ezDL – Operator Guide

Working Group Information Engineering, University of Duisburg-Essen

Contents

1 Installation and setup walk through **5**
1.1 Database . . . 5
1.1.1 MySQL . . . 5
1.2 Simple back end setup . . . 8
1.2.1 Install needed software . . . 8
1.2.2 Prepare the installation directory . . . 8
1.2.3 Copy the Maven projects to the installation directory . 8
1.2.4 Build the system . . . 9
1.2.5 Prepare the start scripts . . . 9
1.2.6 Gather the config files . . . 9
1.2.7 Start the back end . . . 10

2 The back end and agents **13**
2.1 Configuration . . . 14
2.2 Starting agents . . . 15
2.2.1 The low-level view . . . 15
2.2.2 Starting the back end using shell scripts . . . 16
2.2.3 Starting the back end using ant scripts . . . 16

3 Wrapper agents **19**
3.1 Types of wrapper agents . . . 19
3.2 Toolkit wrappers and web proxies . . . 20
3.2.1 The wrapper . . . 20
3.2.2 Squid . . . 20

4 Deployment **23**
4.1 Customizing the Swing client . . . 23
4.2 Signing GFrameDL application . . . 24

4.2.1 Key stores 24
4.2.2 Creating a key store 24
4.2.3 Converting a PCKS12 key store to a default Java key store 24
4.2.4 Signing JARs 24

Chapter 1

Installation and setup walk through

In this chapter we will assume that you work under a Unix-related OS and have already created a system user `ezdl` with group `ezdl` and a home directory `/home/ezdl` that will contain all ezDL related data such as compiled classes, scripts and so on.

The chapter will first introduce the database setup necessary to start the ezDL backend and then take you step-by-step through a typical setup. The following chapters then outline the configuration and start of the agents and wrapper agents.

1.1 Database

This section will walk you through the process for the simplest setup that uses one single database on one single system for all agents. This setup is later assumed in the chapter about the configuration and start of the agents. It is strongly recommended to first setup the system in the way outlined in this document and then moving to the desired target setup.

1.1.1 MySQL

Configuration and encoding issues

One of the most annoying issues with computing in the whole time frame after maybe 1950 is the prevalence (in the epidemiology sense of the word) of

different character encodings—and ezDL is no exception. EzDL uses UTF-8 for transferring information whenever applicable but some infrastructure systems like databases by default don't.

To use MySQL with ezDL, you have to configure MySQL accordingly.

First, set the MySQL server to UTF 8 by adding the following line in the MySQL config (usually found at `/etc/mysql/my.cnf`):

```
[mysqld]
character_set_server = utf8
```

After that character set conversion should work as expected.

Sometimes you will want to implement auxilliary systems (e.g. for automatically adding user accounts). In these software products, the encoding must also be set correctly. There are some configurations of the MySQL server and command line tool that make it very hard to spot problems with the encoding used to insert data into the database. To make sure that your output is actually what is in the database and does not just look correct because your command line client is set so to reverse the wrong encoding used for the data in the database, some variables have to be set. This is what your command line client should look like:

```
mysql> show variables like '%char%';
+--------------------------+----------------------------+
| Variable_name            | Value                      |
+--------------------------+----------------------------+
| character_set_client     | utf8                       |
| character_set_connection | utf8                       |
| character_set_database   | utf8                       |
| character_set_filesystem | binary                     |
| character_set_results    | utf8                       |
| character_set_server     | utf8                       |
| character_set_system     | utf8                       |
| character_sets_dir       | /usr/share/mysql/charsets/ |
+--------------------------+----------------------------+
```

You can set a variable by issuing the following command at the prompt:

```
set <variablename> = <new value>
```

E.g.:

```
set character_set_client = utf8
```

If your output looks as above and also your terminal program is set to UTF-8 encoding, the output of a `SELECT` statement should look correct. If not, it is likely that the data was inserted into the database by an external program such as a web app.

Database creation

After configuring MySQL to use the proper encoding, the database has to be initialized. This section will walk you through the process for the simplest setup that uses one single database on one single system for all agents.

The first step is to create a database as follows.

```
mysql -u <root_user_name> -p
mysql> create database <databasename>;
mysql> grant all privileges on <databasename>.*
 to '<ezdldbuser>'@'localhost'
 identified by '<ezdldbpasswd>';
```

Please note that `mysql` won't echo your password.

The first line creates the database while the second line creates a user ¡ezdldbuser¿ with a password of ¡ezdldbpasswd¿. These values have to be identical with those configured in the properties (see section 2.1).

You are free to choose the name of the database to use but the name has to be the same one as in the properties files that the agents use to initialize themselves (see section 2.1 for details).

