netherlands eScience center

Tutorial
Distributed Computing
with Xenon 1.1.0

Author: Jurriaan H. Spaaks

Date: February 2016

Contents

Chapter 1

Introduction

Many scientific applications require far more computation or data storage than can be handled on a regular PC or laptop. For such applications, access to remote storage and compute facilities is essential. Unfortunately, there is not a single standardized way to access such facilities. There are many competing protocols and tools in use by the various scientific data centers and commercial online providers.

As a result, application developers are forced to either select a few protocols they wish to support, thereby limiting which remote resources their application can use, or implement support for all of them, leading to excessive development time.

Xenon is a library designed to solve this problem. It offers a simple, unified programming interface to many remote computation and data storage facilities, and hides the complicated protocol and tool specific details from the application.

Xenon consists of three pillars: *credentials*, *files*, and *jobs* (Figure 1.1). The credentials pillar contains functionality pertaining to gaining access to (remote) systems, for example by using passwords, public/private keys, certificates, etc. The files pillar contains all functionality relating to the filesystem: creation, deletion, and copying of files and directories, checking if a file or directory exists, listing directory contents, etc. Lastly, the jobs pillar contains functionality relating to jobs, such as submitting and cancelling a job, checking a job's status (*polling*), etc.
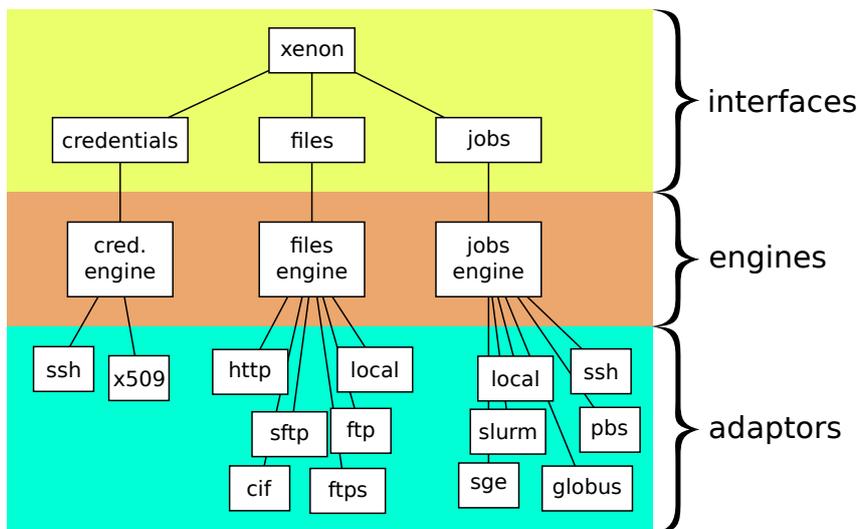
Figure 1.1: Xenon is built on 3 pillars: credentials , files and jobs . Each pillar consists of an interface, an engine, and multiple adaptors.

1.1 Use cases

Xenon is used by several projects at the Netherlands eScience Center. We use it either directly, or via the Python (pyXenon[a]) or JavaScript (Osmium[b]) interfaces. Most of these projects use Xenon to remotely run computations that are too large to run locally, or to distribute a collection of such computations over multiple remote machines. Xenon is also used to copy the input files required by these computations to the remote machines, and to collect the results afterwards. Below are some real-life examples of what Xenon is used for:

1. MAGMa[c] is a web application for the automatic annotation of mass spectrometry data. MAGMa uses Osmium to start calculations from the web interface.

---

[a] https://github.com/NLeSC/pyxenon
[b] https://github.com/NLeSC/osmium
[c] http://www.emetabolomics.org/magma

2. AMUSE[a] is a software framework for astrophysical simulations. Using AMUSE, existing astrophysical models can be easily coupled into larger simulations. The AMUSE framework uses Xenon to transparently provide distributed computing.

3. ABC-MUSE[b] uses the AMUSE software framework described above to couple various types of ocean simulations. Like AMUSE, ABC-MUSE uses Xenon to transparently provide distributed computing.

4. eSalsa-Deploy is an application that can be used to run an ensemble of large ocean simulations on a set of supercomputers. For each simulation in the ensemble, a target supercomputer must be selected, and the input and configuration files prepared on that target. Each simulation requires a large amount of time to complete, and cannot be run in one go as this would exceed the maximum allowed walltime. Instead they are started, stopped and restarted several times. All file transfers, job submission and monitoring in eSalsa-Deploy is handled by Xenon.

5. JSync is a tool used in the Via Appia project[c] to synchronize archeological data between archeologists' laptops and a centralized server located in Germany. This allows the archeologists to view data offline (e.g. during fieldwork) on their laptops. The necessary remote file inspection and file transfer is handled by Xenon.

6. Noodles[d] is a Python based workflow engine that uses pyXenon to run workflow elements in a distributed fashion.

7. In the SIM-CITY project[e], pyXenon is used as part of a webservice allowing traffic simulations to be started on demand.

---

[a]http://amusecode.org/
[b]https://www.esciencecenter.nl/project/abc-muse
[c]https://www.esciencecenter.nl/project/mapping-the-via-appia-in-3d
[d]https://github.com/NLeSC/noodles
[e]https://www.esciencecenter.nl/project/sim-city

1.2   Purpose of this document

This document aims to help users without much prior knowledge about Java programming and without much experience in using remote systems to understand the basics of how to use the Xenon library. At the end of the tutorial you should have enough information to be able to start exploring on your own.

1.3   Version information

It is assumed that you are using one of the Ubuntu-based operating systems (I'm using Linux Lubuntu 14.04.3 LTS). Nonetheless, most of the material covered in this manual should be usable on other Linux distributions with minor changes. The manual is written to be consistent with Xenon release 1.1.0[a].

---

[a]For releases, see `https://github.com/NLeSC/Xenon/releases`

Chapter 2

Basic usage

To use a minimal feature set of the Xenon library, you'll need the following software packages:

1. *Git*, a version management system;

2. *Java*, a general purpose programming language;

The following sections describe the necessary steps in more detail.

2.1   Installing Git

Open a terminal (default keybinding Ctrl + Alt + t). The shell should be Bash . You can check this with:

```
echo $0
```

which should return `/bin/bash`.

