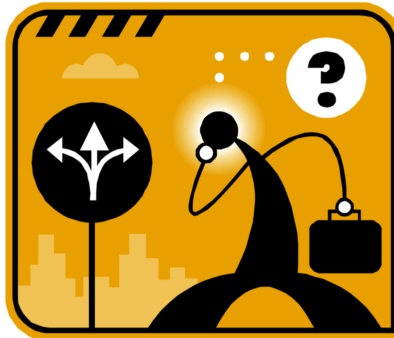
Project no. 033572

CASPAR

Cultural, **A**rtistic and **S**cientific knowledge for **P**reservation, **A**ccess and **R**etrieval

Instrument: Information Society Technologies

Thematic Priority : 2.5.10 Access to and preservation of cultural and scientific resources



CASPAR KEY COMPONENTS BEST PRACTICES

A simple development How-To and State of the Art for developers from the CASPAR Architecture Team

Due date of deliverable: 15 December 2007

Actual submission date: 11 June 2008

Start date of project: 1 April 2006 Duration:42 months

Organisation name of lead contractor for this deliverable: ACS

Version 0.8

Work-Package WP3100

Document Number

Delivery Type Report

Author(s) MF – Marco Fulcoli (ACS) m.fulcoli@acsys.it

LB – Luigi Briguglio (ENG) luigi.briguglio@eng.it

PA – Pasquale Andriani (ENG) pasquale.andriani@eng.it

SC – Steve Crothers (STFC) s.r.crothers@rl.ac.uk

CP – Claudio Prandoni (MW) c.prandoni@metaware.it

MV – Marlis Valentini (MW) m.valentini@metaware.it

LV – Loredana Versienti (CNR) loredana.versienti@isti.cnr.it

HA – Henri Avancini (CNR) henri.avancini@isti.cnr.it

DA – Dimitris Andreou (FORTH) andreou@csd.uoc.gr

Approval

Summary Documentation about three different ways to develop the CASPAR Key Components. Finally a CASPAR Best Practice is provided.

Keyword List Axis2, Spring Framework, Inversion of Control, Configuration, Spring-WS, JAX-WS, Eclipse IDE, Glassfish, Apache Tomcat 5.5, CASPAR Best Practice, Rich Client Applications

Availability ☑ PUBLIC

☐ LIMITED TO EU PROGRAMME DISTRIBUTION

☐ LIMITED TO CASPAR CONSORTIUM DISTRIBUTION

Change History

Issue	Date	Status	Author	Description
0_1	11-Dec-07	Draft	ALL	Creation with existing contributions and submission to ACS
0_2	14-Dec-07	Draft	ALL	Revision with feedback from ACS
0_3	19-Dec-07	Draft	ALL	CASPAR Best practise completed and submission to CASPAR Consortium
0_4	21-Dec-07	Draft	ALL	Typos corrections
0_5	04-Jan-08	Draft	ALL	Added notes for Tomcat Server
0_6	23-Jan-08	Draft	ALL	Added ServiceFactory proposed from DA. Added appendix for SVN structure and build files description.
0_7	08-Feb-08	Draft	ALL	Small correction of Advanced Client Solution.
0_8	11-Jun-08	Draft	ALL	Availability Public

Table of Contents

1. Introduction 4

1.1 How To Read This Document 5

1.2 Experiences Premise 5

2. Developing Web Services with Axis2 6

2.1 Development Environment 6

2.2 Development Process 6

2.2.1 Approach 6
2.2.2 Deployment overview 6
2.2.3 Development process workflow 6

2.3 How-To step by step 7

2.3.1 Server Side 7
2.3.2 Client Side 9

3. Developing Web Services with JAX-WS 11

3.1 Development Environment 11

3.2 Development Process 11

3.2.1 Approach 11
3.2.2 Deployment overview 11
3.2.3 Development process workflow 12

3.3 How-To step by step 12

3.3.1 Server Side 12
3.3.2 Client Side 15

4. Developing Web Services with Spring-WS 17

4.1 Development Environment 17

4.2 Development Process 17

4.2.1 Approach 17
4.2.2 Deployment overview 17
4.2.3 Development process workflow 18

4.3 How-To step by step 18

4.3.1 Server Side 18
4.3.2 Client Side 28

5. Comparing the 3 ways 30

6. The CASPAR Best Practice 31

6.1 Server Side 33

6.2 Client Side 34

6.3 Advanced Client Solution: ServiceFactory 35

7. Future Work 37

References 38

Appendix 1 – SVN Structure and Build Files 39

1. Introduction

The **CASPAR** Architecture Team has identified and specified in [D1301] eleven **CASPAR** Key Components which provide digital preservation functionality, according to the OAIS Functional Model.