The ezDL package provides a set of initialization files that can be used to create the necessary tables. The files reside in the directory `db` at the top level of the ezDL directory tree. Every single file whose name begins with `mysql_` should be piped into the MySQL database as follows.

```
mysql -u <root_user_name> -p <databasename>  <  <filename>
```

Now the database is initialized and ready to be used.

1.2 Simple back end setup

Please note that these instructions are only intended to give a first idea of how things work in the back end. No attempt is being made to make the setup secure, fault-tolerant or get it otherwise to a state where it could be used to serve public users.

1.2.1 Install needed software

In this example we will use ant to start the back end. Both ant and ezDL run using Java 1.6. EzDL is built using Maven. So you need to install the following software packages:

- Java 1.6
- Apache Maven 3.x
- Apache ant

1.2.2 Prepare the installation directory

Choose some directory on your server. One approach is to create a new user that will own all ezDL resources and use that user's home directory to store everything.

1.2.3 Copy the Maven projects to the installation directory

In a production setup you won't probably need every ezDL agent or wrapper but for now we just copy the whole directory tree of every ezDL project to the installation directory:

- `ezdl`
- `framework`
- `service-*`
- `wrapper-*`
- `examples`

1.2.4 Build the system

Go to each of the target directories you created in the step before and build it by typing

```
$ mvn clean install
```

This will take a short time for each project and build and test everything. If any project fails to build you should fix what is wrong and continue then.

1.2.5 Prepare the start scripts

EzDL comes with scripts for the ant build system. They are located under `examples/starter`. Copy the files `start-*.xml` to the installation directory.

Open both `start-single.xml` and `start-common.xml` with a text editor and make sure that in each `target` the `dir` attribute of the `fileset` node points to the right directory. If the project directories are in the same directory as these XML files, the attributes should have values like `${subDir}/target`; the same line in the `startBroker` target should then read `framework/dlbackend/target`.

1.2.6 Gather the config files

The start scripts assume that the config files are located in the default config directory which is system-dependent.

Windows the configs are located in a directory called `ezdl` your APPDATA directory. You can find out what that directory is by opening a command line and issuing `echo %APPDATA%`.

Linux the configs are located in a directory called `.ezdl` in the home directory of the user who starts the back end.

You will find the config files for each project in a subdirectory with the name `setup/properties` located in each project's root directory.

Copy all these files to the config directory suitable for your system.

1.2.7 Start the back end

Go to the installation directory, where the start scripts are located, and issue the following command line:

```
$ ant -buildfile start-spawn.xml start
```

The agents should be started automatically. You can check if everything went fine by pointing a web browser to `http://localhost:3456`.

This page lists all agents that have been started correctly. Sometimes some agents don't get started correctly. The most prevalent error cause is missing database access. For this reason the most likely to fail agents are those that need a database. These are currently (type of agent followed by the name it usually registers with the directory)

- User agent (UA)
- Repository agent (RA)
- Query history agent (QHA)
- Plib agent (PA)
- User log agent (ULA)

To see what when wrong you can start a *single* agent in the foreground using this statement:

```
$ ant -buildfile start-single.xml [target name]
```

Substitute `[target name]` by the name of the target node in the file `start-common.xml`. E.g. the user agent has a target in this file with the name `startUA`. By convention, each agent's start target has the name `start` followed by the agent's name.

So, to see why the user agent fails, you would type

```
$ ant -buildfile start-single.xml startUA
```

For the repository agent you would substitute `startUA` by `startRA`, and so on.

Using this method enables you to see the logging output directly on the terminal. Fix what's going wrong and try again.

Please do not terminate an agent using Ctrl-C or by killing it because the directory agent might keep a stale record for that agent in its roster which could lead to subsequent and strange errors. To terminate an agent always go to the web page of the directory agent and kill the failed agent using the red button on the right hand side of the agent list.

If the agent actually starts, terminate it using the red button and start it again using the formerly used start script `start-spawn.xml`.

Chapter 2

The back end and agents

The back end of ezDL is a set of agents that communicate with each other using a common communication infrastructure like JMS or CORBA. This implies that first the communication infrastructure has to be started and then the agents. See figure 2.1 for a schematic overview.

The first thing that each agent does after being started is connecting to the communication infrastructure and registering with the directory. The directory is just another agent who keeps a list of running agents and their features. This means that the first agent to start is the directory agent.

The agents that provide external clients with access to the ezDL backend are called MTA (message transfer agent). They usually need other agents to run in order to be able to provide their service. E.g. the MTA that the Swing client uses to connect to needs the user agent for authenticating users. This means that the MTA's should be started last.

