University of Zurich
Department of Informatics

Michael Jehle
Kevin Leopold
Linard Moll
Anthony Lymer

Software Evolution Recognition and Visualization Information Service

TECHNICAL REPORT – No. IFI-2009.06

December 2009

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

M. Jehle
K. Leopold
L. Moll
A. Lymer:
Technical Report No. IFI-2009.06, December 2009

Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zurich, Switzerland
URL:

Master Project in Software Systems

September 26, 2009 - February 26, 2009

SerVIS

Software Evolution Recognition and Visualization Information Service

Michael Jehle
03-106-325

Kevin Leopold
04-721-676

Anthony Lymer
04-727-871

Linard Moll
00-916-932

Supervised by
Prof. Dr. Harald Gall
Giacomo Ghezzi
Michael Würsch

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

1 Introduction

The goal of our master project in software systems at the University of Zurich was to take those parts of the Evolizer platform for software analysis that were responsible for analyzing and fetching information about different "software development tools" (Bugzilla, CVS, FAMIX) and decouple them from their Eclipse-based environment. The next task was to set them up as standalone applications, change their underlying database used for storing the extracted data from Hibernate[1] to Sesame (an ontology-based database) and introduce an ontology to describe that data. Furthermore the importers should be created as web services and deployed in an appropriate web service environment that eases the usage of them by a service consumer or web user.

The four importers we dealt with were:

- *Bugzilla Importer:* it extracts data about bugs, solutions, activities, etc. out of a Bugzilla repository.
- *CVS Importer: it extracts* data about users, files, revisions, etc. out of a CVS repository.
- *FAMIX Importer:* it extracts the FAMIX model out of a given Java program
- *SVN Importer:* it extracts data about users, files, revisions, etc. out of an SVN repository. The SVN Importer was not yet part of Evolizer and therefore it was built from scratch.

Since the FAMIX Importer heavily relies on the powerful Eclipse JDT Java compiler. The development of the FAMIX importer was at the same time a proof-of-concept for further development of the FAMIX Importer.

The importers, which used a Hibernate database, were coupled to a Sesame triple store. This allowed us to store the parsed data as triples associated to predefined specific ontologies.

After this prearrangement, we started with our main task, the implementation of those as web services. The motivation behind that is to offer the analysis facilities of the importers to other clients over a simple internet connection.

In addition to the four importer services several management services were created.

- *AAA Service:* handles security issues like getting an account, checking whether a user is allowed to use any of our services or validating the authentication keys.
- *JobStatus Service:* this service allows starting, cancelling or rerunning jobs. Additionally it knows the status of all the jobs.
- *StatusCollector Service:* this service supports the business process and the web GUI through reporting the status of the running importer services.

As a last step, a JSP based web GUI was created to provide user access to our services.

2 Architectural Overview

As shown in Figure 1, one central part of the SerVIS architecture are the four importer services wrapping the Bugzilla, CVS, FAMIX and SVN importers (see section 2.5), the management services managing the import processes and the BPEL loaders acting as connectors to the business processes. Those three service types are described in section 2.6. The logger façade which was used by the importers and the web services is explained in section 2.4. The databases used in the SerVIS architecture are on one hand a MySQL[2] database for the management services to keep track of import states and user management and on the other hand Sesame (see section 2.3) for the importers to store their results. Both components (web services and databases) are running on a Tomcat6 server (see section 2.2) using OpenJDK 7, which is required for the FAMIX Importer to run.

[1] Hibernate 3.3.1 GA was used from http://www.hibernate.org/

[2] MySQL 5.0.45 was used for this project.

The other central part of the SerVIS architecture, the business processes (in BPEL), is described in more detail in section 2.7. All BPEL processes are specified using JBI and are running on a Glassfish 2 server using Java EE 6 (see section 2.2). The two application servers Tomcat and Glassfish are set up on a CentOS Linux machine. The set up of that machine can be found in the appendix in section 5.4.
A possible way to start an import is a web GUI, explained in section 2.8. Finally, an exemplary interaction between web GUI, the web services and the business processes is presented in section 2.9.

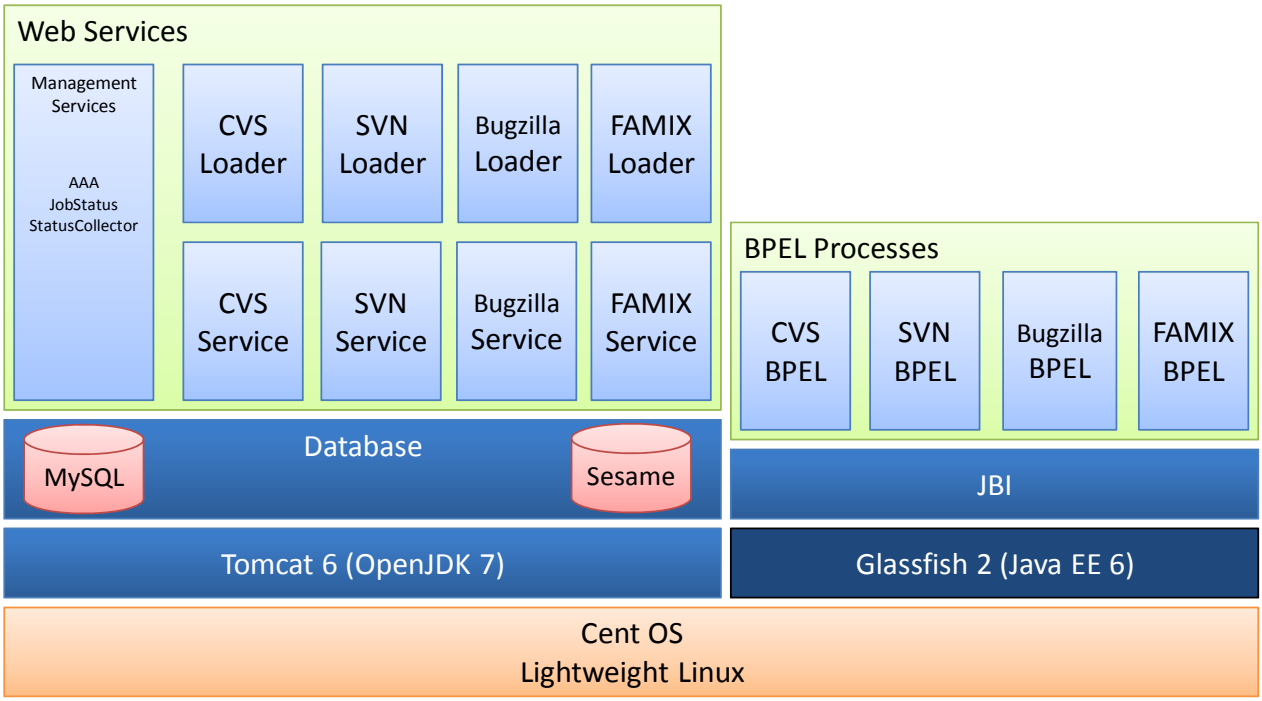


Figure 1: The architectural view.

2.1 Eclipse / NetBeans

Evolizer is a platform that depends strongly on Eclipse[3]. Since the Bugzilla, CVS and FAMIX importers were all Eclipse plug-ins, they couldn't be used independently. To avoid this coupling and to deploy the plug-ins as services, they had to be taken out of the environment. To do this, Eclipse was used for easier understanding of the existing importers (in other IDEs the code wouldn't have compiled). Since the support for web services in eclipse didn't fit our needs, we moved to NetBeans[4] which is more sophisticated in terms of web services and web applications. As it has a big repertoire of tools which support the development of web service based applications. JAX-WS - a large library/Java API for web services which provides the whole package of web service functionality - is natively supported in NetBeans. Thus it should, for example, be easy to extend the present design with WS-Security components to ensure cryptographic communication.

2.2 Tomcat / Glassfish

The SerVIS importer services (see section 2.6.1) and management services (see section 2.7) are built on a Tomcat[5] application server running on OpenJDK[6]. Tomcat was chosen because of the several errors we encountered while trying to get the Sesame DB running under Glassfish[7]. The OpenJDK is used by the

[3] Eclipse 3.4.1 was used from http://www.eclipse.org/downloads/ .
[4] NetBeans 6.1 (ESB edition) and 6.5/7.0M1 were used from http://www.netbeans.org/downloads/ .
[5] Tomcat 6.0.18 was used from http://tomcat.apache.org/download-60.cgi .
[6] OpenJDK 1.7.0-ea-b41 was used from http://download.java.net/openjdk/jdk7/ .
[7] Glassfish ESB was used from https://open-esb.dev.java.net/Glassfish ESB/ .

FAMIX Importer (see section 2.5.3). To get the JBI composite application running the Glassfish application server is used. This decision was influenced by the distinctive integration of WS-* and BPEL functionality into NetBeans development environment as you read in section 2.1. In addition, NetBeans allows debugging a running BPEL process step by step which is a great benefit in such a distributed environment. Since both servers run on the same physical host, the standard port configuration was modified. To access the web administration tools, an Apache reverse proxy setup was made for http://servis.028.ch (Tomcat) and http://servis-admin.028.ch (Glassfish). In the appendix (see section 5.4) the configuration for both application servers as well as the apache reverse proxy setup is described. To improve the building and deployment process a new shared library folder for Tomcat was created containing all libraries used by the services. The resulting overall war file sizes were around 90% smaller, the deployment time changed from minutes to seconds. Again, see section 5.4 for the configuration details.

2.3 Sesame DB

Since the imported data should be stored as subject, predicate and object, a triple store, namely Sesame[8], is used.
Sesame is a database, which allows multiple ways of storage. For our purposes a "Native Store" is used, because it stores the data persistently and performs better than the MySQL approach. To bridge the gap between the object-oriented paradigm and the triple paradigm, a framework called Elmo[9] is used. Like Hibernate for relational data, Elmo maps JavaBeans to triples. With this approach existing code from Evolizer could be reused and thus the adaptation from a relational database to a triple store could be done in a reasonable amount of time.
The specific JavaBeans used by Elmo are located in each importer and for SVN and CVS in org.evolizer.servis.versioning.model.entities, since they share the same data model.
Using the HTTP-Repository directly to save data to Sesame didn't perform well. To fix the performance issue a memory repository and an http repository is used. The model is first created in the memory repository and then exported to the http repository. The export is done step by step (10'000 triples each step) to minimize memory problems.
The Java class (PersistenceManager) which is used to interact with Sesame is placed in org.evolizer.servis.persistence.

2.4 Logger

In an enterprise application logging is very important to understand whether something unexpected (or expected) happened. For the SerVIS project a logger façade which wraps the log4j[10] logger was written. The logger façade can be found in the project org.evolizer.servis.logger. Because the logger uses log4j a log4j.properties file has to be provided.
Since all projects use the logger, the org.evolizer.servis.logger.jar has to be placed in the shared-lib folder of Tomcat along with the log4j library (see chapter 5). If these two files are in the shared library folder, log4j searches for its log4j.properties file in the root folder of Tomcat. The built war files of the different projects neither include the org.evolizer.servis.logger nor log4j, thus a ClassNotFoundException is thrown if log4j and org.evolizer.servis.logger.jar isn't placed in the shared library.

2.5 Importers

In the following sections the different importers are introduced.

[8] Sesame 2.2.3 was used from http://www.openrdf.org/download.jsp.
[9] Elmo 1.3 was used from http://www.openrdf.org/download.jsp.
[10] Additional info's about Log4j can be found at http://logging.apache.org/log4j/1.2/index.html .

2.5.1 Bugzilla

Goal. The Bugzilla Importer extracts data from a Bugzilla repository and stores it in a well-structured ontology based form. This way of saving the Bugzilla data enables semantic analysis of the bug information.
Interface. The Bugzilla Importer presents a handy interface to use its functionality as a standalone application (org.evolizer.servis.bugzilla.service.BugzillaImporterService.java). To create a new BugzillaImporterService only a repository id (string) is necessary, this string will be used as the name of the Sesame store, where the parsed bug data will be stored.
The BugzillaImporter offers the following methods:

- parseSingleBug
- parseAllBugs
- parseARangeOfBugs
- parseAListOfBugs

Workflow. The following part will shortly describe the workflow of a Bugzilla import.

