Installation Manual

OMiLAB Technical Documentation

Version: July 12, 2016

Contents

Revision History

Revision	Date	Author(s)	Description
1.0	19.04.16	D Goetzinger	Created Document

1 Details

Prerequisites

- Basic UNIX shell skills
- Understanding of POSIX systems
- Basic MySQL administration skills

Recommendations

- Basic maven experience
- Experience with Java
- Experience with Tomcat application server

Structure of this document

The first part of this document discusses the building blocks of the OMiLAB Portal and illustrates which dependencies exist between the services. The next part will take a closer look on the hardware and software requirements and discusses fundamental settings for the operating system and server applications. After the general steps to setup the system have been discussed, the document outlines how the installation using pre-build packages looks like, as well as how the software can be built from source.

2 Introduction

The OMiLAB Portal is realised using the microservice[1] architectural pattern. The main idea behind is that the functionality of the system is composed by different (web-)services, running in their own processes and communicating with each other using standard network protocols. This approach brings the advantage of absolute technology independence. The only requirement for services is that they can communicate in HTTP using standardized JSON messages (OMiLAB Generic Interface).

This guide will show how the core components of the OMiLAB Portal can be built from source and deployed, as well as discuss some details of the overall portal architecture. More information on the OMiLAB Generic Interfaces and on how to develop a new service can be found in the *"Service Development Tutorial"*. Additionally this guide will show how an installation from the OMiLAB software repository looks like.

[1] http://martinfowler.com/articles/microservices.html

3 Motivation

A common problem when adding new functionality to application systems is how the features can be fitted in the existing architecture without causing unwanted side effects, breaking things or at least causing the overall software quality to decrease. This is often an issue when it comes to customization of standard software and causes generic data models to prevail.

The presented microservice architecture should address this issues. Most of the services focus on implementing an according data model of their respective domain and constructing a user interface for it. This moves the focus from tasks like dealing with the base platform selection, code insertion and treating side-effects to finding the relevant domain concepts and the transformation into an adequate database schema. The complexity from software-level is moved to infrastructure-level.

4 OMiLAB Portal Architecture

The architecture of the OMiLAB Portal rests on two main building blocks: On one hand the services that provide the actual functionality and on the other the aggregator, who is responsible for composing the OMiLAB Portal of the services.