Now install `git` if you don't have it already:

```
sudo apt-get install git
```

After the install completes, we need to get a copy of the examples. We will use `git` to do so. Change into the directory that you want to end up containing the top-level repository directory. I want to put the Xenon examples in my home directory, so for me that means:

```
cd ${HOME}
```

Then clone the Xenon-examples repository into the current directory:

```
git clone https://github.com/NLeSC/Xenon-examples.git
```

This will create a directory ~/`Xenon-examples` that contains the source code
of the Xenon examples.

2.2   Installing Java

Xenon is a Java library, therefore it needs Java in order to run. Java comes in
different versions identified by a name and a number. The labeling is somewhat
confusing[a]. This is partly because Java was first developed by Sun Microsys-
tems (which was later bought by Oracle Corporation), while an open-source
implementation is also available (it comes standard with many Linuxes). Fur-
thermore, there are different flavors for each version, each flavor having dif-
ferent capabilities. For example, if you just want to *run* Java applications,
you need the JRE (Java Runtime Environment); if you also want to *develop*
Java software, you'll need either an SDK (Software Development Kit) from
Sun/Oracle, or a JDK (Java Development Kit) if you are using the open-source
variant.

Check if you have Java and if so, what version you have:

```
java -version
```

That should produce something like:

```
java version "1.7.0_79"
OpenJDK Runtime Environment (IcedTea 2.5.6) (7u79-2.5.6-0ubuntu1.14.04.1)
OpenJDK 64-Bit Server VM (build 24.79-b02, mixed mode)
```

Note that 'Java version 1.7' is often referred to as 'Java 7'.

If you don't have Java yet, install it with:

```
sudo apt-get install default-jdk
```

this will install the open-source variant of Java ('OpenJDK').

---

[a]See for example `http://stackoverflow.com/questions/2411288/java-versioning-and-terminology-1-6-vs-6-0-openjdk-vs-sun`

2.3 Building with `gradlew`

To check if everything works, we first need to build the example from source and then run the example from the command line.

At this point, ~/`Xenon-examples` only contains files directly related to the source code of the example files. However, in order to build and run the examples successfully, we'll need a few more things. Naturally, we'll need a copy of the Xenon library, but the Xenon library in turn also has dependencies that need to be resolved. Because the process of fitting together the right libraries is quite a lot of work, we have automated it. For this, we use the build automation tool Gradle[a]. Interestingly, you do not need to install Gradle for it to work (although you do need Java). This is because the Xenon-examples repository already includes a script called `gradlew`, which will download a predefined version of the Gradle program upon execution. The advantage of using `gradlew` for this is that the resulting build setup will be exactly the same as what the developers use, thus avoiding any bugs that stem solely from build configuration differences.

The `gradlew` script can be run with arguments. For example, running

```
cd ${HOME}/Xenon-examples
./gradlew dependencies
```

prints a list of the dependencies (of which there are quite many), and

```
./gradlew clean
```

cleans up the files pertaining to any previous builds.

`dependencies` and `clean` are referred to as 'Gradle tasks', 'build tasks' or just 'tasks'. Tasks are defined in a so-called build file called `build.gradle`. To get an overview of all available tasks you could read through `build.gradle`, or you could simply run:

```
./gradlew tasks
```

or

```
./gradlew tasks --all
```

If you run `./gradlew tasks`, you'll see a line at the top that says that the

---

[a]`http://gradle.org/`

default task is called `shadowJar`[a]. `shadowJar` bundles any and all dependencies into one large jar, sometimes referred to as a 'fat jar'. Once you figure out what to put in it, using a fat jar makes deployment more robust against forgotten-dependency errors. Lucky for you, someone has already figured out how to make the fat jar, and has even written it down in a file that Gradle can understand.

Run `./gradlew` (without any arguments) to start the default task. The following things will happen:

1. the correct version of Gradle will be downloaded;

2. the Xenon library will be downloaded;

3. any dependencies that the Xenon library has will be downloaded;

4. all of that will then be compiled;

5. a fat jar will be created.

2.4   Running an example

So at this point we have compiled the necessary Java classes; now we need to figure out how to run them.

The general syntax for running compiled Java programs from the command line is as follows:

```
java <fully qualified classname>
```

We will use the `DirectoryListing` example from the `nl.esciencecenter.xenon.examples.files` package. As the name implies, it lists the directory contents of a given directory. The fully qualified classname for our example is `nl.esciencecenter.xenon.examples.files.DirectoryListing`, but if you try to run

```
cd ${HOME}/Xenon-examples
java nl.esciencecenter.xenon.examples.files.DirectoryListing
```

you will get the error below:

---

[a]https://github.com/johnrengelman/shadow

```
Error: Could not find or load main class \
nl.esciencecenter.xenon.examples.files.DirectoryListing
```

This is because the `java` executable tries to locate our class `nl.esciencecenter.`
`xenon.examples.files.DirectoryListing`, but we haven't told `java` where
to look for it. We can resolve that by specifying a list of one or more directories
using `java`'s classpath option `-cp`.

Directory names can be passed to `java` as a colon-separated list, in which
directory names can be relative to the current directory. Furthermore, the
syntax is slightly different depending on what type of file you want `java` to
find in a given directory: if you want `java` to find compiled Java classes, use
the directory name; if you want `java` to find jar files, use the directory name
followed by the name of the jar (or use the wildcard `*` if you want `java` to
find any jar from a given directory). Finally, the order within the classpath is
significant.

We want Java to find the fat jar `Xenon-examples-all.jar` from `build/libs`.
Using paths relative to ~`/Xenon-examples`, our classpath thus becomes `build/`
`libs/Xenon-examples-all.jar`. However, if we now try to run

```
cd ${HOME}/Xenon-examples
java -cp 'build/libs/Xenon-examples-all.jar' \
nl.esciencecenter.xenon.examples.files.DirectoryListing
```

it still does not work yet, because `DirectoryListing` takes exactly one input
argument that defines the location (URI) of the directory whose contents we
want to list. URIs generally consist of a *scheme* followed by the colon charac-
ter `:`, followed by a *path*[a]. For a local file, the scheme name is `local`. The path
is the name of the directory we want to list the contents of, such as `${PWD}`
(the present working directory).

Putting all that together, we get:

```
cd ${HOME}/Xenon-examples
java -cp 'build/libs/Xenon-examples-all.jar' \
nl.esciencecenter.xenon.examples.files.DirectoryListing local:${PWD}
```

If all goes well, you should now see the contents of the current directory as an
INFO message.

