MÍMIR USER GUIDE

Valentin Tablan   Ian Roberts

Contents

Chapter 1

Introduction

Mímir is a multi-paradigm information management index and repository which can be used to index and search over text, annotations, semantic schemas (ontologies), and semantic meta-data (instance data). It allows queries that arbitrarily mix full-text, structural, linguistic and semantic queries and that can scale to gigabytes of text.

A typical semantic annotation project deals with large quantities of data of different kinds. Mímir provides a framework for implementing indexing and search functionality across all these data types, listed below in the order of increasing information density:

Text
All documents have a textual content. Support for full text search represents the most basic indexing functionality and it is required in most (if not all) cases. Even when semantic annotation is used to abstract away from the actual textual data, the original content still needs to be accessible so that it can be used to provide textual query fragments in the case of more complex conceptual queries.

Mímir uses inverted indexes[1] for indexing the document content (including additional linguistic information, such as part-of-speech or morphological roots), and for associating instance of annotations with the position in the input text where they occur. The inverted index implementation used by Mímir is based on MG4J[2].

Annotations
The first step in abstracting away from the plain text content is the production of *annotations*. Annotations are meta-data associated to text snippets in the documents. Mímir's view of annotations is based on that of GATE, with each annotation described by

- the document it belongs to;

- the start and end offset of the referred text snippet;

- the annotation type;

- an arbitrary set of <feature,value> pairs.

---

[1] *Inverted Indexes* are data structures traditionally used in Information Retrieval to support indexing of text.
[2] http://mg4j.dsi.unimi.it/

An annotation index supports a more generic search paradigm. Depending on the type of annotations available, the user can search across different dimensions. If, for example, the documents are annotated with occurrences of `Person`, `Location`, `Organization` entities, then searches like `{Person}, CEO of {Organization}, based in {Location}` become possible. Storage of annotation data in Mímir indexes is handled by plugins, Mímir ships with two storage plugins by default, one storing annotation data in a relational database and the other in a Knowledge Base to support richer semantic querying.

ANNIC (ANNotations In Context)[3] is a tool predating Mímir that supports the indexing of annotations, and that has been used to inform the design of Mímir.

Knowledge Base Data

Knowledge Base (KB) Data consists of an ontology populated with instances. The ontology represents the data schema and comprises a hierarchy of class types and a hierarchy of properties that are applicable between instances of classes. The instance data represents facts that are known to the systems and is typically at least partially derived from semantic annotation over documents. KB data is used to reach a higher level of abstraction over the information in the documents which enables conceptual queries such as "find all mentions of `Person`s who are employed by any organization based in Yorkshire".

A KB that is pre-populated with appropriate world knowledge can perform other generalisations that are natural to humans users, such as being able to identify Vienna as a valid answer to queries relating to Austria, Europe or the Western Hemisphere.

As mentioned above, Mímir can make use of a Knowledge Base to store information relating to annotations. The links between annotations, the textual data, and the knowledge base information are created by the inclusion into the text indexes of a set specially-created URIs that are associated with annotation data. Furthermore, URIs of entities from the Knowledge Base can be stored as annotation features.

Knowledge bases are typically represented as a collection of triples that are kept in highly-specialised and optimised triple stores, using standards such as RDF or one the versions of OWL[4]. The implementation used by Mímir is based on ORDI and OWLIM[5].

---

[3]See `http://gate.ac.uk/userguide/chap:annic`.
[4]See `http://www.w3.org/RDF/` and `http://www.w3.org/TR/owl-features/`.
[5]See `http://www.ontotext.com/ordi/` and `http://www.ontotext.com/owlim/`.

Chapter 2

Quick Start

This chapter is aimed at the impatient reader who wants a working system as quickly as possible. The technical detail is deliberately kept at a minimum so, while you will hopefully end up with something that works, you will not necessarily understand how it all fits together. For that, please read the remainder of this guide.

2.1 Set Up Your Environment

We suggest you try this on a 64 bit operating system, as that is better suited for running Mímir. A 32 bit system would also work, but the maximum sizes for the indexes wold be limited.

In order to build and run a Mímir server you will need the following pieces of software installed on your system:

Java Development Kit If you don't have one, you can download one from Oracle[1]. Make sure you get the JDK and not the Java Runtime Environment (JRE), as that would not be suitable. Once installed, make sure your `JAVA_HOME` environment variable points to the location where the JDK was installed. Make sure that the `$JAVA_HOME/bin` location is on your `PATH`.

Apache ANT version 1.8.1 or later. You can download it from `http://ant.apache.org/`. Once installed, make sure your `ANT_HOME` environment variable points to the top-level directory of your installation.

Grails version 1.3.7. You can download this from `http://grails.org`. Once installed, make sure your `GRAILS_HOME` environment variable points to the top-level directory of your installation.

Working Internet Connection The next step, described below, is the building of the Mímir library. This starts by automatically downloading all the required dependencies, so it requires a working Internet connection. Once the software is built, it can work without an remote connection.

---

[1]`http://www.oracle.com/technetwork/java/javase/downloads/index.html`

GATE Developer Mímir is an indexer for GATE Documents. The simplest way of generating some GATE documents to be indexed is by using the GATE Developer tool[2].

2.2 Build and Run a Mímir Web Application

After all the prerequisites are installed, we can move to building a Mímir application. For the purposes of this demo, we will build the `mimir-demo` application. In a real deployment you may find the alternative application (`mimir-cloud`) more suitable.

The following steps will help you build the `mimir-demo` application. Commands that you have to execute are formatted in a distinctive font `like this`.

1. **Download the Mímir sources**, if you do not already have a copy. Pick the appropriate version from one of the tag directories listed in the `tags` directory at `http://gate.svn.sourceforge.net/viewvc/gate/mimir/`, then click on the *Download GNU tarball* link. If you want to use the latest development version (which could be unstable!), use the `trunk` directory instead of a tag. Unpack the downloaded file to a location on your computer. If using Windows we recommend not using the popular Winzip utility, as that sometimes mangles the file names. 7-Zip[3] and the Cygwin "tar" utility are known to work correctly in this respect, and other free archiving tools are available that support the `.tar.gz` format.

   Alternatively, if Subversion[4] is installed on your system, you can check out the sources directly from our SourceForge repository by executing the following command:
   `svn co https://gate.svn.sourceforge.net/svnroot/gate/mimir/trunk mimir`. The URL used in the command is for the `trunk` version; adjust accordingly to point to one of the tags, if you require a release version instead. This will create a new directory called `mimir` containing all the source files.

2. **Build the library:** change to the top level directory where you unpacked the downloaded Mímir sources. If you can see the `mimir-core`, `mimir-client`, *etc.* directories, then you are in the correct directory. Execute the `ant` command. This will download all the required dependencies, compile all the Mímir libraries, and build the `mimir-web` Grails plugin.

3. **Generate the demo application:** to use Mímir you need a web application that you can access from your browser. The simplest Mímir-based web application is `mimir-demo`, which can be automatically generated and built by executing the `ant mimir-demo` command. This will produce a new directory, named `mimir-demo`, located under the top level Mímir directory.

4. **Run the demo application:** change to the newly created `mimir-demo` directory (`cd mimir-demo`) and execute the `grails prod run-app` command. This will start the application and will notify you which URL you should use in your browser to access it (normally `http://localhost:8080/mimir-demo/`).

---

[2]GATE Developer is available at `http://gate.ac.uk/download/`. Usage of GATE Developer is beyond the scope of this document, so we assume you have a basic understanding of how to use it. If not, a good place to start is the tutorials page at `http://gate.ac.uk/demos/developer-videos/`.

[3]`http://www.7-zip.org/`

[4]`http://subversion.apache.org/`

2.3 Create, Populate, and Search an Index

5. **Set-up your new Mímir application:** navigate to the administration page. You will be prompted to configure your Mímir instance. After clicking the link, enter the path to a local writable directory where new indexes will be created, and click the *Update*.button.

6. **Create a new index:** navigate back to the administration page (by clicking the link at the top of the page). Under the *Local Indexes* section, click the *create a new local index* link. Give it a name (e.g. 'test'), and click the *create* button. Back on the administration page, click the name of the newly created index. This will take you to the index details page, where you can find the *Index URL* attribute. Make a note of its value, as you will need it later.

7. **Populate the new index:**

   (a) Start GATE Developer, load the ANNIE application (Main Menu → File → Load ANNIE System → with Defaults).

   (b) Open the CREOLE Plugin Manager ((Main Menu → File → Manage CRE-OLE Plugins), and add a new plugin directory pointing at the `mimir-client` directory inside the Mímir distribution. Make sure the new plugin is loaded by checking the appropriate check-box.

   (c) Load a new instance of *Mímir Indexing PR* (Main Menu → File → New Processing Resource → Mímir Indexing PR), and add it to the end of the ANNIE application.

   (d) Make sure that the `mimirIndexUrl` parameter for the new PR is set to the *Index URL* value obtained at Step 6.

   (e) Load some test documents (e.g. some web pages from news web sites), create a GATE Corpus, add all the documents to the corpus, and set the newly corpus as the target for the ANNIE application.

   (f) Run the ANNIE application. This will annotate the documents created during the previous step. The Mímir Indexing PR instance will make sure the annotated documents are sent for indexing to your new Local Index.

8. Go back to the index details page in your browser. Click the *Close* button, and wait for the index to finish closing. When done, go back to the main administration page.

9. **You can now search the new index** by clicking the *search* link next to the name of your new index.

To shut down the running web application, create a file named `.kill-run-app` in the *mimir-demo* directory, and wait for the application to shut itself down.

To deploy the Mímir web application to an application server (such as Apache Tomcat) run the **`grails prod war`** command in the mimir-demo directory. A `mimir-demo-{version}.war` file will be created for you in the `target` sub-directory.

Chapter 3

Installing and Managing Mímir

3.1 Mímir Architecture

Mímir is divided into a number of related modules.

mimir-core The core Java library to create a Mímir index on disk, add GATE documents to the index, and then query the index once it has been built. Also provides some abstract helper classes for the annotation storage layer, but not the actual storage implementations (which are provided by separate plugins, leveraging the CREOLE plugin framework of GATE Embedded).

plugins/db-h2 The default annotation storage implementation. This stores annotation data using H2, an in-process embedded SQL database.

plugins/sesame An alternative annotation storage implementation that stores its annotation data in a triple store using the Sesame API (`http://www.openrdf.org/`). Annotations with an "inst" feature are treated as links into the knowledge base, supporting richer semantic queries.

plugins/sparql A helper that is layered on top of a generic storage implementation to provide semantic querying against a separate knowledge base, accessible at a SPARQL endpoint.

plugins/measurements A special-purpose helper for Measurement annotations created by the GATE `Tagger_Measurements` plugin. Queries are normalised into SI units so can retrieve annotations that express the same measurement in different terms (e.g. an annotation for "90 seconds" would match a query for "1 to 2 minutes").

mimir-client The client side of the Mímir remote protocol, to support distributed indexing and querying.

mimir-web A Grails plugin (`http://grails.org`) providing both the user interface to create and query indexes over the web, and also the server side of the remote protocol to expose several distributed Mímir indexes as a single *federated* index for clients. This is provided as a plugin rather than an application to make it more easily customisable.

mimir-demo The simplest possible Grails application using the above plugin. This application is not actually included in the source tree; it is instead generated by the build process. See Section 3.2 for details on how to do that. Once created, it is ready to use as-is or as a base for customisation to add security, look and feel customisation, etc.

mimir-cloud A fully fledged Grails application, that uses the mimir-web plugin and also includes security support. This is the exact implementation used for Mímir servers supplied on the GATECloud.net platform[1]. This application should be suitable without any modifications for most users.

3.2 Building and Running a Mímir Web Application

3.2.1 Which Mímir Web Application

The `mimir-core` Java library provides support for indexes that are represented as an on-disk directory – named *Local Indexes* in the rest of this document (see the discussion about index types in Section 3.3.1). To get the full functionality of Mímir (including support for *Remote* and *Federated* indexes, as well as user interfaces for system administration and searching indexes) you will need to build and run a web application. All the web elements of Mímir are implemented as the `mimir-web` Grails plugin, which can easily be included in any Grails-based web application. The standard Mímir distribution provides two such web applications:

mimir-demo The `mimir-demo` module is intended to be the simplest possible demonstration of how to use the `mimir-web` Grails plugin in a Grails application. This application is suitable for experimentation and local use, but as it includes no security or authentication support it may not be appropriate for production deployment. These concerns are addressed further in Section 7.3.

If you are just starting with Mímir, or you are only experimenting with Mímir indexes that you do not intend to open to remote users, then this is the recommended web application to use.

You should also use `mimir-demo` (as a starting point) if you are an experienced Grails developer and need a basic application to which you intend to add your own security solution.

mimir-cloud The `mimir-cloud` application is the actual software implementation behind the Mímir servers deployed on the GATE Cloud platform[2]. In addition to the Mímir plugin, it also includes some basic support for user authentication (using the Spring Security Grails plugin), and support for packaging and downloading local indexes. This is probably the most suitable choice if you just need a stand-alone web application with Mímir functionality, and you do not intend to develop your own security solution.

The rest of this chapter talks about the `mimir-demo` application, as the simpler of the two. However, everything said also applies to the `mimir-cloud` application.

---

[1]`https://gatecloud.net`: a platform for running GATE-based processes on the cloud.
[2]`https://gatecloud.net`

3.2.2 Prerequisites

To build the Mímir demo app you will need:

- A Java 6 JDK. Mímir has been tested with the Sun/Oracle and OpenJDK JVMs on Linux and the Apple JVM on Mac OS X.

- Apache Ant 1.8.1 or later.

- Grails (`http://grails.org`). The Mímir plugin was developed with version 1.3.7 but later 1.3.x versions should also work (you will need to run `grails upgrade` in the `mimir-web` directory if you use a version other than 1.3.7). You need to set the JAVA_HOME environment variable to point to your JDK, the GRAILS_HOME environment variable to point to your Grails installation and put $GRAILS_HOME/bin on your PATH.