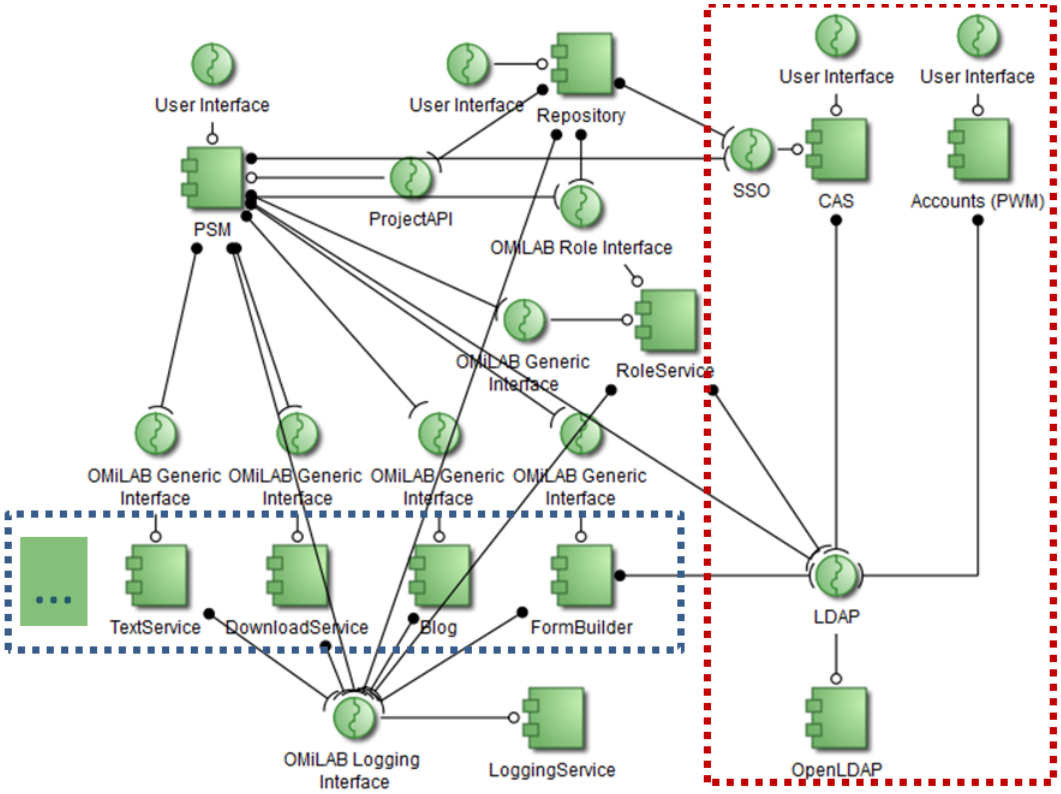

Figure 4.1: Dependencies in the OMiLAB Portal architecture

Figure 4.1 shows an exemplary deployment of the OMiLAB Portal. There are services for most atomic functionalities in the system, like a TextService to store static text or a RoleService to store role assignments. The Project and Service Manager (PSM) in this figure is the application that acts as aggregator. Important is, that there are several dependencies between the services. For example the TextService depends on the LoggingService, like most other services do. In this case the provider of the according service has to make sure that all dependencies are resolved and that his service is fully functional. So the provider of the TextService is responsible for the *configuration at service level*. He has to make sure that the path to the LoggingService is set correctly

and the TextService is working properly. In the Java-based ServiceTemplate this kind of configuration is mainly performed at the *application.properties* files.

The relation between the services and the aggregator, in this case called PSM, is different than between just services. The PSM is usually operated by a franchise administrator and is used to consume services (or functionality). For the *configuration at aggregator* level it has a web interface that allows the management of available services.

As obvious in figure 4.1 the system is designed in a very modular way, what allows it to deploy only the necessary services for a certain environment. The services, presented in the blue box can be added or removed at will, in order to cover the given requirements. The components in the red box are centrally available, in order to facilitate the vision of one OMiLAB account, that can be used for all OMiLABs. The services, that belong to no box, are essential for running the OMiLAB Portal. They are used to provide basic portal functionality, like logging of metadata, management of permissions, management of binary files and the common arrangement of all of these components. Under certain circumstances some of them may also be omitted, but this is not the recommended deployment.

One advantage of the microservice architectural pattern is the encapsulation of all services and the abstraction to a higher-level interface, based on standardized protocols. As of this it is in the responsibility of the according service to choose the right technology stack, especially storage technology (SQL, NoSQL, XML, ...). Furthermore each service is obligated to fetch all the data it needs on its own (Pull, not Push). Also if dependencies fail, the service has to take care of mitigating an outage, as far as possible. Generally data should not be duplicated within the system, as this would lead to anomalies and consistency problems, when it comes to updating or deleting data. If access to data from another service is required, according IDs to identifiy the data set in the remote service should be saved rather than duplicating data at a different service. An asynchronous communication method is not present at the current stage.

Whenever an existing services does not cover the requirements, it is feasible to implement a new one, which realizes a suitable data model and user interface for the domain required, or takes care of wrapping external services, i.e. ADOxx webservices. Also the programming techniques and persistence technologies can be freely chosen for each service. There is no need to worry about the portal management features, as all interaction with the OMiLAB is encapsulated by means of the OMiLAB Interfaces. Further information on how a new service can be developed, can be found in the *"Service Development Tutorial"*.

The following services are depicted in the exemplary deployment in figure 4.1:

- **PSM:** The aggregator is responsible for the coordination of the services and for creating an uniform representation to the user. It supports service management features for various user roles. It accepts all client requests at a central point, transforms them and dispatches them to the according service.
- **LoggingService:** The LoggingService can only be reached by means of the special OMiLAB Logging Interface. It is responsible for storing metadata from the other services, like who performed which action at what specific time.