All in all the start sequence of the back end looks something like this:

1. Communication infrastructure
2. Directory agent
3. All remaining agents
    (a) service agents (e.g. search, user)
    (b) wrapper agents (e.g. ACM wrapper or IEEE wrapper)
    (c) MTA's

How to start agents is described in section 2.2.

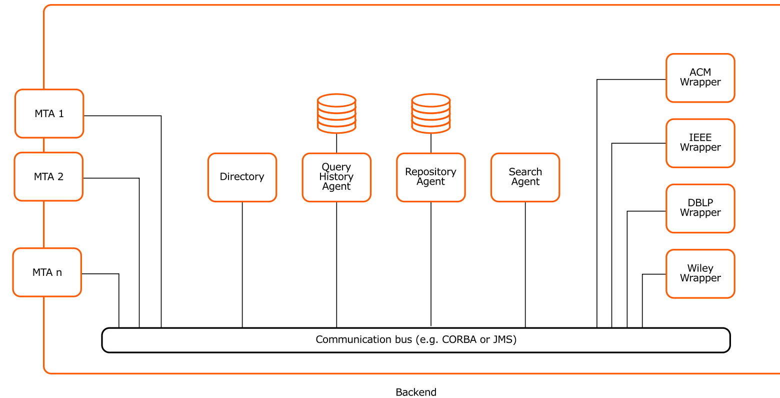

Figure 2.1: The architecture of the ezDL back end

2.1 Configuration

EzDL ships with default configuration files for all agents. These files reside in the directory `setup/properties` under the root directory of each project. E.g. the configuration files for the directory are in `setup/properties` under the project directory of the directory agent, which is usually `service-directory`. A first step is to copy the configuration file from all agents to be started to the target directory `/home/ezdl/.ezdl`.

The agents and wrappers read their configuration as follows:

1. The internal file `global.properties` is read.
2. The file `/home/ezdl/.ezdl/agent.properties` is read, overriding the definitions in global.properties.
3. The properties file given as a parameter at the start of the agent is read, further overriding all previous definitions.

This allows to configure system-wide defaults in `agent.properties` and configure single agents or wrappers to use a different configuration – e.g. for the database.

The example files contain an `agent.properties` that defines the database as follows:

```
db.url=jdbc:mysql://localhost/ezdl
```

```
db.user=ezdl
db.password=ezdl
```

This configures to use a MySQL database named `ezdl` on `localhost`, using a user name of `ezdl` and a password `ezdl`, which is, granted, not very creative or secure, but a good start to get things going. These values can be changed anytime, provided they are changed in the configuration *and* the database.

2.2 Starting agents

2.2.1 The low-level view

At the lowest level, agents are started by calling the `main()` method of the class `AgentApplication`, which accepts the following parameters (in that order):

1. the properties file name (e.g. directory.properties)
2. the class name of the agent to start
3. the name the agent uses to announce itself to the directory

About properties file names

The properties file name given as a parameter to the agent is considered to be in the default config directory for ezDL if the name is not absolute. If the config file is supposed to be in a different directory, just give the full path name of the file.

About the class name

This is usually straight-forward. If you want to start the directory agent you need to find out what the fully qualified class name of that agent's implementation is. For the directory agent it is `de.unidue.inf.is.ezdl.dlservices.backbone.directory.DirectoryAgent`.

Wrapper agents all use the same wrapper agent implementation but that agent is configured to load a specific wrapper class using the properties file. The class that has to be loaded to start a wrapper agent is

`de.unidue.inf.is.ezdl.dlwrapper.WrapperMapper`. `WrapperMapper` takes its configuration from the properties file name given at the command line. This might look like this:

Listing 2.1: acm.properties

```
1 wrapperclass=de.unidue.inf.is.ezdl.dlwrapper.wrappers.cs.acm.ACMWrapper
2 ...
```

The `WrapperMapper` uses the `wrapperclass` line to find out which wrapper class to use and loads it. So all wrapper agents are started using the same agent implementation (`WrapperMapper`) but load the code needed for their specific remove service using a properties file line. Find out more about how to configure a wrapper agent in chapter 3.

About the agent name

The agent name is used for communication between agents and serves as the address of an agent. So the agent name has to be unique. It is possible, though, to start multiple agents of the same class, as long as each one uses a different name. For example: if you need to load-balance searches you could start 5 search agents, all of which using the same configuration and agent class but the agent name might follow a pattern like `SA1`, `SA2`, ...

