Web services for the eJournal

Supervisor: Dr. Denis Gillet

Assistant: Anh Vu Nguyen Ngoc

February 2005

EPFL
helena.wittek@epfl.ch

1. Introduction

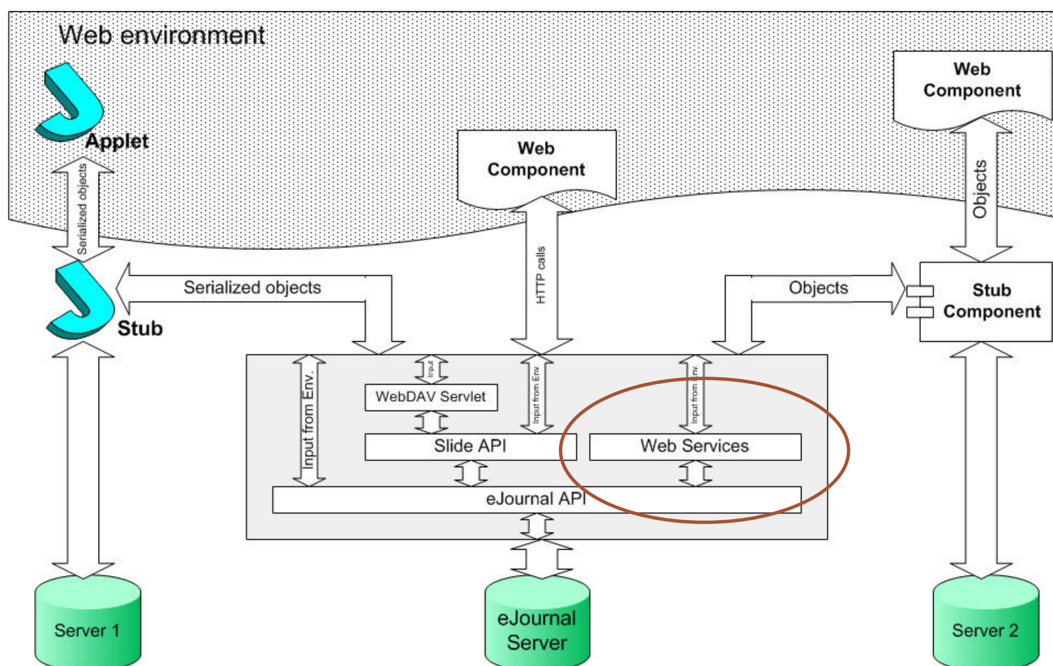
The "Web Services for the eJournal" project was my winter semester project in the 8th semester at the EPFL(Swiss Federal Institute of Technology, Lausanne). Its goal was to integrate Web Services functionalities into eJournal, a collaborative workspace developed as part of the eMersion Project[1].
The eJournal has the goal to sustain a collaboration between students and their tutors, by offering means to share their documents between them.
This paper describes a guideline to create web services for the eJournal as well as the problems encountered and further development of the project.

A web service is a networked application that is able to interact using standard application Web protocols with interfaces and which is described using a standard functional description language.
Web services are platform- and language-independent, which means that a web service can be developed using any language and it can be deployed on any platform. Web services communicate using XML and Web protocols and support heterogeneous interoperability.

An architecture using web services is a service oriented architecture and supports loosely coupled connections. Loose coupling minimizes the impact of changes to the applications. A Web service interface provides a layer of abstraction between the client and server. A change in one doesn't necessarily force a change in the other. The abstract interface also makes it easier to reuse a service in another application.
The development of web services for the eJournal will contribute to the reusability and to an easier maintenance of the eJournal.

1.1. Web services for the eJournal

Below we can see the eJournal and its interactions with the web environment showing implementations that have already been realized (communicating with Applets, an implementation using WebDav).
The goal of this project is to implement web services enabling the eJournal to communicate with the environment and letting the environment communicate with the eJournal. For the eJournal in particular this means that the services created, can and should be reused by other groups that want to create an eJournal-like application for their department. By providing an abstraction layer between the eJournal services and the environment, future clients are not dependent of the exact implementation and the eJournal can still be easily maintained. As web services are platform independent these can be reused by any group having any kind of operating system, enabling them to create an eJournal of their own in their own applications.
Enhancing the eJournal with web services will allow others to benefit from the whole eJournal concept.



2. Services to be implemented

To integrate web service functionalities into the eJournal several services must be implemented.
JournalService:
In the eJournal each group a has its own journal for storing their files thus for the eJournalService several methods have to be implemented such as different lookup methods as well as methods to create, assign a group, rename and delete a Journal.
FolderService:
Each journal contains several folders to store the different fragments (or files). Thus there need to be different methods for creating, renaming, and deleting folders.
FragmentService:
As each folder contains fragments, different services for renaming, copying and moving fragments have to be created.
UserService:
The user service will allow us to look up certain information about users, like which group the belong to, which journal they belong to etc.
AuthenticationService:
An authentication service with a login method is needed to be able to identify the users and let them logon to the eJournal.

With these services future eJournal clients can create their own eJournal like applications.

2.1. Preliminaries

For creating web services we have to write class interfaces and the corresponding implementations. One class will then become a web service and the methods in the interface of the class will become web service operations. Methods not contained in the interface are only helper methods and will not become web service methods.
Based on the interfaces that were given for the different services, I wrote the implementation and added helper methods and classes if needed. Finally I tested if the implementation worked to be able to start into web services with a correct implementation. This did not present any particular problems so I will not go into details of the implementations themselves as this paper should present a guideline to the creation of the web services. From now on I will assume that the basic implementation has been written and tested.

3. Software and components

In this chapter I will introduce the different software and components I have used, how to install them and any special configuration that might be needed. All the technologies used are free and open-source, the advantages being their wide usage, their robustness and good documentation. All the implementation has been done under a Windows XP system.

3.1. Tomcat

The Web server chosen for the implementation is Tomcat Container 5.0. It can be easily downloaded and installed according to the installation instructions [2]. The Tomcat Container is by default configured to listen on port 8080. If there is another server running on this port, the installation port might have to be changed.

3.2. Axis

3.2.1. Installation

The Apache Axis Web Container (from Apache eXtensible Interaction System) is an open source web service toolkit for Java. Axis is an implementation of an XMLSOAP based protocol used for exchanging data in a distributed environment.
It can be easily downloaded and installed[3].
In the Tomcat Server installation there is a directory into which web applications ("webapps") are to be placed. Into this directory we copy the webapps/axis directory from the xml-axis distribution.