- **TextService:** The TextService stores static pages, which may be further partitioned using the configuration at database level.
- **Blog:** The BlogService implements some basic blogging functionality, to spread news about a specific project.
- **DownloadService:** The DownloadService provides a way to present the deployable modelling tools to the enduser, no matter, where they are stored.
- **FormBuilder:** The FormBuilder provides the administrator with a clear GUI to create dynamic forms for the end user, e.g. to support the registration process of an event.
- **RoleService:** The RoleService is responsible for storing role assignments within the context of a project. It is tightly integrated with the PSM, who is responsible for enforcing some of the pre-defined role definitions.
- **CAS and LDAP:** The LDAP is the database where all user information including the profile pictures is stored. The CAS is a Single Sign On solution which access the user data and acts as main login point for a variety of other application. The CAS proofs the identity of the user to the PSM and Repository.
- **Accounts (PWM):** The PWM is responsible for providing a user interface, that gives the end user the means to edit their profile details, e.g. affiliation, photo, etc..
- **Repository:** The Repository acts as a central storage for binary files, like images or executable and can be used by other services, either via its special JSON interface or via JavaScript.

5 Preparation

This chapter should outline, what the software and hardware requirements for the OMiLAB Portal are, and which steps are required to configure the environment for the OMiLAB Portal.

5.1 System configuration

This chapter will give some insight on which software and hardware is supported and how these have to be configured to make the portal work.

Hardware sizing

The minimum hardware requirements for the portal are:

- Server-class computer
- Dual-Core processor
- 2048 MB RAM minimum, depending on service number and simultaneous users (3072 MB or more are recommended)
- 5 GB free disk space, depending on number services and space requirements for pictures, modelling toolkits (more recommended)
- 1 Network card and a static and world-wide routed IP address.

The portal should work pretty much work on any operating system that supports Tomcat, Java and any kind of SQL server. Although the system is only well-tested in the following environment:

- Ubuntu 14.04 LTS
- JRE 1.7
- Tomcat7
- MySQL 5.5

and

- Ubuntu 16.04 LTS
- JRE 1.8
- Tomcat8
- MySQL 5.7

JVM configuration

As each of the simple services is deployed as a separate application, there is a small memory overhead for each service. In order to avoid problems (*"PermGenSpace"*-Error) when it comes to the deployment, the Tomcat application server should be configured accordingly.

The nameserver option should be set to a near server with very good response times, as it might pose a quite strong bottleneck when it comes to accessing services that are on another server and are not reached via "localhost". It is recommended, to add an entry to the local *"hosts"* file[1] for services, that are heavily used, in order to ensure a good response time. Response time delays based on the DNS setting can be debugged by using the *"omilab.debug.performance"* option in the application.properties file of the PSM. Below a standard configuration realizing the changes discussed can be seen. These *"JAVA_OPTS"* may be placed at *"/etc/default/tomcat7"* (This path applies to the package in Ubuntu 14.04 LTS and might be different for other Linux distributions or Tomcat installations.)

```
1 JAVA_OPTS="-Djava.awt.headless=true -Xmx128m -XX:+
    UseConcMarkSweepGC -Xmx1g -XX:MaxPermSize=512m -Dsun.net.
    spi.nameservice.nameservers=131.130.1.11 -Djava.net.
    preferIPv4Stack=true"
```

For development machines and high load system with lots of RAM the maximum heap space of the JVM and the size for the permanent generation of the JVM should be increased, as below:

```
1 JAVA_OPTS="-Djava.awt.headless=true -Xmx128m -XX:+
    UseConcMarkSweepGC -Xmx2g -XX:MaxPermSize=1024m -Dsun.net
    .spi.nameservice.nameservers=131.130.1.11 -Djava.net.
    preferIPv4Stack=true"
```

[1] http://linux.die.net/man/5/hosts

6 Simplified Installation Process

The simplified installation is tested with Ubuntu 16.04 LTS and makes use of the integrated package manager.

Using the pre-packaged components makes it easy to get a standard installation up and running, as well as keeping it updated. A manual installation allows for greater flexibility and customization.

In order to use the package repositories for the OMiLAB Portal you will have to add the following line to the file *"/etc/apt/sources.list"*

```
deb http://jenkins.dke.univie.ac.at/software/ubuntu
```

Furthermore the public key of the repositories has to be imported, like the following:

```
$ wget -O - http://jenkins.dke.univie.ac.at/software/public.
  gpg.key|apt-key add -
```

In order to setup the system, please install the the components you want to have:

```
$ apt-get update
$ apt-get install psm repo loggingservice roleservice
  textservice
```

You will find a list of all packages that can be installed using this mechanism at http://jenkins.dke.univie.ac.at/software/Available.txt

Updates are announced via the OMiLAB Portal mailinglist. Please update the software in a timely manner using:

```
$ apt-get update
$ apt-get upgrade
```