1. One of the import methods of the BugzillaImporterService is called.
2. A new org.evolizer.servis.bugzilla.parser.persistence.BugzillaModelSaver is created. This class uses the PersistenceManager (see section 2.3) to store the parsed data.
3. The org.evolizer.servis.bugzilla.parser.BugParser is triggered with the corresponding settings.
4. As soon as the BugParser finishes its import, we get its parsed data and store it in the org.evolizer.servis.bugzilla.parser.ModelContainer.
5. If the user chose also to parse the activities of the bugs, they are parsed.
6. Eventually the content of the ModelContainer is saved to the database using the BugzillaModelSaver, which uses the functionalities of the PersistenceManager.

For all the different Bugzilla Importer parameters see the SerVIS wiki page “BugzillaImporter.pdf”.

2.5.2 CVS

Goal. The CVS Importer imports the source code and history of a given CVS repository and stores it to Sesame using the Java classes defined in org.evolizer.servis.versioning.model.entities and the source code is checked out to a directory set in a properties file.

Settings. The user of the CVS Importer has the possibility to set whether the source code should get downloaded and which file types he's interested in. However the final decision of checking out is made by the administrator using the properties file.
So if the user decides in his user interface (see section 2.8) that he wants the source code to be checked out, but in the properties file the checkoutSourceCode value is set to false, the source code won't be checked out.
The same pattern is used with the file types. If the user chooses ".java" as the file types he wants to check out, but in the properties file you only allow to import ".c", then the user setting is ignored. The user setting only will have effects if the file types value in the properties file is empty. If the user and the administrator (editing the properties file) don't specify a value for file types, everything will be checked out.

Connection types. At the moment only pserver connections are available to connect to a CVS repository. Unfortunately javacvs (the NetBeans CVS library[11]) doesn't support SSH connection natively. An external shell program has to be configured to enable SSH connections.

Workflow. The following part will shortly describe the workflow of a import.

1. Connect to the given CVS repository (with optional username and password)

[11] Further information about javacvs: http://javacvs.netbeans.org/library

   a. If the connection to the given repository cannot be created, the importer fails throwing a respective Exception
2. Check out the source code if the user and the admin have set the respective flag to true.
3. Create the internal representation of the given CVS repository using the log (Rlog command)
4. Save the created representation to the database

For all the different CVS Importer parameters see the SerVIS wiki page “CVSImporter.pdf”.

2.5.3 FAMIX

Goal. The FAMIX Importer creates a FAMIX model[12] out of object-oriented code files.

Task. The existing FAMIX Importer is based on the broad and reliable Eclipse JDT[13] abstract syntax tree (AST) processing functionalities. After several unsuccessful migration attempts a proof-of-concept reimplementation using the OpenJDK[14] code parsing capabilities was made. Therefore the annotation processing phase of the Sun Java compiler is slightly modified to gain access to the internal code model or ASTs. The OpenJDK compiler uses the com.sun.tools.javac.main.JavaCompiler which is still proprietary Sun Java code at the time this work was written. For the FAMIX Importer a new class org.evolizer.servis.famix.Compiler was created sub classing the Sun Java Compiler to change the way the compiler cleans up the memory after finishing its work. This step was required to gain access to the internal compiler maps, holding all the elements and source codes[15]. In fact a com.sun.tools.javac.util.Context object is holding all important objects like the file manager, the compiler itself and all the maps the compiler created as a single point of access.
First, a model crawler of type AbstractElementVisitor6 processing javax.lang.model.element.Element classes creates the basic FAMIX entities like file, package, class and method entities by visiting all existing code elements the annotation step provides. This visitor works like a regular annotation processor in the precompilation phase of the OpenJDK Java compiler.
To reach all code elements like inner classes, object variables or type casts a new way of accessing the compilers internal state was needed[16]. A second model crawler of type com.sun.source.util.TreeScanner is launched after completion of the attribution phase[17] (Attr) in the annotation-processing phase. This step actually violates JRE 296[18]. This second crawler accesses all elements of type com.sun.source.tree.*, which includes any possible programming construct in a Java source file (e.g. ForLoopTree, IfTree, TypeCastTree).

The FAMIX Importer takes Java source files as input. If a zip file is provided, the zip will be extracted and all Java source files will be added to an internal file list for processing. At the end, the generated FAMIX model is saved in the Sesame DB.

For all the different FAMIX Importer parameters see the SerVIS wiki page “FAMIXImporter.pdf”.

2.5.4 SVN

Goal. The aim of the SVN Importer is to analyze and import a given SVN repository to the internal database using a well-defined ontology. Since the SVN Importer was not yet part of the Evolizer system, it was completely built from scratch.

---

[12] FAMIX meta model http://moose.unibe.ch/docs/famix
[13] Further information about Eclipse JDT: http://www.eclipse.org/jdt/
[14] OpenJDK Compiler Group: http://openjdk.java.net/groups/compiler/
[15] Since this FAMIX importer was created while debugging the compilation process, this setup was very helpful.
[16] Overview on how the OpenJDK compiler internals works: Hacking the OpenJDK compiler; http://www.ahristov.com/tutorial/java-compiler/duplicating-compiler.html.
[17] OpenJDK Compilation Overview, Attr and other phases: http://openjdk.java.net/groups/compiler/doc/compilation-overview/index.html
[18] Please read the comments in the java source files Compiler.java and StructureProcess.java of the package org.evolizer.servis.famix.

Workflow summary. The goal is achieved by executing the following workflow:

1. *Connect to the given SVN repository (with optional username and password)*
   If the connection to the given repository cannot be created, the importer fails throwing a respective Exception
2. *Create the internal representation of the given SVN repository*
   2.1. Loop through all SVN revisions from the repository
   2.2. For each revision: check whether it created a release or not
      2.2.1. If it was a release it creates an internal representation for a release
      2.2.2. Otherwise it creates an internal representation for the author, the change set or the revision
         2.2.2.1. For each changed file / folder in this revision: create an internal representation (entity) for the file / folder, file version, modification report (if desired with calculation of added and deleted source code lines)
3. *Save the created representation to the database*

Like the CVS Importer, the SVN Importer uses the general versioning entities represented by the Java classes from org.evolizer.servis.versioning.model.entities to store the created representation into the ontology-based database.

Settings. The SVN Importer allows the user to specify whether the importer is supposed to calculate the number of changed (meaning added and deleted) lines in source code. Since this calculation significantly increases the import time, the user has the option to turn that feature off.
Additionally, the user may specify whether the importer is supposed to import the whole repository, meaning gathering information about every file or folder that resides anywhere in the repository, even if only a specific subfolder is given to the importer. This feature comes handy, when a user does not know the root of an SVN repository. Generally analysis of the whole repository is encouraged, since otherwise possible connections from the analyzed subfolder to other source units (files or folders) in different locations than that given subfolder are lost.

For all the different SVN Importer parameters see the SerVIS wiki page “SVNImporter.pdf”.

2.6 Web Services

All web services described in this section either extend from the class AbstractService or AbstractImporterService. In the following, those two abstract classes will be described in more detail.

AbstractService. The AbstractService class defines the methods every service has to provide. The following methods are available:

- *isAlive():* Used as ping to find out if the service still reacts. This method is for example used by the business process.
- *getServiceId():* Defines an id of the service. This is always the fully qualified class name. This method is for example used by the business process and loaders.
- *verifyConfiguration(String[] configuration):* Checks whether the configuration map (see below) has all the mandatory key/value-pairs. Throws a java.lang.IllegalArgumentException in case of missing values. Using this method a configuration can be verified, before it is executed.
- *execute(String[] configuration):* Starts the wrapped functionality (e.g. other web service or Java class).

The class also defines some constants which are used to get values out of the configuration map:

- CONFIG_KEY_LINE_SEPARATOR = "=";
- CONFIG_KEY_JOBID = "ISerVISWebService.jobId";
- CONFIG_KEY_AUTHKEY = "ISerVISWebService.authKey";
- CONFIG_KEY_PROCESSID = "ISerVISWebService.processId";
    - Value for this key is the fully qualified class name of the loader which is executed.
    - This information is used by the JobStatusService to invoke the service.

AbstractImporterService. Extends from AbstractService and adds functionality which the importers can use. Thus every importer extends this abstract class.
The AbstractImporterService manages the lifecycle of (Runnable) import sessions. It reports all necessary information like the start, failure and end of import session to the StatusCollectorService. The state of the job itself isn't changed; the BPEL process is in charge of setting the status of the job.

To be able to let the importers run, some kind of configuration has to be provided by the user (e.g. the URL of an SVN repository). This configuration is being held in a configuration map to make it easily exchangeable between multiple services:

Configuration Map. Every importer service takes a map, which defines the configuration for the given service. Because a web service interface which takes a java.util.Map can lead to serialization problems a String array is used holding "key=value" elements. Using this configuration map the syntactical coupling between the service and the service consumer is highly reduced. The service could easily add a new option, without the need to change its interface. Of course semantical coupling exists, because a client that wants to use the new option provided by the service has to add the option to the configuration map which it then passes to the service. This approach is still a lot more flexible than just adding a new parameter to the web service method signature.

2.6.1 Importer Services and Loaders

For every importer there are two specific web services available: Importer services and importer loaders.

Importer Services. For each one of the importers an importer service was defined. As mentioned before, all these importer service classes extend from the AbstractImporterService class, as the name suggests. They generally implement the verification of a given configuration for their importers, set up the importer environment (like configuration parameters etc.), create the importer itself and use the AbstractImporterService to run the import.

Importer Loaders. Additionally, for each importer service, an importer loader service was created (see Figure 1). Those loaders act as a mediator between the JobStatus service (see section 2.6.2) and the Business Processes (see section 2.7). When executed by the JobStatus service, the loaders set the correct input variables and use them to start the business process. Furthermore, the loaders are responsible for calling the right method to verify the importer configuration. The loader was introduced to have an indirection between the JobStatus and the BPEL process, such that these two components could be developed independently.

2.6.2 Management Services

AAA Service. Since most of the web methods use an authorization key for checking user access, an authentication and authorization service is needed. The AAA service provides some basic authentication and authorization features to all other web services like a sign-up, log-in or is-valid method. The project where the web service resides provides a standalone jsp page for user sign-up, log-in and e-mail verification. All the data is stored by Hibernate in a MySQL database. Please refer to the AAA web service schema in the appendix (chapter 5) for further details about the AAA web service and MySQL setup.

JobStatus Service. The JobStatus service is used to add and manipulate an entire job. When a job is added it verifies the configuration and stores the job alongside its configuration to the MySQL database using Hibernate.
After adding a new job it is checked whether it can get started using a job scheduler. To avoid millions of jobs working in parallel the maximum of running jobs can be set in a properties file. The file is located in org.evolizer.servis.web.JobStatusService/WEB-INF/classes and is called jobstatus.properties.
When a job, according to the job scheduler, can be started the execute(String[] configuration) method (see section 2.6, AbstractService) is invoked. This is done via reflection. Using reflection the source code doesn't have to be changed when a new business process is introduced. Still the web service references

have to be present in the org.evolizer.servis.web.JobStatusService project. The JobStatus service uses the value given in the configuration map stored in CONFIG_KEY_PROCESSID (using the fully qualified class name it can call the service) to invoke the loader which in turn starts the respective process.

StatusCollector Service. The StatusCollector service maintains the status of all importer sessions running "inside" a job. It provides functionality to insert status information about an import session (e.g. set SVN import status to successfully finished) and to read inserted status. Furthermore, it offers the functionality to check whether a given importer session is finished, still at work or failed. This information is for example required by the business process to determine whether it can start a new service or an entire job has finished.
The status information is stored in a MySQL database using Hibernate.