- Google Web Toolkit (`http://code.google.com/webtoolkit`). The GUS web interface is implemented in GWT, using the Grails gwt plugin, but a standalone installation of GWT is required to build the app. Version 2.4.0 or later is required, and you need to set the GWT_HOME environment variable to point to your GWT installation.

While not strictly a pre-requisite, Mímir performs much better on 64-bit systems than on 32-bit ones, partly due to simply being able to assign more memory to the process, but also because the larger address space allows MG4J to memory-map many of the files that make up the index.

To run a local instance of Mímir you can use the standard `grails prod run-war` command, but to deploy a production instance you will need a separate servlet container such as Tomcat.

3.2.3 Building

There is a top-level Ant build.xml file that should build all the modules in the correct order. To do that simply change to the top level directory containing the Mímir source code, and run the `ant` command. To perform the same build process manually, you need to change to the following directories and run the following `ant` commands in this order:

1. `mimir-core`: `ant publish`

2. `mimir-client`: `ant publish`

3. `plugins`: run `ant` in each sub-directory of the plugins directory in turn (order is not important here, the plugins do not depend on one another).

4. `mimir-web` - `grails compile` followed by `grails compile-gwt-modules`.

Once the plugin is built you need to generate the source folder for the demo application. You can do that by running the `ant mimir-demo` command in the top level folder of the Mímir source tree. This will create a new directory (named `mimir-demo`) inside the top

level directory, which contains a simple Grails-based web application, that uses the Mímir Grails plugin.

The next step is to configure the `mimir-demo` web application, and is described in the following Section.

3.2.4   Configuring

When the Mímir Grails plugin is installed into a Grails application, it creates a base configuration file at `grails-app/conf/MimirConfig.groovy`. This file contains a number of settings that affect the running of the Mímir components. In many cases the default options will be sufficient, but you should nevertheless check the configuration and make sure it is appropriate for your needs.

```
1  gateInit {
2    gateHome = "WEB-INF/gate-home"
3    userConfigFile = "WEB-INF/gate-home/user.xml"
4  }
```

Since Mímir is based on GATE, the plugin initialises the GATE environment at start-up. These parameters control the initialisation process. In most cases you can leave the values at their defaults, which use a deliberately cut-down set of GATE configuration files installed into `web-app/WEB-INF` by the Mímir Grails plugin. The available parameters are gateHome, pluginsHome, siteConfigFile, userConfigFile and builtinCreoleDir, which correspond to the standard settings on the Gate class, and their values can be either absolute URLs (such as `file:/opt/gate`) or paths which are taken relative to the web application (i.e. the web-app directory of the Grails application).

```
1  plugins {
2    h2 = "../plugins/db-h2"
3    myCustomPlugin = "file:/data/mimir/plugins/myCustomPlugin"
4  }
```

This section specifies the Mímir plugins that should be loaded, and determines the kinds of annotation helpers you will be able to use in your indexes. You generally need at least one of the standard `db-h2` and/or `sesame` plugins to be able to do anything useful with Mímir, and you may want the `measurements` plugin as well if you will be searching on Measurement annotations and/or the `sparql` plugin if you have an external knowledge base. Section 4.1 has more information about the standard annotation helpers, and section 7.1 discusses how to implement your own custom ones.

Mímir uses the GATE plugin mechanism, so Mímir plugins are actually very simple CREOLE plugins[3], used to add a set of `jar` files to the current classpath.

Plugins can be specified either as absolute URLs or as paths relative to the Grails application base directory. Absolute URLs will be loaded as such both in `run-app` and in WAR deployments, but plugins specified as relative paths are treated slightly differently. They will be loaded directly from the specified paths in `run-app`, but when building a WAR

---

[3]See `http://gate.ac.uk/userguide/chap:creole-model`.

file the referenced plugins will be packaged inside the WAR file and loaded from there at runtime.

```
1  queryTokeniserGapp =
2      "WEB-INF/gate-home/default-query-tokeniser.xgapp"
```

Whereas GATE's usual data model deals with annotations in terms of their *character* offsets from the start of the document, Mímir deals in terms of *tokens*. Queries for plain text strings in Mímir must be tokenised before they can be matched against the index, and the tokenisation applied to the queries must match that applied to the documents that have been indexed. The Mímir Grails plugin uses a saved GATE application state (gapp file) to perform query tokenisation, the location of which is specified here. Again, the location can be an absolute URL or a path relative to the web-app directory, and the default value refers to a simple app installed by the Mímir Grails plugin that contains a single ANNIE tokeniser with its default settings.

If your tokenisation requirements are more complex, you can provide your own saved application, or alternatively you can use your application's `resources.xml` or `resources.groovy` to override the definition of the Spring bean named "queryTokeniser" – this bean must define a GATE LanguageAnalyser that will produce annotations of type Token in the default annotation set.

Note that because of the special handling at build time of plugins referenced as relative paths (see above), if you want to load additional plugins into a WAR-packaged Mímir using runtime settings in an external configuration file, then the plugins must be specified using absolute URLs, i.e. `gate.mimir.plugins.custom = "file:/opt/mimir/plugins/custom"`. Relative plugin paths are ignored at runtime by Mímir when running from a WAR deployment. However, since Mímir plugins are simply standard GATE CREOLE plugins and the Mímir Grails plugin initialises GATE Embedded using Spring you can load extra plugins relative to your web app by using Spring configuration in `WEB-INF/spring/resources.xml` (see `http://gate.ac.uk/userguide/sec:api:spring` for details):

```
1   <beans xmlns="http://www.springframework.org/schema/beans"
2       xmlns:gate="http://gate.ac.uk/ns/spring"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="
5         http://www.springframework.org/schema/beans
6         http://www.springframework.org/schema/beans/spring-beans.xsd
7         http://gate.ac.uk/ns/spring
8         http://gate.ac.uk/ns/spring.xsd">
9     <gate:extra-plugin>WEB-INF/custom-plugin</gate:extra-plugin>
10  </beans>
```

3.2.5 Running

The easiest way to run the Mímir demo web app is to use the normal Grails commands `run-app` or `run-war`. For performance, `grails prod run-war` is preferable. For anything more than the smallest toy index it is advisable to increase the memory available to Mímir by using the JAVA_OPTS environment variable. For example (using bash or a similar POSIX shell):

```
$ JAVA_OPTS='-Xmx4G' grails prod run-war
```

To shut down a web app started using `run-app` or `run-war`, simply create an empty file in the `mimir-demo` directory named ".`kill-run-app`". Grails watches for this file and will shut down gracefully when it detects that the file has been created.

For production deployments, a better option is to build a WAR file using `grails prod war` and deploy that to a standalone servlet container such as Apache Tomcat. If you are using Ubuntu or Debian GNU/Linux, it is better to download the standard Tomcat ZIP package from Apache and use that rather than installing the Tomcat available through `apt-get` as the latter is configured by default with a security manager that interferes with Mímir.

When deployed to a servlet container the demo application reads configuration at runtime from two locations using the Grails standard "externalised configuration" mechanism:

- `WEB-INF/classes/mimir-app-config.groovy` inside the web application.

- `mimir-config.groovy` in the working directory of the Java process.

Any values in these files override values specified in `MimirConfig.groovy` or the main application `Config.groovy`. For production deployments, you should be sure to specify the public URL of your Mímir server in one of these configuration files. For example:

```groovy
1  gate.mimir.indexBaseDirectory = "/data/mimir/indexes"
2  grails.serverURL = "http://example.com/mimir"
3  // or just http://example.com if you have deployed Mímir
4  // as the ROOT web application
```

3.3 Indexes in Mímir

3.3.1 Types of Index

A single instance of Mímir can host several indexes. Mímir supports *local* indexes, stored in the file system of the Mímir server, and *remote* indexes, which are a view of an index hosted in another Mímir instance (possibly on a different machine). Several indexes (of any type) can be combined into a *federated* index, which presents the group of indexes as a single virtual index. All the indexing and searching functionality of Mímir applies equally to all three index types.

Each Mímir index has a *state*, and the operations that can be performed on the index depend on which state it is currently in. When first created, a local index will be in the *indexing* state, meaning it is waiting for documents to be added to the index. When all the documents have been added to the index an administrator will close the index, putting it into the *closing* state. For large indexes the closing process can take several hours, and when it is complete the index will enter the *searching* state, at which point it is available for querying. The other possible state for a local index is *failed*, indicating a problem with the index. Typically a failed index will need to be deleted by the administrator. Thus it is apparent that an index cannot be used simultaneously for searching and indexing. An existing set of index files can

be imported into a running Mímir instance as a local index, which will then immediately
be in the *searching* state.

Remote indexes inherit their state from the remote server, and federated indexes inherit
their state by combining the states of their component indexes. A federated index may
occasionally appear in the *working* state if its component indexes are not all in the same
state (for example if some of them have started closing but others are still in *indexing*
mode), but the working state will usually resolve to a normal state once the component
indexes have synchronised.

Note that once a local index has moved from *indexing* to *closing* to *searching* it is not
possible to add more documents to the same index. The suggested way to add to an index
is to create a new index to hold the new documents, fill it, close it, and then create a
federated index consisting of the original index plus the new one (or if the original index
was itself federated, add the new index to the existing federation).

A typical setup for a large-scale indexing task would be to have a number of identical
"slave" servers running Mímir, each with a single local index. A single "master" Mímir
instance could then have one remote index definition pointing to each of the slaves, and a
single federated index combining the remote indexes. This federated index would be the
point of entry into the system and would share out indexing jobs (round-robin among the
slaves) or search requests (to all the slaves in parallel) as appropriate.

3.3.2   Creating a Local Index

Indexes in Mímir are managed through the web interface. The front page of a newly-
installed Mímir is shown in Figure 3.1. The *index templates* mentioned at the bottom, are
used to define the properties of new indexes, and are described in more detail in Chapter 4.
The Mímir Grails plugin provides a single example template based on ANNIE annotation
types.

To create an empty local index ready to receive documents for indexing, select the *create
a new local index* link. This will present a form (Figure 3.2) asking for the name of the
new index and the template from which it should be created. The "Document URIs are ex-
ternal links" option affects the way documents are presented in the search interface. Every
document in Mímir is identified by a URI, and if you intend to use document URIs that
are actually resolvable URLs (for example if your documents came from a web crawl) then
you should select this option to add a link to the original document to the search results.
If the document URIs will not be resolvable URLs then leave the option un-selected. The
index will be assigned a unique identifier and a new directory will be created under the
`indexBaseDirectory` you configured earlier to hold the index data. The newly-created
index will be in the *indexing* state (see Figure 3.3), ready to receive documents for indexing.
For details of how to submit documents to the index, see Chapter 4.

This *index information* page can be accessed at any time by clicking the link for the relevant
index name from the Mímir front page (Figure 3.4). Once all the documents to be indexed
have been submitted the index can be closed using the *Close* link on the index information
page. This will change the state of the index to *closing* as described above and begin the
closing process. The information page will show a live-updating progress bar (Figure 3.5)
giving some indication of the time remaining until the index has completely closed.
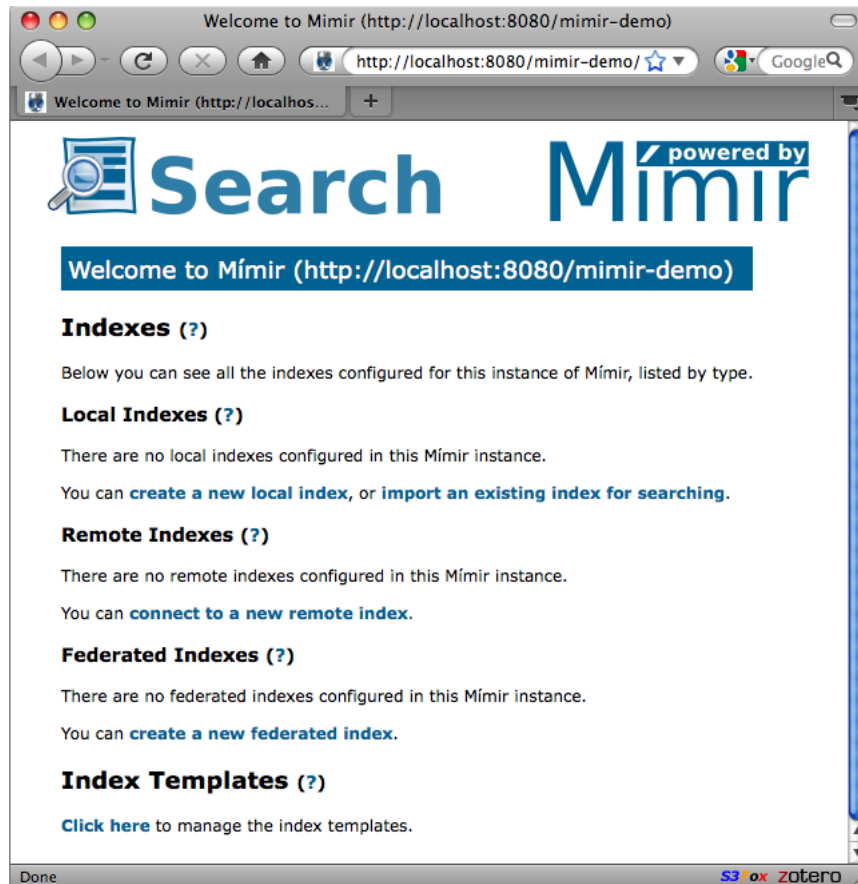
Figure 3.1: The default front page of a new Mímir

When the closing process is complete (the progress bar reaches 100%) the index will switch into *searching* mode, and the index can then be searched using the tools described in Chapter 5.

3.3.3   Working with Remote and Federated Indexes

The architecture of Mímir is designed to make working with remote and federated indexes as transparent as possible. The setup process will obviously vary for the different index types, but once created the process of submitting documents for indexing or of performing queries is exactly the same for all indexes.