---

[a]The full specification of a URI is (optional parts in brackets):
```
scheme:[//[user:password@]domain[:port]][/]path[?query][fragment]
```

2.5 Setting the log level

Xenon uses the logback[a] library for much of its verbose output. For this, developers have sprinkled the code with so-called *logger* statements that produce messages. Each message has been annotated as an error, a warning, debugging information, or as a plain informational message. The logback library lets the user configure where each type of message should be routed. For example, warnings and informational messages may be routed to standard output, while error messages may be routed to standard error. Furthermore, it gives the user control of how each message should be formatted, for example with regard to what class produced the message, and at what line exactly. The behavior of the logger can be configured by means of a file called `logback.xml`. It is located at `src/main/resources/`. There should not be any need for you to change `logback.xml` too much, but if you do, make sure to re-run `./gradlew` for your changes to take effect.

By default, `logback.xml` uses a logging level of `INFO`, which means that warnings, errors, and informational messages are routed to standard output, but debug messages are not visible. At some point, you may find yourself in a situation where you want to change the logging level. There's two ways you can do that: first you can change the default behavior by editing the `loglevel` line in `src/main/resources/logback.xml`, saving it, and re-running `./gradlew`.

---

[a]`http://logback.qos.ch/`

Secondly, you can keep the defaults as they are, but only run a specific call with altered `loglevel` settings, by using command line parameters to the `java` program. For example, you could lower the logging level from `INFO` to `DEBUG` as follows:

```
cd ${HOME}/Xenon-examples
java -Dloglevel=DEBUG -cp 'build/libs/Xenon-examples-all.jar' \
nl.esciencecenter.xenon.examples.files.DirectoryListing local:${PWD}
```

Standard output will now include messages of the `WARN`, `ERROR`, `INFO`, and `DEBUG` level, e.g.:

```
time    : 11:49:58.473 (+315 ms)
thread : main
level  : DEBUG
class  : nl.esciencecenter.xenon.examples.files.DirectoryListing:74
message: Creating a new Path...

time    : 11:49:58.473 (+315 ms)
thread : main
level  : DEBUG
class  : nl.esciencecenter.xenon.examples.files.DirectoryListing:78
message: Getting the directory attributes...

time    : 11:49:58.481 (+323 ms)
thread : main
level  : INFO
class  : nl.esciencecenter.xenon.examples.files.DirectoryListing:95
message: Directory local:/home/daisycutter/Xenon-examples exists
         and contains the following:
         .travis-ssh.sh
         run-examples.sh
         .gitignore
         doc
         build
         bin
                          <some content omitted>
```

2.6   Xenon's SSH adaptor

Up till now, you've used Xenon to perform tasks on your own system (by means of Xenon's local adaptor). Usually though, you'll want to use Xenon to perform tasks on remote systems, such as the Amazon cloud, the Lisa cluster computer, the Cartesius supercomputer, or perhaps just a single machine (but not your own). To do so, you first need to connect to them. You can use Xenon's SSH adaptor to do so. The SSH adaptor works in a similar way as the so-called *secure shell* program `ssh` on the Linux command line. Through SSH, you can connect to a remote system and do all the things a user is allowed to do on that system (which in turn is defined by the remote system's administrator).

To keep things simple, we'll first set up an SSH connection to `localhost`, thus effectively connecting ourselves with ourselves. Later on in this document we'll also look at using SSH to connect to a remote system (see Section 2.15: *Live systems*). Additionally, we'll set up Docker to run virtual remote systems on your local machine. That way, you can emulate the behavior of one or more remote machines (see Sections 2.16: *Installing Docker* and 2.17: *Using Docker containers*) without the need for any additional hardware besides your own machine. Anyway, that's all for later—let's get started on setting up SSH to `localhost` first.

2.6.1 `ssh` to `localhost`

SSH uses the server-client model. Normally, your system is the client, while the remote system is the server. Since we will connect to `localhost` however, your system will play the part of both client and server.

The client needs to have a so-called *public-private key pair*. The key pair can be generated according to different algorithms: `dsa`, `ecdsa`, `ed25519`, or `rsa`. The Linux program `ssh-keygen` implements these algorithms. We will use it to generate a public-private key pair using the `rsa` algorithm, as follows:

```
# generate key pair using rsa algorithm
ssh-keygen -t rsa
# (press enter to accept the default file location ~/.ssh/id_rsa; if it
# tells you the file already exists, you can skip creation of a new key)
# (press enter to set the passphrase to an empty string)
# (press enter to confirm setting the passphrase to an empty string)
```

You should now have a directory ~/`.ssh` with two files in it: `id_rsa`, the private part of the key pair and `id_rsa.pub`, the public part of the key pair. Note that the contents of `id_rsa` should remain a secret.

That's it for the client side, but you still need to do some stuff on the server side. First, install the package `openssh-server` from the Ubuntu repositories:

```
# install a server
sudo apt-get install openssh-server
```

The server side keeps a list ~/`.ssh/authorized_keys` of trusted identities. You have to add the identity information from ~/`.ssh/id_rsa.pub` to it. Linux provides an easy way of doing this, as follows:

```
# copy identity, type 'yes' when prompted
ssh-copy-id ${USER}@localhost
```

On the server side, file ~/`.ssh/authorized_keys` should now contain the information from ~/`.ssh/id_rsa.pub`.

When you did `ssh-copy-id`, SSH tried to connect the current user (whose credentials are found in ~/.ssh/id_rsa.pub) to the 'remote' system `localhost`. If that was successful, there should now be a new file ~/.ssh/known_hosts on the client, that contains a list of machines that we have connected to previously; i.e. these are systems we trust.

Each line in `known_hosts` is formatted according to:
`|<flag>|<salt>=|<hostname>= <key> <value>=`, where

- `<flag>` is a flag. Mine says `1` to signify that the third element `<hostname>` is hashed using the SHA1 algorithm;

- `<salt>` is the (public) salt used to encrypt the host name;

- `<hostname>` is the (hashed) host name;

- `<key><value>` pairs, e.g. the key `ecdsa-sha2-nistp256` and its value `AAAAE2V...<characters omitted>...RpXi/rE`, representing the public key of the 'remote' system which was generated when we installed `openssh-server` and which is stored at `/etc/ssh/ssh_host_ecdsa_key.pub`[a] on hostname.

Xenon uses `known_hosts` to automatically connect to a (known) remote system, without having to ask for credentials every time.