2.7 Business Processes

Since all SerVIS importers and management services are deployed and accessible as web services, business processes were built to orchestrate the authentication, fault and job handling as well as message collecting. To get the right choreography a simple BPEL process invokes all the sub-processes in the expected order. All the different importers are managed by a single importer BPEL process which figures out which importer to run based on the provided variables. For running the BPEL processes a JBI composite application was built. Access to this application is given by an additional SOAP port. To ensure a well-defined configuration setup a loader web service was added. Its verifyConfiguration web method calls all the needed web services to check the provided configuration array. These processes and the JBI application were built using NetBeans 6.1. The process loader web service was built in NetBeans using Tomcat as application server. Please refer to the BPEL and CASA schemata in the appendix (chapter 5) for further information.

2.8 Web GUI

There are three main JSP pages, which provide access to the different web services.

Log-in. The log-in is handled by the index.jsp page. It accesses the AAA Service and checks whether the given user data is valid or not. If e-mail and password are correct, the authentication key of the user is stored in the JSP session and the user is forwarded to the job overview site.

Job overview. The job overview site provides an overview over all the jobs of the user that is logged in. The user can see the status of each job and can access detailed information about every action which was triggered considering this job. If a job is successfully finished, the result can be downloaded on this page. Additionally a job can be deleted or rerun (if e.g. the repository data has changed).

Create new job. This site offers the possibility to start new jobs. The user can select which importer he wants to use and may then enter the input parameters. If the input configuration was not valid (e.g. missing input parameter), the user is given the possibility to reenter a correct value. After starting a job, the job overview site is displayed again. The user can now track the progress of his job.

2.9 Service Interaction

Figure 2 shows an exemplary execution of two imports. First a user enters the importers configuration in the web GUI and submits that to the JobStatus service. The latter creates a new job, adds a respective entry in the job database with status WAITING and starts the business process by using the respective loader. As soon as the business process receives a new job, it sets it to status RUNNING, calls each execute method of the composed services and waits for all importer services to finish. It does so by polling: it periodically calls the isFinished method of the StatusCollector for every importer started and stops waiting as soon as that method returns true for all importers. An importer service that was started may create an arbitrary number of WORKING status with a message describing what it currently does in

the status database, to report the current status of the import. As soon as an import is finished (successfully or erroneously), a respective entry in the status database is made. From that moment on, the isFinished method, which is periodically called by the business process will return true and the

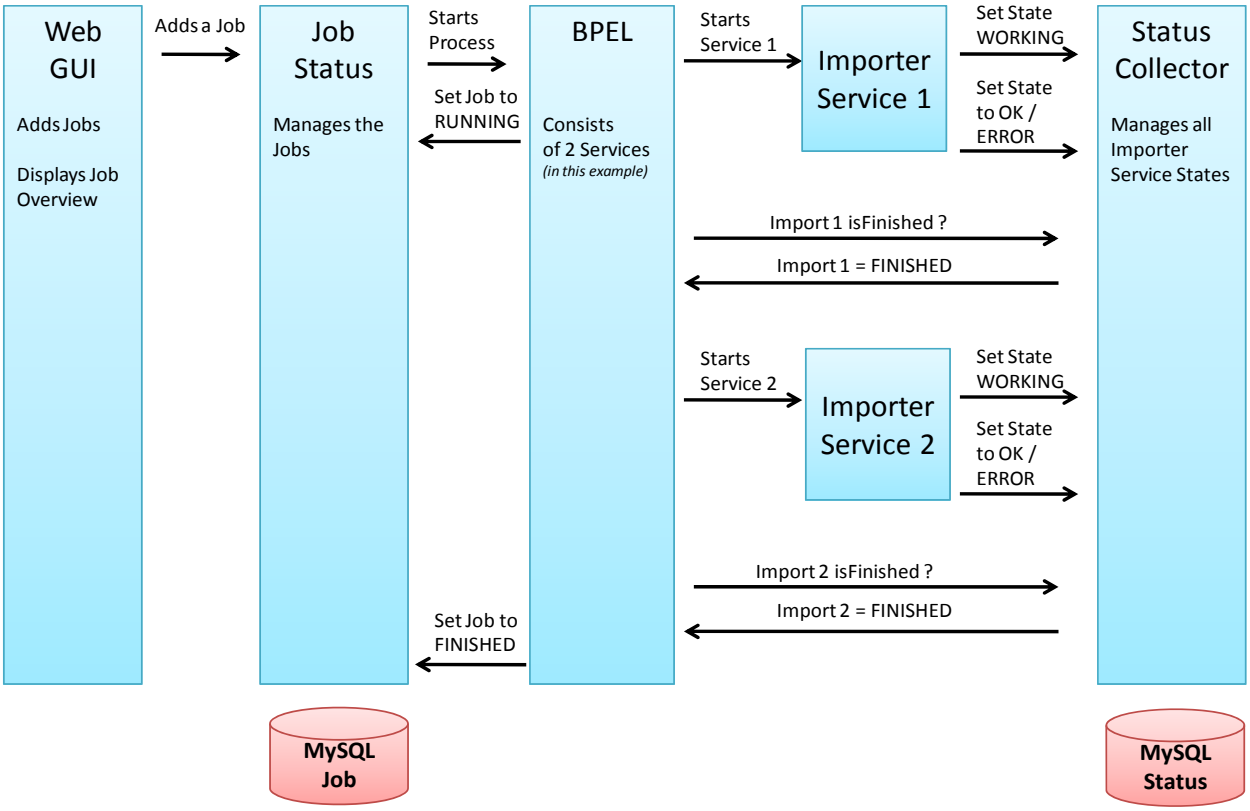


Figure 2: An example of a possible service interaction.

business process will set the job to FINISHED in the job database.

3 Extending the Platform

Integrating new services into our environment is possible; the service itself needs to be integrated into our business process and the web GUI needs to be extended to match the new services input fields.
Here are the steps:

1. Create/include new web service or web service client
    a. Extend the class from AbstractService or AbstractImporterService according to the existing services.
    b. Take an existing web service as reference (see section 5.1).

2. Create new BPEL process
    a. Set a new CONFIG_KEY_PROCESSID (equals to the class name, see information below)
    b. Existing BPEL Processes can be used as a "template" or just extend and compile the process.bpel or importer.bpel file (see section 5.2)

3. Modify the CASA setup
    a. This step is only needed if a new SOAP/Mail/FTP access port into the BPEL process is required.
    b. Modify according to the ProcessPL SOAP input port for the process.bpel process (see section 5.3)
    c. Connect the new SOAP service port to the process input port. (You may want to first read http://www.netbeans.org/kb/60/soa/casa-quickstart.html)

4. Create new Loader
    a. Extend the class from AbstractService
    b. Starts the new BPEL Process by sending a SOAP message to the newly created SOAP port.
    c. The execute method is called by the JobStatusService, if Job scheduling, resource allocation and control is desired.
    d. Take an existing importer loader as reference.

5. Modify the JobStatusService
    a. Add the WS references of the loader by inserting a new "WSDL from existing web service"
    b. Source code doesn't have to be changed, because reflection is used
    c. Uses CONFIG_KEY_PROCESSID = class name to invoke the service method (see information below)

    The CONFIG_KEY_PROCESSID has to be the fully-qualified class name of the WS reference (on the client side). Generally this would be:

```
getClass().getName() + "Service"
```

    If a Service is called Generic and it is in the package org.example the CONFIG_KEY_PROCESSID has to be "org.example.GenericService". A client then would use the Service as follows:

```
org.example.GenericServce gs = new org.example.GenericServce();
org.example.Generic generic = gs.getGenericPort();
```

    Because the JobStatusService uses reflection to create a web service instance the source code doesn't have to be extended. It just uses the CONFIG_KEY_PROCESSID and creates the service via reflection.
    To ensure that the web service can be invoked a new WS client for the new loader can be created in the JobStatusService. If the class name of the generated class equals the class name defined in CONFIG_KEY_PROCESSID, invocating the loader service (and subsequently the BPEL process) works.

6. Web GUI
    a. Extend the web site to create a job containing the new BPEL process

7. Start the new Job
    a. Call the addJob(configuration) method of the JobStatusService
    b. Configuration has to contain at least the constants defined in AbstractService

4 Conclusion

4.1 Summary

The SerVIS project dealt with four different importers, a Bugzilla, CVS, a FAMIX and an SVN Importer. Those importers were set up as standalone applications and attached to a Sesame triple store. In a second phase, they were integrated in a web service environment. Additionally to the four importer services, three management services were developed, which support the handling of the importer services. Those are: an AAA service, which handles accounts and security issues, a job status service, which keeps track of all the running jobs and a status collector service, which is responsible to collect the status of the different importer services. All the importer services are coordinated by BPEL business processes.

4.2 Lessons Learned

There were a couple of challenges we came across in our project, that are worth mentioning.

Sesame. Using Elmo was quite challenging. It's a rather young technology with a small community. Getting answers to open questions was often quite difficult, especially concerning performance problems. Furthermore, there were only very little examples for supposedly simple tasks like creating a new repository or the like. To get Sesame and Elmo work the way we wanted to, workarounds were needed sometimes.

Stable IDE. Having a stable IDE is very important. A lot of time was lost with NetBeans crashes. An alternative for web service development worth trying would be the commercially available IBM WebSphere or Oracle SOA Suite.

Debugging. Debugging in a distributed environment was rather difficult. Appropriate tool support and exhaustive logging were indispensable helpers in that matter.

Concurrent development. Concurrent development of web services is conflict prone, even if the developers work in the same room. Continuous integration would have been a useful aid.

4.3 Future Work

There are some features that may be added to increase efficiency and usability of the SerVIS environment as it is now.

FAMIX Importer. Future improvements should only use the second model crawler (of type TreeScanner) to get a consistent picture of the code tree. Since the TreeScanner get access to all elements from the internal code model tree, almost all entities can be type casted to their origin Element classes the AbstractElementVisitor6 crawler would visit.

AAA. An accounting aspect for resource planning and billing may be added in future development together with an administration view including user management.

JobStatus Service. Possible future work could be the introduction of a UDDI to easily find new web services/business processes and bind them dynamically. This would avoid including the web service references in the JobStatus service project every time a new business process has to be called.

Web GUI. The “create new job” page currently is hard coded to match the input parameters of the given web services. To make the pages more dynamic and extensible, a dynamic build up of this page would be a good approach.
One possibility would be to introduce a configuration file for each web service, where a definition of the input parameters of the service is saved. Along with the input parameters, a specification of the input widget could be saved. With such a set up, the whole create new job page could be generated and if a new service should be added to the web GUI, it would only be necessary to create such a configuration file.

JobStatus Service / Importer Loaders. The JobStatus scheduler may be extended by the functionality of the loaders and call the business process directly.