During the simplified installation it is not possible to change the exact path and thus the applicationcontext, where the application is deployed to. This means that the default values for the service dependencies will be fine in most cases. Configuration changes should not be done directly at the *"application.properties"* file, but rather using the according tool, for example:

```
$ dpkg-reconfigure psm
```

7 Manual Installation Process

Using the manual installation it is possible to setup a very specific installation and use strongly customized software. It should only be used, when the simplified installation cannot be used. The manual installation and the installation from the source code brings a lot of flexibility, but this needs high maintenance efforts, as this process has to be repeated whenever there is an update. Figure 7.1 gives an overview of the steps needed to setup a manual installation.

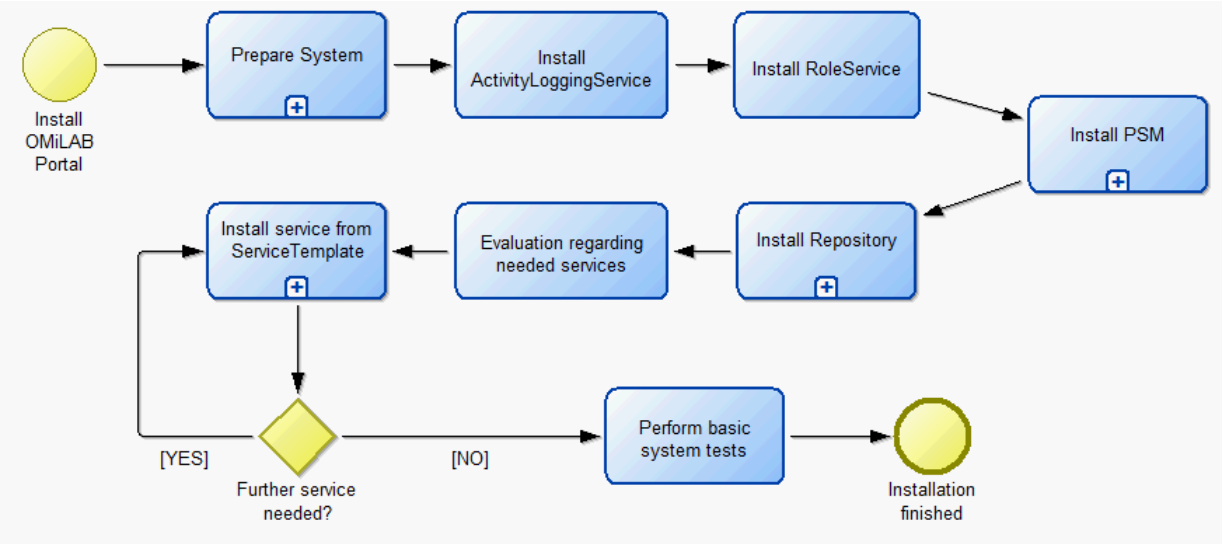

Figure 7.1: Manual installation process of the portal

The first steps consist in the preparation of the environment, already discussed in section 5.1. The steps are visualized in detail in figure **??**.

7.1 Database setup

Each of the services in section 3 requires its own MySQL database. This does not need to be the case at all times. There might be services that do not no require a database at all or use a different kind of database (NoSQL, XML, plain filesystem, ...). In this case a MySQL database for each one is required. The SQL file accompanying this document will take care of the creation of the databases, tables and populate everything with example data. In a production environment it is strongly suggested to create separate users for each database, whereas in a development environment using the same user for