For passwordless SSH to work correctly, permissions are important. You can check the octal representation of a file's permissions by:

```
stat -c "%a" <filename>
```

and you can change them with:

```
# change <filename>'s permissions to rwx------ or octal 700
chmod 700 <filename>
```

On the server-side, verify that permissions are set correctly on the following files and directories:

---

[a]You can show the fingerprint of the server's public key file using:
```
ssh-keygen -l -f /etc/ssh/ssh_host_ecdsa_key.pub
```
The number that `ssh-keygen` returns should be the same number that was displayed in your terminal when you did `ssh-copy-id $ USER@localhost`.

1. `${HOME}/` and `${HOME}/.ssh/` should be writable only by owner, not by group or all. On my system, I have octal permission 700.

2. `${HOME}/.ssh/authorized_keys` should be writable only by owner, not by group or all. On my system, I have octal permission 600.

On the client-side, verify that permissions are set correctly on the following files and directories:

1. `${HOME}/` and `${HOME}/.ssh/` should be writable only by owner, not by group or all. On my system, I have octal permission 700.

2. `${HOME}/.ssh/id_rsa` should be octal permission 600. `ssh-copy-id` should have set the permissions correctly.

3. `${HOME}/.ssh/id_rsa.pub` should be octal permission 644. `ssh-copy-id` should have set the permissions correctly.

```
# now try to log in without entering key:
ssh ${USER}@localhost
```

If all goes well, you should be logged in automatically without having to enter a password. You can log out of the SSH session with:

```
logout
```

2.7 DirectoryListing through SSH

So now that we have configured passwordless SSH to `localhost`, listing the contents of a directory on `localhost` is actually pretty simple. In the URI that you pass as a command line argument to `DirectoryListing` (see page 9), replace `local` by `ssh`, and supply `localhost` as the domain name. Also note that the inclusion of a domain makes it necessary to include `//` between `ssh` and `localhost` (see footnote on formatting URIs on page 9). So the complete call becomes:

```
cd ${HOME}/Xenon-examples
java -cp 'build/libs/Xenon-examples-all.jar' \
nl.esciencecenter.xenon.examples.files.DirectoryListing \
ssh://localhost${PWD}
```

If all goes well, you should see the same directory contents as before, but now viewed through SSH.

2.8 Example usage of local and SSH adaptors

We have compiled a list of examples demonstrating various uses of the local adaptor and the SSH adaptor. Your system is now configured correctly to run all of them. This is most easily accomplished by starting the `run-examples.sh` script we have prepared:

```
cd ${HOME}/Xenon-examples
./run-examples.sh
```

Note that you can edit the loglevel by changing the parameter `LOGLEVEL` in `run-examples.sh`.

2.9 A minimal Eclipse installation

So now that we've verified that everything works, we can start thinking about doing some development work. Let's first look at the Java editor Eclipse.

Eclipse is a very powerful, free, open-source, integrated development environment for Java (and many other languages). It is available in most Linuxes from their respective repositories. By default, Eclipse comes with many features, such as Git (version control), Mylyn (task management), Maven (building), Ant (building) , an XML editor, as well as some other stuff. While these features are nice, they can get in the way if you're new to code development with Java using Eclipse. We will therefore set up a minimal Eclipse installation which includes only the Eclipse platform and the Java related tools (most importantly, the debugger). Feel free to skip this next part if you're already familiar with Eclipse.

Go to `http://download.eclipse.org/eclipse/downloads/`. Under 'Latest release', click on the link with the highest version number. It will take you to a website that has a menu in the upper left corner. From that menu, select the item 'Platform Runtime binary', then download the file corresponding to your platform (for me, that is `eclipse-platform-4.5-linux-gtk-x86_64.tar.gz`). Go back to the menu by scrolling up, then select the item 'JDT Runtime binary', and download the file (there should be only one; for me that is `org.eclipse.jdt-4.5.zip`).

Now go to where you downloaded those two files to. Uncompress `eclipse-platform-4.5-linux-gtk-x86_64.tar.gz` and move the uncompressed files to a new directory `~/opt/minimal-eclipse/` (they can be anywhere, really,

but ~/**opt** is the conventional place to install user-space programs on Linux). Start Eclipse by running `eclipse` from ~/opt/minimal-eclipse/eclipse.

In Eclipse's menu go to Help, then select Install New Software... . Near the bottom of the dialog, uncheck Group items by category. Then click the top-right button labeled Add... and click Archive... . Then navigate to the second file you downloaded, `org.eclipse.jdt-4.5.zip` and select it. In the dialog, a new item Eclipse Java Development Tools should appear. Make sure it's checked, then click Next and Finish. When Eclipse restarts, you should have everything you need for Java development, without any of the clutter!

Adding a Bash alias to ~/.bash_aliases will make it easier to start the program. I've used

```
echo "alias miniclipse='${HOME}/opt/minimal-eclipse/eclipse/eclipse'" >> \
${HOME}/.bash_aliases
```

to do so (restart your terminal to use the `miniclipse` alias).

2.10 Automatic project setup with `gradlew`

Normally, when you start a new project in Eclipse, it takes you through a series of dialogs to set up the Eclipse project in terms of the directory structure, the classpath, etc. The configuration is saved to (hidden) files `.project`, `.classpath`, and `.settings/org.eclipse.jdt.core.prefs`. The dialogs offer some freedom in setting up the project. This flexibility is great when you're working on some project by yourself, but when there are multiple people working together, one developer may have a different project setup than the next, and so bugs are introduced. That's why we will use `gradlew` to generate a standard project setup for us:

```
cd ${HOME}/Xenon-examples
./gradlew eclipse
```

2.11 Opening the Xenon examples in Eclipse

After the Eclipse files have been generated, start Eclipse by typing the Bash alias `miniclipse` at the command line. From Eclipse's menu, select File→Import. In the Select dialog, select Existing projects into Workspace, then click the button labeled Next.

In the next dialog (Import projects), use the Browse... button to select the

project's root directory, e.g. `/home/daisycutter/Xenon-examples`, then click
Finish. An item Xenon-examples should now be visible in the Package explorer
pane. Expand it, and navigate to `src/main/java/`, then double-click Directo-
ryListing.java from the nl.esciencecenter.xenon.examples.files package to view its
code in the editor pane.