2.2.2 Starting the back end using shell scripts

A home-made shell script should obey the above restrictions about the start order. The class path is most easily constructed using the `target` directories that the Maven builder generates. It contains the JAR files of the agents and a subdirectory (`dependencies`) contains needed dependencies. It is a good idea to keep dependencies of different agents separate since they might conflict each other: agent A might use version α of a certain library while agent B uses version β so if both versions of this library are in the class path it depends on the order in which the libraries are searched for the classes. While this might work sometimes by pure chance, you typically would want a deterministic way of starting things and using the right library version.

2.2.3 Starting the back end using ant scripts

The easiest way to start an agent is using the supplied `ant` scripts. Three scripts are placed in the top level of the ezDL directory tree: `start-common.xml`

contains start information for all agents and wrappers in the package. This file cannot be used for starting as it is included in the other two files. `start-spawn.xml` is a start script that starts agent processes in the background. Doing so has one huge advantage: all agents can be started at once. The huge disadvantage is that there is no way to see any output of the agents except for the log files. The other file, `start-single.xml`, can be used to start exactly one agent because the agent is started in the foreground and blocks until it is halted. Since the agent is started in the foreground, it is possible to see the agent's output on the console. The latter file is mostly used for debugging.

Starting all agents at once is done like follows:

```
ant -buildfile start-spawn.xml start
```

The command results in some output that shows which agents were started but nothing else. To verify the success of the start, the directory agent's web admin page can be used. The location of it is given in `directory.properties`. The files shipped with ezDL have the following configuration in them:

```
webadmin.port=3456
webadmin.user=ezdl
webadmin.password=ezdl
```

That means that the web site is located at `localhost`, port `3456` using the given credentials for log-in. You can point your web browser to the location and hopefully see a list of agents that are running.

Chapter 3

Wrapper agents

Wrapper agents are agents that abstract from accessing (remote) resources. Wrapper agents are ”‘normal”’ agents whose functionality is defined by a `Wrapper` class. Wrapper agents are often called ”‘wrappers”’ for short, even though that actually only refers to the concrete class that implements the `Wrapper` interface.

Since wrapper agents aren't so different from the other agents, they are started and configured as such. There are some wrapper-specific configuration options, though, that are outlined in this chapter.

3.1 Types of wrapper agents

There are two different types of wrappers. The first one—called “Solr wrapper”—wraps a Solr server that is normally operated locally. Local Solr servers are useful for caching data that is easily available. This makes accessing the data very fast and reliable. The second kind wraps an external web service. This is done by downloading the web pages from that server and parsing them. The process is facilitated by an internal component called the “toolkit”. This is why this kind of wrappers is called “toolkit wrappers”. There is another difference between Solr wrappers and toolkit wrappers: when a Solr wrapper has a code defect that leads to abnormal behavior like ongoing multiple requests, usually only your own resources suffer from the load. When a toolkit wrapper freaks out, somebody else's resources suffer from what might seem to them as a DoS attack. See section toolkit-proxies for a way to make that less likely.

3.2 Toolkit wrappers and web proxies

Toolkit wrappers post requests for web resources to remote web sites and parse the responses. This might lead to problems if a wrapper has a defect that leads to hammering the remote site with requests. To prevent this from happening, caching the web requests can be used. Since the wrappers' requests often contain a query string, most proxies will not cache these resources out-of-the box. Additional configuration has to be done to both wrapper and proxy.

3.2.1 The wrapper

To tell a wrapper to use a web proxy, the following lines have to be included in the wrapper's properties:

```
toolkit.proxy.host=<hostname>
toolkit.proxy.port=<port number>
```

E.g., to configure a wrapper to use a proxy at port 8080 on `localhost`, one would add:

```
toolkit.proxy.host=localhost
toolkit.proxy.port=8080
```

The wrapper then has to be restarted since the properties file is only read at the start.

3.2.2 Squid

Squid is a free-as-in-speech web proxy. Getting a Squid proxy to cache resources whose URL's have query strings, the following changes to the `squid.conf` have to be performed:

There should be two lines that read

```
acl QUERY urlpath_regex cgi-bin \?
cache deny QUERY
```

These lines basically assign the name `QUERY` to URLs that contain either the key word `cgi-bin` or a question mark and then tells Squid to not cache resources whose URL match this pattern. Since this is exactly what we do not want, remove or comment-out these lines.