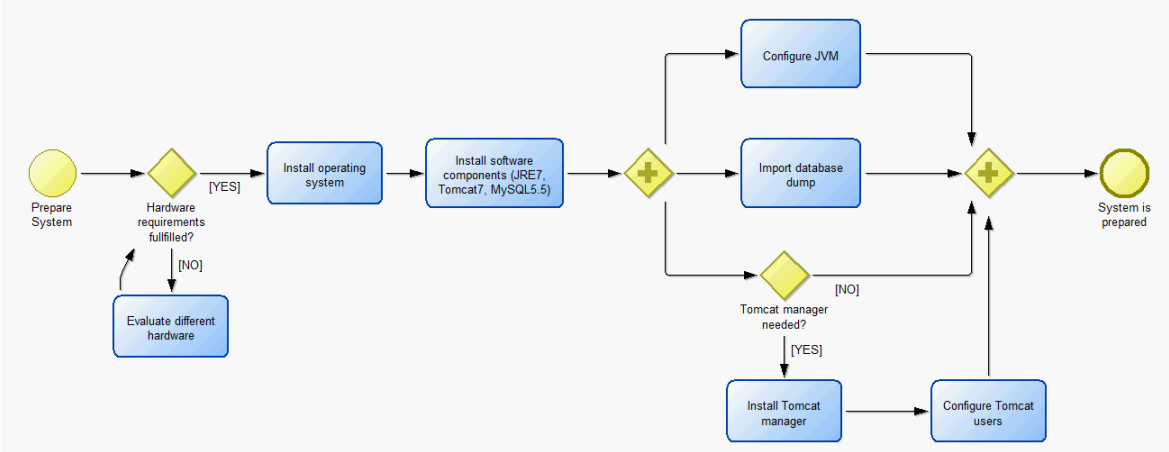

Figure 7.2: Preparation process for the installation

all databases might be tolerated. See the documentation of your database server on how to create database users and assign permissions to them[1]. In order to import the sample SQL file for all services into MySQL you may use the command below:

```
1 $ mysql -u root -p < db_init.sql
```

If you do not want to use the attached SQL file, you will find a standardized SQL file that takes care of the creation of the according tables and necessary data in the *"INSTALL"* folder of each service. See the documentation of your database server on how to create databases and import SQL scripts[2]. Sometimes it might be necessary to adjust the configuration of the according services, to match the database names of the example sql script.

7.2 Useful Paths

The following applies to the standard Tomcat installation on Ubuntu 14.04 LTS:

- **Webapps:** (deployed applications): */var/lib/tomcat7/webapps*
- **Logs:** */var/log/tomcat7/*
  - *catalina.out* is the main log for debugging and includes the stacktraces
  - Continuous viewing of the file is possible with

    ```
    1 $ tail -fn 300 /var/log/tomcat/catalina.out
    ```

  - or view the current content of the whole file without continuously updating to the newest entries with

[1] https://dev.mysql.com/doc/refman/5.5/en/user-account-management.html
[2] https://dev.mysql.com/doc/refman/5.5/en/mysqlimport.html

```
1 $ less /var/log/tomcat/catalina.out
```

- **Tomcat7 configuration:** */etc/tomcat7/*
- **Tomcat7 startup configuration:** */etc/default/tomcat7*

8 Manual Deployment Process

By now the base system, consisting in the operating system, the Tomcat application server and MySQL database should have been installed. This chapter will outline how the services, based on the ServiceTemplate can be build from source, configured and deployed at the application server.

In most cases (all, except the Repository) it is sufficient to download the latest binaries of the software from the build server[1]. In this cases, the section 8.1 can be skipped.

8.1 Build system

The build system in use by most services is Maven[2]. It takes care of managing the dependencies and basic setup of the build. The most important feature in use in almost every project is the *"Maven Profile"*. It is used to customize the build to a specific environment. This boils down to running certain minifier for CSS or JavaScript files and choosing the right configuration file. The configuration can either take place before the build or afterwards. When using Maven it will simply copy the appropriate properties file from *"src/main/resources/config"* to *"WEB-INF/classes/application.properties"* in the war artefact, where it is picked from the according service. Of course this can be done manually at any time.

When a new profile should be added, one should not forget, that not only the application.properties has to be added, but also the *"pom.xml"* has to be edited to add a new adapted profile there. When building from source the profile has to be specified with the *"-P"* option. The goal mostly used is *"package"* which will create a deployable war file that can be directly copied to the Tomcat webapps folder or may be deployed using the manager application. Alternatively, especially during development, it is quite useful to configure the *"tomcat7 maven plugin"*. The goal *"tomcat7:redeploy"* can then be used and it will take automatically care of deploying the application on the application server. In order to make this work, the manager application has to be configured accordingly. A user with the *"manager-script"* role has to be configured. Furthermore the credentials for this user have to be present in the *"settings.xml"* of the local Maven installation. In order to support the development a local GitLab installation is used. The services can be found at the following locations: The core services can be found at:
https://gitlab.dke.univie.ac.at/groups/omilab-core-infrastructure.
Other services that can be deployed and used in the application system can be found at:
https://gitlab.dke.univie.ac.at/groups/omilab-services

[1] http://jenkins.dke.univie.ac.at/release/
[2] https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html

8.2 Generic Deployment Process

The following build process will work for most services based on the Java ServiceTemplate, with the exceptions listed in the next chapter. The steps needed to get to the final installation consist in 1) cloning the repository 2) configuration 3) building and 4) deployment. In the following the process will be shown based on the example of the ActivityLoggingService and the generic profile *"distribution"*. One should note that variances of this process are possible. For example it is also possible to change the configuration after deployment at *"WEB-INF/classes/application.properties"*. This will be necessary, if the builds from the build server are used.