Remote indexes

A *remote* index is a mechanism whereby one Mímir instance can transparently index documents in, or send queries to, an index that is located in a different Mímir instance, typically running on separate hardware. To connect one *master* Mímir instance to an index
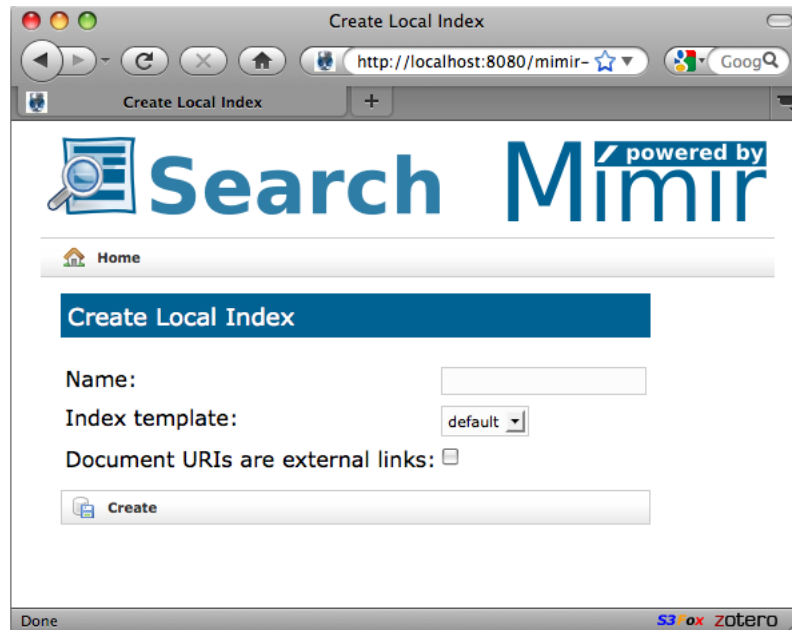
Figure 3.2: Creating a new local index

Figure 3.3: Results of creating a new local index

Figure 3.4: List of local indexes on the Mímir front page
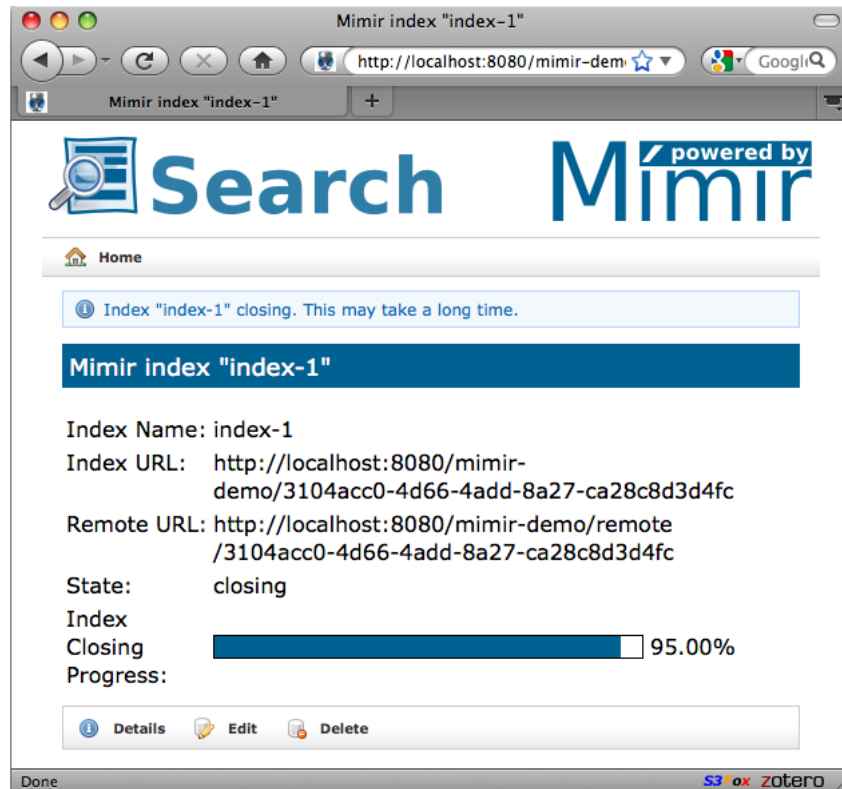
Figure 3.5: Closing a local index

running in another *slave* instance, first visit the index information page for the relevant index on the slave and make a note of its *remote URL* (typically a URL of the form `http://server:port/mimir/remote/{UUID}`). Now on the front page of the master instance, select the *connect to a new remote index* link. This will present a form (Figure 3.6) asking for a name for the remote index (which need not be the same as the name of the index on the slave), and a *remote URL* which is the one you made a note of from the slave above. You should never create a remote index pointing to another index in the same Mímir instance. Such a configuration is not supported and will lead to errors!

The remote index defined on the master server will synchronise its state with that of the underlying index on the slave, and once created will be usable exactly like a local index. However remote indexes are rarely used directly, as in most cases it is more efficient to operate on the slave instance itself. The main benefit of remote indexes comes when they

Figure 3.6: Connecting to a remote index

are used as part of a *federated index*.

Federated indexes

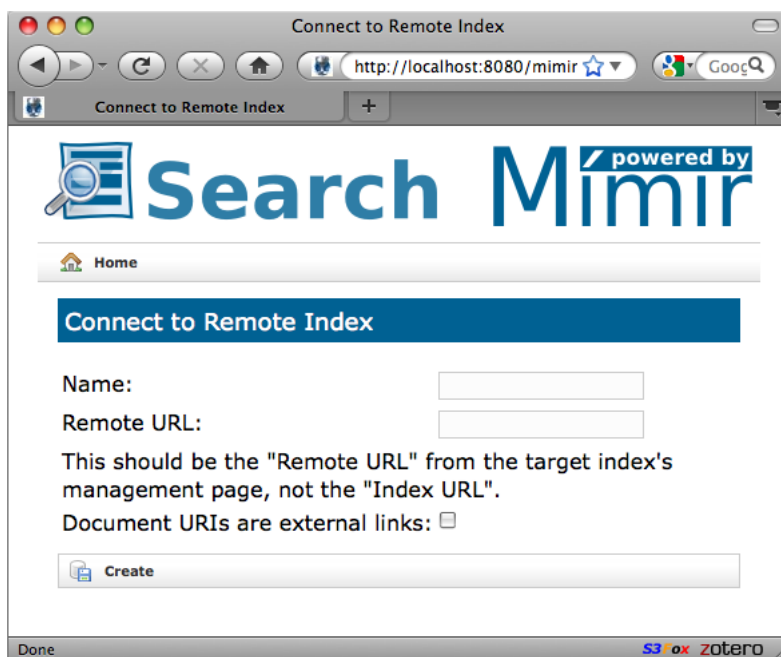A *federated index* is a device to bundle several indexes (which can themselves be local, remote or federated) together so they can be used as a single index. Documents for indexing are shared out between the component *sub-indexes*, and searches are performed by all sub-indexes in parallel. Thus a federation of five indexes each containing 200,000 documents will typically run queries faster than a single index containing 1 million documents. To create a federated index, go to the Mímir front page and select the *create a new federated index* link. This will present a form (Figure 3.7) asking for a name for the federated index. The form also includes a multiple-selection list to specify the sub-indexes to be included in the federated index. Select the appropriate entries from this list using the usual multiple list selection mechanism (ctrl-click on Windows or Linux, cmd-click on Mac OS X) and press the *Create* button to create the index. Once created the federated index will be usable exactly like a local or remote index.

3.3.4   Deleting Indexes

If an index registered with Mimir is no longer required it can be deleted by selecting the *Delete* button from the index information page (accessible by clicking on the name of the relevant index on the Mímir front page). For remote and federated indexes this simply deletes the "registration" of the index with Mímir, which can be easily re-created as above.
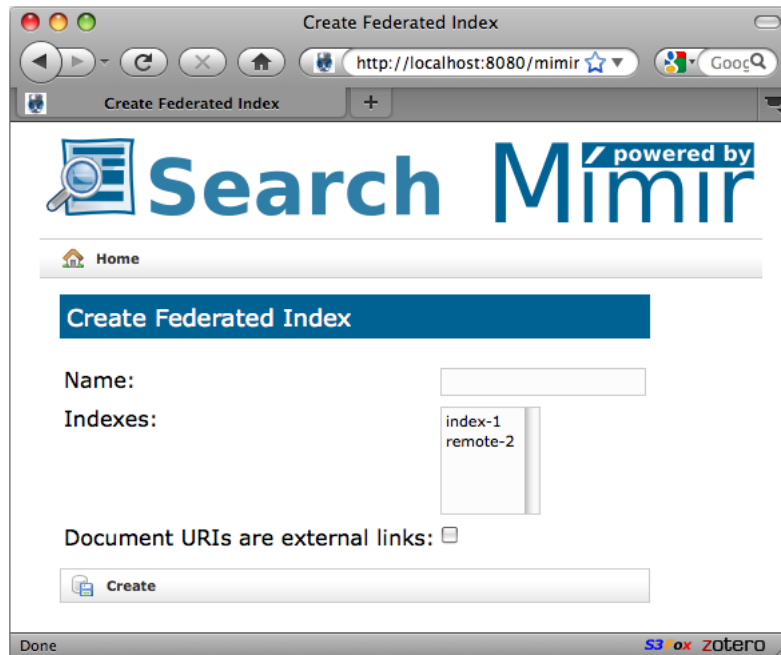
Figure 3.7: Creating a new federated index

For local indexes it also offers the option to delete the underlying index files from disk. If a local index in *searching* state is deleted without deleting the disk files then the index can be re-created later using the *import an existing index for searching* option from the Mímir front page. However, if a local index in the *indexing* or *closing* state is deleted without having been properly closed then the index files will be unusable and will need to be deleted manually.

Mímir will not allow the deletion of an index which is currently part of a federated index in the same Mímir instance. To delete such an index, it must first be removed from the federated index. This guarantee only applies to indexes within a *single* Mímir instance — Mímir does not prevent the deletion of an index on a slave instance which is being used as a remote index by a master instance (it prevents the deletion of the remote index definition in the master but not the slave index it points to). However to do so would put the remote index on the master (and hence any federated index that it is part of) into the *failed* state, preventing further use until the problem is resolved.

3.4 "Deleting" Documents from a Mímir Index

While Mímir indexes are not directly modifiable once they have been created, there are situations in which it is necessary to remove documents that should not have been indexed in the first place, or documents that may be considered libellous, etc. To support this, Mímir provides a mechanism to mark individual documents in the index as "deleted", and any documents so marked will be excluded from future queries. It is not possible to completely delete the data from the index files on disk, short of completely re-building the index from

scratch, but documents marked as deleted are not accessible through any of the public
Mímir APIs or user interfaces.

To mark a document as deleted (or to remove an existing deletion marker, making the
document available for queries again), use the "Manage deleted documents" link from the
index's administration page. This will present the screen shown in figure 3.8, with a text
box into which you can type one or more (space-separated) document IDs, and choose
whether to mark them as deleted or as "not deleted" (i.e. to remove any existing deletion
markers for those document IDs).
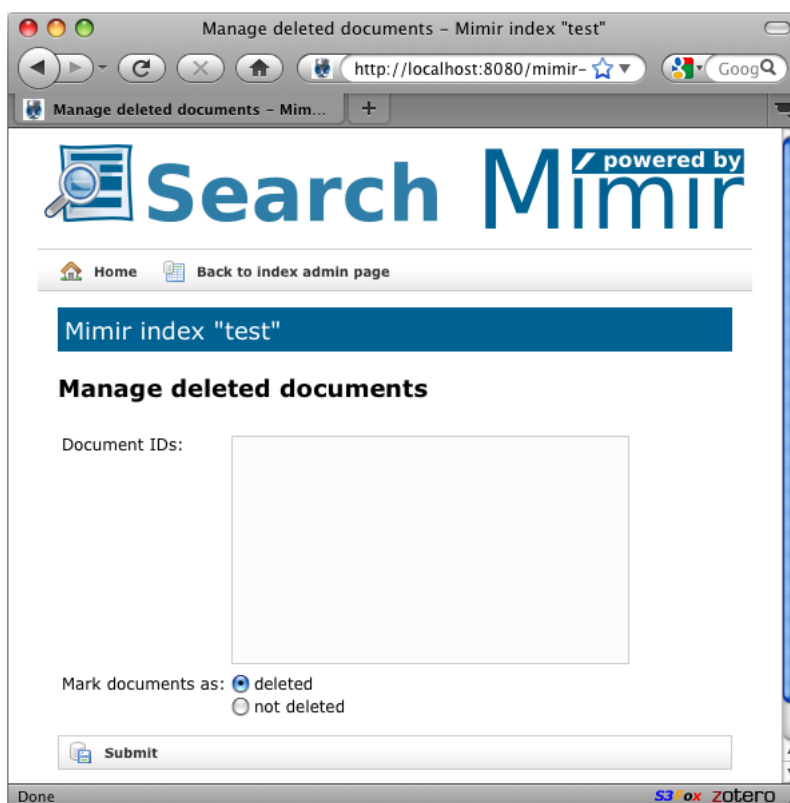
Figure 3.8: Managing deleted documents

Note that the IDs required here are not the URIs that were provided with the documents
when they were indexed, but the internal Mímir document IDs which are numbers start-
ing from 0, as returned in the hit lists and "docStats" by the Mímir query APIs (see sec-
tion 5.2.1). In the GUS interface (section 5.2.2) the document ID is part of the URL to the
"cached" document text, `.../search/gusDocument/<documentID>?queryId=....`

Chapter 4

Indexing Documents with Mímir

Mímir is designed to index semantically annotated documents. It accepts as input GATE documents[1] and produces a set of indexes as a result. The way the text and annotations of the input documents are converted into indexes is controlled through configuration options.