5 Appendix

5.1 SerVIS Infrastructure Overview

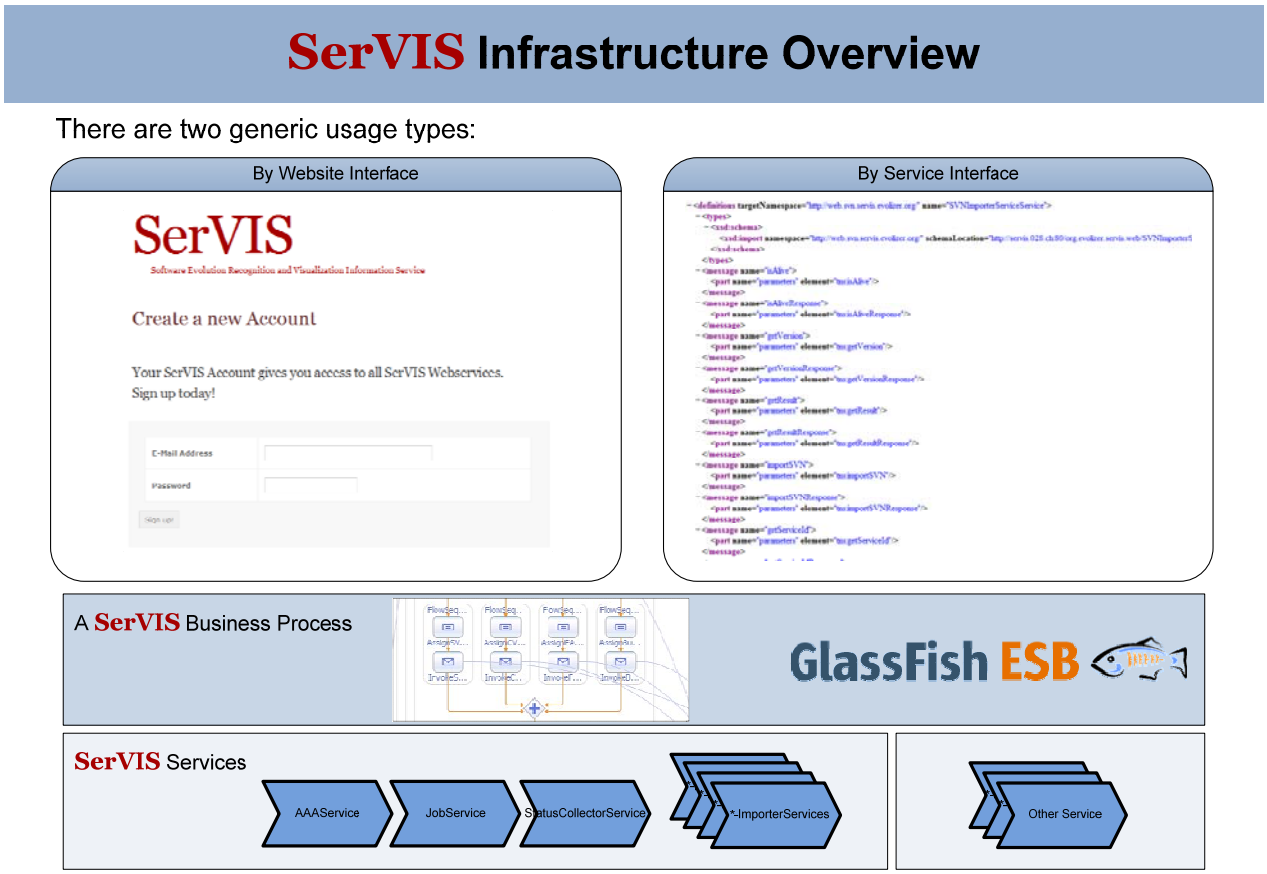


Figure 3: The SerVIS Infrastructure Overview.

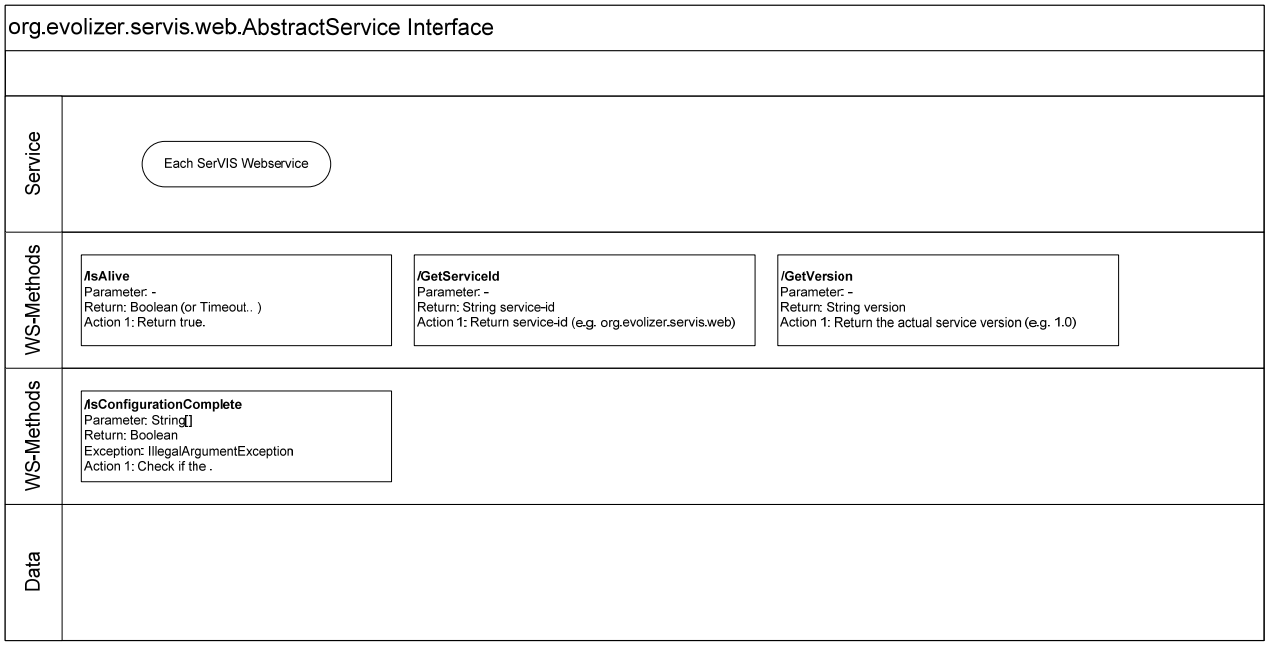


Figure 4: The AbstractService Interface.

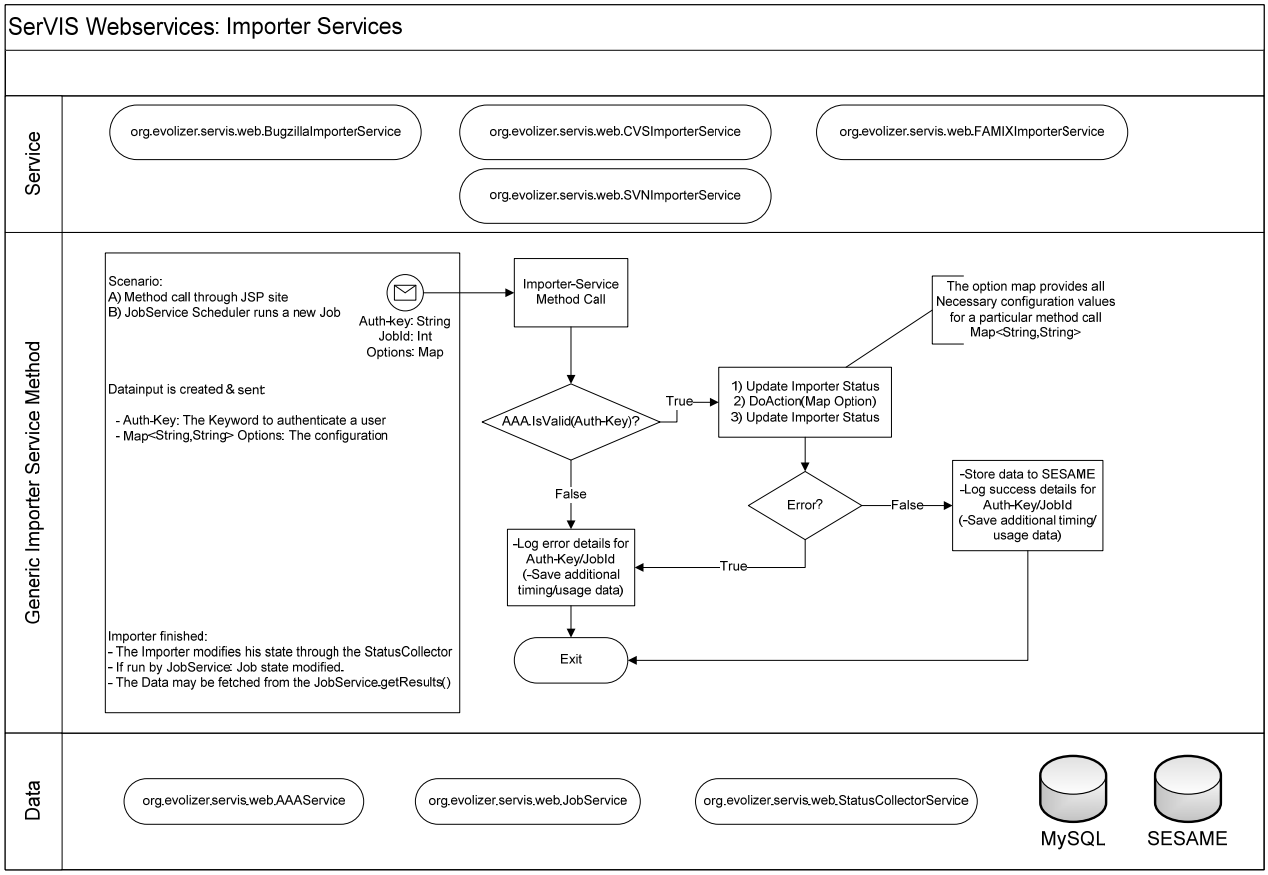


Figure 5: A generic SerVIS Importer service.

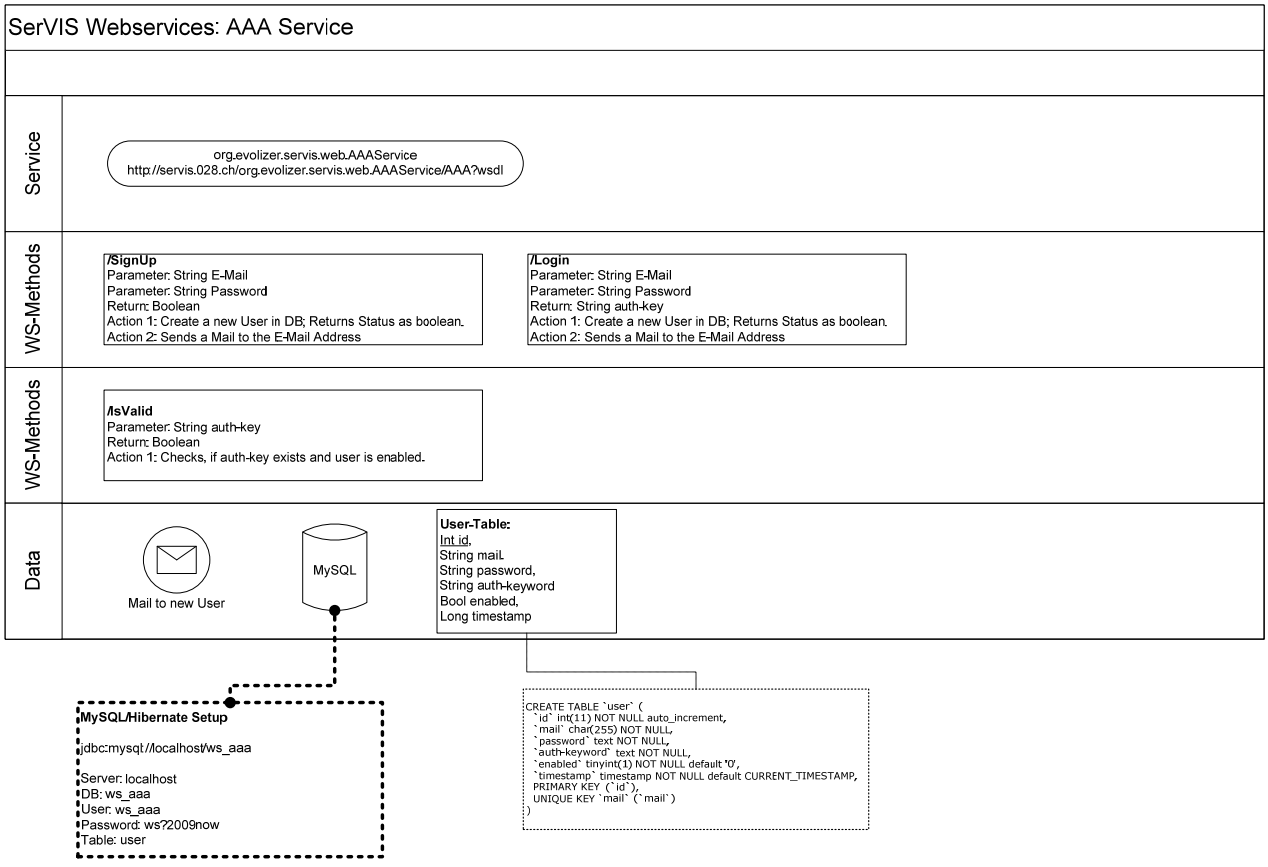


Figure 6: The AAA service.