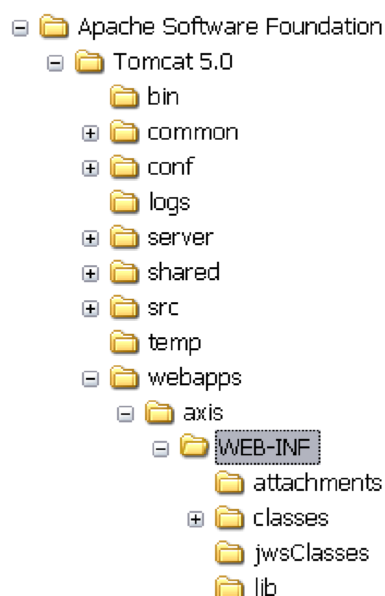


Figure 1 : Folder Structure

After installation Axis must be validated with happyaxis. We follow the link *Validate the local installation's configuration* . This brings us to a *happyaxis.jsp,* a test page that verifies that needed and optional libraries are present. The URL for this is http://localhost:8080/axis/happyaxis.jsp
If any of the needed libraries are missing, Axis will not work and we cannot proceed until all needed libraries are found, and this validation page is happy. In my case there were some libraries missing which I downloaded and installed.

3.2.2. Features

Axis has several tools for implementing Web Services:

- full support of SOAP 1.1 (Simple Object Access Protocol, see section 4.1)
- WSDL support (Web service description language, see section 4.2)
- Java2WSDL tool
    - if Java service implementation already exists, it generates a WSDL from Java service implementation classes
- WSDL2Java tool
    - generates client stubs, these stubs represent the service on the client-side
    - generates service skeletons for implementing services described by WSDL, these represent the service on the server-side
    - generates other necessary server-side files
- automatically generates WSDL for deployed web services, clients can access by appending “?wsdl” to the web service URL http://host:port/axis/services/service-name
- AdminClient tool to deploy the services (make them available)
- SOAPMonitorApplet to view requests and responses to/from the services

3.3. MySQL

We used MySQL Server because it is used in the original implementation of the eJournal. The database stores the metadata of the eJournal, this being user information, group information, folder and fragment information.
I installed MySQL 4.0.21[4] (the current release is 4.1.9, but this version fulfilled the requirements for the project). I also installed MySQL Query Browser and Administrator, not necessary but helpful graphical user interfaces instead of using the command line.

3.4. Eclipse & plugins

As development environment I used Eclipse 3.0.1. [5]
Eclipse is a an open extensible Java IDE that can be enhanced with plug-ins which can facilitate the development of the web services:
I installed several plugins for Eclipse:

- Tomcat Plugin for starting/shutting down Tomcat[6]
  This plugin can be used for starting, stopping and restarting Tomcat 5.0.x
  - Registering Tomcat process to Eclipse debugger
  - Creating a WAR project (wizard can update server.xml file)
  - Adding Java Projects to Tomcat classpath
  - Setting Tomcat JVM parameters, classpath and bootclasspath

- Java2WSDL Plugin from improve-technologies[7]
  On the server side:
  - Create a project with a default webapp directory structure and classpath with axis libraries in the WEB-INF/lib directory
  - Generate a WSDL file from a Java file
  - Deploy a web service under Axis from a WSDD file

  On the client side:
  - Generate a Java File (proxy) from a WSDL file to use a web service

  Unfortunately this plugin did not work with my Eclipse, there was an error about an illegal file name that could not be resolved.

- WSDL2Java from MySpotter [8]
  This plugin, though not supported anymore, did work but as it does only create client side stubs, it is useful for creating a web service clients only, not for providing a web service.

3.5. General Configuration

To use the Axis command line tools (Java2WSDL, WSDL2Java) the classpath must be correctly configured:
Under Windows all these libraries must be included:

```
%AXISCLASSPATH% :
D:\Libs\axis\axis-ant.jar;
D:\Libs\axis\axis.jar;
D:\Libs\axis\commons-discovery.jar;
D:\Libs\axis\commons-logging.jar;
D:\Libs\axis\jaxrpc.jar;
D:\Libs\axis\log4j-1.2.8.jar;
D:\Libs\axis\saaj.jar;
D:\Libs\axis\wsdl4j.jar;
D:\Libs\axis\xercesImpl.jar;
D:\Libs\axis\xml-apis.jar;
D:\Libs\axis\xmlParserAPIs.jar;
D:\Libs\axis\activation.jar;
D:\Libs\axis\mail.jar
```

And more importantly the classpath to the actual java classes must be defined, for example:

```
D:\EPFL\eclipse\workspace\eJournalProject\classes ;
```

4. Web Services Basics

In this chapter I will introduce some basic notions about web services to understand how clients and providers of web services communicate.

4.1. SOAP

SOAP(Simple object access protocol) is an XML messaging protocol providing conventions on how to structure headers and body in an XML message. SOAP is declared to be transport independent, so that SOAP messages can be sent over arbitrary transport protocols. SOAP specifies exactly how to encode an HTTP header and an XML file so that it can execute a remote procedure call and pass information to the procedure. It also specifies how the called program can return a response. As the message is sent over an HTTP protocol it can also avoid firewalls. SOAP provides an attractive alternative to traditional proprietary protocols, such as CORBA and DCOM.

4.2. WSDL

The web services interface is modeled using Web Services Description Language (WSDL). A WSDL document is essentially an XML file that describes the location and the operations with their parameters for a given web service. The current standard is WSDL 1.1

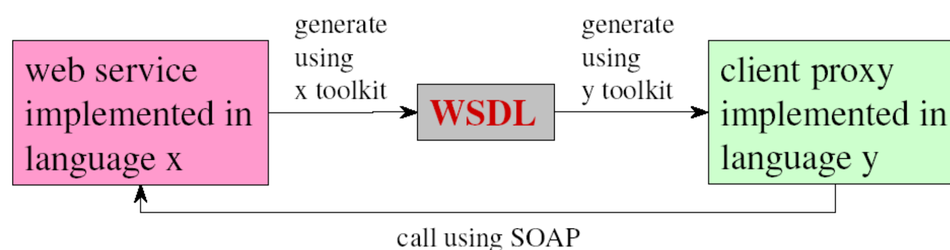


Figure 2: WSDL

Here I use the example of the eJournalFolderService to explain the WSDL language. A WSDL description is composed of the following:

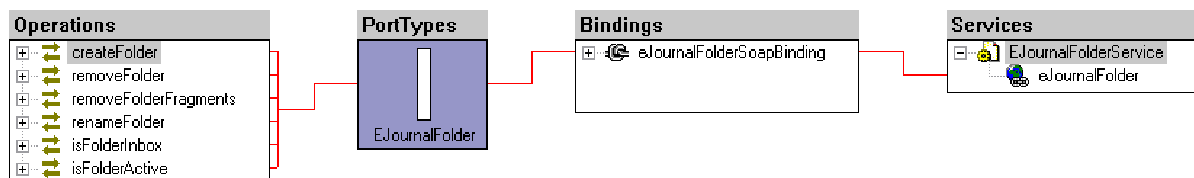


Figure 3 : WSDL of ejournalFolderService in XMLSpy

- **types**

If a web service returns a user-defined type, like the method “login” from eJournalAuthenticationService, a container for abstract type definitions using XML Schema is created. This container is used for user-defined types, like the class User or the class Fragment. The following represents the structure of the class User.

```
<wsdl:types>
  <schema targetNamespace="http://core.eJournal.sti.epfl.ch"
         xmlns="http://www.w3.org/2001/XMLSchema">
     <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
     <complexType name="User">
        <sequence>
           <element name="password" nillable="true" type="xsd:string"/>
           <element name="section" nillable="true" type="xsd:string"/>
           <element name="loginName" nillable="true" type="xsd:string"/>
           <element name="sciper" nillable="true" type="xsd:string"/>
           <element name="username" nillable="true" type="xsd:string"/>
           <element name="email" nillable="true" type="xsd:string"/>
        </sequence>
     </complexType>
  </schema>
</wsdl:types>
```

- **message**
  The message element shows the parameters of the request and the return type of a service operation. The operation *createFolder* needs 2 parameters of type String and returns a parameter of type int.

```
<wsdl:message name="createFolderRequest">
     <wsdl:part name="journalID" type="xsd:string"/>
     <wsdl:part name="folderName" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="createFolderResponse">
     <wsdl:part name="createFolderReturn" type="xsd:int"/>
   </wsdl:message>
```

- **portType**
  The portType element includes a set of operations that are supported by this service, here the supported operation of eJournalFolderService is *createFolder*. Each operation includes the input and the output messages of the operation.

```
<wsdl:portType name="EJournalFolder">
  <wsdl:operation name="createFolder" parameterOrder="journalID folderName">
     <wsdl:input message="impl:createFolderRequest" name="createFolderRequest"/>
     <wsdl:output message="impl:createFolderResponse"
        name="createFolderResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

- **binding**
  The binding element in the Web service's description file describes the supported protocol. It contains information on what protocol is being

used. For SOAP protocol, the binding is <soap:binding> and the transport is SOAP messages on top of HTTP protocol.

```
<wsdl:binding name="eJournalFolderSoapBinding" type="impl:EJournalFolder">
   <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="createFolder">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="createFolderRequest">
         <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://folder.service.eJournal.sti.epfl.ch" use="encoded"/>
      </wsdl:input>
      <wsdl:output name="createFolderResponse">
         <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://folder.service.eJournal.sti.epfl.ch" use="encoded"/>
      </wsdl:output>
   </wsdl:operation>
</wsdl:binding>
```

- **service**
  The service element describes where to access the service and through which port to access the Web service.

```
<wsdl:service name="EJournalFolderService">
   <wsdl:port binding="impl:eJournalFolderSoapBinding" name="eJournalFolder">
      <wsdlsoap:address location="http://localhost:8080/axis/services/eJournalFolder"/>
   </wsdl:port>
</wsdl:service>
```

4.3. WSDD

The WSDD (Web Service Deployment Descriptor) is an XML file containing the deployment descriptor for statically configuring the Axis engine. A deployment descriptor contains information about what we want to make available to the Axis engine. This is proprietary to Axis. Axis uses the information contained in this deployment descriptor to track operations and type mappings.
For each WSDL file, deployment descriptors (deploy.wsdd, undeploy.wsdd) are created. They are used by the AdminClient to deploy the service (make the service available).

This is the deploy.wsdd file for the Folder Service:

```
<deployment xmlns=http://xml.apache.org/axis/wsdd/
         xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <!-- Services from EJournalFolderService WSDL service -->
  <service name="eJournalFolder" provider="java:RPC" style="rpc" use="encoded">
      <parameter name="wsdlTargetNamespace"
value="http://folder.service.eJournal.sti.epfl.ch"/>
      <parameter name="wsdlServiceElement" value="EJournalFolderService"/>
      <parameter name="wsdlServicePort" value="eJournalFolder"/>
      <parameter name="className"
         value="ch.epfl.sti.eJournal.service.folder.EJournalFolderSoapBindingSkeleton"/>
      <parameter name="wsdlPortType" value="EJournalFolder"/>
      <parameter name="allowedMethods" value="*"/>
      <typeMapping xmlns:ns="http://folder.service.eJournal.sti.epfl.ch"
```

```
        qname="ns:ArrayOf_tns2_Fragment" type="java:ch.epfl.sti.eJournal.core.Fragment[]"
        serializer="org.apache.axis.encoding.ser.ArraySerializerFactory"
        deserializer="org.apache.axis.encoding.ser.ArrayDeserializerFactory"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </service>
</deployment>
```

Essentially this WSDD is describing:

- The name of the service, e.g., eJournalFolder.
- The class that handles the service, `ch.epfl.sti.eJournal.service.folder.EJournalFolderSoapBindingSkeleton`.
- Allowed methods: The methods that are to be made available to clients of the Web Service. Here the * means all of the public methods of the class declaration. If we wish to restrict which methods are available, we could instead provide a space-separated or a comma-separated list of the names of the methods that are to be made available (if some are still in construction for example)
- As Axis has to map between Java and XML types the type mapping tells Axis which serializer has to be used for serializing objects like Fragment[] so that it can be sent with the SOAP protocol.

5. Possible implementations

Web services can be implemented in numerous ways. Having chosen Axis as a web service container there still are different ways to create web services.

5.1. Instant deployment with JWS:

The simplest and most straightforward way to deploy a web service is to use instant deployment. Simply rename the implementation .java file to .jws and place it in the web service container application. JWS stands for Java Web Service. If you have a .java file named EJournalServiceFolder.**java** you can rename it to EJournalServiceFolder.**jws**, place it in the web service container (in the webapps directory) and Axis will autodeploy the application. You can access the WSDL contents of the Web service as http://localhost:8080/axis/EJournalServiceFolder.jws?wsdl, it is then compiled and executed. The service is ready and clients for this service can be written using the WSDL description.
Unfortunately this is only for very simple applications or the very first web service creation. One of the problems of using this technique is that you have to use simple data types and common Java classes such as String, int etc. as method arguments and method return types (an exact mapping list between Java and XML types is available on the Axis Web site[9]). We cannot use user-defined classes which is not very practical as we need to use complex objects like the `User` class as method parameters or return types. Also the use of packages is not possible with this implementation. So a more complex way of

creating web services was chosen: Custom deployment using Axis tools, as will be described in the next chapter.

5.2. Custom deployment using Axis tools

Using the tools provided by Axis seemed a good start into web services and how to deploy a web service using these tools will be described in the following chapters.

6. Implementation

In this chapter I will describe first how to provide a web service and then how to become a client of a web service. My guideline shows the creation of the eJournalFolderService but also applies to the creation of the other services. I assume here that the services have been written with the interfaces that contain the future web service operations and the implementation classes, all in working order according to section 2.

6.1. Providing a web service

6.1.1. General Structure

The main package is `ch.epfl.sti.eJournal.service` and contains the following subpackages with:

```
.authentication
.fragment
.folder
.journal
.communication
```

I chose to create a package for each service because for each service several files will be created so it seemed to be more logical to create a package for each service.

Structure in Eclipse:

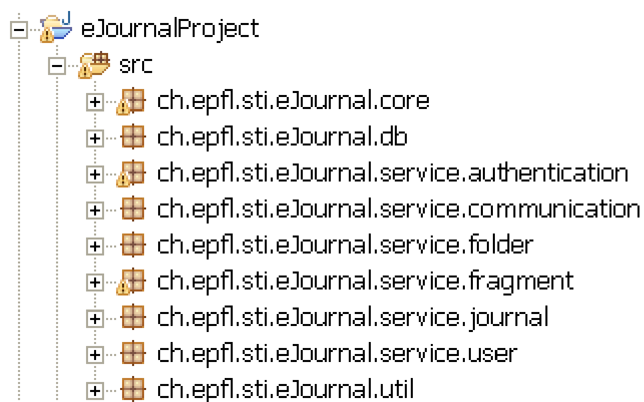


Figure 4 : Structure in Eclipse

The `.core` package contains all the user defined classes (all objects like the objects `User or Fragment`) and some constants.
The `.db` package contains all classes used to access the MySQL database
The `.util` contains utility classes like a Calendar class etc.

6.1.1.1. Java2WSDL[10]

To create the classes that are needed to provide a web service we use the Axis command line tool Java2WSDL:

```
java -cp %AXISCLASSPATH% org.apache.axis.wsdl.Java2WSDL
-o eJournalFolder.wsdl
-l"http://localhost:8080/axis/services/eJournalFolder"
-i ch.epfl.sti.eJournal.service.folder.EJournalFolderImpl
ch.epfl.sti.eJournal.service.folder.EJournalFolder
```

-o defines the name of the output file
-l defines the url of the service
-i defines the implementation of the class
(Passing the implementation class means that method parameters will be named eg. journalID, newFolderName etc. instead of being unnamed like in0, in1)
and finally the interface is passed as an argument
This creates the WSDL document: `eJournalFolder.wsdl`

6.1.1.2. WSDL2Java [10]

Once the WSDL document is created we use WSDL2Java to create the files needed to provide a Web Service:

```
EJournalFolder.java
EjournalFolderImpl.java
EJournalFolderSoapBindingImpl.java
EJournalFolderBindingSkeleton.java
```

The skeleton class is the class that sits between the Axis engine and the actual service implementation, it gets the web service client requests and sends the responses in SOAP (hence the “SOAP” in the name). The SoapBinding class the calls the implementation classes to execute the request. The WSDL2Java also generates the deployment descriptors needed to deploy the service, these can then be adapted further (see section 4.3 and section 6.3)

```
deploy.wsdd
undeploy.wsdd
```

```
java -cp %AXISCLASSPATH% org.apache.axis.wsdl.WSDL2Java
-o ..\src
-s
-S true
-Nurn:eJournalFolder
ch.epfl.sti.eJournal.service.authentication eJournalFolder.wsdl
```

-o is the output directory
-s, --server-side emit server-side bindings for web service
-S, --skeletonDeploy: for emitting server.side bindings
Assumes --server-side.
-Nurn maps a namespace to the package

Java2WSDL requires all user defined classes that are parameters or return types of a service operation to model a JavaBean, requiring each class to have a default constructor and getter/setter methods. For more information and problems encountered due to this see section 7.

Executing WSDL2Java created the file EJournalFolderSoapBindingImpl.java
By default, this file contains an empty implementation for the service. Manual coding is still required to connect the server-side implementation code to the existing Java application.

```
package ch.epfl.sti.eJournal.service.folder;

public class EJournalFolderSoapBindingImpl implements
ch.epfl.sti.eJournal.service.folder.EJournalFolder{
//this line has to be added
      EJournalFolder folder = new EJournalFolderImpl();