4.1 Configuring the Indexer

In the Mímir web interface, the configuration of a new index is represented by an *index template*. This specifies:

- which annotation types and features to index

- which annotation sets contain these annotations

- (optionally) which document features should be indexed

- how to handle the document format and metadata

Index templates can be managed using the "Click here to manage the index templates" link at the bottom of the Mímir front page. An index template is specified in a structured "domain specific language" using Groovy — Listing 4.1 shows an example of the default template provided by the Mímir Grails plugin.

---

[1]`http://gate.ac.uk/userguide/chap:corpora`

```
1    import gate.creole.ANNIEConstants
2    import gate.mimir.SemanticAnnotationHelper.Mode
3    import gate.mimir.index.OriginalMarkupMetadataHelper
4    import gate.mimir.db.DBSemanticAnnotationHelper as DefaultHelper
5
6    tokenASName = ""
7    tokenAnnotationType = ANNIEConstants.TOKEN_ANNOTATION_TYPE
8    tokenFeatures = {
9      string()
10     category()
11     root()
12   }
13
14   semanticASName = ""
15   semanticAnnotations = {
16     index {
17       annotation helper:new DefaultHelper(annType:'Sentence')
18     }
19     index {
20       annotation helper:new DefaultHelper(annType:'Person',
                                        nominalFeatures:["gender"])
21       annotation helper:new DefaultHelper(annType:'Location',
                                        nominalFeatures:["locType"])
22       annotation helper:new DefaultHelper(annType:'Organization',
                                        nominalFeatures:["orgType"])
23       annotation helper:new DefaultHelper(annType:'Date',
                                        integerFeatures:["normalized"])
24       annotation helper:new DefaultHelper(annType:'Document',
                                        integerFeatures:["date"], mode:Mode.
                                        DOCUMENT)
25     }
26   }
27   documentRenderer = new OriginalMarkupMetadataHelper()
28   documentMetadataHelpers = [documentRenderer]
```

<div align="center">Listing 4.1: The default index template provided with Mímir</div>

The various sections of the template are as follows:

Imports

```
1    import gate.creole.ANNIEConstants
2    import gate.mimir.SemanticAnnotationHelper.Mode
3    import gate.mimir.index.OriginalMarkupMetadataHelper
4    import gate.mimir.db.DBSemanticAnnotationHelper as DefaultHelper
```

The template can optionally start with import statements to import any Java classes that are used further on in the template.

Token Definitions

```
6    tokenASName = ""
7    tokenAnnotationType = ANNIEConstants.TOKEN_ANNOTATION_TYPE
```

```
 8    tokenFeatures = {
 9      string()
10      category()
11      root()
12    }
```

The next section of the template deals with the *tokens* that Mímir will base its index on. Mímir sees every document as a stream of tokens rather than a stream of characters, and all the annotations indexed by Mímir are stored in terms of their starting and ending *tokens*, not character offsets. Thus for Mímir to work correctly it needs to know how to split up the document into tokens and what information to store about each token. For this purpose it uses GATE annotations, and indexes the values of features on the annotations.

The following options can be configured:

tokenASName The name of the annotation set in which the token annotations can be found (for example `tokenASName = "mimir"`). To use the default annotation set, which has no name, this may be left unspecified, or explicitly set to the empty string: `""`, or to `null` (without quotes).

tokenAnnotationType The annotation type that should be used as tokens. This entry is required, and can generally be simply set to the default `ANNIEConstants.TOKEN_ANNOTATION_TYPE` (with a suitable `import` at the top of the template).

tokenFeatures A block of code giving the features from each token annotation that should be indexed.

The *tokenFeatures* block should list the features to be indexed as shown in the example, each feature name followed by parentheses. For advanced users an MG4J `TermProcessor` instance may be provided inside these parentheses. By default, if no term processors are specified, the *first* feature is converted to lowercase and the subsequent features are not modified. Since terms in a query are processed using the same processor as those in the index, this has the effect of making searches on the first feature case-insensitive, and searches on the other features case-sensitive. To stop any processing being done, you should supply a `it.unimi.dsi.mg4j.index.NullTermProcessor` value, by specifying e.g. `string(NullTermProcessor.getInstance())`, after including the relevant `import` statement at the top.

Semantic Annotations

```
14    semanticASName = ""
15    semanticAnnotations = {
16      index {
17        annotation helper:new DefaultHelper(annType:'Sentence')
18      }
```

The next section defines the *semantic annotations* that Mímir will include in the index. Each `index` block defines a set of semantic annotation types that will be indexed and stored together in one index. The choice of how to group annotation types together into indexes

can affect the indexing speed, as the annotations within one index are processed sequentially by a single thread, whereas types in separate indexes can be indexed in parallel.

Each annotation type to be indexed is introduced by "`annotation`". This is a method call in the Groovy DSL which takes the following named arguments:

helper The *semantic annotation helper* Java object that should be used to index this annotation type.

type The annotation type that is to be indexed. When using the default semantic annotation helper types this can be omitted.[2]

Semantic Annotation Helpers

Semantic annotations are stored in special indexes that associate URIs with document positions. During indexing, the role of the helper implementations is to store the necessary information about each annotation to be indexed in a persistent form and return one or more URIs that identify it.

One could make a distinction between *generic* semantic annotation helper types, which can be configured to handle any annotation type and features, and *special-purpose* helpers that are designed to handle specific annotation types. Mímir supplies two generic helper implementations in the `db-h2` and `sesame` plugins[3] that store annotation information in a relational database and a knowledge base respectively. For the most standard cases, one or other of these default helper implementations should be sufficient. One sample special-purpose helper for `Measurement` annotations (as generated by the GATE `Tagger_Measurements` plugin) is also provided, in the `measurements` plugin. This is intended both to be useful in its own right and to serve as a template for how to implement your own helpers for other complex annotation types. The `sparql` plugin provides a helper that can wrap any other helper and add the ability to query for URI-valued features by making a query to a SPARQL endpoint. These plugins are discussed in detail in chapter 6.

Note for users upgrading from Mímir 3.2.0 and earlier: the previous index template DSL style using the annotation type as the method name and the nominalFeatures etc. as parameters is still supported but should be considered deprecated. You should consider porting your index templates to the new style, as support for the old style may be removed in a future release.

Document Features

Starting with Mímir version 3.4.0, annotation helpers can also be used to index document features.

---

[2]In particular, if the specified helper has a method "`getAnnotationType()`" then this will be called and the returned value used as the annotation type. All the standard helpers provided with Mímir extend `AbstractSemanticAnnotationHelper` which implements this method.

[3]A third implementation in the `ordi` plugin is now deprecated. This stores the data in the same underlying OWLIM semantic repository format but accesses it through a different API abstraction.

```
24        annotation helper:new DefaultHelper(annType:'Document',
                               integerFeatures:["date"], mode:Mode.
                               DOCUMENT)
```

The above declaration (note the `mode` parameter!) creates a new Semantic Annotation Helper that uses the document features instead of the features from any given annotation. The helper behaves as if a single annotation, of the declared type (in our case *Document*), existed that covers the whole extent of the document, and has the same features as the GATE Document being indexed.

Things to note:

- All helper implementations supplied with Mímir are capable of working in `DOCUMENT` mode, so you can use them for indexing document features.

- The default value for the `mode` parameter for all supplied helper implementations is `ANNOTATION`. Not specifying a `mode` value preserves the default functionality (from versions preceding 3.4.0), i.e. indexing **annotation** features.

- You can have as many helpers as you want working in `DOCUMENT` mode, in parallel.

- The specified value for the `annType` parameter is used by the helper for simulating the presence of an actual annotation spanning the whole document; Mímir then behaves as if such annotations actually existed. This implies that you cannot re-use the name of an annotation type that is already being indexed. For example, if you are already indexing actual annotations of type `Document`, then you will need to choose a different name for the virtual annotation type used when indexing document features.

Document Rendering and Metadata

```
27    documentRenderer = new OriginalMarkupMetadataHelper()
28    documentMetadataHelpers = [documentRenderer]
```

The final part of the template concerns how document-level metadata is indexed, and how this can be combined with the document text to render the document content at search time, with matches of the query highlighted. These tasks are performed by objects that implement the interfaces `DocumentMetadataHelper` and `DocumentRenderer` respectively (both in the `gate.mimir` package). Mímir provides a single class `gate.mimir.index.OriginalMarkupMetadataHelper` which implements both interfaces, so in most cases the same object can be used for both jobs.

An index template must define one `documentRenderer` and may define any number of `documentMetadataHelpers` (in a square-bracketed list). If the renderer is an `OriginalMarkupMetadataHelper` (or a subclass) then the renderer object must be included in the list of metadata helpers in order to function correctly. Other metadata helpers may be added to the list if required.

4.2　Adding Documents to an Index

Once an index has been created in *indexing* mode, the next stage is to add documents to the index. Mímir provides an HTTP API for this which accepts documents for indexing via HTTP POST requests that include the document in Java serialised format. The easiest way to make use of this API is via GCP (the GATE Cloud Paralleliser batch processing tool) using a `MimirOutputHandler`. This GCP output handler makes use of the `gate.mimir.index.MimirConnector` (in the `mimir-client` module) to actually make the remote call, and you can use the same API in your own code. To add a GATE document to an open index simply call:

```
1   MimirConnector.defaultConnector().sendToMimir(document, uri, indexUrl);
```

. . . with the following parameters:

document a `gate.Document` for indexing.

uri the URI that should be used to identify the document in the Mímir index. May be `null`, in which case Mímir will generate a URI, but in most cases there will be a more meaningful identifier that could be used.

indexUrl a `java.net.URL` pointing to the location of the Mímir index. This is the "Index URL" given on the index information page.

The document to be indexed must, of course, contain the token and semantic annotations that the index expects.

It is possible to create your own private instance of `MimirConnector` rather than simply using the default one, but this is not necessary in normal use.

4.3　The Default Representation Scheme

The default generic SAH implementations try to minimise the amount of data stored in their underlying database or semantic repository by creating representation templates that are shared between all occurrences of annotations with the same values for the features. There are two levels of templates, the first defined by the values of nominal features, and the second that uses the values of all the other features. This is intended to reflect the typical scenario where most annotations are defined by a small set of nominal features, with a few of them having features with arbitrary values. Most annotation types would then only make use of level-1 templates, with a few of them employing both level-1, and level-2 templates.

The representation schema used by the Sesame helper is illustrated in Figure 4.1. Figure 4.2 shows the data created for an example annotation, with the two mention URIs displayed in bold. These URIs will be stored in the mentions index. The DB helper uses a similar strategy, with separate level 1 (nominal) and level 2 (everything else) database tables for each annotation type. Annotation types that only have nominal features need just a level 1 table.

The execution flow for each annotation includes the following main steps:

- Given the annotation's nominal features, find an appropriate level-1 (L1) template. If none is found, create one.
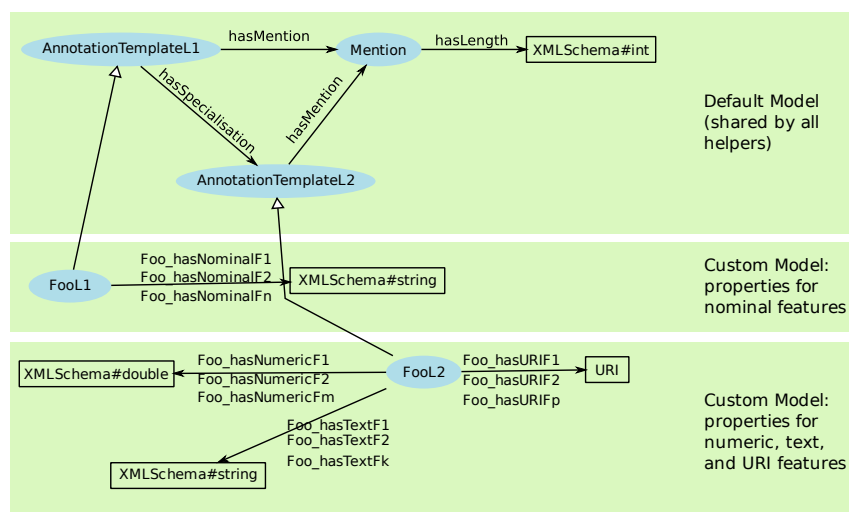
Figure 4.1: Default semantic annotation helper representation schema.

- For the L1 template, find a mention of appropriate length (the number of tokens covered by the annotation). If none is found, create one. Add the mention URI to the mentions index.
- If the annotation has non-nominal features:
  - Find an appropriate level-2 (L2) template, based on the feature values. If none is found, create one.
  - For the L2 template, find (or create) a mention of appropriate length; add the mention URI to the index.

All annotations sharing the same feature values will share the same database entries or knowledge base entities (the resources with URIs 'FooL1:1' and 'FooL1:1_1'), and the same mention objects. This has the advantage of reducing the size of the database, which allows more documents to be indexed, and helps achieve better execution speeds during search. The downside of this is that the indexing process is somewhat slowed down, as pre-existing entities need to be retrieved at every step.

Figure 4.2: Default semantic annotation helper data example.