After the first specification, a refinement specification has been done in order to add more details to the conceptual models of each **CASPAR** Key Component and to identify common interfaces and functionalities.

After the specification of the **CASPAR** Key Components, the **CASPAR** Consortium has defined the Framework Architecture as [D3101] and investigated practical solutions about implementation aspects from the current state of the art.

That analysis has identified a set of suitable solutions and choices such as:

- A first implementation of the **CASPAR** Key Components can be provided adopting the web service paradigm and by using Axis as suitable soap engine;

- **CASPAR** Architecture Team has identified and modelled in the refined specifications a set of interfaces and abstract classes for common functionalities such as: registration and unregistration of the **CASPAR** Key Component, component descriptive information, initialisation. In this perspective, those models represent a common layer for the integration of the **CASPAR** Key Components;

- **CASPAR** Architecture Team agrees on focusing aspects concerning the business logic of the **CASPAR** Key Components rather than focusing on underlying framework, which is usually “more evolvable and less preservable”. In this perspective, **CASPAR** Architecture Team has identified the Spring Framework as a practical solution to integrate **CASPAR** Key Components;

- During the analysis of the state of the art, the **CASPAR** Architecture Team has identified a new promising standard to support JAVA Web Services. It’s JAX-WS and is the evolution of the well known and widely adopted JAX-RPC. In this perspective, the **CASPAR** Architecture Team has analysed the possibility to adopt JAX-WS for **CASPAR** Key Components development.

For the above reasons, the **CASPAR** Architecture Team has involved a three small groups in order to test and evaluate the three suitable ways to develop web services:

1. Developing Web Services with *Axis2*
2. Developing Web Services with *JAX-WS*
3. Developing Web Services with *Spring-WS*

This document gathers the experiences done from the three groups, providing detailed information for starting development of web services with solutions from the current state of the art.

Each section of this document identifies the tools and the environment adopted for the experiences and the steps to be done to successfully compile, deploy and test a simple sample web service. Moreover, in each section will be shown the same “`Sample`” web service, which essentially exposes a method inherited from the common `CASPAR` interface (i.e. `getComponentDescInfo`). That will allow the reader to understand the issues/advantages coming from the adoption of the specific approach.

After the presentation of all of the three ways, the **CASPAR** Architecture Team provides a comparison of them, and finally shows the chosen solution in the **CASPAR** Best Practice section and identifies a suitable road map for future work.

1.1 How To Read This Document

There are mainly two ways to read this document:

1. Read it from the first page to the last one – This is the longest way, but it can give you a complete idea of the experiences we have done;
2. Read only the section “The CASPAR Best Practice” – This is the shortest way, and allows you to understand a best practice to develop your component, by following the sample code and steps.

You are free to choose your favourite way. *Good reading!*

1.2 Experiences Premise

The three development experiences have been done starting from common conditions:

- All the models are defined in the **CASPAR** shared project with the tool Enterprise Architect, which has allowed to generate Java code for interfaces and classes;
- The web service used for the experience is the implementation of the interface `Sample`, which exposes the method `getComponentDescInfo;`
- The common models adopted for the experiences are shown in the picture below:

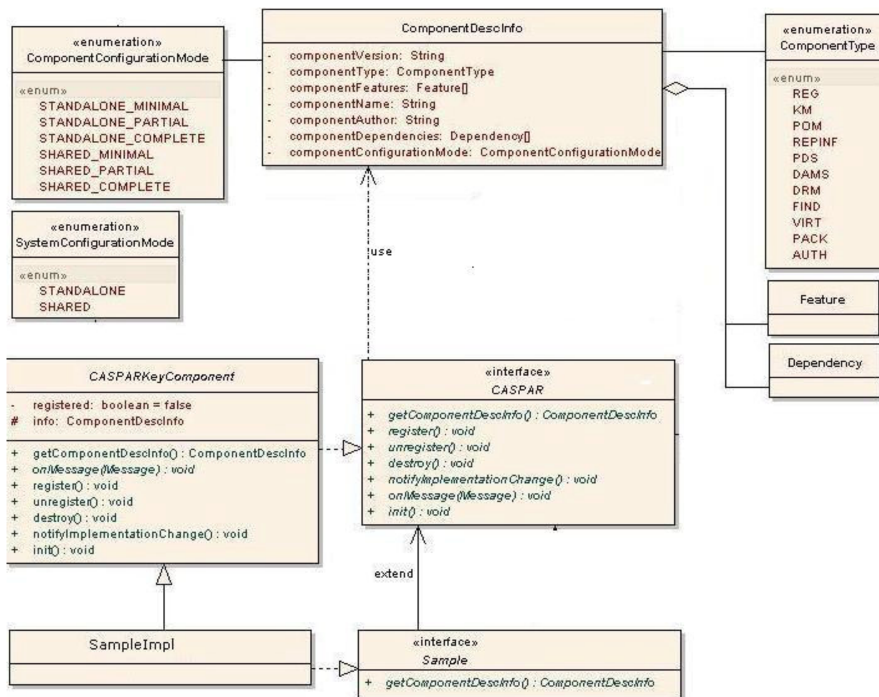


Figure 1 - CASPAR Common models for the experiences

2. Developing Web Services with Axis2

2.1 Development Environment

In this section it's explained how to create, deploy and invoke a Web Service by using Axis2 Soap Engine. The used technologies are:

- IDE - Eclipse 3.3 Europa - http://www.eclipse.org/europa/
- Application Container - Tomcat 5.5.23 - http://tomcat.apache.org/
- Build tool - Ant 1.7.0 - http://ant.apache.org/
- JDK 1.5 – http://java.sun.com
- SOAP Engine - Axis2 - http://ws.apache.org/axis2/index.html
- Eclipse Plugin - Axis2 Codegen Eclipse Plug-in - http://ws.apache.org/axis2/tools/1_3/eclipse/wsdl2java-plugin.html

2.2 Development Process

2.2.1 Approach

First of all, it's necessary to generate a `WSDL` document starting from the implementation class. Next, by using the resulting `WSDL` the skeleton and stub codes are generated. The skeleton represents the server side code, allowing integration with the underlying framework. And the stub code represents the client side code, which depends from the chosen technology and framework. A `build.xml` file is created too, in order to build and manage (using Ant) client and server software artefacts.

2.2.2 Deployment overview

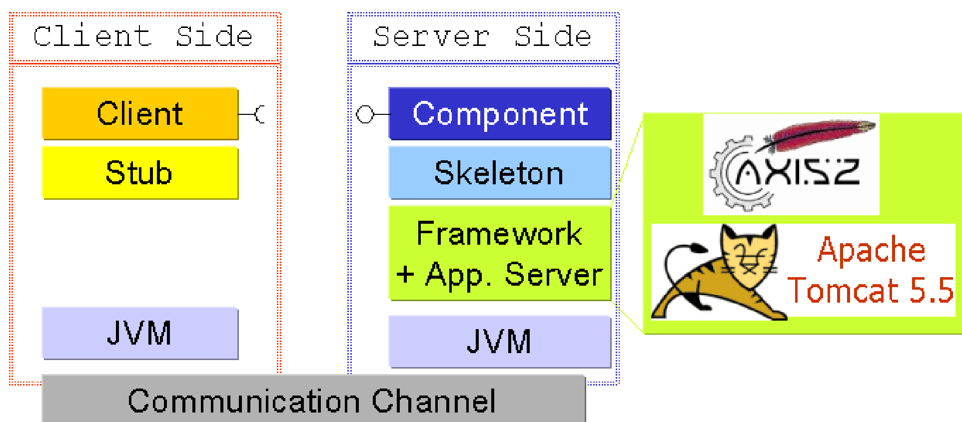


Figure 2 Web Service Deployment overview with Axis2

2.2.3 Development process workflow

In next figure is shown the development process used in order to build a web service with Axis2. Every step is then explained in the detail in next sections.

The BPMN diagram shows the overall development process followed for this experience with Axis2:

- the main actors, human (Developer) and automatic tools (Enterprise Architect, Axis2 Code Generator, Apache Ant Builder);
- the relative activities (i.e. from Generate Interface to Deployment);
- the artefacts (i.e. from Component Model to Component Archive).