2.12   Running a Java program in Eclipse

So now that we have the source code open in the editor, let's see if we can
run it. You can start the program in a couple different ways. For example,
you can select Run→Run; you can use the key binding Ctrl+F11, or you can
press the 'Play' icon in Eclipse's GUI. If you try to run the program, however,
you will get the error we saw earlier at the command line (Eclipse prints the
program's output to the pane labeled Console):

```
time   : 18:42:04.689 (+219 ms)
thread : main
level  : ERROR
class  : nl.esciencecenter.xenon.examples.files.DirectoryListing:51
message: Example requires a URI as parameter!
```

So somehow we have to tell Eclipse about the URI (including both its scheme
and its path) that we want to use to get to the contents of a directory of
our chosing. You can do this through so-called *Run configurations* . You can
make a new run configuration by selecting Run from the Eclipse menu, then
Run configurations.... In the left pane of the dialog that pops up, select Java
Application, then press the New launch configuration button in the top left of the
dialog. A new run configuration item should now become visible under Java
Application. By default, the name of the run configuration will be the name of
the class, but you can change the name to whatever you like. When you select
the DirectoryListing run configuration in the left pane, the right pane changes to
show the details of the run configuration. The information is divided over a few
tabs. Select the tab labeled Arguments. You should see a field named Program
arguments where you can provide the arguments that you would normally pass
through the command line. Earlier, we passed the string `local:${PWD}`, but
that won't work here, since `${PWD}` is a Bash environment variable, and thus
not directly available from within Eclipse. Eclipse does provide a workaround
for this by way of the `env _var` variable. `env_var` takes exactly one argument,
namely the name of an environment variable, such as `PWD`. The correct text
to enter into Program arguments thus becomes `local:${env_ var:PWD}`. Note
that `${env_var:PWD}` refers to the directory that Eclipse was started from.

2.13 Debugging a Java program in Eclipse

In my opinion, one of the most helpful features of the Eclipse interface is the debugging/inspecting variables capability. This lets you run your program line-by-line. To start debugging, you have to set a breakpoint first. Program execution will halt at this point, such that you can inspect what value each variable has at that point in your program. Setting a breakpoint is most easily accomplished by double-clicking the left margin of the editor; a blue dot will appear. Alternatively, you can press Ctrl+Shift+b to set a breakpoint at the current line.

Set a breakpoint at the line

```
URI uri = new URI(args[0]);
```

Now we need to set the *debug configuration* in a similar manner as we did for the run configuration. Select Run from the menu, select Debug configurations... (not Run configurations...), then select the configuration we used previously.

Run the program up to the position of the breakpoint. There are various ways to start a debug run: e.g. by selecting Run→Debug; or by pressing F11.

You can add all kinds of helpful tools to the Eclipse window; for an overview of your options, click the Window menu item, then select Show view, then select Other.... Select whatever tools you like, but make sure to select the Variables tool from Debug. This tool allows you to view information about your variables while you're debugging. Use drag and drop to lay out the Eclipse window to suit your needs. Eclipse refers to its window layout as a *perspective* ; perspectives can be saved by subsequently selecting Window, Perspective, and Save perspective as.... This allows you to have custom perspectives for development in different languages (Java, Python, C, etc.), or for different screen setups (laptop screen v. side-by-side 1920x1080 for example), or for different tasks (Java development v. Java debugging for example).

Getting back to DirectoryListing, execution has been halted just before the line URI uri = new URI(args[0]); was executed. If you now look in the Variables tool pane, there should be only one variable visible: args, which contains the string we supplied through the Program arguments of the debug configuration.

Press F6 to evaluate the line. You'll see a new variable uri of type URI appear in the Variables pane. Expand the object to inspect it in more detail.

When you're done inspecting, press F8 to make Eclipse evaluate your program, either up to the next breakpoint, or if there are no breakpoints, up to the end

of your program.

Finally, you can terminate a debug run by pressing Ctrl+F2. Table 2.1 summarizes some of the most common Eclipse key bindings used in running and debugging Java programs.

Table 2.1: Default key bindings used for running and debugging Java programs in Eclipse.

Default key binding	Description
F5	Step in
F6	Step over
F7	Step return
F8	Continue to the next breakpoint
F11	Start a debug run
Ctrl+F2	Terminate a debug run
Ctrl+Shift+b	Set a breakpoint at the current line
Ctrl+F11	Start a (non-debug) run

2.14   Setting the log level

Earlier, we looked at how to pass custom `loglevel` values to `java` using command line parameters such as `-Dloglevel=DEBUG` (see section 2.5). Passing command line parameters is also possible in Eclipse, by altering the debug configuration (or run configuration) as follows. In Eclipse's menu, go to Run and select either Run configurations... or Debug configurations... as appropriate. Then, in the left pane, select the Java application whose configuration you want to adapt. In the right-hand pane, subsequently select the tab labeled Arguments. The second text field from the top should be labeled VM arguments. Here you can add command line parameters to the java program, such as `-Dloglevel=DEBUG`.

2.15   Live systems

Earlier, we set up passwordless SSH to connect to `localhost`. Obviously, using SSH to connect to your own system is a bit silly. Normally, you'll want to connect to a physically remote system. In the next two sections, I'll explain

how to connect to SURFsara's cluster computer, Lisa, and VU University's DAS-4 cluster, respectively. If you want to follow along, you need to make sure you have an account on at least one of these systems. Check their respective websites[a,b] to see how you can apply for an account.

If for some reason you can not or do not want to connect to these systems, make sure to check Sections 2.16: *Installing Docker* and 2.17: *Using Docker containers* to learn about using virtual remote systems.

2.15.1   `ssh` to SURFsara's Lisa cluster computer

Cluster computers typically have a dedicated machine (the so-called 'headnode') that serves as the main entry point when connecting from outside the cluster. For Lisa, the headnode is located at `lisa.surfsara.nl`.

First, I verify that I can connect to Lisa's head node, using the `ssh` command below:

```
# (my account on Lisa is called jspaaks)
ssh jspaaks@lisa.surfsara.nl
```

If this is the first time you connect to the remote machine, it will generally ask if you want to add the remote machine to the list of known hosts. For example, here's what the Lisa system tells me when I try to `ssh` to it:

```
The authenticity of host 'lisa.surfsara.nl (145.100.29.210)' can't be
established.
RSA key fingerprint is b0:69:85:a5:21:d6:43:40:bc:6c:da:e3:a2:cc:b5:8b.
Are you sure you want to continue connecting (yes/no)?
```

If I then type `yes`, it says[c]:

```
Warning: Permanently added 'lisa.surfsara.nl,145.100.29.210' (RSA) to
         the list of known hosts.

                    <some content omitted>
```

and asks for my password.