Chapter 5

Searching Mímir Indexes

From a user's point of view, Mímir is a tool for searching a collection of semantically annotated documents. It provides facilities for searching o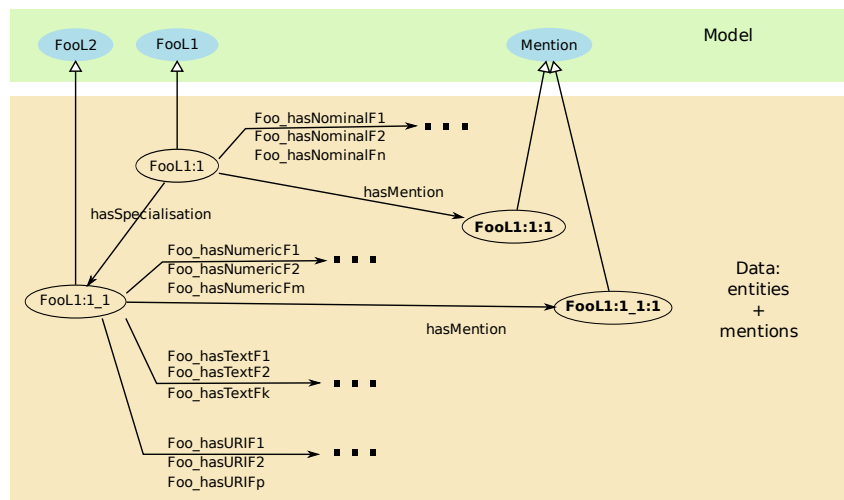ver different views of the document text, for example one can search the document words, the part-of-speech of those words, or their morphological roots. Beside searching the document text, Mímir also supports searches over the documents' semantic annotations, where queries are based on annotation types and restrictions over the values of annotation features. These different search paradigms can be combined freely into complex queries, with support for sequences, repetitions, and Boolean operators.

A search session entails the formulation of a query, running the query with the Mímir query engine, and consuming the query results. Mímir queries are expressed in a text-based query language which is described in section 5.1. The way these queries are submitted to Mímir depends on how it is deployed, the various interfaces are discussed in section 5.2.

5.1 The Mímir Query Language

A Mímir query is either a simple query (i.e. a `String` query, section 5.1.1, or an `Annotation` query, section 5.1.1), or a compound one, which comprises a set of sub-queries linked by operators. If no operator is placed between any two sub-queries, then the `Sequence` operator (see section 5.1.6) is implied. This means that several queries written one after another are interpreted as one sequence query. For example, a query like '*the brown dog*' is interpreted as a sequence query, having three sub-queries, each of them being a String query. This would match occurrences of the exact phrase '*the brown dog*' in the indexed documents. Note that this is different from the standard behaviour of search engines, which would simply match documents in which all three query terms occur, in whichever order. That type of search is also supported in Mímir, through the `AND` operator (see section 5.1.2). Parentheses can be used for grouping where the syntax would otherwise be ambiguous.

5.1.1 String Queries

The simplest form of query is a query term. This will match all occurrences of the query term in the indexed documents.

If the Mímir index being interrogated includes multiple token indexes, then the particular index to be searched can be specified by prefixing the query term with the index name and a colon, for example the query '*root:be*'[1] will match all morphological forms of the verb *to be*. If the name of the string index is omitted, then the first configured index is used. By convention (reflected in the default Mímir configuration) the first string index is used to store the terms text, so the default behaviour is to search over the document text, as expected. Double-quoted strings are treated as plain term queries against the first token index in a similar way.

In fact the above is a slight simplification, as bare terms (and double-quoted strings) are actually tokenised before being searched for. This is because Mímir views documents as streams of tokens, not characters, and the query must match the tokenisation that was used to index the documents. For example, the default GATE tokeniser treats "don't" as two tokens, "do" and "n't", so a query for *don't* as a single token would fail. To get around this, Mímir runs a GATE application over the string of the query to generate Token annotations, and then constructs a query for these tokens in sequence (see section 5.1.6). Named index queries ("root:be") are not tokenised, so if you want to avoid tokenising a particular query for any reason (e.g. if you suspect there is a mis-tokenised document in your index) you can explicitly name the appropriate index (typically "string", i.e. string:don't).

Annotation Queries

If annotation indexes were used during indexing, Mímir allows searching for annotation-based patterns. An annotation is a piece of metadata associated with a text segment, with a **type** and optionally **features**. An annotation query takes the following form: {`Type feature1=value1 feature2=value2 ...`}. The annotation type is required, the feature constraints are optional.

While the example above uses equality for the feature constraints, other operators are also available. Here is the full list:

equality: represented by the sign =, matches annotations which have the given value for the specified feature. The equality operator is applicable to features of any type.

comparison operators: represented by one of the following symbols: <, <=, >, >=, with the usual meaning. These operators can apply to features of type `nominal`, `numeric`, or `text`.

regular expressions: can be specified using the syntax `REGEX(pattern, flags)`, where the `pattern` represents the regular expression sought, and the `flags` are optional, and can be used to change the way matching is performed. See `http://www.w3.org/TR/xpath-functions/#regex-syntax` for a full specification of the regular expression support. The `REGEX` operator can only be used for `nominal`, and `text` features.

Examples:

---

[1] This assumes that an index named `root` exists, and was used to store the morphological root of the words.

{Person gender = female} – searches for annotations of type `Person`, which have a feature named `gender`, with the value *female*.

{Measurement type = scalar normalisedValue > 0 normalisedValue < 10 normalisedUnit = m} – searches for scalar measurements, with a unit of *metre*, and a normalised value between 0 and 10^2. Note that the same feature name can be used in several constraints, in which case only annotations where the feature meets *all* the constraints will be matched by the query. For disjunctive queries, use the OR operator described below.

In order to be able to search on a particular feature, that feature must have been specified in the index template when the index was created – Mímir indexes only the features it has been told to index. There may be additional "synthetic" features available at query time depending on the semantic annotation helper that the index uses for the given annotation type, for example the SPARQL helper allows queries on the "feature" named "sparql", the measurements helper allows queries for "spec" etc.

5.1.2   AND Operator: "&"

The 'AND' (also '&') operator can be used to specify queries that should match document segments that include at least one hit from each of the sub-queries. The results returned will always be the shortest document segments that satisfy the query.

5.1.3   OR Operator: "|"

OR queries are used to search hits that match one of a set of alternative query expressions. This is indicated by using the 'OR' (also '|') operator between the sub-queries. A query of the form `Query1 | Query2` will return hits that match either sub-query `Query1` or sub-query `Query2`.

5.1.4   IN and OVER Operators

The operators `IN` and `OVER` are used to search for hits of a query that contain, or are contained in the hits of another query. For example:

Query1 IN Query2 will match all the hits of `Query1` that are contained in a hit of `Query2`.

Query1 OVER Query2 will match all hits of `Query1` that contain (are overlapping) a hit of `Query2`.

5.1.5   Repeats Operator: "+"

The + operator can be used to match text segments that comprise a sequence of hits from the same sub-query. The length of the sequence is specified though a number (representing the **maximum** number of repetitions) or through two numeric values (representing the **minimum** and **maximum** number of repetitions). For example:

---

[2]The extended support for Measurement annotations is discussed further in section 6.3.

to+3 will match one, two, or three repeated occurrences of the word *to*. The returned hits will be of the form "*to*", "*to to*", or "*to to to*").

{Person}+2..5 will match sequences of 2, 3, 4, or 5 adjacent Person annotations.

({Location locType = city} | {Location locType = country})+3 will match any sequence of up to three Location annotations where each one refers to either a city or a country.

Note that there is no support for a repetition count of zero (an optional match) – you will need to reformulate the query to cover the versions with and without the optional match separately and combine them with an OR, for example (term1 term2+2 term3) | (term1 term3). Similarly there is no support for unbounded wildcards (*n* times or more).

5.1.6   Sequence Queries and Gaps

As sequence is the default operator in Mímir, there is no graphical sign for it: simply writing a set of queries one after another will cause a search for sequences of hits, one from each sub-query. For example, the query "the energy level" is actually a sequence query where the first sub-query searches for the word "*the*", the second for "*energy*", and the last for "*level*".

It is sometimes useful to include gaps in a sequence query, that is to allow arbitrary text fragments (of specified length) to occur in-between the hits from some of the sub-queries. This can be done by using the gap markers "[n]", or "[m..n]". These will match a sequence of length *n*, or with a length of between *m* and *n* of arbitrary tokens.

For example the query "the [2] root:time" will match phrases like "*the best of times*" or "*the worst of times*", whereas the query "the [2..10] root:time" would also match "*the best use of one's time*" (where the gap consists of six tokens – five words and an apostrophe).

5.1.7   Escaping Reserved Words

Some words are part of the query language definition so they cannot be used directly as query terms. If that is desired, then these constructs must be escaped as shown in the following table:

Reserved input	Escaped form
{, }	\{, \}
(,)	\(, \)
[,]	\[, \]
:	\:
+	\+
|	\\|
&	\&
?	\?
\	\\
.	\.
"	\"
=	\=
IN	"IN"
OVER	"OVER"
OR	"OR"
AND	"AND"

Escaping reserved constructs in the Mímir query language

5.2   Search Interfaces – How to Submit Queries to Mímir

The Mímir Grails plugin supplies two search interfaces by default, with the infrastructure
to implement other interfaces as required. An XML-based service interface allows other
applications to submit queries to the indexes hosted by a Mímir web application by POST-
ing requests over HTTP (described in section 5.2.1). There is also an example user-facing
search interface called *Web UI*, intended primarily for testing and demonstration purposes
(described in section 5.2.2). Both of these interfaces interact with the underlying indexes
through the `SearchService` Grails service provided by the plugin. When embedding the
Mímir Grails plugin in another Grails application this service is the primary means for
application code to interact with Mímir, and is described in section 5.2.3.

5.2.1   Mímir Search Web Service

The Mímir web application exposes the search functionality as a web service that can be
accessed through a simple HTTP interface. All requests are performed by calling an action
with a set of parameters; the results of a call are encoded in XML and returned as the
response to the request. All the example URLs in this section assume the `mimir-demo`
application with its default URL mappings, running on `localhost` port 8080.

The Mímir web service can be accessed at a URL like:
`http://localhost:8080/mimir-demo/{index ID}/search/{action}`, where the `action`
value is the name of one of the supported actions, described below. The actual URL (with
the correct index ID included) can be obtained from the *index information page* presented
by the Mímir web application. Parameters may be supplied as query parameters with a `GET`
request or in normal `application/x-www-form-urlencoded` form in a `POST` request.
Alternatively, they may be supplied as XML (if the request content type is `text/xml` or
`application/xml`) of the form:

```
<request xmlns="http://gate.ac.uk/ns/mimir">
  <firstParam>value</firstParam>
  <secondParam>value</secondParam>
</request>
```

The first request to the service will return a session cookie, which must be passed back with all subsequent requests.

When accessing the service URL with no value provided for `action`, a help page will be returned presenting the documentation associated with the XML web service.

The following actions are available:

help

Function	Obtain service documentation.
Parameters	none
Returns	A help message describing how to use the service.

postQuery

Function	Starts a new query. The query will execute asynchronously in a background thread, and will initially run until it has found a maximum of 1 million hits or has been running for 30 seconds, whichever comes first. If the query runner hits one of these limits before the search is complete it can be restarted by calling the `getMoreHits` action.
Parameters	**queryString:** the text of the query, using the Mímir query language.

Returns	An XML message with the ID of the new query, or an error message if there were any problems while parsing the query. Example request:
	`http://localhost:8080/mimir-demo/a4300d00-2dd1-4797-8eaa-e65b0c7d879b/search/`
	`postQuery?queryString=%22the%22`
	Example response:
	`<?xml version="1.0"?>`
	`<message xmlns='http://gate.ac.uk/ns/mimir'>`
	`  <state>SUCCESS</state>`
	`  <data>`
	`    <queryId>a28656e2-18f4-4b58-b9d3-9a9378eb14d0</queryId>`
	`  </data>`
	`</message>`

documentsCount

Function	Gets the number of result documents.
Parameters	**queryId:** the ID for the query, as returned by the `postQuery` action.

Returns	An XML message encapsulating a numeric value, or an error message if there were any problems. The value returned is -1 if the search has not yet completed, or the total number of result documents otherwise.
	
	Example request:
	
	`http://localhost:8080/mimir-demo/a4300d00-2dd1-4797-8eaa-e65b0c7d879b/search/`
	`documentsCount?queryId=a28656e2-18f4-4b58-b9d3-9a9378eb14d0`
	
	Example response:
	
	``xml
	<?xml version="1.0"?>
	<message xmlns="http://gate.ac.uk/ns/mimir">
	  <state>SUCCESS</state>
	  <data>
	    <value>8209</value>
	  </data>
	</message>
	``
	
	Example error response:
	
	``xml
	<?xml version="1.0"?>
	<message xmlns='http://gate.ac.uk/ns/mimir'>
	  <state>ERROR</state>
	  <error>Query ID a28656e2-18f4-4b58-b9d3-9a9378eb14d1 not known!</
	      error>
	</message>
	``

documentsCurrentCount

Function	Gets the number of result documents found so far.
Parameters	**queryId:** the ID for the query, as returned by the `postQuery` action.

Returns	An XML message encapsulating a numeric value, or an error message if there were any problems. After the search compeltes, the value returned is identical to that returned by calling `documentsCount`.
	
	Example request:
	
	`http://localhost:8080/mimir-demo/a4300d00-2dd1-4797-8eaa-e65b0c7d879b/search/`
	`documentsCurrentCount?queryId=a28656e2-18f4-4b58-b9d3-9a9378eb14d0`
	
	Example response:
	
	``xml
	<?xml version="1.0"?>
	<message xmlns="http://gate.ac.uk/ns/mimir">
	  <state>SUCCESS</state>
	  <data>
	    <value>142</value>
	  </data>
	</message>
	``
	
	Example error response:
	
	``xml
	<?xml version="1.0"?>
	<message xmlns='http://gate.ac.uk/ns/mimir'>
	  <state>ERROR</state>
	  <error>Query ID a28656e2-18f4-4b58-b9d3-9a9378eb14d1 not known!</
	      error>
	</message>
	``

documentId

Function	Obtains the document ID for the document at a given rank (position in the results list).
Parameters	**queryId:** the ID for the query, as returned by the `postQuery` action. **rank:** the rank (position on the results list) for the requested document.