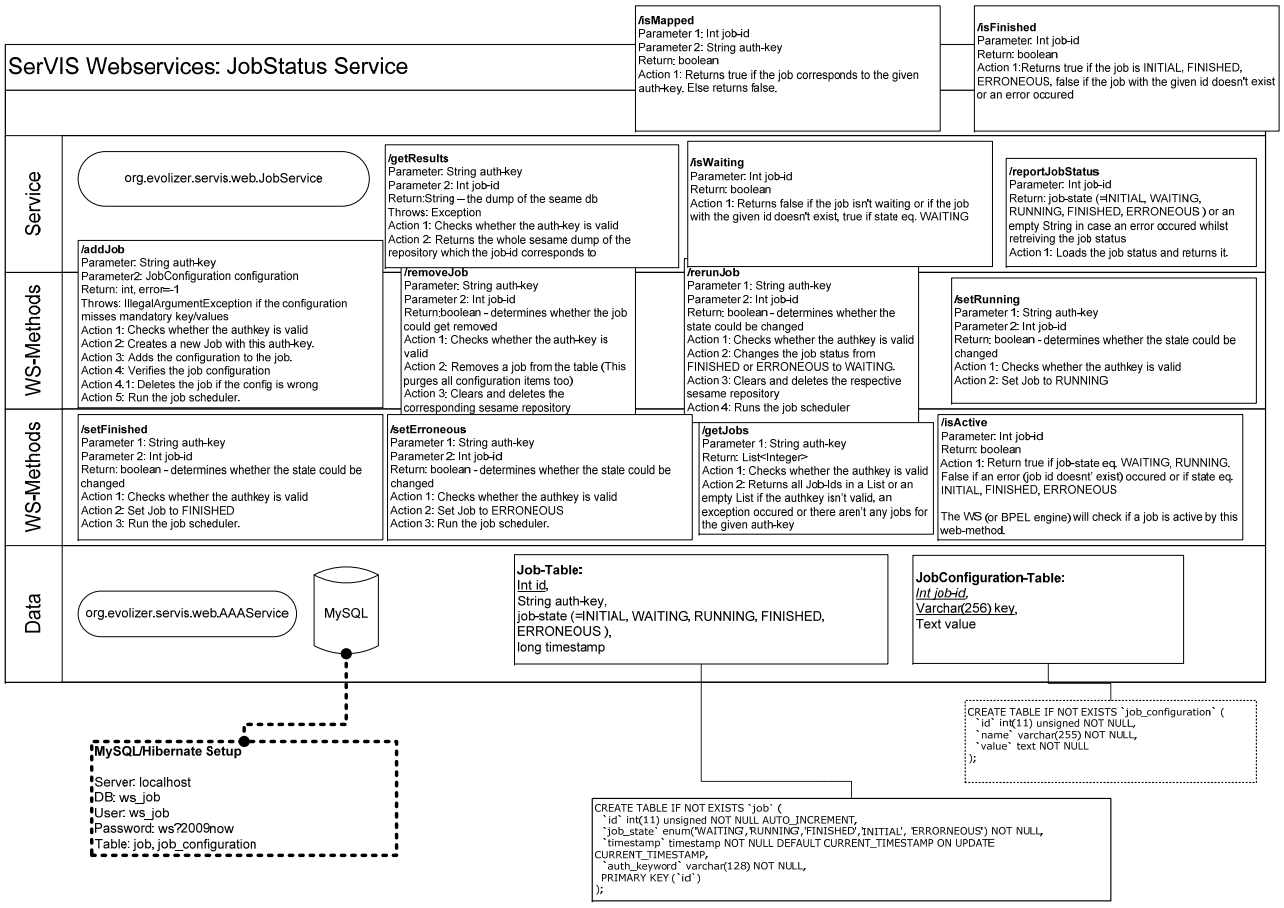


Figure 7: The JobStatus service.

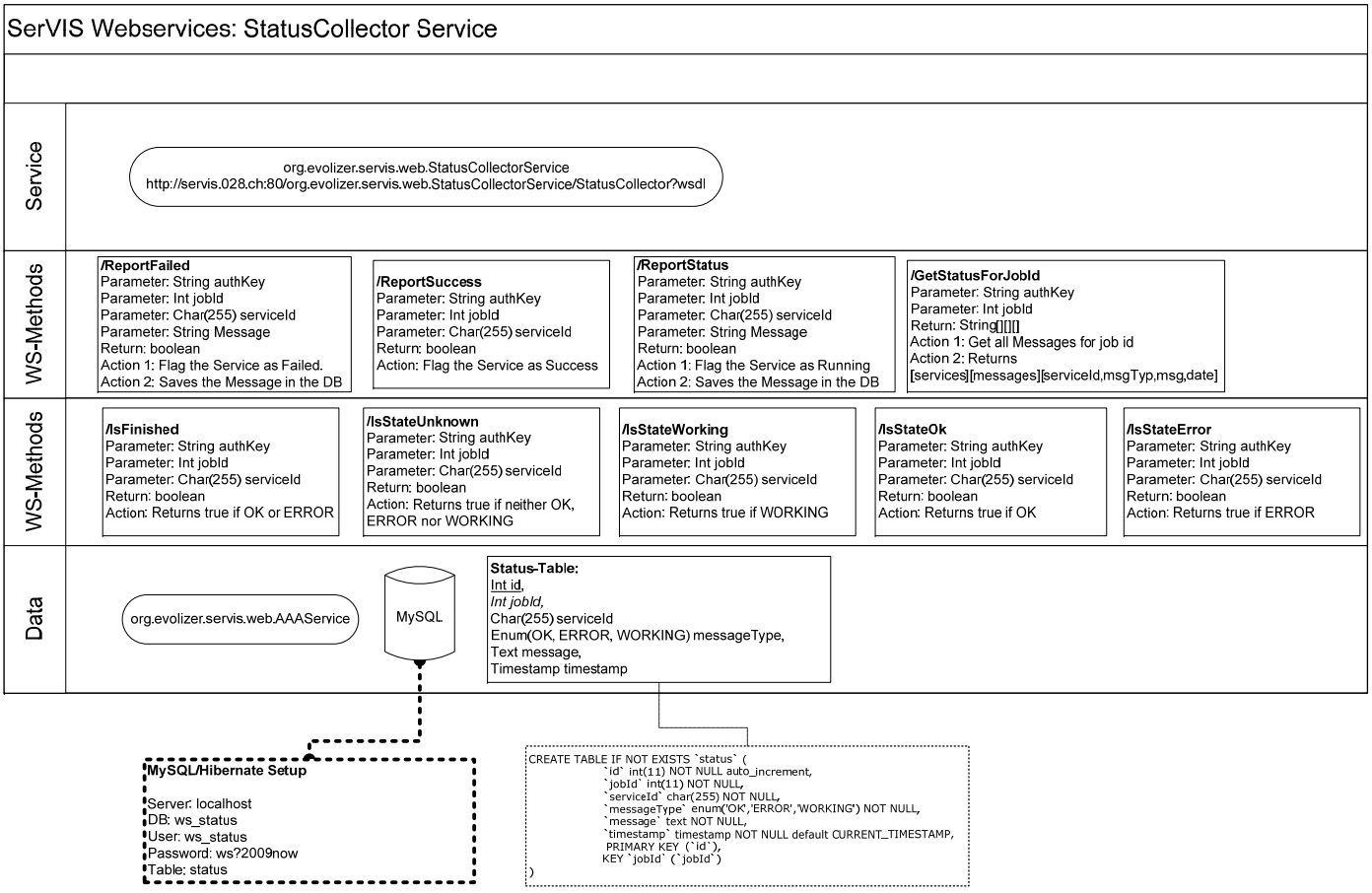
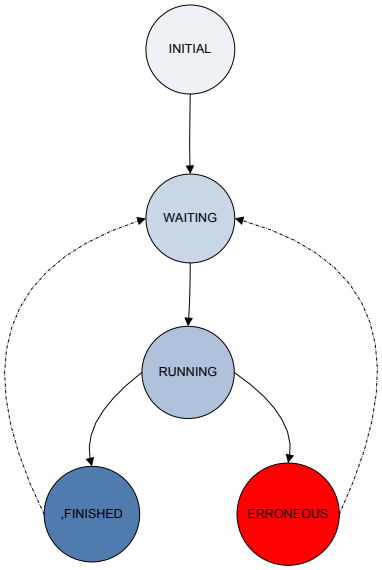
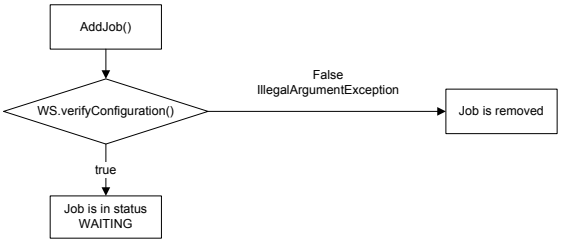


Figure 8: The StatusCollector service.

SerVIS: Job Service State Chart





INITIAL: When a job is added it is in state INITIAL. After the job configuration is verified it changes to WAITING.

WAITING: When a job is in WAITING it waits for the job scheduler to run the job. After the job schedulers runs the job it is changed to RUNNING

RUNNING: The job is running until it is either FINISHED or ERRONEOUS.

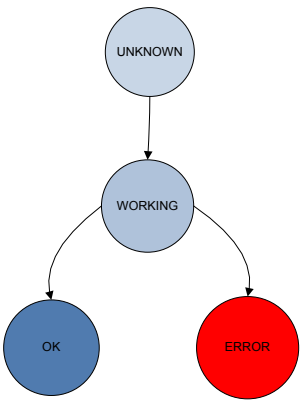
FINISHED: No errors occured

ERRONEOUS: Errors occured while the job was running.

FINISHED (and ERRONEOUS - if corrected) jobs can be rerun again.

Figure 9: The JobStatus service state chart.

SerVIS: StatusCollector Service State Chart



These four states define the state of a specific workpart (e.g. one Importer in a big business process).

A workpart is identified by
- JOB-ID: The job which contains this workpart
- SERVICE-ID: The identifier of this workpart (e.g. org.evolizer.servis.famix.web.FAMIXService)

UNKNOWN: The StatusCollectors has no entries/knowledge about this workpart (e.g not started yet). No state messages were received until now.

WORKING: The job is running meaning neither in an UNKOWN, OK nor in an ERROR state. This state is reached, if min. one state message was received until now.

OK: No errors occured. The workpart ended in an OK state. (IsFinished is now true.)

ERROR: Errors occured while the workpart was running. The workpart ended in an ERROR state. (IsFinished is now true.)

Figure 10: The StatusCollector service state chart.

