. Just change this setting for the job agent installation and all plug-ins will work. For plug-ins that are outside of this directory you may have to manually register the path if it is different from the main path. It will also be a lot easier if you plan to move all plug-ins to a different directory. Just move the JAR files and change this setting. There is no need to change the paths for each plug-in in the database.

Appendix D. 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.

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. 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.

Multiple configuration files

Starting with BASE 2.6 it is possible to use multiple configuration files for extended properties. This is useful only if you want to add new columns. To remove or change existing columns you

still have to modify the original extended properties file. It is rather simple to use this feature. Create a new directory with the name `extended-properties` in the same directory as the `extended-properties.xml`. In this directory you can put additional extended property files. They should have the same format as the default file.

Tip

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.

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"> • string • date
Table D.1. Attributes for the <property> tag		Note that the given types are converted into the most appropriate database column type by Hibernate.
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> .
averagable	no	<p><i>This attribute has been deprecated and replaced by the <code>averagemethod</code> attribute!</i></p> <p>If it makes sense to calculate the average of a set of values for this property or not. By default, all numerical columns are averagable. For non-numerical columns this attribute is ignored.</p>
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>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 E. 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 29.3.1, “Using files to store data” (page 269).

E.1. Default platforms/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

E.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.

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. 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 E.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 E.2. Attributes for the <property> tag

Attribute	Required	Comment
*		All attributes defined by the <property> tag in extended-properties.xml. See Table D.1, “Attributes for the <property> tag” (page 344).
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 E.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 E.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 JEP parser. It supports the common mathematical operations such as +, -, *, /, some mathematical function like, log2(), ln(), sqrt(), etc. See the API documentation for JEP 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 F. 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>: 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>: 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.

`<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 G. 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`.

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. Instead, 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.

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.

Property	Default value	Estimated execution time
agent.shortest.slots	1	< 1 minute
agent.shortest.priority	4	
agent.short.slots	1	< 10 minutes
agent.short.priority	4	
agent.medium.slots	2	< 1 hour
agent.medium.priority	3	
agent.long.slots	2	> 1 hour
agent.long.priority	3	

¹ <http://java.sun.com/javase/6/docs/api/java/lang/Thread.html>

Appendix H. 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 G, *jobagent.properties reference* (page 351) 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
Job:1
Job:5
```

Status can be either `Running` or `Paused`. For each job that is currently running, the ID is given. In the future, the **info** command may output more information. For example, there is already infrastructure code for CPU and memory usage. The only problem is that the information is not easy to get for a Java program.

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 I. migrate.properties reference

TODO

I.1. mysql-migration-queries.sql

TODO

Appendix J. Other configuration files

J.1. mysql-queries.xml and postgres-queries.xml

TODO

J.2. log4.properties

TODO

J.2.1. Migration logger

The migration logger is by default set to warn and the output will be put in a file called `migration.log`, located in the same directory as the migration is executed from (normally `base2root/bin/`).

The common logger for the migration is called `log4j.logger.net.sf.basedb.clients.migrate` and sets general properties for the whole migration logging, but each class/item transfer can also have separate levels of logging. Logging for a separate class is set by defining the level for the class logger to use. Setting the level for a sublogger is done like this:

```
log4j.logger.net.sf.basedb.clients.migrate.Classname = level
```

J.3. hibernate.cfg.xml

TODO

J.4. ehcache.xml

TODO

Appendix K. 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 29.1, “The Public API of BASE” (page 228) for more information about what we mean with the *Public API* and backwards compatible.

Note

There is no history for releases prior to BASE 2.2 because we did not actively keep track of them. We believe that if such changes exists, they are minor and does not affect many plug-ins since in those days very few 3rd-party plug-ins existed.

K.1. BASE 2.13 release

Reporters, Raw data, Features and Array design blocks are now proxied

This was previously not possible due to a Hibernate problem with stateless sessions. See <http://opensource.atlassian.com/projects/hibernate/browse/HHH-3528> for more information about this problem.

This change means that items linking to reporter, raw data, feature or array design blocks will no longer load the linked items automatically. This is usually not a problem since the proxies will be initialised if needed. The exception is when a stateless session was used to create the proxy since the stateless can't initialise proxies. In BASE, stateless sessions are only used by `DataQuery` instances, eg. queries that returns reporter, raw data or features. When this type of query is used and when linked items are used in a way that causes proxy initialization the linked item must be explicitly `FETCH JOIN`-ed by the query. Here is an example:

```
RawBioAssay rba = ...
DataQuery<RawData> rawQuery = rba.getRawData();
rawQuery.join(
    Hql.leftJoin(null, "reporter", Item.REPORTER.getAlias(), null, true));
// NOTE! Last parameter is 'true' to FETCH JOIN the reporter!!
...
DbControl dc = ...
DataResultIterator<RawData> rawData = rawQuery.iterate(dc);
while (rawData.hasNext())
{
    RawData rd = rawData.next();
    ReporterData reporter = rd.getReporter();
    int reporterId = reporter.getId();
    // Always safe since getId() doesn't cause proxy initialization

    reporter.getName();
    // The above statement will fail in BASE 2.13 if
    // the FETCH JOIN is no included
    ....
}
```

The error message to look out for is: `org.hibernate.SessionException: proxies cannot be fetched by a stateless session`

Re-attaching a detached item no longer assumes that it has been modified

The `DbControl.reattachItem(BasicItem)` method has been deprecated and replaced with `DbControl.reattachItem(BasicItem, boolean)`. The boolean parameter is used to tell BASE if the item has been modified or not while it was detached from the session. The previous behaviour of `DbControl.reattachItem(BasicItem)` was equivalent to `DbControl.reattachItem(BasicItem, true)`, but this is now changed to `DbControl.reattachItem(BasicItem, false)` since in most cases there are really no changes to the item. The problem with the old behaviour was seen in the new (in BASE 2.13) change history functionality which would create UPDATE events even when there was no change.

K.2. BASE 2.12 release

Log-2 and log-10 transformed spot intensity data is now allowed

Prior versions of BASE only allowed unlogged spot intensity values. Analysis plug-ins that operate on spot data should be updated to check the kind of values that are present in the source bioassay set and either:

- Use an appropriate algorithm if it encounters logged data
- Give an error message that says that it requires unlogged data

Plug-ins that are not aware of the type of data may produce unexpected results if they are applied on logged data. The core plug-ins that are shipped with BASE has been fixed and they should work with any kind of data. The `Base1PluginExecutor` that is used for executing BASE 1 plug-ins can be configured to work with only a specific kind of data. After upgrading to BASE 2.12 a server admin should manually update the configuration of all registered BASE 1 plug-ins with information about what kind of source data that is required and what kind of result data the plug-in produces. The default setting is that a plug-in works with any kind of data and produces the same kind of data used as source.

This change also affects formulas, which now has two additional properties: source and result intensity transform. The source intensity transform property tells BASE what kind of source data that the formula can be used with. If this property is not specified the formula can be used with any kind of data. The result intensity transform property tells BASE what kind of result the formula generates. If this property is not specified the formula is expected to generate the same kind of data as the source data. All existing user-defined formulas will not have any of the properties set. After upgrading to BASE 2.12 user should check their formulas and set appropriate values for the source and result intensity transform attributes.

The `ch()` function automatically converts logged intensities to unlogged

In order to maintain as much backwards compatibility as possible the `ch()` function will automatically convert logged data back to unlogged. This means that many formulas will continue to work unmodified, but some may create unnecessary complex formulas. Consider, for example, the log-ratio formula: $\log_2(\text{ch}(1) / \text{ch}(2))$, which will be converted to: $\log_2(2^{\text{rawCh}(1)} / 2^{\text{rawCh}(2)})$ if it is applied on logged values. A better re-write is: `rawCh(1) - rawCh(2)`.

K.3. BASE 2.11 release

Biomaterial batch importers uses a different coordinate system to target biowells

The batch importers previously used the same coordinate system for locating biowells on a plate that BASE uses internally. A 0-based numerical coordinate pair. This has now been changed to use the more logical alphanumeric 1-based coordinate system typically found on plates. As an example files should now specify A1, B2, C3 instead of [0,0], [1,1] or [2,2]. Files that use the "old" coordinate system must be updated to the new coordinate system, or the imported data will end up in incorrect wells or in no well at all. The change affects three batch importers:

- `SampleImporter`
- `ExtractImporter`
- `LabeledExtractImporter`

Note

It is still possible to use purely numerical coordinates, but they must be 1-based and not 0-based as before!

Warning