Returns	An XML message encapsulating a numeric value, or an error message if there were any problems. Example request:
	`http://localhost:8080/mimir-demo/a4300d00-2dd1-4797-8eaa-e65b0c7d879b/search/`
	`documentId?queryId=a28656e2-18f4-4b58-b9d3-9a9378eb14d0&rank=3`
	Example response:
	```\n<?xml version="1.0"?>\n<message xmlns='http://gate.ac.uk/ns/mimir'>\n <state>SUCCESS</state>\n <data>\n <value>11</value>\n </data>\n</message>\n```

documentScore

Function	Obtains the score for the document at a given rank (position in the results list).
Parameters	**queryId:** the ID for the query, as returned by the `postQuery` action. **rank:** the rank (position on the results list) for the requested document.
Returns	An XML message encapsulating a numeric value, or an error message if there were any problems. Example request:
	`http://localhost:8080/mimir-demo/a4300d00-2dd1-4797-8eaa-e65b0c7d879b/search/`
	`documentScore?queryId=a28656e2-18f4-4b58-b9d3-9a9378eb14d0&rank=3`
	Example response:
	```\n<?xml version="1.0"?>\n<message xmlns="http://gate.ac.uk/ns/mimir">\n <state>SUCCESS</state>\n <data>\n <value>12.330469310919446</value>\n </data>\n</message>\n```

documentHits

Function	Obtains a set of hits. Each hit is defined by a document ID, a position and a length, both of which are defined in terms of tokens, not characters (see Section 4.1 for details).
Parameters	**queryId:** the ID for the query, as returned by the postQuery action. **rank:** the rank (position on the results list) for the requested document.

Returns	An XML message encapsulating a set of \<hit\> elements, one for each individual hit.
	Example request:
	`http://localhost:8080/mimir-demo/a4300d00-2dd1-4797-8eaa-e65b0c7d879b/search/`
	`documentHits?queryId=a28656e2-18f4-4b58-b9d3-9a9378eb14d0&rank=3`
	Example response:
	<pre><?xml version="1.0"?>
	<message xmlns='http://gate.ac.uk/ns/mimir'>
	  <state>SUCCESS</state>
	  <data>
	    <hits>
	      <hit documentId='11' position='257' length='1'/>
	      <hit documentId='11' position='266' length='1'/>
	      <hit documentId='11' position='290' length='1'/>
	      <hit documentId='11' position='303' length='1'/>
	      <hit documentId='11' position='309' length='1'/>
	      <hit documentId='11' position='316' length='1'/>
	      <hit documentId='11' position='320' length='1'/>
	      <hit documentId='11' position='332' length='1'/>
	      <hit documentId='11' position='335' length='1'/>
	      <hit documentId='11' position='342' length='1'/>
	      <hit documentId='11' position='348' length='1'/>
	    </hits>
	  </data>
	</message></pre>

documentText

Function	Action for obtaining (a segment of) the text of a document.
Parameters	**queryId:** the ID for the query that has returned the document ID being used, as returned by the postQuery action. **rank:** the rank (position on the results list) for the requested document. **termPosition** the position of the first returned token. This parameter is optional; defaults to 0 is not provided, which means the first token of the document. **length** the number of tokens to be returned. This parameter is optional, if omitted, all the document tokens will be returned.

| Returns | An XML message containing the text of all the individual tokens and, if available, the spaces between them.
This action could be used, for example, to obtain text snippets around a query hit.
Example request:
`http://localhost:8080/mimir-demo/a4300d00-2dd1-4797-8eaa-e65b0c7d879b/search/`
`documentText?queryId=a28656e2-18f4-4b58-b9d3-9a9378eb14d0&rank=1&termPosition`
`=100&length=10`
Example response:
<pre><?xml version="1.0"?>
<message xmlns='http://gate.ac.uk/ns/mimir'>
  <state>SUCCESS</state>
  <data>
    <text position='100'>25</text>
    <text position='101'>C</text>
    <space> </space>
    <text position='102'>1</text>
    <text position='103'>/</text>
    <text position='104'>08</text>
    <space> </space>
    <text position='105'>C</text>
    <text position='106'>25</text>
    <text position='107'>C</text>
    <space>   </space>
    <text position='108'>1</text>
    <text position='109'>/</text>
  </data>
</message></pre> |

documentMetadata

Function	Returns the title and URI associated with a document. Optionally, other metadata fields can also be obtained. All these values were provided at indexing time.
Parameters	**queryId:** the ID for the query that has returned the document ID being used, as returned by the `postQuery` action. **rank:** the rank (position on the results list) for the requested document. **fieldNames:** (optional) a comma-separated list of other field names to be returned.

Returns	An XML message encapsulating the several string values, or an error message if there were any problems.
	Example request:
	`http://localhost:8080/mimir-demo/a4300d00-2dd1-4797-8eaa-e65b0c7d879b/search/`
	`documentMetadata?queryId=a28656e2-18f4-4b58-b9d3-9a9378eb14d0&documentId=1`
	Example response:
	<pre><?xml version="1.0"?>
	<message xmlns="http://gate.ac.uk/ns/mimir">
	  <state>SUCCESS</state>
	  <data>
	    <documentTitle>
	      Virtual job-hunting: Technology fills situations vacant
	    </documentTitle>
	    <documentURI>http://www.bbc.co.uk/news/business-12194581</
	        documentURI>
	  </data>
	</message></pre>

renderDocument

Function	Renders the document text and hits for a given document, in the context of a given query. The HTML of the document is rendered directly to the response stream of the connection.
Parameters	**queryId:** the ID for the query, as returned by the `postQuery` action. **rank:** the rank (position on the results list) for the requested document.

Returns	HTML content. The hits are rendered as `<span class="mimir-hit">`...`</span>`.
	 Example request:
	 `http://localhost:8080/mimir-demo/a4300d00-2dd1-4797-8eaa-e65b0c7d879b/search/`
	 `renderDocument?queryId=a28656e2-18f4-4b58-b9d3-9a9378eb14d0&rank=1`

close

Function	Closes a query, releasing all resources allocated for supporting it. After a query is closed, no more actions can be performed for it. It is important to close queries, as each running query uses up memory on the server. Queries are also closed automatically after a period of inactivity (upon session expiration, the time for which is defined in the configuration of the web application server – this is why it is important to pass the session cookie you received from `postQuery` back to the server with subsequent calls).
Parameters	**queryId:** the ID for the query, as returned by the `postQuery` action.

Returns	An XML message with a success or failure value.
	 Example request:
	 `http://localhost:8080/mimir-demo/a4300d00-2dd1-4797-8eaa-e65b0c7d879b/search/`
	 `close?queryId=a28656e2-18f4-4b58-b9d3-9a9378eb14d0`
	 Example response:
	
	 ```<?xml version="1.0"?>```
	 ```<message xmlns='http://gate.ac.uk/ns/mimir'>```
	 ``` <state>SUCCESS</state>```
	 ```</message>```
	
	 Example failure (using the ID for an already closed query):
	
	 ```<?xml version="1.0"?>```
	 ```<message xmlns='http://gate.ac.uk/ns/mimir'>```
	 ``` <state>ERROR</state>```
	 ``` <error>Query ID a28656e2-18f4-4b58-b9d3-9a9378eb14d0 not known!</```
	 ``` error>```
	 ```</message>```

5.2.2   The Web UI Example User Interface

The Web UI search tool is a browser-based search interface intended to serve as a platform for experimentation with a Mímir index, and as a demonstration of the capabilities of the Mímir framework and API. It is written using the Google Web Toolkit, and the source code is included in the Mímir Grails plugin.

In the demo web application with its default URL mappings, the Web UI interface for an index in searching mode is available at
`http://localhost:8080/mimir-demo/{index ID}/search/index`.   The   initial
page, shown in figure 5.1, provides a text area into which you can type queries in the

Figure 5.1: Front page of the Web UI user interface

Mímir query language. It provides auto-completion for annotation types and features (by asking the index what types it was configured with when it was created). Clicking the Search button starts a search on the server.

Hits are shown below the search box, as shown in figure 5.2, with the hit text highlighted in bold and with three tokens of left and right context. The document title is a link, in this example to the original document as the index was created with the "Document URIs are external links" option. The "cached" link shows Mímir's cached copy of the document, with all the hits from that document highlighted in red. For indexes where the document URIs are not external links the document title would link directly to the cached version and there would be no separate "cached" link.

At the bottom of the page is a row of pagination links (figure 5.3).

5.2.3   Embedding Mímir in a Grails Application

Both the XML web service and the GUS interface ultimately use a Grails service provided by the Mímir plugin to search their indexes. If you install the Mímir plugin into your own Grails application this service will be your primary entry point to make use of Mímir functionality, so this section explains what you need to know to use it effectively.

The `searchService` is a normal Grails service which can be autowired into your own services, controllers, etc. The service itself is very simple, offering only the following methods:

postQuery(index, queryString) start running a query against the given index. The index can be specified either as a string containing the indexId (the last component of the index URL, typically a UUID) or as an `Index` domain object (the database object representing a local, remote or federated index). Returns a *query ID* string.

getQueryRunner(queryId) retrieves the `QueryRunner` for the given running query ID. `QueryRunner` is the interface through which you can interact with the running query.

Figure 5.2: Web UI search results page

Figure 5.3: Web UI pagination links for a large search

closeQueryRunner(queryId) indicates that the given query runner is no longer required. It is important to call this method when you have finished with a query runner, as each runner owns resources such as background threads which need to be properly cleaned up.

Once a query has been started, its `QueryRunner` provides access to the statistics, the hits themselves, and the text in the matched documents. The most important methods are summarised below, but for full details you should look at the interface definition itself, in the `gate.mimir.search` package of `mimir-core`.

getDocumentsCount() Gets the number of result documents that have been found. While the query has not yet finished running, this method returns −1.

getDocumentsCurrentCount() Gets the number of distinct documents that have so far been found to contain hits. This number may increase at any time while the query is currently active. Once the query completes, this method returns the same value as `getDocumentsCount`.

getDocumentHits(rank) Gets the details for the hits found inside a given document. The document is pecified by its rank (the position in the list of result documents). The value supplied for the `rank` paramter must be between 0 (inclusive) and the value returned by `getDocumentsCount()` (exclusive). The return value from this method is a list of `Binding` objects, each representing one hit.

getDocumentID(rank) Gets the ID in the underlying index of the *n*th document that matched this query. This ID is needed when deleting documents from the index.

getDocumentTitle/URI(rank) Gets metadata about the document with the given rank.

getDocumentText(rank, start, length) Gets the text of the document with the given rank, starting at the *start*th token and extending for *length* tokens. The return value is a pair of parallel string arrays, one containing the text of the tokens and the other containing the text between each token and the following one.

renderDocument(rank, Appendable) Render the document content, with hits highlighted, using the document renderer configured for the index. The content is written to the specified Appendable (a StringBuilder, Writer, etc.).
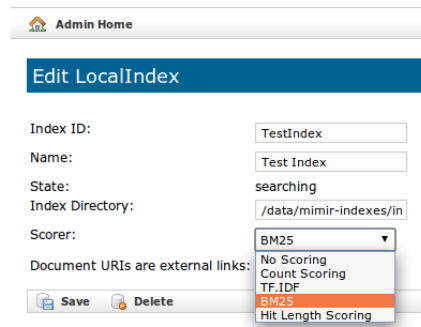
The `getDocumentHits()` method returns a list of `Binding` objects, which provide several methods, the most important ones being `getTermPosition()` (the offset of the first token covered by the hit) and `getLength` (the number of tokens it covers).

5.3   Ranking of Results

Starting with version 4.0, Mímir supports the ranking of results. This means that a scoring algorithm is applied to assess the relevance of each returned document and that documents are returned in decreasing order of their scores. To support this, Mímir will need to execute the query twice: once to calculate the scores, and a second time to collect the actual hits. Consequently, the query execution will be a bit slower when using ranking.

5.3.1   Configuring Ranking

Each searchable index can be configured to use a scorer. When initially created, all indexes have scoring disabled. This can be changed from the admin interface, by editing the index

Figure 5.4: Configuring a Scorer

configuration, as shown in Figure 5.4.

If the provided scoring algorithms are not suitable for your needs, you can add new ones as discussed in Section 7.2.

Chapter 6

Standard Mímir Plugins

Mímir uses the GATE Embedded *CREOLE plugin* mechanism to load semantic annotation helper classes. A number of plugins are supplied by default with the Mímir distribution, and those plugins are described in this chapter. Information on how to create new plugins to provide user-defined helper classes can be found in section 7.1.

6.1 The `db-h2` Plugin

The `db-h2` plugin is one of two plugins (the other being the `sesame` plugin) that provides a *generic* semantic annotation helper implementation that can be configured for any annotation type with any features. The helper provided by `db-h2` uses an embedded relational database engine (`http://www.h2database.com/`) to store the annotation data, and generally provides the best performance of the standard generic helpers.

`gate.mimir.db.DBSemanticAnnotationHelper` is the helper class provided by the `db-h2` plugin. It has a constructor that takes a `Map` of configuration parameters, and Groovy provides special "named argument" support for `Map`-valued method and constructor parameters, allowing the following idiom in the index template DSL:

```
1  // note the "import X as Y", which is another Groovy feature to create an
2  // alias for an imported class name
3  import gate.mimir.db.DBSemanticAnnotationHelper as DefaultHelper
4  // ...
5  semanticAnnotations = {
6    index {
7      annotation helper:new DefaultHelper(annType:'Person',
                                            nominalFeatures:["gender"])
8    }
9  }
```

The supported constructor arguments are:

annType: the annotation type which the helper is to process.

nominalFeatures: the names of the features to be indexed that have nominal values. An annotation feature is said be nominal if the range of possible values is clearly defined

and limited in size. There is no hard rule regarding the size of the set of permitted values, but, for optimal results, this should not exceed a few tens of values.

integerFeatures: the names of the features to be indexed that have integer values (i.e. values that can be converted to a Java `long` value).

floatFeatures: the names of the features to be indexed that have floating-point numeric values (i.e. values that can be converted to a Java `double` value).

textFeatures: the names of the features to be indexed that have arbitrary text values (as opposed to the nominal case of a fixed list of possible values).

uriFeatures: the names of the features to be indexed that have URIs as values.