5.2 BPEL Processes

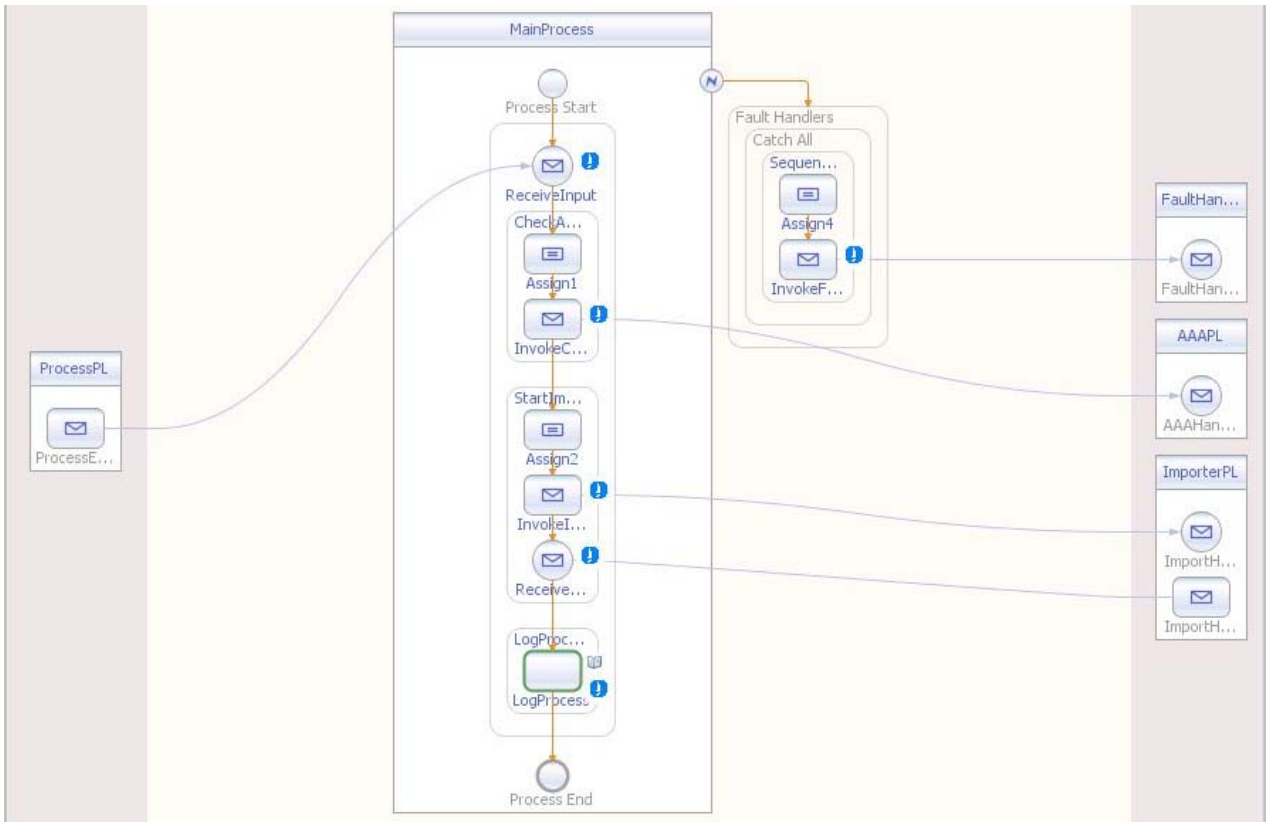


Figure 11: The main process.

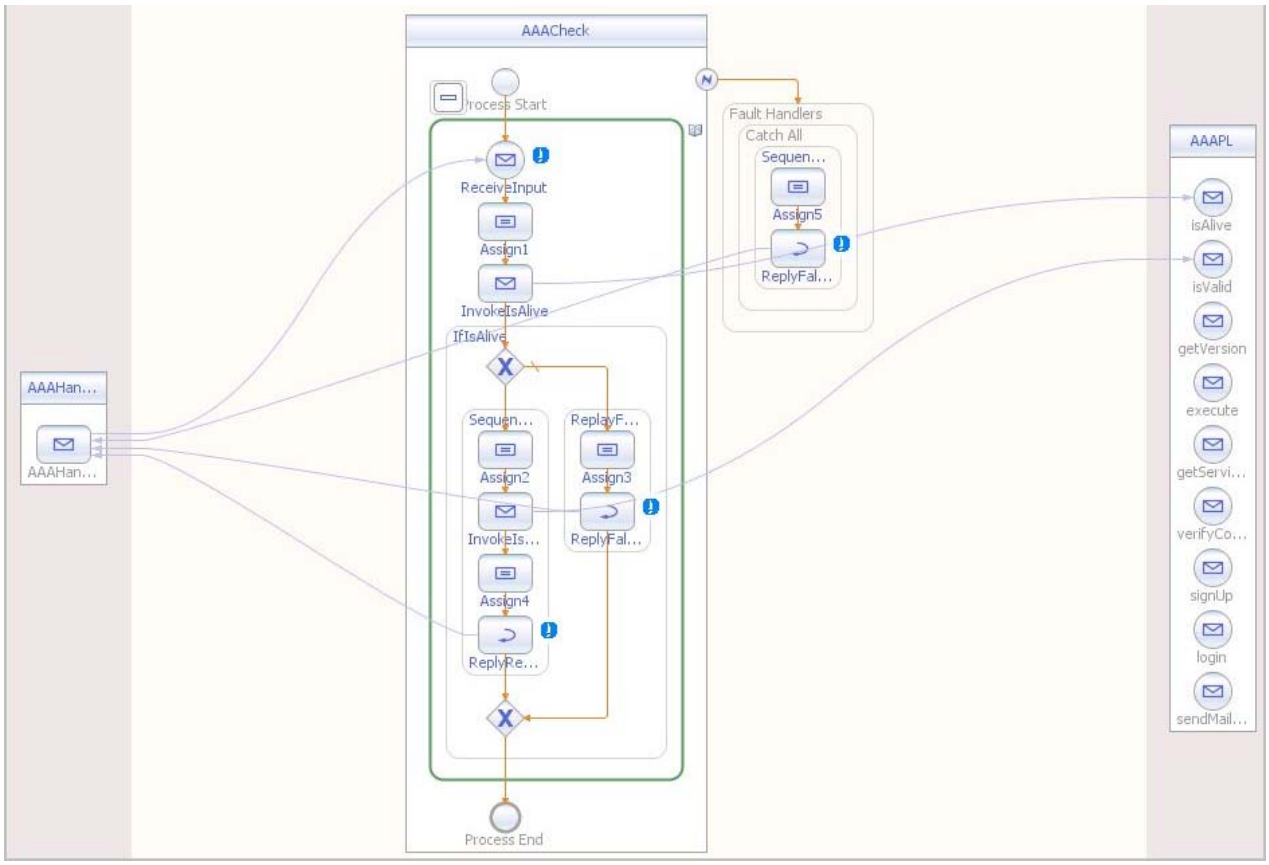


Figure 12: The AAACheck process.

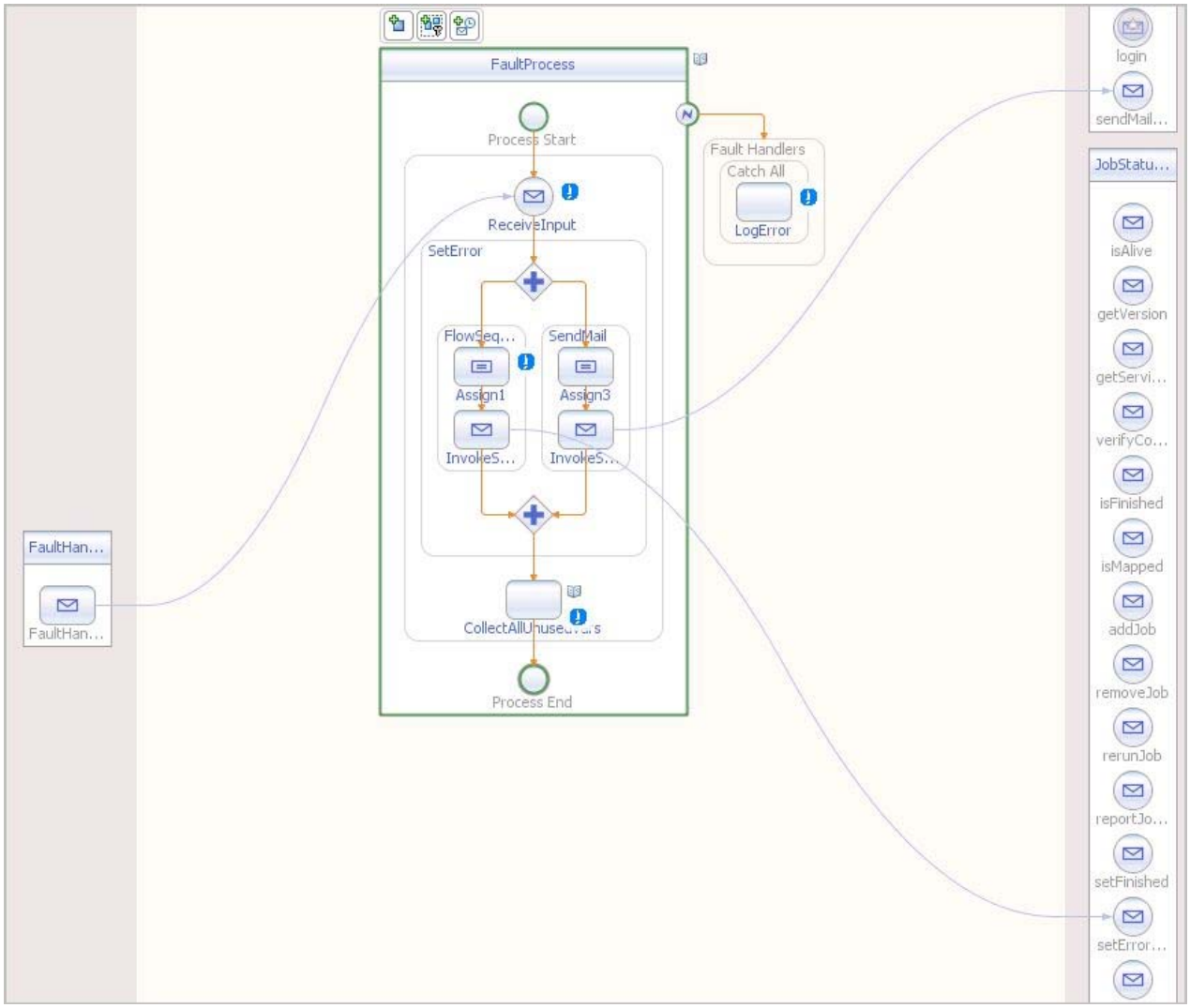


Figure 13: The FaultHandler process

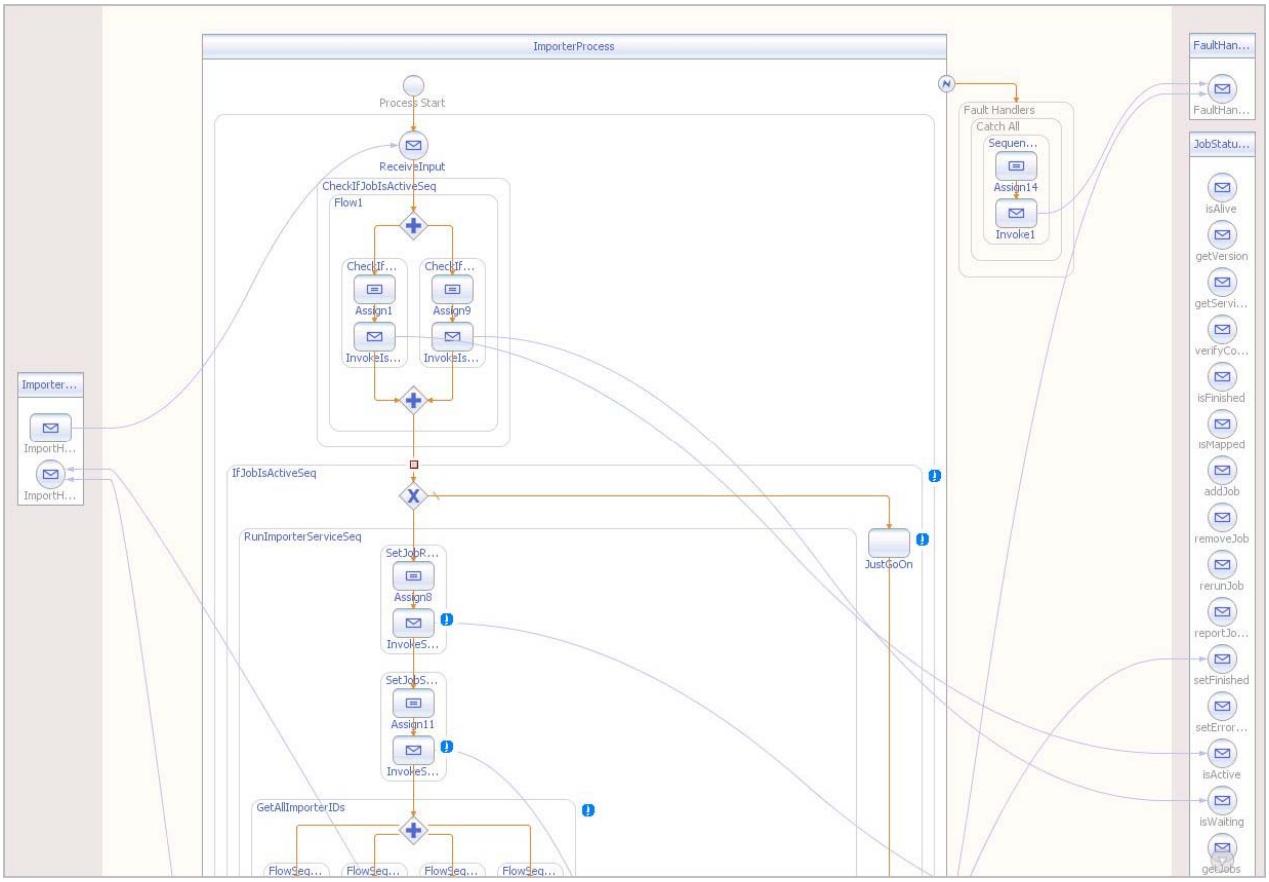


Figure 14: The Importer process part 1.