1. **Clone** the repository of the service that should be built

```
$ git clone git@gitlab.dke.univie.ac.at:omilab-core-
   infrastructure/ActivityLoggingService.git
```

   The respective URL of the repository can be obtained via its GitLab page.

2. **Configuration**

```
$ vi src/main/resources/config/application-distribution.
   properties
```

   The configuration file of the service may be edited at this step. A good documentation of the available options can always be found in the file *"application-distribution.properties"*. The files are usually separated in different sections and only the "Configuration" section should be edited. Common attributes like *"spring.datasource"* require the according credentials for the database and the attribute *"app.url"* requires the external reachable url of the application. Please pay attention to match the application context (= filename of the war file). Usually in the resources folder a folder called *"INSTALL"* can be found. It provides further installation instructions, database schema or other useful information.

3. **Build** the service

```
$ mvn clean -Pdistribution package
```

   This command cleans the target directory and then builds the war file for the profile *"distribution"*. It should be in the root of the cloned directory, where the *"pom.xml"* is located. If the Tomcat7 plugin has been configured in the *"pom.xml"*, package could also be replaced with *"tomcat7:redeploy"* or *"tomcat7:deploy"* and it will automatically deploy the resulting war to the application server, thus making the next step unnecessary.

4. **Deploy** the service

```
$ cp target/activity-0.2-SNAPSHOT-distribution.war \
> /var/lib/tomcat7/webapps/logging.war
```

Finally the resulting war file in the target folder has to be deployed to the Tomcat server. This can either be done by using the manager application of Tomcat[3] or by simply copying the resulting war file to the *"webapps"* directory, as shown in the command above. To make the database dump and most example config work the following contexts are assumed:

- "logging" for the ActivityLoggingService
- "textservice" for the TextService
- "role" for the RoleService
- "psm" for the ProjectStructureManager
- "repo' for the FileManager

Generally the context can be defined freely, although one has to pay attention that every configuration respects that, especially the service definitions at the PSM and the *"app.url"* in the service configuration (*"application.properties"*). Please see the *"Service Development Tutorial"* or the *"PSM Service Documentation"* on how custom service definition files can be created.

This process is valid for all services based on the generic ServiceTemplate, although further steps might be required, depending on the service. A summary of this process can be seen in figure 8.1.

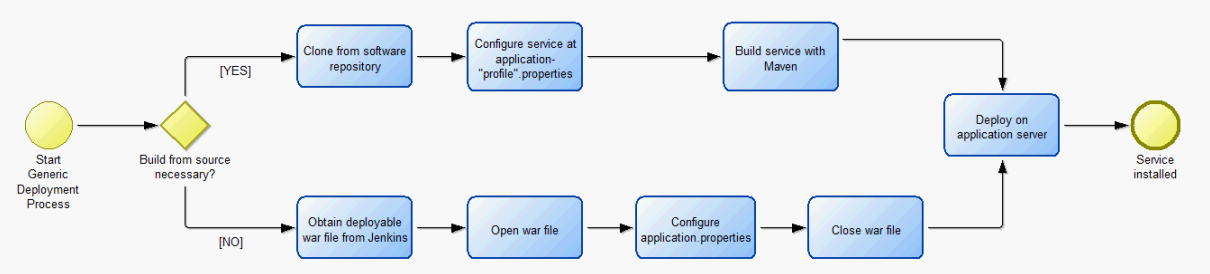

Figure 8.1: Generic Deployment Process

8.3 Exceptions of the Generic Deployment Process

As discussed, sometimes services do use external tools and frameworks, that require special configuration. In this case the steps from the Generic Deployment Process may not be enough. Further information on exceptions can be found at their respective service documentation.