The DB-based helper does not distinguish between text- and URI-valued features, indexing both types in the same way, but it accepts both kinds as arguments to be consistent with the Sesame helper (described below). When using the `import as` mechanism, switching an index template from the DB helper to the Sesame helper or vice versa is simply a matter of changing the name of the class being imported.

6.2   The `sesame` Plugin

The `sesame` plugin provides the other standard generic helper, `gate.mimir.sesame.SesameSemanticAnnotationHelper`. This helper stores annotation data as triples in a Sesame repository (by default using OWLIM as the underlying storage engine).

The `SesameSemanticAnnotationHelper` constructor takes the same named arguments as the DB helper for the annotation type and feature names (see the previous section for details). Internally all numeric features (integer and float) are stored as floating point numbers, but the helper accepts both "integerFeatures" and "floatFeatures" parameters for consistency with the DB helper.

For advanced users it also provides a mechanism to specify the location of the Sesame repository configuration template file (in the standard Turtle RDF format), as a parameter named either "absolutePath" (for an absolute path to the file) or "relativePath" (for a path relative to the `sesame` plugin directory). The default configuration file is `resources/owlim.ttl` under the `sesame` plugin. When specifying a custom repository configuration it is important to retain the repositoryID of "owlim" as this is assumed by the helper code.

6.2.1   Searching in the Knowledge Base

For annotations that have a URI-valued feature named "inst", the Sesame helper provides an additional "synthetic" feature named `semanticConstraint` that provides a way to query for annotations based on information in the knowledge base.

```
1  {Organization semanticConstraint = "?inst <http://proton.semanticweb.
       org/2005/04/protont#locatedIn> <http://example.com#Sheffield> ."}
```

The value of the semanticConstraint feature is a SPARQL fragment that has access to the `?inst` variable, referring to the URI in the "inst" feature of the annotation. This mechanism

can only query for one specific feature (inst) and only against triples stored in the same semantic repository as the Mímir annotation data. For a more flexible approach to semantic queries that can make use of semantic data stored in a remote knowledge base, see the `sparql` plugin (section 6.4).

6.3 The `measurements` Plugin

The GATE `Tagger_Measurements` plugin, introduced in GATE 6.1, is able to recognise many different kinds of measurement expressions in text. It normalizes the value and unit of each measurement into the SI system of measurements and stores these values as features of the Measurement annotation. For example, the text "45 cm" would be annotated with a normalized unit of metres and a normalized value of 0.45, the text "18 in" would also be normalized to metres, in this case with a normalized value of 0.4572.

The Mímir `measurements` plugin provides a SAH that implements the same normalization on queries. It processes queries for a "synthetic" feature called "spec" which represents a measurement specification in a controlled language and converts constraints on this feature into the corresponding constraints on the real normalized value and unit features that have been indexed. For example, a search for {Measurement spec="1 to 3 feet"} would be treated as a query for measurements whose normalized unit is metres and whose normalized value is between 0.3048 and 0.9144, which would match both the "45 cm" and "18 in" examples above.

6.3.1 Configuring the Measurements SAH

To use the measurements helper you need to first ensure that the `measurements` plugin is loaded into your Mímir instance, then create an index template that specifies an instance of the helper:

```
1  import gate.mimir.measurements.MeasurementAnnotationHelper
2
3  // ...
4  semanticAnnotations = {
5    index {
6      // Measurement helper with default settings
7      annotation helper:new MeasurementAnnotationHelper(
8            delegateHelperType:DefaultHelper)
9    }
10 }
```

Note that the measurement helper does not need any "annType" or "*xxx*Features" parameters, as it is hard-coded to work only for annotations of type "Measurement" with the features that are produced by the measurement tagger PR. However the constructor does take a `Map` with other named arguments:

```
6        // Example of how to configure a custom "units" file
7        annotation helper:new MeasurementAnnotationHelper(
8              delegateHelperType:DefaultHelper,
9              unitsFile:'resources/americanUnits.dat',
10             locale:'en_US')
```

The following parameters are supported:

delegateHelperType (required) a `Class` object representing the type of generic helper that the Measurements helper should delegate to. This class must provide a 6-argument constructor taking the annotation type (a String) and five String arrays for the nominal, integer, float, text and URI feature names respectively. Both the DB and Sesame helpers provide this constructor.

unitsFile the location of the `units.dat` file used to configure the measurements parser. If not specified, a default file provided with the `measurements` plugin is used. This value can be an absolute URL (file:/path/to/units.dat) or a relative path which will be resolved against the `measurements` plugin directory.

commonWords the location of the common words file used by the measurements parser. As with the `unitsFile` parameter, if omitted a default file bundled with the plugin is used.

locale the locale under which the measurements will be parsed. Defaults to "en_GB" if unspecified.

encoding the character encoding used to read the configuration files. Defaults to "UTF-8" if unspecified.

The measurements SAH is pre-configured with the feature names that the measurements tagger produces, and can only be used for annotations of type `Measurement`.

Measurements helper implementation

The `MeasurementAnnotationHelper` extends the `DelegatingSemanticAnnotationHelper` base class described above. It does not add any behaviour at indexing time, simply passing all the annotations through directly to its delegate. However it overrides the `getMentions` search method to support the "spec" feature.

When a query including a spec feature constraint is received, the helper parses this spec using the measurements parser to obtain a normalized unit and value or values for the measurement sought. It then constructs a number of new constraint sets that match annotations compatible with the spec and then for each of these alternatives, runs these constraints in combination with the other non-spec constraints of the original query against the delegate helper. The final set of URIs returned is the union of the results obtained from the delegate for all the alternative reformulations of the spec constraint.

As well as being useful in its own right for Measurement annotations, the measurements helper serves as an example of how to implement your own special-purpose helper based on the delegating base class. Feel free to use it as a template for your own helper implementations.

6.4   The `sparql` Plugin

The `sparql` plugin provides a semantic annotation helper that wraps another helper, adding flexible *semantic query* support. It is intended to be used with annotations that have one

or more URI-valued features whose values refer to entities in an external knowledge base (accessible at a standard *SPARQL endpoint*). The SPARQL helper has no effect at indexing time, simply delegating all calls through to the underlying helper, but at search time it allows queries for the synthetic feature "sparql". This feature value is taken to be a SPARQL "SELECT" query, which is posted to the configured SPARQL endpoint. The variables in the SELECT query must correspond to the names of features that have been indexed by the underlying helper, and each row in the result set becomes a standard Mímir query to the underlying helper. Any annotations that match any of these new queries will be treated as a match for the `sparql` constraint. This process is described in detail below.

For example, given a helper configured for the public DBpedia endpoint `http://dbpedia.org/sparql`, the following Mímir query:

```
{Person sparql = "SELECT DISTINCT ?inst WHERE {
  ?inst <http://dbpedia.org/ontology/birthPlace>
    <http://dbpedia.org/resource/Sheffield> }"}
```

would search for all Person annotations that have an "inst" feature containing the URI of an entity in DBpedia that represents a person born in Sheffield.

6.4.1   Creating a SPARQL Helper

The SPARQL semantic annotation helper class is called `gate.mimir.sparql.SPARQLSemanticAnnotationHelper`. It has a `Map` constructor taking the following parameters:

delegate (required) the underlying semantic annotation helper that this SPARQL helper should wrap.

sparqlEndpoint (required) the address of the SPARQL endpoint that this helper should use when making SPARQL queries.

queryPrefix an optional prefix that will be prepended to the string specified in the `sparql` synthetic feature to form the actual SPARQL query that will be sent to the endpoint. Typically this would be used to define appropriate namespace prefixes.

querySuffix an optional suffix that will be appended to the end of the SPARQL queries. Thus the actual SPARQL query submitted to the endpoint is *queryPrefix* + `sparql` feature + *querySuffix*.

sparqlEndpointUser and sparqlEndpointPassword username and password used to authenticate to the SPARQL endpoint (only HTTP basic authentication is supported). May be omitted if your endpoint does not require authentication.

The helper also accepts the usual "annType" and "*xxx*Features" parameters but these are not normally required – if the delegate helper is a subclass of `AbstractSemanticAnnotationHelper` (which is the case for all the standard helpers) then the SPARQL helper will take its feature names from the delegate, so the only time the features need to be specified explicitly for the SPARQL helper is if the delegate is a custom helper type that does not extend `AbstractSemanticAnnotationHelper`.

For example, the following configuration would set up a helper for Person annotations operating against DBpedia, to support the example query above:

```
1  import gate.mimir.db.DBSemanticAnnotationHelper as DBH
2  import gate.mimir.sparql.SPARQLSemanticAnnotationHelper as SPARQLHelper
3  // ...
4  semanticAnnotations = {
5    index {
6      annotation helper:new SPARQLHelper(
7          sparqlEndpoint:'http://dbpedia.org/sparql',
8          delegate:new DBH(annType:"Person", uriFeatures:['inst']))
9    }
10  }
```

Alternatively, the helper could be configured with a queryPrefix to set up some useful namespace prefixes:

```
1  semanticAnnotations = {
2    index {
3      annotation helper:new SPARQLHelper(
4        sparqlEndpoint:'http://dbpedia.org/sparql',
5        queryPrefix:
6         'PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#> \
7          PREFIX xsd:<http://www.w3.org/2001/XMLSchema#> \
8          PREFIX owl:<http://www.w3.org/2002/07/owl#> \
9          PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#> \
10          PREFIX dbo:<http://dbpedia.org/ontology/> \
11          PREFIX dbr:<http://dbpedia.org/resource/> ',
12        delegate:new DBH(annType:"Person", uriFeatures:['inst']))
13    }
14  }
```

Note the backslashes which are a Groovy feature to permit a String literal to be broken across several lines, and also the trailing space before the closing quotation mark – the helper simply concatenates the prefix, query and suffix without any additional space, so required spaces must be part of the prefix or suffix string itself. This would allow the example query above to be rewritten more succinctly as:

```
{Person sparql = "SELECT DISTINCT ?inst WHERE {
  ?inst dbo:birthPlace dbr:Sheffield }"}
```

For annotation types that have only one URI-valued feature it may be desirable to include the "SELECT DISTINCT ?inst WHERE { " in the prefix and add a suffix of " }", which would reduce the query down to

```
{Person sparql = "?inst dbo:birthPlace dbr:Sheffield"}
```

If your index template includes several ontology-based annotation types sharing the same SPARQL endpoint and prefixes then listing these in full for each annotation type will result in a large and unwieldy template. However, since the index template is itself a Groovy script it is possible to declare methods to factor out the common code. Method declarations must be placed *outside* the semanticAnnotations block:

```
1  import gate.mimir.db.DBSemanticAnnotationHelper as DBH
2  import gate.mimir.sparql.SPARQLSemanticAnnotationHelper as SPARQLHelper
3
```

```
4   def standardHelper(type) {
5     return new SPARQLHelper(
6         sparqlEndpoint:'http://dbpedia.org/sparql',
7         queryPrefix:
8          'PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#> \
9           PREFIX xsd:<http://www.w3.org/2001/XMLSchema#> \
10          PREFIX owl:<http://www.w3.org/2002/07/owl#> \
11          PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#> \
12          PREFIX dbo:<http://dbpedia.org/ontology/> \
13          PREFIX dbr:<http://dbpedia.org/resource/> ',
14        delegate:new DBH(annType:type, uriFeatures:['inst']))
15  }
16
17  // ...
18  semanticAnnotations = {
19    index {
20      annotation helper:standardHelper('Person')
21      annotation helper:standardHelper('Location')
22      annotation helper:standardHelper('Organization')
23    }
24  }
```

6.4.2   Format of SPARQL Queries

This section describes in more detail how the SPARQL queries relate to the annotations indexed by the underlying semantic annotation helper. As a simple example we consider the query for people born in Sheffield:

```
{Person sparql = "SELECT DISTINCT ?inst WHERE {
  ?inst dbo:birthPlace dbr:Sheffield }"}
```

The helper will submit the SPARQL query to its configured endpoint, and receive a response of the form:

inst
http://dbpedia.org/resource/Gordon_Banks
http://dbpedia.org/resource/Michael_Palin
http://dbpedia.org/resource/David_Blunkett
...

This will then generate a series of queries to the underlying helper of the form:

```
{Person inst = "http://dbpedia.org/resource/Gordon_Banks"}
{Person inst = "http://dbpedia.org/resource/Michael_Palin"}
...
```

and any annotation that matches any of these queries will be returned as a match for the sparql constraint.

The SPARQL query can bind more than one variable, and the values of the variable bindings can be RDF literals as well as URIs, they convert to queries on the underlying helper in the same way.

Chapter 7

Extending and Customising Mímir

The standard semantic annotation helpers provided by Mímir are adequate for many use cases, but if your application needs more functionality that they cannot provide it is easy to add your own custom helper implementations using a plugin mechanism. This process is described in section 7.1. Similarly, the basic Mímir demo application shows the simplest way to use the Mímir Grails plugin, but it provides no authentication or security, for example. To add these kinds of features you can install the Mímir plugin into your own custom Grails application, as described in section 7.3.

7.1 Creating New Semantic Annotation Helpers

Semantic annotation helpers (SAHs) are the mechanism that Mímir uses to store information about annotations and allow this information to be queried at search time. A SAH is associated with a particular annotation type in the Mímir index configuration, and performs two functions:

During indexing for each annotation of the relevant type, store information about that annotation in some persistent form and return to Mímir one or more URIs that represent that annotation. These URIs are included in the main MG4J index and associated with the location in the document where the annotation was found.

During searching given a set of feature value constraints, use the persistent store created during indexing to determine the URIs associated with annotations that satisfy the constraints.