Further, Squid needs to know when to refresh cached resources. Find a proper spot for a new line by looking for lines starting with `refresh_pattern` and change the lines so that the last two lines read:

```
refresh_pattern -i (/cgi-bin/|\?)  5  50%  1440
refresh_pattern .                  0  20%  4320
```

The last line containing the full stop (".") may contain whatever values it had before. The first line in the above listing have the following meaning. First, the pattern is defined to match both URLs that have a part `/cgi-bin/` or a question mark in them, which indicates that they contain dynamic content. The first number, 5, then defines that the minimum time to keep that resource in the cache is 5 minutes. The next number, 50%, means that a resource is not refreshed, if it has been in the cache less than 50% of its age (time since last change) if it has no explicit expiry time. The last number, 1440, means that resources are refreshed if they are at most 1440 minutes (a day) in the cache. These numbers are by no way god-given. Adapt them to suit your needs.

The file `/var/log/squid/access.log` can be inspected to see if the configuration works as intended. The following snippet shows a resource that was not cached:

```
134.91.35.179 - - [17/Apr/2010:09:18:20 +0200]
"GET http://portal.acm.org/results.cfm? HTTP/1.1" 200 165084
"-" "Apache-HttpClient/4.0 (java 1.5)" TCP_MISS:DIRECT
```

Please note the `TCP_MISS`, that indicates a cache miss.

If the configuration leads to items being cached, lines like the following should be found in the log:

```
134.91.35.179 - - [17/Apr/2010:09:19:48 +0200]
"GET http://portal.acm.org/results.cfm? HTTP/1.1" 200 165121
"-" "Apache-HttpClient/4.0 (java 1.5)" TCP_HIT:NONE
```

The dead give-away here is the TCP_HIT, which means that there was a cache hit for that resource.

Please note that the search agent maintains an internal cache for queries and their result lists. So to test the cache, you not only have to restart the Squid daemon but also the search agent. Otherwise the toolkit wrapper isn't even asked again.

In order to see the query string of the URL, set the following option in the Squid config:

```
strip_query_terms off
```

This clearly has privacy implications as all queries sent to remote sites are logged in full. So this setting should only be used for debugging and testing and not in a production environment.

Chapter 4

Deployment

4.1 Customizing the Swing client

Listing 4.1: perspective.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<perspective id="usertest">
  <title lang="en">User Perspective</title>
  <title lang="de">Benutzerperspektive</title>
  <tools>
    <tool id="search" path="de.unidue.inf.is.ezdl.gframedl.tools.search.DefaultSearchTool" />
    <tool id="detail" path="de.unidue.inf.is.ezdl.gframedl.tools.details.DetailTool" />
    <tool id="queryhistory" path="de.unidue.inf.is.ezdl.gframedl.tools.queryhistory.QueryHistoryTool" /
    <tool id="clipboard" path="de.unidue.inf.is.ezdl.gframedl.tools.clipboard.ClipboardTool" />
  </tools>
  <rootWindow>
    <splitWindow orientation="horizontal">
      <splitWindow orientation="vertical" splitPos=".3f">
        <tabbedWindow>
          <toolView toolId="queryhistory" viewId="default" />
          <toolView toolId="search" viewId="tabbedqueryview" />
        </tabbedWindow>
        <toolView toolId="search" viewId="resultview" />
      </splitWindow>
      <splitWindow orientation="vertical">
        <tabbedWindow>
          <toolView toolId="clipboard" viewId="view" />
          <toolView toolId="search" viewId="wrapperchoice" />
        </tabbedWindow>
        <toolView toolId="detail" viewId="default" />
      </splitWindow>
    </splitWindow>
  </rootWindow>
</perspective>
```

4.2 Signing GFrameDL application

Signing JAR files is especially recommended for deployment via Java Web Start if the users should not be scared by warning dialogs.

4.2.1 Key stores

A so called key store is required for signing JAR files.

4.2.2 Creating a key store

```
keytool -import -trustcacerts -alias <alias> -file <certificate> -keystore <ke
```

4.2.3 Converting a PCKS12 key store to a default Java key store

```
keytool -importkeystore -deststorepass testtest -destkeypass keypass -destkeys
```

4.2.4 Signing JARs

With the generated key store (see previous section) the GFrameDL JAR file can be signed. A bash script can be found in the ezDL Mercurial repository (jnlp/ezdl-sign-jar). This script signs JAR files and deploys them via scp to a remote host.

```
tbeckers@artus:~$ ./ezdl-sign-jar
JAR file to sign (default is gframedl.jar):
Signed JAR file (default is gframedl-signed.jar):
Keystore (default is /home/tbeckers/.ezdl/keystore):
Host name to where the signed JAR will be copied (default is ezdl.de):
Final target of signed JAR file (default is /var/www/webstart):
jarsigner -keystore /home/tbeckers/.ezdl/keystore -signed-jar gframedl-signed.
Enter Passphrase for keystore:
scp gframedl-signed.jar ezdl@ezdl.de:/var/www/webstart
```