The result of this connection is that ~/.`ssh/known_hosts` now includes a line

---

[a]`https://userinfo.surfsara.nl/systems/lisa`

[b]`http://www.cs.vu.nl/das4/`

[c]SURFsara publish RSA public key fingerprints for their systems at `https://userinfo.surfsara.nl/systems/shared/ssh`. The number posted there should be the same as what you have in your terminal.

for the Lisa system.

In Eclipse, go to `DirectoryListing`'s Debug configurations like you did before and bring up the Arguments tab. Instead of connecting to `localhost`, we now want to connect to the Lisa cluster's head node at `lisa.surfsara.nl`. The full URI[a] should now be `ssh://jspaaks@lisa.surfsara.nl:22/home/jspaaks`, although you can get away with leaving the `port` value (`22`) and the `path` value (`/home/jspaaks`) unspecified, because they are the same as the default settings for the Lisa system.

However, if you now run `DirectoryListing`, it still won't work, because we still have to set up passwordless login using a public/private key pair.

Add your public key information to `~/.ssh/authorized_keys` on the Lisa system with:

```
# (from your own system)
ssh-copy-id jspaaks@lisa.surfsara.nl
```

Verify that you can now log in without being asked for a password by:

```
ssh jspaaks@lisa.surfsara.nl
```

You should now be able to list the directory contents of your home directory on the Lisa using `DirectoryListing`.

2.15.2   `ssh` to DAS-4 cluster computer

Connecting to DAS-4 works in a similar way as connecting to Lisa, except that DAS's headnode is located at `fs0.das4.cs.vu.nl`, so the full URI becomes: `ssh://jspaaks@fs0.das4.cs.vu.nl:22/home/jspaaks`. Due to the default settings on DAS-4, only the port number is superfluous here; omitting the path will result in listing the directory contents of `/`.

2.16   Installing Docker

So far, we've been using SSH to connect to `localhost` as well as to some live systems. Each option comes with its own drawbacks: live systems require

---

[a]see page 9 for URI's format specification

additional hardware, and they may be temporarily unavailable (for instance when you are on a flaky wireless network, or when other users have higher-priority jobs). On the other hand, development on `localhost` can be tricky as well, for example when one job requires software that conflicts with that of another job. With Docker, you can combine the pros of running on `localhost` with the pros of running on live systems, with few of the drawbacks. We will set up an environment in which a virtual remote system is used. Multiple virtual remote systems may in actual fact run on one physical machine, because each is running in its own *container*.

For now, let's just install the Docker software that we'll use to set up virtual remote systems. Note that much of this section is based on: `https://docs.docker.com/engine/installation/ubuntulinux/`, so that's where you need to go if you want to learn more.

Docker requires that your operating system (the host system) is 64-bit. Also, it needs a minimum Linux kernel version of 3.10 (on the host).

Check your kernel version with:

```
uname -r
```

Mine is `3.13.0-67-generic`.

The Ubuntu repositories contain an older version of Docker, which you should not use. Instead, use the newer version from Docker's own PPA.

First check if you have the older version by:

```
docker -v
```

Mine says:

```
Docker version 1.9.0, build 76d6bc9
```

If your version is lower, go ahead and uninstall as follows. First find out where your Docker program lives with:

```
which docker
```

and then find out which package your Docker is a part of with:

```
dpkg -S `which docker`
```

If you already had Docker installed, then the package name is likely either `docker.io` or `lxc-docker`. Either way uninstall the entire package, including its settings with:

```
sudo apt-get remove --purge docker.io
```

or

```
sudo apt-get remove --purge lxc-docker*
```

We will use software from a third-party repository, `https://apt.dockerproject.org`. For this, we'll need to add the new repository's PGP key to our installation as follows:

```
sudo apt-key adv --keyserver hkp://pgp.mit.edu:80 \
--recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

The details of the next step vary depending on the operating system you are using, so let's first check which version you are running:

```
lsb_release -dc
```

Make a note of your distribution's codename for the next step (mine is `trusty`).

Open or create the file `/etc/apt/sources.list.d/docker.list` in an editor such as nano, gedit, leafpad, etc. I'm using nano:

```
sudo nano /etc/apt/sources.list.d/docker.list
```

Delete any existing entries in `/etc/apt/sources.list.d/docker.list`, then add one of the following options:

1. `deb https://apt.dockerproject.org/repo ubuntu-precise main`

2. `deb https://apt.dockerproject.org/repo ubuntu-trusty main`

3. `deb https://apt.dockerproject.org/repo ubuntu-vivid main`

4. `deb https://apt.dockerproject.org/repo ubuntu-wily main`

(I chose the second because I'm on trusty).

Next, save and close `/etc/apt/sources.list.d/docker.list`.

Now that we have added Docker's PPA to the list of software sources, we need to update the list with the package information as follows:

```
sudo apt-get update
```

Check if your are now using the right docker:

```
apt-cache policy docker-engine
```

Mine says:

```
docker-engine:
Installed: 1.9.0-0~trusty
Candidate: 1.9.0-0~trusty
Version table:
***1.9.0-0~trusty 0
    500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
    100 /var/lib/dpkg/status
   1.8.3-0~trusty 0
    500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.8.2-0~trusty 0
    500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.8.1-0~trusty 0
    500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.8.0-0~trusty 0
    500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.7.1-0~trusty 0
    500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.7.0-0~trusty 0
    500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.6.2-0~trusty 0
    500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.6.1-0~trusty 0
    500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.6.0-0~trusty 0
    500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.5.0-0~trusty 0
    500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
```

Now for the actual install. If your Ubuntu version is Ubuntu Wily 15.10, Ubuntu Vivid 15.04, or Ubuntu Trusty 14.04 (LTS), you're in luck, as these OS'es have everything you'll need already. If you're not on one of these Ubuntu versions, refer to `https://docs.docker.com/engine/installation/ubuntulinux/` for instructions on installing some additional packages before proceeding with the next step.

Install Docker with:

```
sudo apt-get install docker-engine
```

The Docker service should have started; if for some reason it hasn't, you can start it manually by:

```
sudo service docker start
```

Now let's try a small example to see if Docker works:

```
sudo docker run hello-world
```

This command downloads a test image `hello-world` from DockerHub, an external repository for storing Docker images. Just to be clear, an 'image' in this context refers to an image of an operating system—it has nothing to do with a picture.

When the container runs, it prints an informational message. Then, it exits.

You can check where docker images are stored by:

```
docker info
```

Mine are stored under `/var/lib/docker`; whatever the location, make sure you have enough disk space there, as Docker will download any new containers to that location. Docker images are typically less than 1 GB each.

The Docker daemon binds to a Unix socket instead of a TCP port. By default, that Unix socket is owned by the user `root` and other users can access it with `sudo`. For this reason, the Docker daemon always runs as the `root` user.

To avoid having to use `sudo` when you use the `docker` command, we will create a Unix group called `docker` and add users to it. When the Docker daemon starts, it makes the ownership of the Unix socket read/writable by the `docker` group.

Add yourself to the `docker` group with:
```
sudo usermod -G docker -a <name-of-user>
```

Log out and back in. You should now be able to re-run the `hello-world` example without the need for `sudo`:
```
docker run hello-world
```

2.17   Using Docker containers

In this section, I'll show you how you can `ssh` into a Docker container instead of `localhost`. The advantage of this approach is that the configuration of the server is nicely separated from that of the client, making it easier to deal with conflicting configurations (as long as each is contained in its own Docker container).

While you could configure your own Docker container, there's no need—you can simply re-use one of the Docker containers we use for integration testing of the Xenon library. There's a whole bunch of them on DockerHub [a]. In this section and the next, I'll be using the container named `nlesc/xenon-torque`.

---

[a]`https://hub.docker.com/search/?q=xenon`

Download and start `nlesc/xenon-torque` with:

```
docker run -d --cap-add=SYS_RESOURCE -p 2222:22 \
--name tremendous-torque  nlesc/xenon-torque
```

The `docker` command breaks down as follows: basically we want `docker` to `run` the container `nlesc/xenon-torque` for us, using some options:

- `-d` to signify that we want the Docker container to run in the background;

- `--cap-add=SYS_RESOURCE` to tell `docker` that the container can use all the resources that are physically available given the hardware it's running on;

- `-p 2222:22` to tell `docker` that the host system's port 2222 should map to port number 22 inside the container;

- and finally, `--name tremendous-torque` to set the container's name.

You can get an overview of all Docker containers that your system knows about with:

```
docker ps --all
```

The Docker container named `tremendous-torque` should be running in the background (because we started it with the `-d` flag). Verify that its `STATUS` is `up` and that the port numbers have been mapped correctly.

Let's try if we can `ssh` into the container. For this we need to identify ourselves with a username/password combination. The `tremendous-torque` container has a user whose username is `xenon` and whose password is `javagat`. Run:

```
ssh xenon@localhost -p 2222
```

and fill in the password when prompted.

An `ls -l` should yield something like this:

```
Last login: Wed Dec 16 13:07:17 2015 from 172.17.0.1
[xenon@localhost ~]$ ls -l
total 12
-rw-r--r-- 1 root  root  1165 Dec 16 13:08 supervisord.log
-rw-r--r-- 1 root  root     2 Dec 16 13:05 supervisord.pid
drwxr-xr-x 4 xenon xenon 4096 Sep 30 09:24 xenon_test
```

Now that you've verified that the Docker container works, `logout` of it.

In Eclipse, make a copy of `DirectoryListing` and call it something like `DirectoryListingWithPwd`. We'll use `DirectoryListingWithPwd` to demonstrate a second way of logging in to a remote machine, namely using a name and password combination. For this, we need to create a `credential`, as follows. In `DirectoryListingWithPwd`, find the lines that say:

```
String scheme = uri.getScheme();
String auth = uri.getAuthority();
FileSystem fs = files.newFileSystem(scheme, auth, null, null);
```

and add the following code just above it to specify a credential:

```
// Retrieve the Credentials API
LOGGER.debug("Getting the credentials interface...");
Credentials credentials = xenon.credentials();

// Create a password credential
LOGGER.debug("Creating a password credential...");
String theScheme = "ssh";
String theUsername = "xenon";
char[] thePassword = "javagat".toCharArray();
Credential credential = credentials.newPasswordCredential(theScheme,
        theUsername, thePassword, null);
```

Make a new run/debug configuration. For Program arguments, use `ssh://`
`xenon@localhost:2222/home/xenon`.

Now try to run the new file `DirectoryListingWithPwd` (don't forget to re-run `./gradlew` if you're testing at the command line). If all goes well, you should see the directory contents of `xenon`'s home directory in the Docker container named `tremendous-torque`.

Containers can be stopped by:

```
docker stop <identifier>
```

Where `<identifier>` can be a container's `NAME`, such as `tremendous-torque`, or its `CONTAINER_ID`, such as `902ab7ee1143`, for example:

```
docker stop tremendous-torque
```

Removing Docker containers from your system works in a similar manner:

```
# remove Docker container by NAME
docker rm tremendous-torque

# remove Docker container by CONTAINER_ID
docker rm 902ab7ee1143
```

2.18   Xenon's Torque adaptor

Torque[a] is a so-called *resource manager*. There are quite a few resource managers, but the most important ones are Torque and SLURM[b] . Strictly speaking, Torque is just one member a closely related group of resource managers, including for example PBS, OpenPBS[c,d], Sun Grid Engine (SGE), Oracle Grid Engine[e] [f], Open Grid Engine[g], and Torque itself.

A resource manager is a program that typically runs on remote systems such as clusters, clouds, or supercomputers. As the name implies, the resource manager's job is to manage how the remote system's resources are allocated to the users. In doing so, it takes into account what type of resources are available, and what type of resources are needed in terms of processing power, number of cores, network speeds, etc. The resource manager works together with a *scheduler* to determine when the required resources will be made available to the user. Note that schedulers can prioritize certain users' jobs, for example if these users have paid more than others to make use of the system.

To make a job's requirements known to Torque, a user typically writes a plain text file (a *jobscript*) laying out the required walltime, number of cores, number of nodes, and so on. Perhaps most importantly, the jobscript also defines the name of an executable, i.e. the work that the user wants done. The jobscript is then submitted to the resource manager, who together with the scheduler finds a time slot during which the required resources are available simultaneously.

---

[a]http://www.adaptivecomputing.com/products/open-source/torque/
[b]http://slurm.schedmd.com/
[c]http://www.pbsgridworks.com/Default.aspx
[d]http://www.mcs.anl.gov/research/projects/openpbs/
[e]http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html
[f]http://www.univa.com/oracle.php
[g]http://gridscheduler.sourceforge.net/

2.18.1 Submitting jobs to Torque

Here is an example of a very simple jobscript:

```
1  #PBS -l nodes=1:ppn=1
2  #PBS -l walltime=00:10:00
3  #PBS -o stdout.txt
4  #PBS -e stderr.txt
5
6  # print the start time
7  echo `date`: starting the job
8
9  # the actual work I want done:
10 /bin/sleep 30
11
12 # print the end time
13 echo `date`: job completed
```

At the top of the script, there are a few lines that start with `# PBS`. Those lines define options to the resource manager. Note that these are options to Torque—the `#PBS` part is just part of Torque's inheritance from its parent PBS.

With the options used here, I ask the resource manager for one machine (`-l nodes=1`), on which I will run 1 process (`ppn=1`), which I estimate will take less than 10 minutes (`-l walltime=00:10:00`). I further ask Torque to print any standard output message to a file `stdout.txt` (`-o stdout.txt`), and any standard error messages to a file `stderr.txt` (`-o stderr.txt`). Note that there are many more PBS options, the ones I'm using here are just a few of the more common ones. In line 10 of the jobscript, I'm simulating doing some useful work by `sleep`ing for 30 seconds.

The normal workflow for executing a job is as follows:

1. `ssh` to the remote system;

2. copy the required program to the remote machine or copy its source code and compile on the remote machine;

3. open a command line editor like `nano` or `vi` and write a jobscript;

4. submit the jobscript to the resource manager;

5. wait for the job to start;

6. wait for the walltime to pass while repeatedly asking the resource manager about the status of the job (*polling*) in hopes of it finishing sooner;

7. copy the results back to your own system for further analysis.

Let's see if we can follow these steps and submit the job to the Torque resource manager inside `tremendous-torque`.

Check that `tremendous-torque` is still up and running and `ssh` into it. The container has just 1 command line editor program, called `vi`. Start it with `vi`. Write the jobscript (you may need to refer to the `vi` cheatsheet[a]). Save the jobscript as `jobscript` by first getting into 'command mode' by pressing Esc, then typing `:w` and pressing Enter. Pressing i while in command mode will get you back to normal mode ('insert mode'). When you're done, get into command mode again and type `:q` to quit the program.

After verifying that you now have a new file `jobscript` in `xenon`'s home directory, submit the jobscript to Torque as follows:

```
qsub jobscript
```

If all goes well, `qsub` should reply with the name of the job that was just submitted to the *queue* (most likely something like `0.localhost`).

Since you're the only user on this virtual remote system, the job will start almost immediately. You can ask the resource manager about the status of your jobs by:

```
# qstat -u <username>
qstat -u xenon
```

which will give you an overview of user `xenon`'s jobs.

At some point, the job's status will be listed as `C` for 'completed'. You should then also have two new files, `stderr.txt` and `stdout.txt`. The latter should contain the start and end times of your job, as written by the jobscript's `echo` statements.

2.18.2   Submitting jobs to Torque with Xenon

The example `SubmitSimpleBatchJob` from package `nl.esciencecenter.xenon.examples.jobs` does more or less the same as what you just did manually (although it still requires a few small changes to make it work).

Open `SubmitSimpleBatchJob` in Eclipse. As you can see, the beginning and

---

[a]`http://www.lagmonster.org/docs/vi.html`

the end are much the same as we had for `DirectoryListing`, but the middle is a bit different. In particular, we now retrieve the `Jobs` interface instead of the `Files` interface, and we do some stuff with it. The middle part also includes a `JobDescription`, that defines what useful work we want done (although here, we are using the not-so-useful `sleep` again to simulate doing useful work).

We then submit the `JobDescription` to the `scheduler` using `jobs.submitJob()`, and then wait until it's either done, or until the job times out. Here, `jobs.waitUntilDone()` takes care of the polling for us. After `waitUntilDone` returns, we check its `status` to see if the job was successful or timed out.

For the SSH adaptor, we used `ssh://xenon@localhost:2222/home/xenon` to list the directory contents of `xenon`'s home (see p. 27). For submitting jobs, we simply replace the scheme value `ssh` by `torque`, and remove the now obsolete path value (`/home/xenon`), so the Program arguments in our run/debug configuration should now become `torque://xenon@lisa.surfsara.nl:2222`.

However, if you try to run the program as is, you'll get the error below:

```
time   : 16:54:56.932 (+837 ms)
thread : main
level  : ERROR
class  : nl.esciencecenter.xenon.examples.jobs.SubmitSimpleBatchJob:97
message: SubmitSimpleBatchJob example failed: ssh adaptor: Auth cancel
```

This is because you still need to gain access to the Docker container named `tremendous-torque`. While you could set up passwordless login again, let's try something different for a change: Xenon Credentials.

Make a copy of `SubmitSimpleBatchJob` and call it `SubmitSimpleBatchJobWithPwd` or something. Adapt the script to let it use `Credentials` like you did before (see page 27). You should now be able to run `SubmitSimpleBatchJobWithPwd` and have it return something like:

```
time   : 17:00:16.571 (+310 ms)
thread : main
level  : INFO
class  : nl.esciencecenter.xenon.examples.jobs.SubmitSimpleBatchJobWithPwd:53
message: Starting the SubmitSimpleBatchJobWithPwd example.

time   : 17:00:48.479 (+32218 ms)
thread : main
level  : INFO
class  : nl.esciencecenter.xenon.examples.jobs.SubmitSimpleBatchJobWithPwd:103
message: Job ran succesfully!

time   : 17:00:48.485 (+32224 ms)
thread : main
level  : INFO
class  : nl.esciencecenter.xenon.examples.jobs.SubmitSimpleBatchJobWithPwd:112
message: SubmitSimpleBatchJobWithPwd completed.
```

This concludes the tutorial. For further exploration, the JavaDoc is an excellent starting point. You can find it online at `http://nlesc.github.io/Xenon/versions/1.1.0/javadoc/`. You can also generate a local copy by running:

```
# (in Xenon-examples)
./gradlew javadoc
```

After this, open your browser and navigate to the local file `build/doc/javadoc/index.html` to inspect the JavaDoc.

Index