Conceptually, SAH implementations can be divided into two types. *Generic* helpers are those that can index any annotation types and features, and *special-purpose* helpers are those that are designed to work with specific types of annotation. There are two generic SAH implementations provided with Mímir by default. You would create a new generic SAH implementation if you wanted to store annotation data in a different underlying storage format. Mímir provides one example special-purpose SAH for Measurement anno-

tations, which can serve as a template for how to implement your own helpers for other annotation types.

7.1.1   The `SemanticAnnotationHelper` **Interface**

The `gate.mimir.SemanticAnnotationHelper` interface is the contract that all helpers must implement. It specifies three groups of methods that must be implemented:

Lifecycle Methods

The interface includes two pairs of init/close lifecycle methods, one pair taking an `Indexer` parameter (used when the helper is indexing annotations) and the other pair taking a `QueryEngine` parameter (used when the helper is searching). Both the Indexer and QueryEngine provide access to an `IndexConfig` object which defines the configuration of the index, including the location of the index files on disk, and provides a mutable "context" map that can be used to share objects among the various SAH objects (for example the Sesame helper uses the context to share a single connection to the semantic repository among all the helpers associated with the index). The appropriate `init` method is called by Mímir when the index is opened (in whichever mode), before any other requests are passed to the helper, and the corresponding `close` method is called when the index is shut down.

Indexing Methods

When indexing annotations Mímir calls the following methods:

documentStart(document) Called when the indexer starts processing a particular document to allow the helper to perform any per-document setup tasks. This method is guaranteed to be called once per document, before any calls to getMentionUris.
getMentionUris(annotation, length, indexer) Called once for each semantic annotation of this helper's type in the document. The helper is expected to use the annotation's length (in tokens) and feature values to determine the relevant URI or URIs that represent this annotation, and return them.
documentEnd() Called after all the annotation for a particular document have been processed.

These methods are always called from a single thread, as long as the same helper object is not used for more than one annotation type.

Note that the annotation length passed to `getMentionUris` is measured in *tokens*, not characters. Because Mímir operates on streams of tokens, semantic annotations that partially overlap a token will be considered by Mímir to cover the whole token. I.e. given the hypothetical example:

```
... started on 10/05/1987 by John Smith ...
    ------- -- -- -- ---- -- ---- -----
```

where tokens are represented by `---`, an annotation that covers just the "87" would be indexed as if it covered the whole "1987" token.

Search Methods

The final method in the interface is `getMentions(annotationType, constraints, queryEngine)`. This method is called by Mímir when searching for annotations, and the helper must use its stored data to determine all the possible mention URIs that satisfy the provided constraints, and return them along with their lengths (in tokens) as provided to `getMentionUris` when the annotations were indexed.

There is a second overloading of this method specified in the interface which is a convenience for callers when all the constraints are simple feature value equality constraints, but generally implementors of new SAH types can ignore this as Mímir provides an abstract base class that converts the Map form of constraints into the more general List<Constraint> form and calls the other method.

The `getMentions()` methods may be called from multiple threads at the same time, so implementations should be thread-safe.

7.1.2   Abstract Base Classes

Mímir provides an abstract class `AbstractSemanticAnnotationHelper` which, as described above, implements the Map version of `getMentions` in terms of the List<Constraint> version, and also provides empty implementations of `documentStart` and `documentEnd`. As well as this, it provides accessor methods to access the list of feature names of each of the five types (nominal, integer, float, text and URI) that a particular helper object supports. `AbstractSemanticAnnotationHelper` enjoys special support in the Mímir Grails plugin, allowing clients to determine what feature names an index supports for each annotation type whose helper extends `AbstractSemanticAnnotationHelper`. All the standard helper implementations provided with Mímir extend this base class.

Special-purpose helpers for particular annotation types typically operate by mapping the features of their target annotations and/or the feature constraints in a query into a different set of features or constraints which can then be handled by a generic helper. The Measurement helper described in section 6.3 operates in this way. To support this pattern, Mímir provides an abstract `DelegatingSemanticAnnotationHelper` which implements all the SAH interface methods to simply delegate to another helper instance. Subclasses can then override the methods as appropriate to map their features or constraints into terms that the underlying helper can understand and then call the `super` method to pass these parameters on to the delegate.

`DelegatingSemanticAnnotationHelper` extends `AbstractSemanticAnnotationHelper` so it advertises the features it supports in the usual way. However it is important to note that the various `get*FeatureNames` methods of the delegating helper do *not* call their counterparts in the delegate, which allows a delegating helper to advertise different features from those supported by its delegate.

The SAH lifecycle

Semantic annotation helper objects go through a specific lifecycle in Mímir. When creating a new index for indexing, the helpers defined for each semantic annotation type are instan-

tiated by calling their constructors from the Groovy DSL (see the measurements example below). Once instantiated, the `init(Indexer)` methods of each helper in turn are called (one after the other, in a single thread, so if you are sharing objects among your helpers through the context you can be sure that you have exclusive access to the context map during the call to your init method).

The actual indexing process takes place in several threads in a pipelined manner. When a document arrives for indexing it is first processed by the token indexer (to index the token features), then the semantic indexers specified by the `index { ... }` blocks in the DSL in turn. Each indexer operates in its own thread, with documents passing from one to the next via queues. So each document is only processed by one thread at a time but under load you may have the token indexer dealing with document 3 at the same time as semantic indexers are dealing with documents 2 and 1.

When indexing is complete the helpers' `close(Indexer)` methods are called (again, in sequence in one thread). The index is now closed and the SAH objects can be garbage collected.

The index configuration, including all the SAH objects, is serialized to XML using XStream (`http://xstream.codehaus.org`). Therefore it is important to mark as `transient` any fields of your helper class that should not be serialized (e.g. temporary in-memory caches, etc.).

When an index is opened for searching the XML configuration is deserialized to re-create the helper objects, and their `init(QueryEngine)` methods are called. Note that as with Java object serialization XStream does *not* call object constructors when deserializing, so any initialization must happen in the init method or in a `readResolve` method, and not in the constructor.

Annotation queries result in calls to the relevant helper's `getMentions` method, which has been discussed in detail above.

Finally, when the index is shut down the `close(QueryEngine)` methods of the helpers are called in sequence.

7.1.3   Packaging New Helper Types for Use with Mímir

To create a new helper type and make it available to Mímir you need to first write your Java class that implements `SemanticAnnotationHelper`, typically via one of the abstract base classes. To make your helper available to Mímir requires a couple of extra steps.

Helper implementations are loaded into Mímir using the standard GATE CREOLE plugin mechanism. The helper class or classes must be packaged up into a JAR file which is placed in a directory containing a simple `creole.xml` file.

```
1   <CREOLE-DIRECTORY>
2     <JAR>my-plugin.jar</JAR>
3   </CREOLE-DIRECTORY>
```

This directory is now a CREOLE plugin that can be loaded into Mímir, for example by using the configuration options described in Section 3.2.4.

7.2   Registering new Scorers

The set of scoring algorithms made available in the admin interface is specified in the application configuration file (described in Section 3.2.4). The default configuration includes the following options:

```
1  scorers.'Count Scoring' = {
2    new DelegatingScoringQueryExecutor(new CountScorer())
3  }
4  scorers.'TF.IDF' = {
5    new DelegatingScoringQueryExecutor(new TfIdfScorer())
6  }
7  scorers.'BM25' = {
8    new DelegatingScoringQueryExecutor(new BM25Scorer())
9  }
10 scorers.'Hit Length Scoring' = {
11   new BindingScorer()
12 }
```

Each declaration maps a Groovy closure to the name of a scorer. The closure must return an object implementing the `gate.mimir.search.score.MimirScorer` interface, which itself extends the MG4J interface `it.unimi.dsi.mg4j.search.score.DelegatingScorer`.

Mímir provides the `gate.mimir.search.score.DelegatingScoringQueryExecutor` class which can be used to wrap one of the scorer implementations provided by MG4J. If none of those are suitable, you can write your own custom implementations and make them avaialble to Mímir in the form of a GATE CREOLE plugin. This will ensure the new classes are added to the classpath and can be found by the system. See Section 7.1.3 for a description of how to declare new plugins, and Section 3.2.4 for information about loading additional plugins.

7.3   Embedding Mímir in a Custom Grails Application

The demo Grails application provided with Mímir provides access to the functionality of the Mímir Grails plugin but is deliberately kept simple. It does not address concerns such as security and authentication, but these can be handled by installing the Mímir plugin into a custom Grails application and adding these functions there.

7.3.1   Installing the Mímir Plugin

The Mímir plugin is installed like any other Grails plugin, by running `grails package-plugin` in the `mimir-web` directory to create a plugin zip file and then `grails install-plugin grails-mimir-web-{version}.zip` in your application's base directory to install the plugin into your application. On installation the plugin adds a number of files to the host application:

grails-app/conf/MimirConfig.groovy a default configuration file for Mímir settings, which can be customised as described in section 3.2.4.

grails-app/conf/MimirUrlMappings.groovy a default set of URL mappings for Mímir controllers. These are discussed in more detail below.

web-app/WEB-INF/gate-home a default GATE home directory, containing user and site configuration files and the default query tokeniser application, as required by the template configuration file.

7.3.2   URL Mappings

The standard URL mappings for Mímir adhere to the following conventions:

- All administrative actions have URLs that start with `/admin`. This is intended to simplify the job of providing security via an external mechanism, such as the default container-provided security defined by the servlet specification.
- The URLs for all actions referring to a given index have a common prefix (by default `/{indexId}/`). This is used to support remote access through the `mimir-client` library, which relies on a base URL to which it adds known suffixes to access given functionality (e.g. the search actions are all found under `/{indexId}/search/{actionName}`). The mimir-client library is used, for example, in the implemenation of remote indexes.

These default mappings may be modified, but the protocols used by client code to push documents into an index, to search using the XML API, and to interact with remote indexes all make assumptions about certain relative paths. Therefore, if you need to allow remote access via the `mimir-client` library, then you must make sure that:

- there is a URL prefix that is common to all actions provided by the `IndexManagementController` and the `SearchController`. Let us call this common prefix the `{indexUrl}`;
- all actions provided by the `IndexManagementController` should be published under `{indexUrl}/manage/{actionName}`;
- all actions provided by the `SearchController` should be published under `{indexUrl}/search/{actionName}`;
- all actions of the search controller must have `parseRequest:true`;
- the names of the actions in the controllers above must not be changed.

7.3.3   Security Considerations

There are a number of established and mature Grails plugins that implement user registration, authentication and security in various ways for Grails applications. By design, the Mímir plugin and demo app do not provide any security of their own, but for production deployments you may wish to consider any or all of the following.

- Place your application behind an Apache HTTPD or other similar front-end server, and proxy only those URLs that need to be public (/css, /gwt, /images, /plugins, /{yourindex}/search, etc.)

- Install a security plugin such as spring-security-core or nimble, and restrict the Mímir pages to certain users. For example the /admin pages could be restricted to administrative users, the gus pages to registered searchers, etc. This is how the `mimir-cloud` application is set up.

- If your Mímir instance is a slave in a federated index, use firewall rules or a Grails filter to restrict access to the */search URLs to only the master Mímir server.

Note that if you use user authentication on the XML search API or the remote protocol then you will need to modify the default client code to authenticate itself appropriately. For example if you require HTTP basic authentication for the search or manage URLs then you will need to configure the master Mímir server that uses this as a remote to make it pass the relevant Authorization HTTP header. The remote protocol uses the `WebUtils` class in `mimir-client` to make its HTTP calls, and there is a subclass of this class that is able to provide basic authentication headers, you can wire this in by overriding the definition of the webUtilsManager Spring bean supplied by the Mímir plugin – see the plugin source code for full details.

Appendix A

Change Log

This appendix details the main changes in each Mímir release.

A.1 Version 4.0 (Mmmm 2012)

- Changed the results presentation to be document-centric, as opposed to hit-centric.

- Overhauled the query API (in all modalities: Java local, Java remote, and XML remote) to work in document centric mode and to remove the main pain points identified.

- Simplified all the query APIs by making them almost completely synchronous.

- Added support for ranking the results (see Sections 5.3 and 7.2).

- New implementations for all the query runners (used when searching local, remote and federated indexes).

- Replaced the old GWT based UI with a new implementation (see Section 5.2.2.

- Added the mimir-cloud web application to the source tree (see Section 3.2.1).

A.2 Version 3.4.0 (November 2011)

- Added support for indexing document metadata, i.e. features (see Section 4.1).

- Mímir Grails Plugin: moved some configuration options from the external file to a database field, so that it can now be changed using the admin web UI.

- API: simplified the construction of all default Semantic Annotation Helpers. They all get a single no-argument constructor, and set of setter method for editing the various properties (Java Bean style). The Groovy interface does not change, as Groovy will automatically convert a constructor call that takes a Map to a call for the no-argument constructor, followed by all the required setPropertyXYZ calls.

- Completely removed the (previously deprecated) `ordi` plugin, as it relies on software that is no longer supported by the original authors.

- Removed the `mimir-demo` example application from the source tree. It can now be automatically generated using an Ant call (see Section 3.2).

- Licence changed to LGPL.

A.3 Version 3.3.0 (October 2011)

- Added support for marking documents as "deleted" (see section 3.4).

- Major changes to the format of the Index Template Groovy DSL (see section 4.1). The old format provided by Mímir 3.2.0 is still supported for existing semantic annotation helper types, but new helper types in future may not be supported in the old style DSL.

- Added the *SPARQL* semantic annotation helper (see section 6.4).

- Updated versions of a number of libraries (H2 database to 1.3.160, OWLIM to 3.5, MG4J to 4.0, fastutil to 6.4, dsiutils to 2.0).

- The `ordi` semantic annotation helper plugin is now deprecated. Use the `sesame` plugin instead, which supports the same on-disk format for its annotation storage but uses a different library to access it.

- Fixed various bugs and memory leaks (see subversion logs for full details).

A.4 Version 3.2.0 (May 2011)

First public release of Mímir, under an AGPL licence.