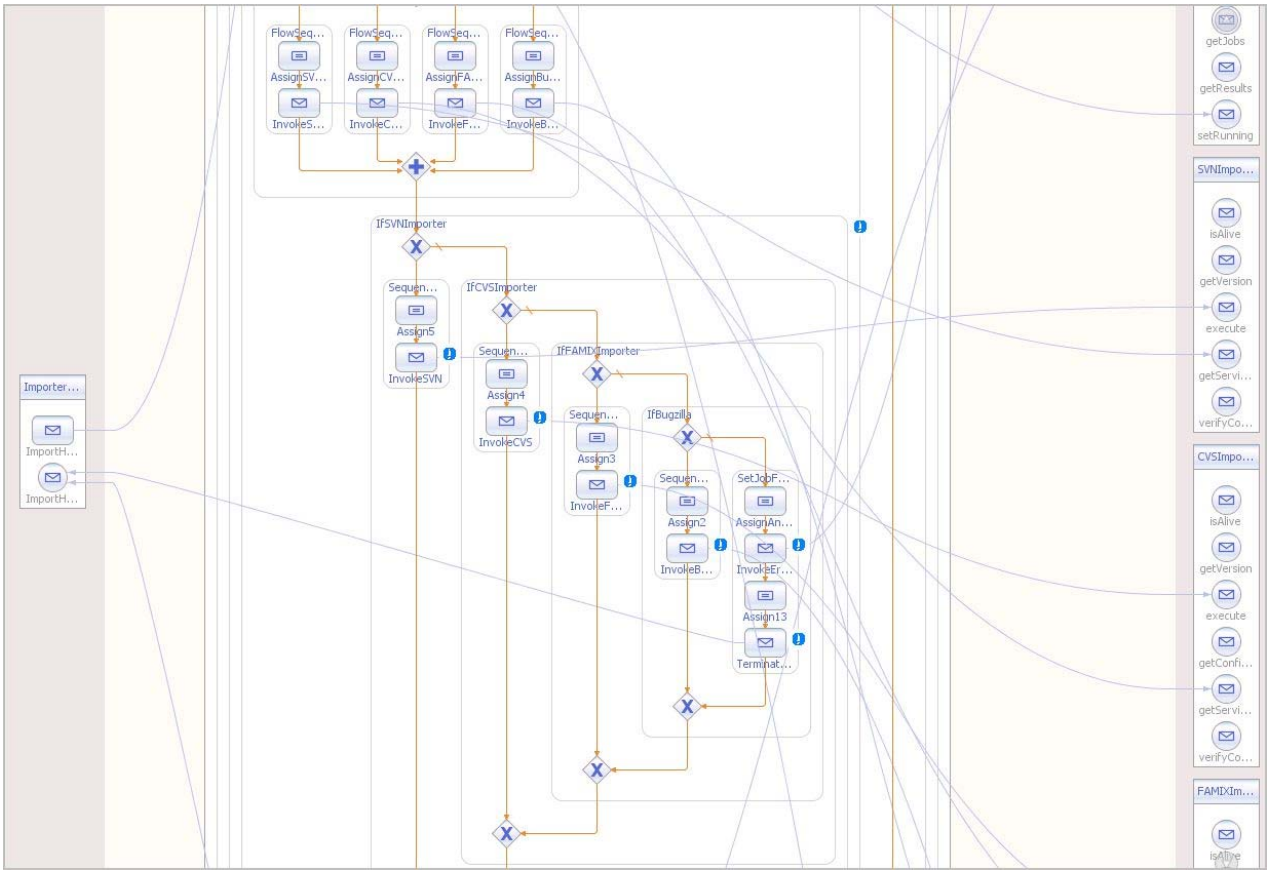


Figure 15: The Importer process part 2.

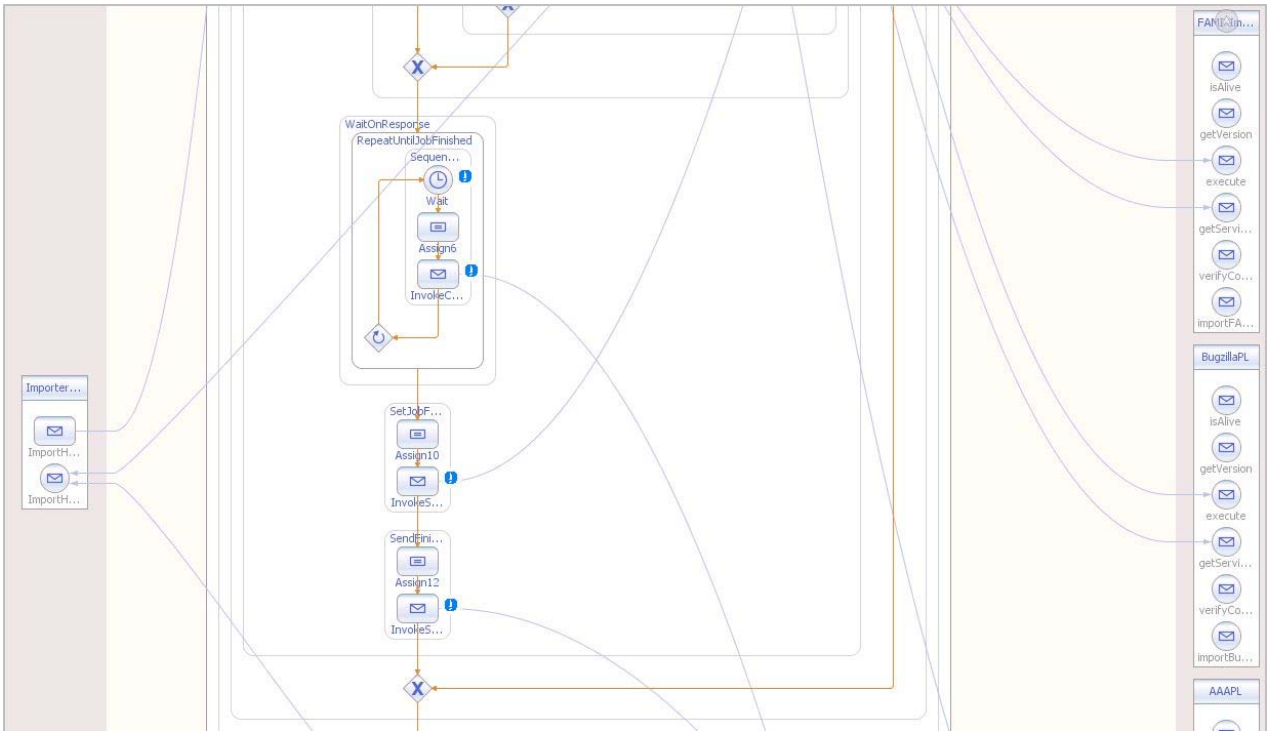


Figure 16: The Importer process part 3.

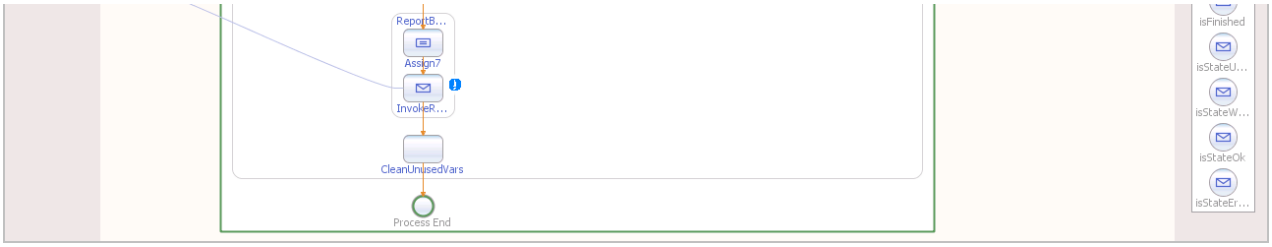


Figure 17: The Importer process part 4.

5.3 The JBI CASA view

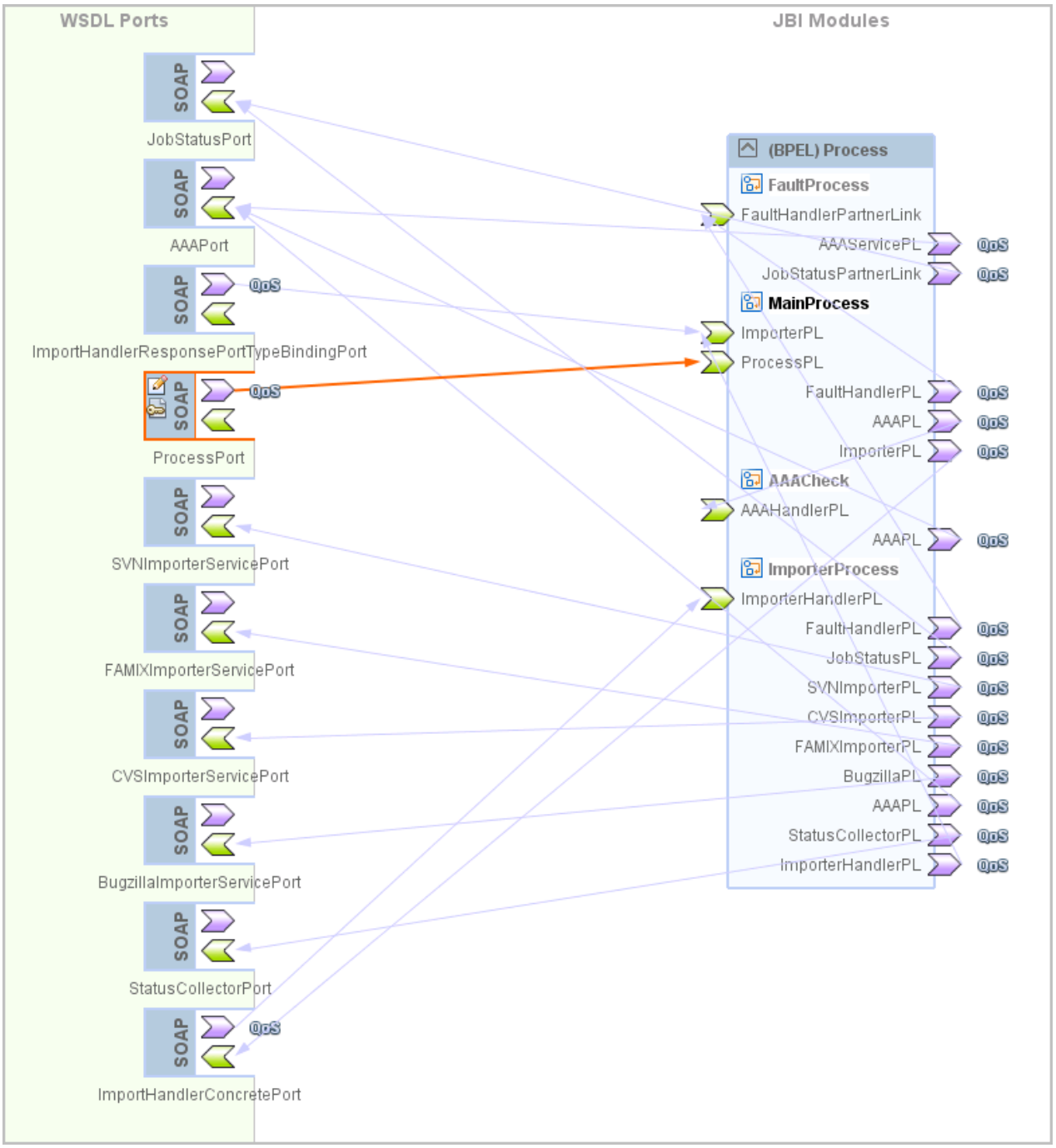


Figure 18: The composite application CASA file.