[3]https://tomcat.apache.org/tomcat-7.0-doc/manager-howto.html

8.3.1 Repository

The Repository (FileManager) is not based on Spring Boot and the ServiceTemplate, but is rather an ordinary Java web application based on existing open source projects [4] [5].

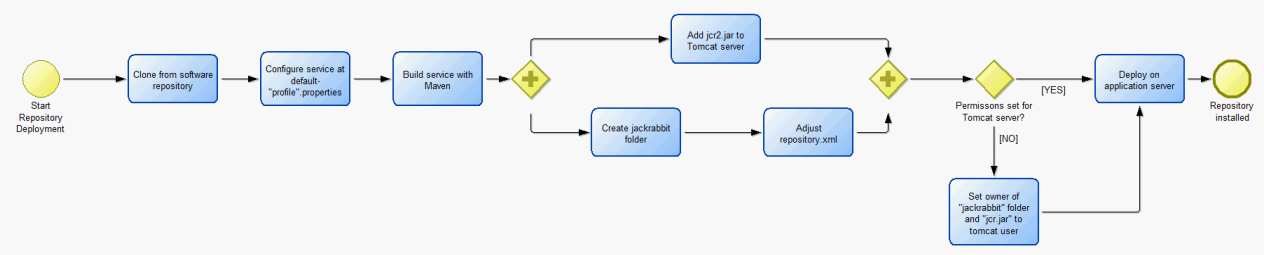

Figure 8.2: Deployment process for the Repository

This makes the installation somehow special. The configuration *has* to be done before the build of the project, as Maven will set several configuration options during the build. Furthermore after the git repository has been cloned, the Tomcat installation has to be adjusted. Within the *"INSTALL"* directory a file called *"jcr-2.0.jar"* can be found. This file has to be copied to the shared lib directory of the Tomcat installation. In the case of Ubuntu 14.04 it is located at *"jcr-2.0.jar"*. Additionally in the root of the Tomcat installation a directory *"jackrabbit"* has to be created (absolute path in Ubuntu 14.04 is *"/var/lib/tomcat7/jackrabbit"*). Within the directory *"INSTALL"* two example configuration files (repository.xml.X) can be found. The second one has to be copied to the previously created jackrabbit folder and renamed to *"repository.xml"*. The file has to be filled with database credentials. Please use the same database and credentials at both of the two settings. Also make sure that all of these files and folders have the correct permissions (*"jcr2.jar"* and the *"jackrabbit"* folder). Commands similar to the following might be used:

```
1 $ chown -R tomcat7:tomcat7 <file or folder>
2 $ chmod -R 755 <file or folder>
```

Furthermore the configuration file is not called application-profilename.properties in this case, but rather default-profilename.properties. The final steps are identical to the other applications.

The variation of the generic process is visualized in figure 8.2

8.3.2 PSM

The PSM makes use of the Apache Lucene[6] project. This requires to presence of a folder, called *"indexes"* in the root directory of the Tomcat installation. All further

[4] https://github.com/simogeo/Filemanager
[5] https://github.com/th-schwarz/C5Connector.Java
[6] https://lucene.apache.org/core/

steps, like the indexing are handled automatically. Nevertheless this folder has to be created manually with the correct permissions. Again, the variation of the generic process is visualized in figure 8.3

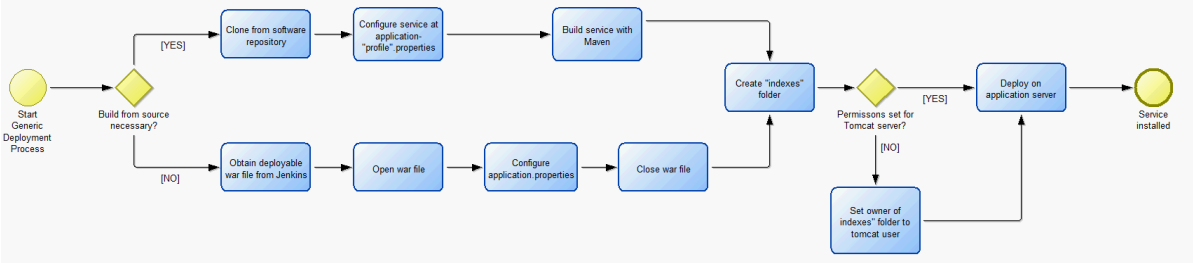

Figure 8.3: Deployment process for the PSM