public int renameFolder(java.lang.String folderID, java.lang.String
newName) throws java.rmi.RemoteException {
// this line has to be changed from return –3; to the following
        return folder.renameFolder(folderID,newName);
    }
```

The SOAPBindingImpl class is not really needed, you could also let the Skeleton class call your implementation classes directly.
This Binding class is not deleted if you repeat the operation.

Once the SoapBinding class has been changed, all the classes need to be compiled and placed in the axis services directory in Tomcat:
%TOMCAT_HOME%\webapps\axis\classes or lib (in the classes directory for compiled classes, in the lib directory for jar files containing the compiled classes).

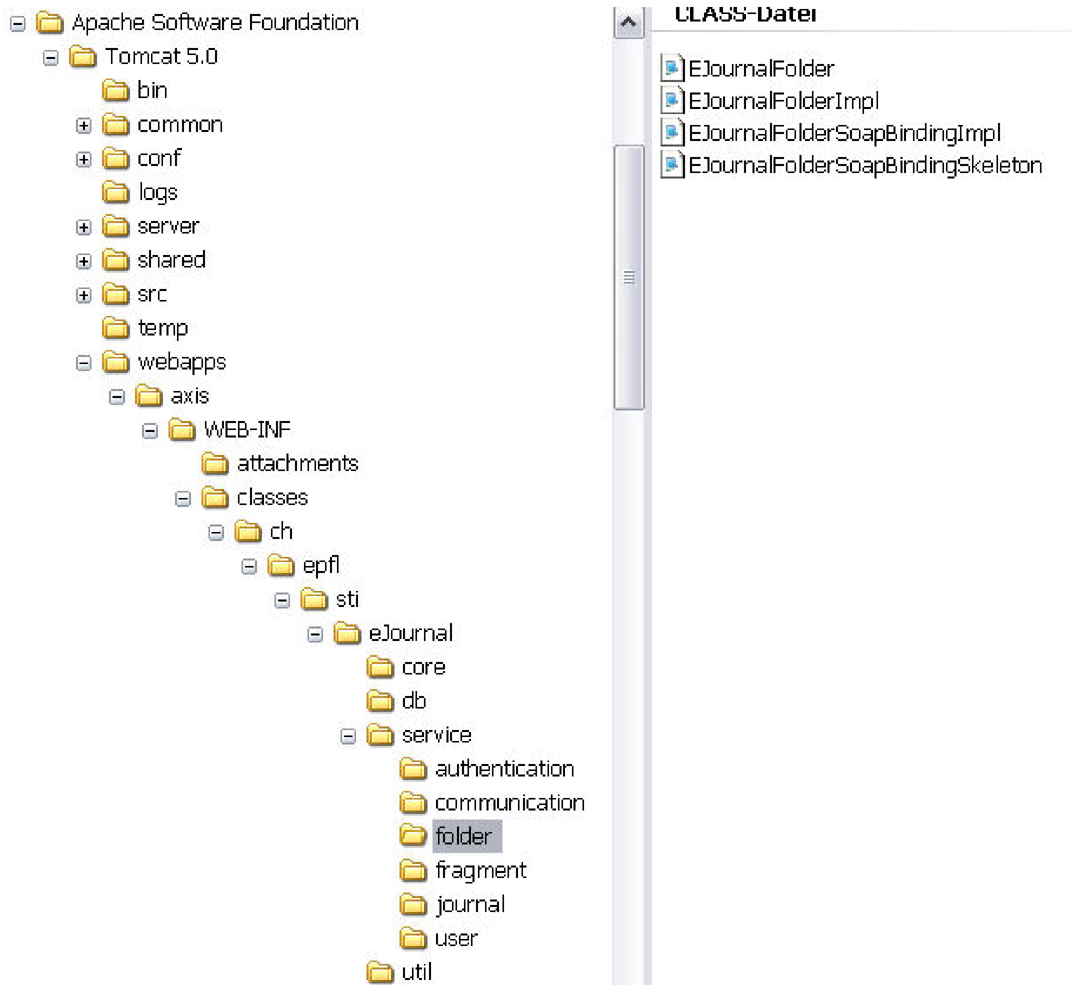


Figure 5: Compiled classes in Tomcat/Axis

6.1.2. Deploying the web service

In the previous section WSDL2Java created the files needed to deploy and undeploy the services: `Deploy.wssd` and `undeploy.wsdd`

Axis provides a deployment tool, AdminClient and with the following command we can deploy the service:

```
java –cp %AXISCLASSPATH% org.apache.axis.client.AdminClient
ch\epfl\sti\eJournal\service\folder\deploy.wsdd
```

(use undeploy.wsdd to undeploy the service)

Then if we go to our services with our browser (http://localhost:8080/axis/services) we should see our service deployed. By clicking on the WSDL link we can see the associated WSDL file.
The web service is now functioning.

Figure 6 : deployed services

6.2. Creating a web service client

There are many ways to create a web service client, it could be JSP pages or Java clients or even clients in other programming languages.
This approach shows how to create a client using the tools provided by Axis and the web service description file.
The advantages of using this approach are:

- Provide type-safe invocation of web services
    - rather than passing parameters as an array of Objects, calls are made using an interface that specifies parameter types
    - rather that receiving an Object result, the interface specifies the actual result type
- Types are already checked at compile-time, not at execution

6.2.1. WSDL2Java

Apache Axis provides the WSDL2Java utility for creating the client proxy code. So we can execute WSDL2Java on the eJournalFolder.wsdl file like described in section 6.1.1.2 without emitting server side bindings.
These are the files needed to create a client of the Web Service:

```
eJournalService.java
eJournalFolderService.java
eJournalFolderServiceLocator.java
eJournalFolderSoapBindingStub.java
```

The BindingStub class is the counterpart of the SkeletonStub class on the server side. This class invokes the web services on the server by sending requests in SOAP. The Locator class locates an implementation of the service using the URI specified in the WSDL. The Service class returns an instance of Folder on which the operations can be called.

6.2.2. create the client (Java)

This is a very simple example client using the eJournalFolderService and the rename operation:
After running the command WSDL2Java a package containing the above mentioned files is created which can now be imported in our client code and invoked as a service.

```
package eJournalClient;

import ch.epfl.sti.eJournal.service.folder.*;

public class FolderClient {

   public static void main(String[] args){

   // Make a service
```

```
        EJournalFolderService service = new
              EJournalFolderServiceLocator();
        EJournalFolder folder = service.geteJournalFolder();
        int result = folder.renameFolder("1","newName");
        System.out.print("Result: " + result);
    }
}
```

6.2.3. create the client (JSP)

The JSP client can be created by getting the WSDL description, in this case eJournalFolder.wsdl, executing WSDL2Java and compiling the 4 needed classes (Interface, Service, ServiceLocator and BindingStub). Then a .jar file containing the compiled classes is placed in the WEB-INF/lib directory of the JSP Project.
This is an example page for using the the Folder Service and calling the rename folder operation:

```
<%@page contentType="text/html"
    import="java.net.*,
        java.util.*,
        org.apache.soap.*,
        org.apache.soap.rpc.*,
        java.lang.*,
        ch.epfl.sti.eJournal.service.folder.*%>
<%!
String ResultMessage ="";
%>
<%          //here we get a folder service
EJournalFolderService folderService = new
EJournalFolderServiceLocator();
EJournalFolder folder = folderService.geteJournalFolder();
%>

<%
      //first we get the parameters entered in the JSP form
String folderNewName = (String)request.getParameter("folderNewName");
String folderIDRename = (String)
request.getParameter("folderIDRename");
      if ((folderNewName != null)&& (folderIDRename != null)){
            int result =
      folder.renameFolder(folderIDRename,folderNewName);
      ResultMessage = new Integer(result).toString();}
%>

<html>
<head><title>eJournal Folder</title></head>
<body>
<text> <b>Following services are available:</b></text>
 <br/>
<text> ----- Rename a folder (folder ID, newName) ------</text>
<form method="get" action="folder.jsp">
<input type="text" name="folderIDRename" id="folderIDRename"
      value=""/>
 <input type="text" name="folderNewName" id="folderNewName" value=""/>
 <input type="submit" value="Rename folder" />
```