The new coordinate system only affects the batch importers. The BASE web client will still display the internal 0-based coordinate values. BASE users should be aware of this, particularly if they use the table exporter to generate template files intended for import at a later time. In this case the coordinates in the template file needs to be edited before importing.

K.4. BASE 2.10 release

Added 'entryDate' property to a lot of items

Including reporters and users. Since those two items are extendable by the server admin this addition may break a custom property with the same name.

K.5. BASE 2.9 release

Must use a database that supports UTF-8

If you have been running BASE on a database that is not using UTF-8 as the character set you need to convert the database as part of the update. Read Section L.1, "BASE 2.9 must use a database that supports UTF-8" (page 365) for more information.

K.6. BASE 2.7.1 release

`Type.BOOLEAN.parseString()`

This method now converts the empty string, "no", "0", and "false" (ignoring case) to boolean `false`. A `null` value as input still returns `null` as output. All other values are converted to `true`. Previously, all values except `null` and the string "true", was converted to `false`. The change was made to make this method behave the same as other string conversion methods.

K.7. BASE 2.7 release

Tomcat 6 or higher is required

If you have been running BASE with Tomcat 5.5 you need to upgrade your Tomcat installation before installing BASE 2.7.

K.8. BASE 2.6 release

Feature identification methods

Array design features can now be identified by three different methods: COORDINATES, POSITION and FEATURE_ID. The coordinates method was the only one supported earlier.

- Client and plug-in code that depends on features having unique COORDINATES may no longer work if used with an array design using a different identification method. Code that is, directly or indirectly, using a `FeatureBatcher` or `RawDataBatcher` are probably affected by this. For example, a raw data importer which doesn't know how to set the position or the feature ID can't import data to an array design that is using one of the new identification methods.
- The POSITION identification method will require a unique position number. This value used to be an auto-generated sequence starting at 1. The other identification methods will still do that, but when using the POSITION identification method the only requirement is that the position value is a unique positive integer. Client or plug-in code that depends on the position being a number between 1 and the number of features may fail if used with an array design using the POSITION identification method.

Data file types

The `DataFileType.setValidatorClass()` and `DataFileType.setMetadataReaderClass()` no longer verifies that the specified class names exists and implements the required interfaces. This validation will instead happen at commit time. The reason for this change is the added support for having the validator/meta data reader in external JAR files. This means that the validation can't happen until both the class names and JAR paths has been set. If a client application need validation before commit time it should use `DataFileType.getValidator()` and `DataFileType.getMetadataReader()` after the class names and JAR paths has been set.

Job.Status.ABORTING

Added the ABORTING option to the `Job.Status` enumeration. This option is used when the ABORT signal has been sent to an executing job, but the job has not yet responded to it. Code that uses the job's status to decide what action to take may fail since this is a new option. In most cases, it should be handled in the same way as if the job is EXECUTING.

Hybridization to Labeled extracts link

This link can now hold information about which sub-array a labeled extract belongs to on a multi-array hybridization. Code that is unaware of the concept of sub-arrays may find hybridizations where the number of labeled extracts doesn't match the number channels in the platform used, and that more than one labeled extract has the same label. This was previously considered as an error condition by the experiment validator. With the new scheme the validation has to be done on a sub-array basis instead of on the complete hybridization.

A similar issue arises when inheriting annotations to a raw bioassay which stems from a specific sub-array on a multi-array hybridization. This raw bioassay should only inherit annotations from the labeled extracts that are part of the same sub-array. For API compatibility reasons the

`Hybridization.getAnnotatableParents()` will still return *all* labeled extracts. Code that is calling this method in order to find the parents to a raw bioassay should instead call the new method, `Hybridizations.getAnnotatableParents(int)`, where the `int` is the sub-array index value for the raw bioassay.

A related issue arise when querying labeled extracts and joining the source events collection to use the linked hybridizations in the query. Here is an example:

```
// Find all labeled extracts on a given hybridization
Hybridization current = ...
ItemQuery<LabeledExtract> labeledExtractQuery = LabeledExtract.getQuery();

// This no longer works
// labeledExtractQuery.join(Hql.innerJoin("sourceEvents", "evt"));

// Replace the above line with these two line
labeledExtractQuery.join(Hql.innerJoin("sourceEvents", "srcevt"));
labeledExtractQuery.join(Hql.innerJoin("srcevt", "event", "evt"));

labeledExtractQuery.restrict(
    Restrictions.eq(
        Hql.property("evt", "hybridization"),
        Expressions.integer(current.getId())
    )
);
```

The good new is that the modifications makes it possible to filter the query on used quantity and sub-array. See the Javadoc for `Hybridization` to get more information.

K.9. BASE 2.5 release

Raw data types

The storage attribute of the `<raw-data-type>` has been deprecated for the `raw-data-types.xml` file. BASE will refuse to start if it finds this attribute. Raw data types which doesn't use the database for storing data should be registered as `Platform:s` instead.

Applications or plug-ins that filters on the `rawDataType` property for `RawBioAssay` or `Experiment` may need to change. The ID given to raw data types that doesn't use the database for storing data are prefixed with "platform.". The ID for the Affymetrix platform has changed from `affymetrix` to `platform.affymetrix`.

Raw bioassays

The property `spots` which used to hold the number of spots in the database OR in the file(s) has been split into two properties:

- `spots`: Now only holds the number of database spots
- `numFileSpots`: Holds the number of spots that are stored in files

Applications or plug-ins that filters on the `spots` property may no longer work as expected for file-only platforms, since this value is now always 0. They should use the `numFileSpots` property instead.

Array designs

The `ArrayDesign.isAffyChip()` method has been deprecated. Applications or plug-ins that filter on the `affyChip` property may no longer work as expected. The applications should instead filter on the `platform.externalId` and look for the value given by the constant `Platform.AFFYMETRIX`.

```
// This may no longer work
query.restrict(
    Restrictions.eq(
        Hql.property("affyChip"),
        Expressions.parameter("affyChip", true, Type.BOOLEAN)
    )
);

// Use this instead
query.restrict(
    Restrictions.eq(
        Hql.property("platform.externalId"),
        Expressions.string(Platform.AFFYMETRIX)
    )
);
```

K.10. BASE 2.4 release

Plug-in API

The `InteractivePlugin.isInContext()` method may now throw exceptions to indicate error-level messages. Messages that are returned by the method are considered as a warning message and are by default no longer shown to the users. See the section called “The `net.sf.basedb.core.plugin.InteractivePlugin` interface” (page 173) and the section called “The Plugins tab” (page 19).

Custom JSP pages for plug-ins

Plug-ins using custom JSP pages for parameter input are recommended to pass along the `requestId` parameter. The parameter is optional, but if used it will prevent data from two different plug-ins configurations to get mixed up. See Section 26.2.3, “Using custom JSP pages for parameter input” (page 182).

JEP parser

The `Jep.nodeToExpression()` and `Jep.nodeToString()` methods return `NULL` if they find an unquoted `NULL` string in the expression. This allows JEP to convert expressions like `ch(1) == NULL` into a Query API expression testing for null. To get the old behaviour use `ch(1) == 'NULL'`.

Parsing strings into numeric values

The `Type.parseString(String)` method for `Type.FLOAT` and `Type.DOUBLE` has changed its behaviour for `NaN` and `Infinity` values. The methods used to return `Float.NaN`, `Float.NEGATIVE_INFINITY`, `Float.POSITIVE_INFINITY` or the corresponding `Double` values. Since databases doesn't like these special values and eventually most values will go into the database, the `parseString` method now returns `null` instead.

Extended properties and raw data types

We have added validation code to check for invalid values. If you have modified the `extended-properties.xml` or the `raw-data-types.xml` file and they contain invalid values, you may not be able to start BASE until they are fixed. The validation is rather strict and things that may have worked before (because you were lucky or because the database has been forgiving) may no longer work. Here is an overview of the most important validation rules:

- Names and identifiers for extended properties and raw data type can only contain letters, digits and underscores. They must not start with a digit.

- Names of database tables and columns can only contain letters, digits and underscores. They must not start with a digit.
- There mustn't be any duplicate tables, columns, properties, etc. for a given context. For example, no duplicate tables in the database, no duplicate columns in a table, and no duplicate properties for a raw data type.

K.11. BASE 2.3 release

FlatFileParser

The `hasMoreData()` method has changed the order of checks made to determine the line type. The checks are now made in the following order:

1. Check if the line should be ignored.
2. Check if the line is a data footer.
3. Check if the line is the start of a new section.
4. Check if the line is a data line.

The data line check has been moved to the last since it was difficult to create settings that made sure section and data footer lines were matched correctly.

BASE 1 Plug-in executer

Changed to store information about plug-in parameters as XML in the database instead of in the original BASE version 1 plug-in definition file. Existing BASE version 1 plug-ins must be re-configured before they can be used. To do this:

1. Go to **Administrate** > **Plugins** > **Configurations**
2. Step through the configuration wizard for all BASE version 1 plug-in configurations. You do not need to change any parameters. Just click on the **Next** button until the configuration is complete.

K.12. BASE 2.2 release

BASE 1 Plug-in executer

No longer provides a default mapping between BASE version 1 and BASE version 2 raw data columns. To solve this add a **Formula** item with the same name as the BASE version 1 column name and an expression that picks the BASE version 2 raw data property. For example:

Name	BCh1Mean
Type	Column expression
Raw data type	GenePix
Channels	2
Expressions	1: raw('ch1BgMean')

Appendix L. Things to consider when updating an existing BASE installation

This document is a complete list of things that may have to be considered when updating a BASE installation to a newer version. The Section 21.1, “Upgrade instructions” (page 121) section only include the most recent information that is needed for updating the previous BASE version to the current version.

L.1. BASE 2.9 release

BASE 2.9 must use a database that supports UTF-8

If you are upgrading from BASE 2.8 or lower and your existing database is not using UTF-8 you must convert the database to UTF-8 *before you execute the* `./updatedb.sh` script.

BASE 2.9 includes a utility that can convert an existing MySQL database. After installing the BASE 2.9 files, but *before* running the `./updatedb.sh` script, execute the following on the command line:

```
cd <base-dir>/bin
./onetimefix.sh utf8 -x
```

The `-x` option makes the script update the database immediately. You can leave this option out to have it generate a SQL script file (`convert-to-utf8.sql`) instead. The script will by default not try to convert tables that it already thinks are using UTF-8. If the script for some reason is incorrect when detecting this, you can use the option `-f` to force conversion of all tables.

The conversion utility only works with MySQL. PostgreSQL users should instead use a backup and restore using as described in the PostgreSQL manual¹. Eg. dump the existing BASE database, create a new database that uses UTF8 and restore the backup into the new database.

L.2. BASE 2.7.2 release

Bug in the LOWESS plug-in affecting BASE version 2.0 -- 2.7.1

BASE 2.7.2 fixes a serious bug in the LOWESS plug-in shipped as a part of the BASE package. The bug is found in all BASE versions between 2.0 and 2.7.1, and has caused incorrect normalization values to be calculated. All data that has been normalized with the LOWESS plug-in prior to BASE 2.7.2 should be considered invalid and needs to be re-normalized with the fixed version. Downstream analysis steps that has used the incorrectly normalized data also needs to be redone. For more information about the bug see <http://base.thep.lu.se/ticket/1077>

BASE 2.7.2 includes a utility for finding all experiments/bioassay sets that includes data normalized with the LOWESS plug-in. An administrator can use this utility to extract a list of all experiments/bioassay sets that needs to be fixed. The utility can also tag the name of the found experiments/bioassay sets with `FIX LOWESS` to make it easier to find data that needs to be fixed.

¹ <http://www.postgresql.org/docs/8.1/static/backup.html>

Things to consider
when updating an existing BASE installation

The utility can't see any difference between data normalized with the old version and the fixed version. It will simply report all data that has been normalized with the LOWESS plug-in. Only use the utility a single time right after the upgrade to BASE 2.7.2.

The utility is a command line program that should be executed on the BASE application (web) server.

```
cd <base-dir>/bin
./onetimefix.sh lowess_warn -u <login> -p <password> -f
```

We recommend running the utility as the root user. The `-f` option is optional. If it is included the found experiments/bioassay sets are tagged with `FIX LOWESS`, otherwise only a list with the information is generated.

L.3. BASE 2.7 release

BASE 2.7 requires Tomcat 6 or higher

If you are upgrading from older BASE versions to BASE 2.7 or higher you must also make sure that you are running Tomcat 6 or higher. Earlier BASE versions only required Tomcat 5.5. If you are not already running Tomcat 6 you need to upgrade. Other servlet engines may work if they implement the Servlet 2.5 and JSP 2.1 specifications.

L.4. BASE 2.4.4 release

Upgrading from BASE 2.4.3 or lower to 2.4.4 or higher

Older releases of BASE 2 used to create indexes for many columns in the dynamic database. The same columns are part of the primary key for the tables so the indexes are not really needed. The result is very bad performance since the database engine sometimes get stuck in "index update" mode making the entire server very slow. BASE 2.4.4 no longer creates the indexes. Indexes on existing tables should be dropped to increase the performance. Tests have shown a 50-90% decrease in execution time for some plug-ins (<http://base.thep.lu.se/ticket/294>).

Removing the indexes is very simple. *After the server has been upgraded* following the usual instructions below, issue the the following commands:

```
cd <basedir>/bin
./dynamicdb.sh -v -dropindexes
```

Skip the `-dropindexes` option to do a dry run without actually deleting the indexes.