5.4 SerVIS Server Installation

Please note: There are TGZ files of the used SerVIS wiki, Tomcat 6 and Glassfish 2 server in the DVD folder "Report/Additional Stuff".

The SerVIS server runs on a CentOS Linux 5.2 system. Fetch the distribution ISOs from http://www.centos.org/ .
After installation set the right hostname ("system-config-network") and check that the host resolves the FQDN of the system in the expected manner (file /etc/hosts). For our SerVIS server this was:

```
130.60.156.191          servis.028.ch servis
127.0.0.1               localhost.localdomain localhost
::1                     localhost6.localdomain6 localhost6
```

5.4.1 OpenJDK 1.7.0, Java JDK 1.6.0

Since the FAMIX Importer builds on OpenJDK, Tomcat must use OpenJDK as Java Runtime environment. Glassfish doesn't support a Java runtime greater than 1.6.999 (at the time of writing) therefore a Sun Java JDK 1.6.0 is required too.

1. Install OpenJDK 1.7.0 in "/usr/local/jdk1.7.0" by extracting the newest archive from http://download.java.net/openjdk/jdk7/.
2. Install OpenJDK 1.6.0 by yum which uses "alternatives" to adjust all system-, binary- and library-links to this Java version. (yum install java-1.6.0-openjdk)
3. Check that "java -version" returns the right Java version strings:

```
[root@servis local]# java -version
java version "1.6.0_0"
IcedTea6 1.3.1 (6b12-Fedora-EPEL-5) Runtime Environment (build 1.6.0_0-b12)
OpenJDK Client VM (build 1.6.0_0-b12, mixed mode)

[root@servis local]# /usr/local/jdk1.7.0/bin/java -version
java version "1.7.0-ea"
Java(TM) SE Runtime Environment (build 1.7.0-ea-b41)
Java HotSpot(TM) Client VM (build 14.0-b08, mixed mode, sharing)
```

5.4.2 Tomcat

If you don't know Tomcat, please read http://tomcat.apache.org/tomcat-6.0-doc/index.html first. Check the sections about installation, the shared class loader and security.

1. Fetch the newest Tomcat 6 archive from http://tomcat.apache.org/download-60.cgi (e.g. apache-tomcat-6.0.18.tar.gz) and extract it to /usr/local/apache-tomcat-6.0.18.
2. Create a group "tomcat" and a user "tomcat" (Home-Dir: /usr/local/apache-tomcat-6.0.18, Shell: /bin/sh)
3. Create a folder "/usr/local/apache-tomcat-6.0.18/shared-lib"; Extract the file shared-lib.zip from the org.evolizer.servis.web project into it. Adjust the file access rights.
4. Adjust the file conf/catalina.properties:

```
shared.loader=/usr/local/apache-tomcat-6.0.18/shared-lib/*.jar
```

5. Adjust the file: startup.sh:

```
JRE_HOME=/usr/local/jdk1.7.0/jre; export JRE_HOME
```

6. Adjust the file: bin/catalina.sh

```
JAVA_OPTS="$JAVA_OPTS -Xss8M -Xmx800M -D/usr/local/apr/lib"
```

7. Add a startup script e.g. /etc/init.d/tomcat6

```
#!/bin/bash
#
# tomcat6      This shell script takes care of starting and stopping Tomcat
#
# chkconfig: - 80 20
#
### BEGIN INIT INFO
# Provides: tomcat6
# Description: tomcat
# Short-Description: start and stop tomcat
### END INIT INFO
#

RETVAL="0"


function echo_failure() {
    echo -en "\\033[60G"
```

```
    echo -n "[  "
    echo -n $"FAILED"
    echo -n "  ]"
    echo -ne "\r"
    return 1
}

function echo_success() {
    echo -en "\\033[60G"
    echo -n "[  "
    echo -n $"OK"
    echo -n "  ]"
    echo -ne "\r"
    return 0
}

case "$1" in
    start)
        /bin/su - tomcat -c "sh bin/startup.sh" >> /dev/null
        sleep 2
        echo_success
        ;;
    stop)
        /bin/su - tomcat -c "sh bin/shutdown.sh" >> /dev/null
        sleep 2
        echo_success
        ;;
    restart)
        /bin/su - tomcat -c "sh bin/shutdown.sh" >> /dev/null
        sleep 2
        /bin/su - tomcat -c "sh bin/startup.sh"
        echo_success
        ;;
    *)
        echo "Usage: tomcat6 {start|stop|restart}"
        exit 1
esac

exit 0
```

8. Activate it and run Tomcat:

```
chkconfig --add tomcat6
chkconfig tomcat6 on
service tomcat6 start
```

The Tomcat log file is saved in logs/* like catalina.out. The logger output goes there too.

5.4.3 Glassfish

If you don't know Glassfish, please read first https://glassfish.dev.java.net/downloads/quickstart/index.html and visit the Glassfish wiki http://wiki.glassfish.java.net .

Get the Glassfish ESB installer from https://open-esb.dev.java.net/Downloads.html and run it. Install Glassfish in /usr/local/glassfish and use the JDK 1.6 as Java runtime. If there is already another application using the default internet ports (like on our SerVIS server) change them according to:

```
HTTP:  8081
HTTPS: 8181
```

Start the application server using the asadmin CLI. You may create a simple init.d script according to the tomcat init.d script and change:

```
case "$1" in
    start)
```

```
        /usr/local/glassfish/bin/asadmin start-appserv >> /dev/null
        sleep 5
        echo_success
        ;;
    stop)
        /usr/local/glassfish/bin/asadmin stop-appserv >> /dev/null
        sleep 5
        echo_success
        ;;
```

Register and activate it as a new system service (according to Tomcat). The logfile is saved in /usr/local/glassfish/domains/domain1/logs/server.log.

Check the options from asadmin since most of them are used once in a while.

5.4.4 Logger and DB Configuration

Add the following log4j configuration file log4j.properties to your apache base directory (e.g. /usr/local/apache-tomcat-6.0.18/):

```
log4j.rootLogger=DEBUG, A1, R1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n

log4j.appender.R1=org.apache.log4j.RollingFileAppender
log4j.appender.R1.File=logs/logger.out
log4j.appender.R1.MaxFileSize=10MB
log4j.appender.R1.MaxBackupIndex=10
log4j.appender.R1.layout=org.apache.log4j.PatternLayout
log4j.appender.R1.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
```

Add the following log4j configuration file dbconfig.properties to your apache base directory (e.g. /usr/local/apache-tomcat-6.0.18/):

```
#Config for Sesame

#the url where the sesame db is located
url=http://localhost:8080/openrdf-sesame/

#the ID of the specific repository.
#If there isn't an ID specified, then the test repositories will be used
repositoryID=
```

5.4.5 MySQL

Install and activate the MySQL database server. Use the packetmanager yum for installing mysql-server and all related packets. Activate the server by:

```
chkconfig mysqld on
service mysqld start
```

The database configuration file is located in /etc/my.cnf. The DB data is stored in /var/lib/mysql/* .

For the SerVIS Management Services, the SerVIS database archive file from the DVD may be used. Just copy and paste the database folders and restart the MySQL server. Additionally, check the “MySQL .txt” file in the “Report/Additional Stuff” folder.
For creation of the MySQL users please use the credential information from “SerVIS-Services.vsd” – again in “Report/Additional Stuff” folder.

5.4.6 Apache Web Server

Install and activate the Apache HTTP server. Use the packet manager yum for installing “httpd” and all related packets. Activate the server by:

```
chkconfig httpd on
```

The apache configuration file is located in /etc/httpd. The DB data is stored in /var/www/html/. Add the following lines to your configuration:

```
NameVirtualHost *:80

# Virtual host TOMCAT
<VirtualHost *:80>
   ServerAdmin webmaster@servis.028.ch
   DocumentRoot /var/www/html
   ServerName servis.028.ch
   ErrorLog /var/log/httpd/error_log
   CustomLog /var/log/httpd/access_log combined

   #Tomcat RevereProxy
   ProxyPreserveHost On
   ProxyPass            /       http://localhost:8080/
   ProxyPassReverse     /       http://localhost:8080/
</VirtualHost>


# Virtual host GLASSFISH
<VirtualHost *:80>
   ServerAdmin webmaster@servis.028.ch
   DocumentRoot /var/www/html
   ServerName servis-admin.028.ch
   ErrorLog /var/log/httpd/error_log
   CustomLog /var/log/httpd/access_log combined

   #Tomcat RevereProxy
   ProxyPreserveHost On
   ProxyPass            /       http://localhost:4848/
   ProxyPassReverse     /       http://localhost:4848/
</VirtualHost>
```

Change the domain name according to your needs and verify the configuration:

```
[root@servis ~]# apachectl -S
VirtualHost configuration:
wildcard NameVirtualHosts and _default_ servers:
_default_:443          servis.028.ch (/etc/httpd/conf.d/ssl.conf:81)
*:80                   is a NameVirtualHost
         default server servis.028.ch (/etc/httpd/conf/httpd.conf:994)
         port 80 namevhost servis.028.ch (/etc/httpd/conf/httpd.conf:994)
         port 80 namevhost servis-admin.028.ch (/etc/httpd/conf/httpd.conf:1009)
Syntax OK
```

You may have to verify if the proxy modules are activated in httpd.conf.

Start the server: "apachectl start"

5.4.7 Firewall

Add the following lines to your iptables config file (Redhat: /etc/sysconfig/iptables) and reload iptables.

```
##### Apache Access Port:
```

```
-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 443 -j ACCEPT
-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 80 -j ACCEPT
##### Tomcat direct access:
-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 8080 -j ACCEPT
##### Glassfish direct access:
-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 8081 -j ACCEPT
-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 8181 -j ACCEPT
-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 9081 -j ACCEPT
-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 18181 -j ACCEPT
###############
```

Since a reverse proxy on port 80 is active, port 8080 may also be closed.

5.4.8 Backup

You have to backup the following directories:

/var/lib/mysql
/usr/local/glassfish
/usr/local/apache-tomcat-6.0.18

and the /etc/ for additional configuration items.

5.5 E-Mail Notifications

The AAA service provides an easy way to inform customers about their jobs. Here are some of the messages if a regular SVN import job is started.

Job is started message:

```
Return-Path: <do.not.reply@servis.028.ch>
Received: from servis.028.ch ([130.60.156.191])
       by www3.lin-art.cc (8.13.8/8.13.8) with ESMTP id n1PBvtq8019273
       (version=TLSv1/SSLv3 cipher=DHE-RSA-AES256-SHA bits=256 verify=NO)
       for <test123@lin-art.cc>; Wed, 25 Feb 2009 12:58:00 +0100
Received: from servis.028.ch (localhost.localdomain [127.0.0.1])
       by servis.028.ch (8.13.8/8.13.8) with ESMTP id n1PBvsXR029776
       for <test123@lin-art.cc>; Wed, 25 Feb 2009 12:57:55 +0100
Date: Wed, 25 Feb 2009 12:57:54 +0100
From: do.not.reply@servis.028.ch
To: test123@lin-art.cc
Message-ID: <13557582.0.1235563074743.JavaMail.tomcat@servis.028.ch>
Subject: SerVIS Status Notification
MIME-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit

This is a message from the SerVIS Webapplication:
------------------

Your job with Job-ID: 11 was successfully started.

------------------
Thanks for using SerVIS.
```

Job is finish message:

```
This is a message from the SerVIS Webapplication:
------------------

Your job with Job-ID: 11 finished successfully. The results are now available.
```

```
------------------
Thanks for using SerVIS.
```