```
</form>
 <br/>
<text>The result of your web services call is:<%=ResultMessage%>
</b></text>
<br/>
</body>
</html>
```

Below is the whole structure for the JSP client:

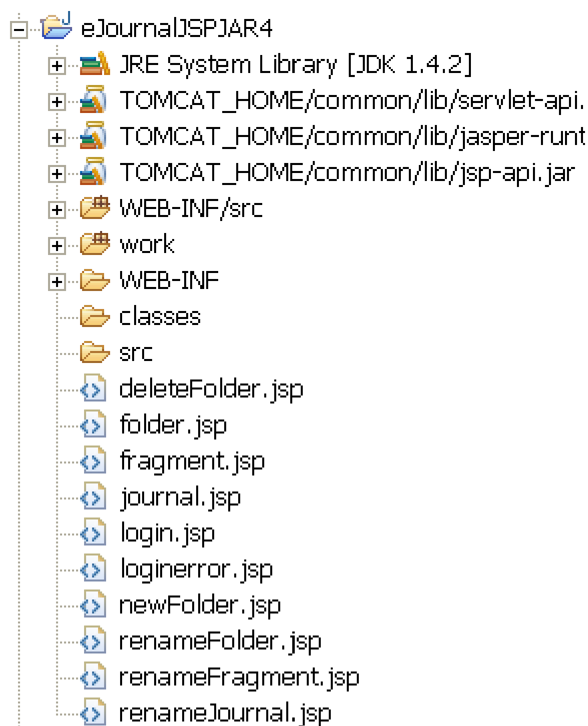


Figure 7: Implementation of the JSP Client

6.3. Soap Monitor

The Soap Monitor Applet provided by Axis can be used for monitoring of SOAP requests and responses via a web browser. For each service called it will show the SOAP message for the operation that was invoked and the response that was sent.
It needs to be compiled, installed and enabled[11].
Then the deployment descriptors of the services have to be changed and the following has to be added after the <service tag>:

```
<service name="eJournalFolder" provider="java:RPC" style="rpc"
use="encoded">
    <requestFlow>
       <handler type="soapmonitor"/>
    </requestFlow>
    <responseFlow>
       <handler type="soapmonitor"/>
     </responseFlow>
```

. . .

The service has to be undeployed and redeployed using the modified wsdd file. Then using a web browser we can go to http://localhost:8080/axis/SOAPMonitor and view the SOAP requests and responses in the applet.

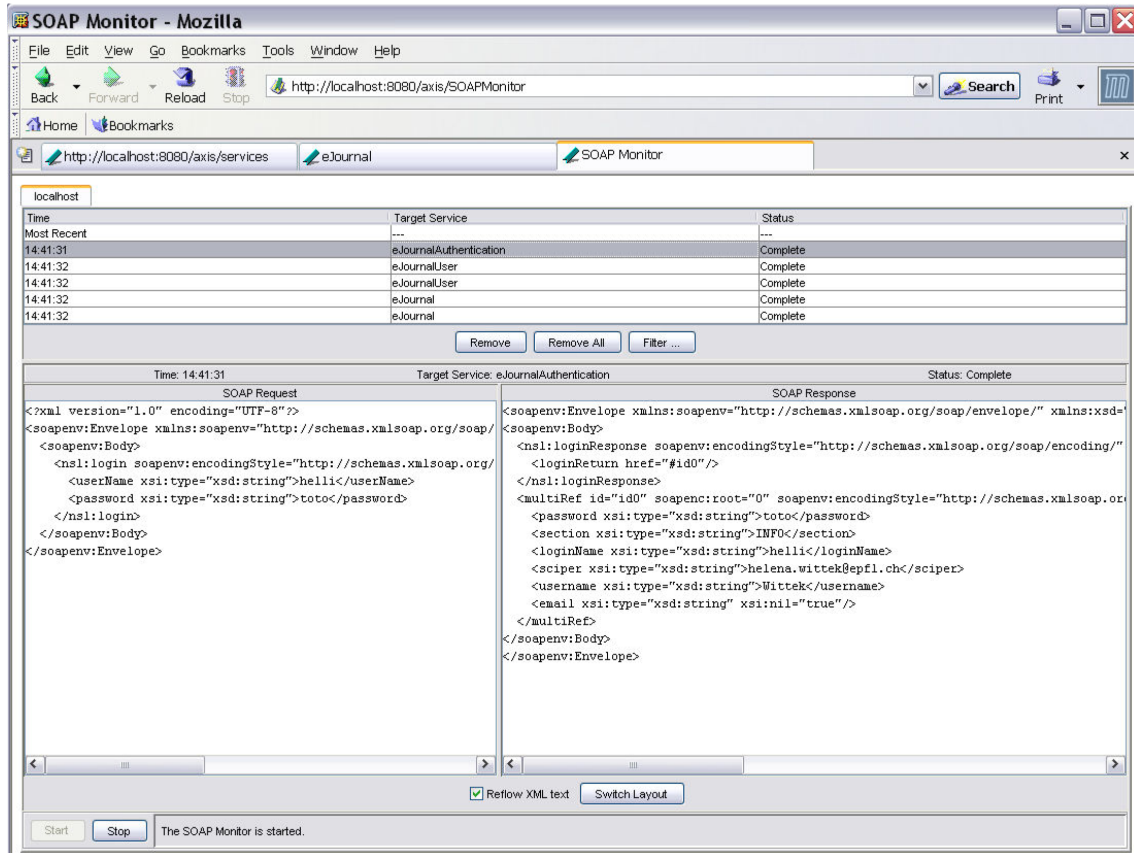


Figure 8: SoapMonitorApplet with eJournalAuthentication Service

7. Problems encountered

This chapter describes the problems I encountered while creating the web services and possible solutions.

7.1. User defined Types

Axis maps between Java types and XML Schema types so it can only process user-defined Java objects if there is a registered Axis serializer for it.
When using Axis it is mandatory that all user-defined types that are parameters or return types of web service operations model a Java bean by having a default public constructor and getter/setter methods. They cannot contain any other methods so all other methods contained in those classes must be moved elsewhere. This was the case for the fragment class which contained some additional methods, these had to be moved to a `FragmentOperations.java` class.

7.2. ArrayLists

One problem encountered was that Axis could not create corresponding types for the java.collections type ArrayList. This is because Axis needs to reach building blocks (standard types or user-defined types) but an ArrayList is a list that contains instances of Object and these are not building blocks. So a Java Array can be of type String[] or User[] but not a java.util collections type like ArrayList or Vector.
As methods in eJournalService and eJournalFolderService return ArrayLists they had to be changed to return objects like User[] or Folder[] and corresponding java beans have to be created (if we would have left them as ArrayLists they would have got serialized/deserialized as Object[]).
One solution to this problem would be to write custom serializers/deserializers. Once the serializers written we can tell Axis which types they should be used for by using a typeMapping tag in the deployment descriptor , which looks like this:

```
<typeMapping qname="ns:MyJavaClass" xmlns:ns="urn:SomeService"
languageSpecificType="java:my.java.MyJavaClass"
serializer="my.java.Serializer"
deserializer="my.java.DeserializerFactory"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
```

7.3. The Fragment Class

The Fragment class is an abstract class which is supposed to model other classes like External Fragment. This caused some problems to make this class compatible with web services.

7.3.1. Abstract classes

It was possible to create a WSDL file for the abstract Fragment class and the corresponding client and server classes, but at execution there was a problem creating the bean class of the abstract class ("missing the default constructor" error) that could not be resolved.

7.3.2. Protected constructors

As already mentioned before, all user-defined types used as parameters or return types for a web service methods(like User or Folder) must model Java beans. This means they must have a default public constructor. For Fragment, having a protected constructor, the WSDL file could not be created. The structure of fragment had to be changed to use a public constructor.

7.3.3. fragState

Fragment has an attribute representing its state: FragState of Type FragmentState. The class FragmentState being an interface (for implementing other Fragment States like FragmentNormalState etc.) does not have a default (public) constructor which means that a WSDL file cannot be created. The implementation has thus to be changed to be able to make the class Fragment web service compatible.

8. Further development

In this chapter I will describe further possible developments of this project and improvements that could be applied to the services.

8.1.1. MVC

The Model – View – Controller paradigm is architectural design pattern for interactive applications. Its goal is to separate business model functionality from the presentation and control logic. The model object holds the application's data and knows all about the data that is displayed. It also knows about all the operations that can be applied to transform that object,
The view obtains data from the model and then displays the information.
The controller handles events that change the model or the view.
An MVC structure allows greater flexibility, more possibilities for reuse and better maintainability.
Up to now the JSP clients called the services directly from the JSP pages. To make the code easier to maintain and reusable it would be better to use the MVC paradigm. This can be implemented by using for example STRUTS, as explained in the next chapter.

8.1.2. STRUTS (An MVC framework)

STRUTS[12] is an open source framework for building Java web applications proposed by the Apache Software Foundation and based on the MVC paradigm.

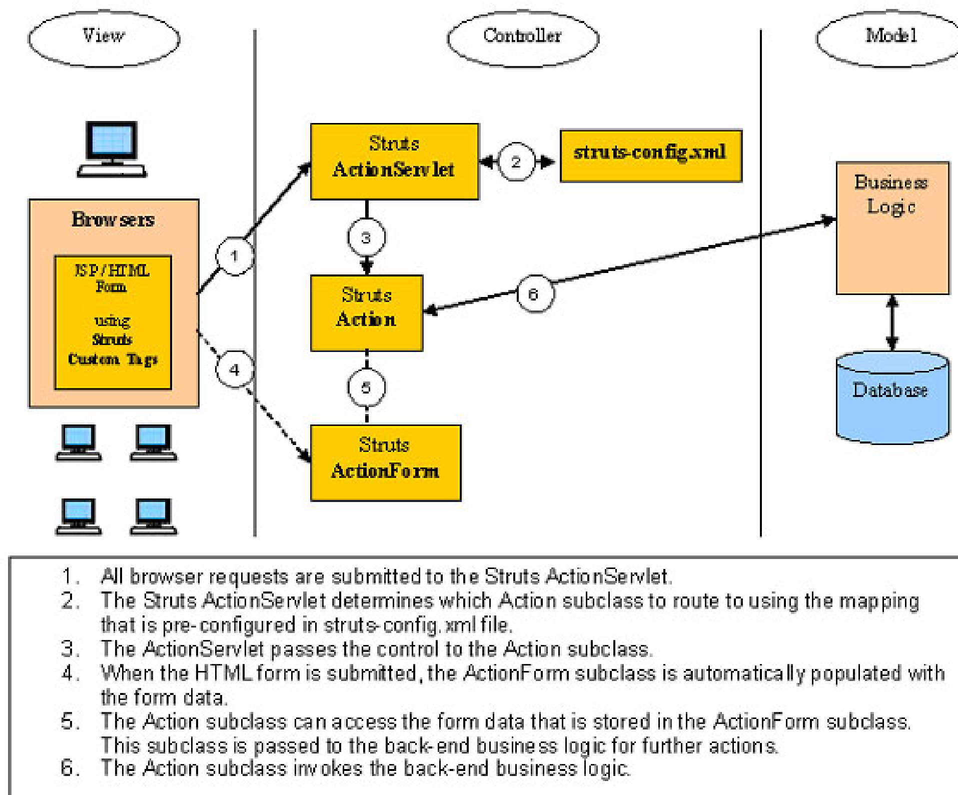


Figure 9: STRUTS Architecture

STRUTS uses ActionForms to model the data. These are represented as a JavaBean with getter/setter methods. The View is implemented using JSP pages but this time they only display data or collect data, they do not call the web services directly.
The ActionForm (representing the controller) will then get the data from model (the ActionForm) and call the web services to collect data and update the model or the view.

8.1.3. Implementation

Here is an example of how to use STRUTS with the web services that we provided, we become once again the client of our web services:
For the implementation of the AuthenticationService we have a JSP page with a form containing 2 text boxes and a submit button. The LoginForm is a Java Bean with 2 attributes: username and password. In the struts-config.xml file the

login action from the JSP page is mapped to the LoginAction.java file with the parameters of the JSP page retrieved by the LoginForm.
For using the web service functionalities I have retrieved the WSDL file and used the Java2WSDL command to create the client side stubs. These are in the `ch.epfl.sti.eJournal.service.authentication` and the `ch.epfl.sti.eJournal.core packages`. (the core package was created because we want to return the user defined type User).
Having the client stubs we can implement the Action class.
In the Action class we retrieve the values from the LoginForm class (password and username) and retrieve the authentication service as already explained in section 6.2.2 . We call the authentication service with the password and the username and are able to update the JSP page according to the result of the web service call.
Now the logic has been separated from the view and the eJournal enhanced with web services becomes even more maintainable and flexible.

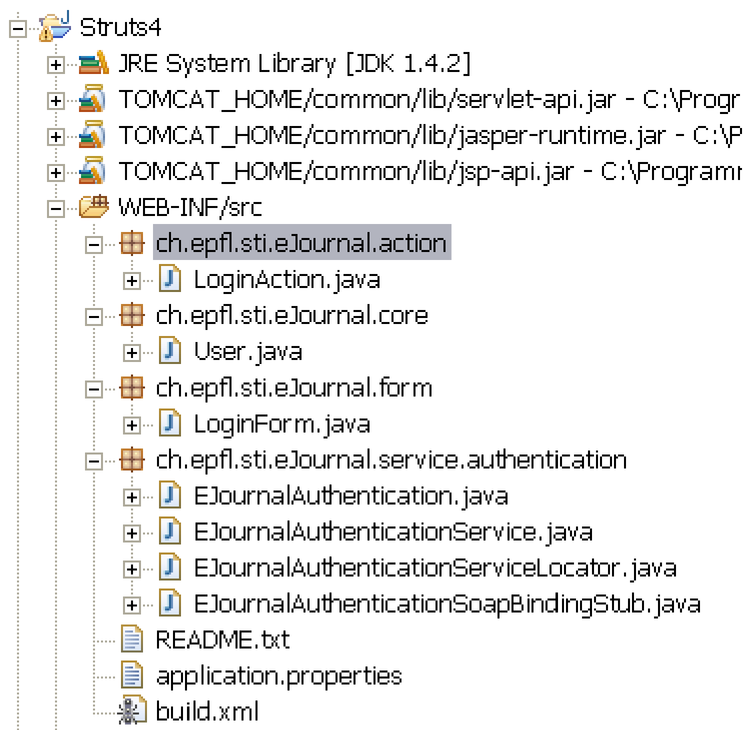


Figure 10: Struts implementation

8.2. Custom development

Having started into web services using the tools provided by Axis, we see that towards the end of this project the ways to create web services for complex projects like the eJournal are becoming limited. We need to look more deeply into more custom ways of creating web services, especially to remedy to problems such as the Fragment class. One approach would be to write custom serializers to be able to use complex objects in the services.

8.3. Integration of other services

As this paper describes how to become a web service client we could also imagine that the eJournal could become a client of other web services that would enhance its functionalities.

8.4. Integration of the eJournal web services into other applications

The ultimate goal of creating web services is to make these available for others. Now that the services are created any other entity (for example another EPFL laboratory) could reuse the services in their applications, for creating an eJournal-like application on their own. Of course the question of security and disk space sharing will then have to be considered but integrating web services in the eJournal allows more users to benefit from the whole eJournal structure.

9. Conclusion

Enhancing the eJournal with web services shows a lot of possibilities for the future. By creating three different clients we have shown the reusability of the web services. This functionalities will allow more users to benefit from the eJournal regardless of what operating system their using or which language their application is in.
As we know interoperability between platforms and different technologies becomes more and more a challenge in today's world and this project tries to show some solutions to this problem.

I found it very interesting and challenging trying to create web services.
The project involves a lot of different technologies eg. WSDL, SOAP, Java, JSP that need to be understood and have to work together, which has not been easy at times.
The web services technology is still quite new so there isn't much documentation in books or on the Internet and sometimes problems can be hard to resolve. Especially as there are only small web service examples, using basic types, which makes it difficult to find documentation for larger problems like user-defined classes, serialization problems etc.
Still it provides an interesting view of the future and what can be done to unite different programming concepts and languages as well as letting others benefit from the developments.

10. Index

Web services for the eJournal **1**
1. Introduction **3**
1.1. Web services for the eJournal 4
2. Services to be implemented **5**
2.1. Preliminaries 5
3. Software and components **6**
3.1. Tomcat 6
3.2. Axis 6
3.2.1. Installation 6
3.2.2. Features 7
3.3. MySQL 7
3.4. Eclipse & plugins 7
3.5. General Configuration 8
4. Web Services Basics **9**
4.1. SOAP 9
4.2. WSDL 9
4.3. WSDD 11
5. Possible implementations **12**
5.1. Instant deployment with JWS: 12
5.2. Custom deployment using Axis tools 13
6. Implementation **13**
6.1. Providing a web service 13
6.1.1. General Structure 13
6.1.2. Deploying the webservice 16
6.2. Creating a web service client 18
6.2.1. WSDL2Java 18
6.2.2. create the client (Java) 18
6.2.3. create the client (JSP) 19
6.3. Soap Monitor 20
7. Problems encountered **22**
7.1. User defined Types 22
7.2. ArrayLists 22
7.3. The Fragment Class 22
7.3.1. Abstract classes 22
7.3.2. Protected contructors 23
7.3.3. fragState 23
8. Further development **23**
8.1.1. MVC 23
8.1.2. STRUTS (An MVC framework) 23
8.1.3. Implementation 24
8.2. Custom development 25
8.3. Integration of other services 26
8.4. Integration of the eJournal web services into other applications 26
9. Conclusion **26**
10. Index **27**
11. Figure Index **28**
12. References **28**

11. Figure Index

Figure 1 : Folder Structure 7
Figure 2: WSDL 9
Figure 3 : WSDL of EjournalFolderService in XMLSpy 9
Figure 4 : Structure in Eclipse 13
Figure 5: Compiled classes in Tomcat/Axis 16
Figure 6 : deployed services 17
Figure 7: Implementation of the JSP Client 20
Figure 8: SoapMonitorApplet with eJournalAuthentication Service 21
Figure 9: STRUTS Architecture 24
Figure 10: Struts implementation 25

12. References

---

[1] http://emersion.epfl.ch/index.html
[2] http://jakarta.apache.org/tomcat/
[3] http://ws.apache.org/axis/java/install.html
[4]Download page for MySQL Server http://dev.mysql.com/downloads/
[5] Download page for Eclipse: http://www.eclipse.org/downloads/index.php
[6] http://www.sysdeo.com/eclipse/tomcatPlugin.html
[7] http://www.improve-technologies.com/alpha/axis/
[8] http://www.myspotter.com/wsdl2java.shtml
[9] Axis type mappings: http://ws.apache.org/axis/java/user-guide.html
[10]For more info on Axis and its tools see: http://ws.apache.org/axis/java/user-guide.html
[11] http://ws.apache.org/axis/java/install.html#soapmon
[12] For more information on STRUTS see http://struts.apache.org/

The result of this project should be seen as a working prototype for some of the services not as a complete result, the guidelines for creating the services should save time and avoid frustration for further development.