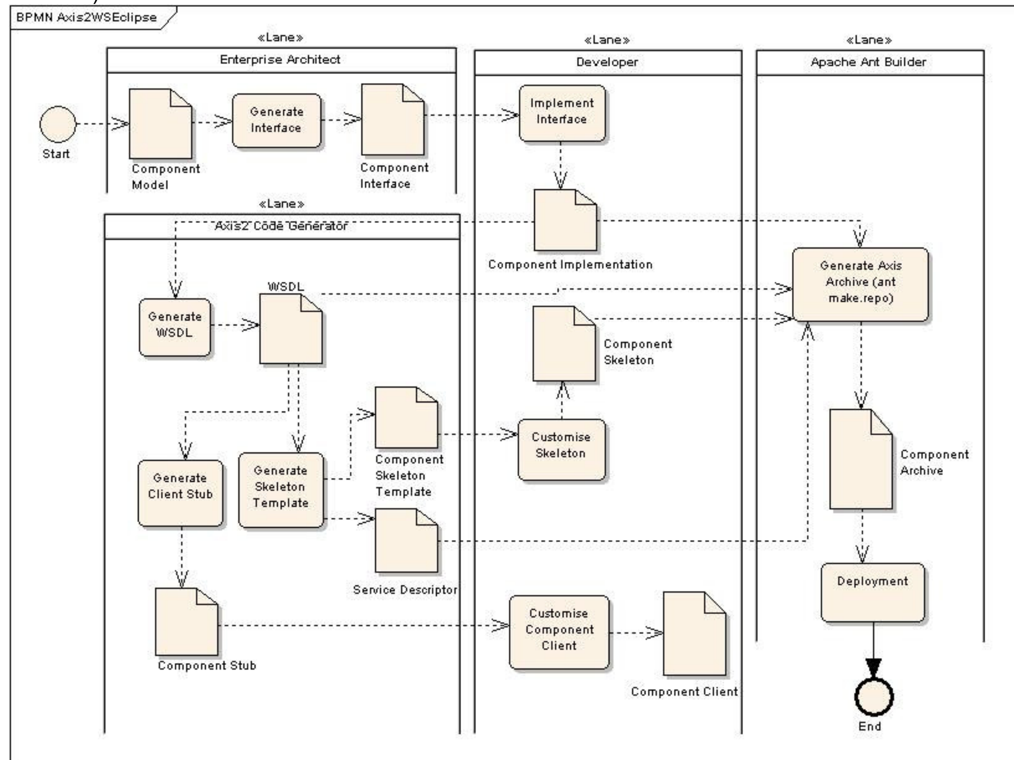


Figure 3 - Axis2 Development Process

2.3 *How-To step by step*

In this section the development process is explained, starting from the generation of `WSDL`, through server side code, client side code, to customization, deploy and invocation.

2.3.1 Server Side

Creation of WSDL

As explained in the Development Process section it's possible to start developing a web service directly with an implementation class. The current situation is that a `Sample` interface is just implemented in the class `SampleService`.

It could be possible to start from implementation class `SampleService` in order generate, through Axis2 Codegen, a valid `WSDL`.

It's important to remark that Axis2 CodeGen plug-in creates a `WSDL` without exposing inherited methods. So in this case the workaround is to override the inherited method `getComponentDescInfo` just calling the `super` method.

```
@Override
public ComponentDescInfo getComponentDescInfo(){
      return super().getComponentDescInfo();
}
```

Server side code

Again, by using Axis2 Codegen, the generated WSDL is used to generate server side code. It's enough to:

- Select option "Generate Java source code from a WSDL file"
- Click the "Browse…" button and select the "WSDL file location"
- Set the options as:
    - Codegen option = **custom**
    - Output language = **java**
    - Service Name = *insert here the name of your service*
    - Port Name = *choose the option for your service binding (i.e. HTTP, SOAP1.1, SOAP1.2)*
    - Databinding Name = **adb**
    - Custom package name = *insert here the package of skeleton classes*
    - Select "Generate server side code", "Generate a default service.xml" and "Generate an Interface for Skeleton"
- Select "Browse and select a project on current eclipse workspace"
    - The resulting artefacts are:
        - Skeleton Class Template
        - Skeleton interface
        - Parameter XSD Classes used for the request and response messages
        - services.xml - the service descriptor for deployment on Axis2

Skeleton Class Template is not implemented, so the following method is added in order to create an association between the Skeleton class and the original Implementation class.

```
private Service service;


private Service getService(){
      if (service == null)
            this.service = new ServiceImpl();
      return service;}
```

So each method can be implemented by invoking the `ServiceImpl` class. Unfortunately, it's not enough, because of each skeleton method must return a **Response* object, for instance `GetComponentDescInfoResponse`. So in the case of `getComponentDescInfo` skeleton method a translation from `ComponentDescInfo` business object to `GetComponentDescInfoResponse` is requested. The code is shown below.

```
	public eu.casparpreserves.
impl.sample.xsd.GetComponentDescInfoResponse getComponentDescInfo()
{
		SampleImpl n = getServiceInstance();
		ComponentDescInfo c =  n.getComponentDescInfo();

		GetComponentDescInfoResponse response = new
GetComponentDescInfoResponse();
		eu.casparpreserves.framework.xsd.ComponentDescInfo cxsd
= new eu.casparpreserves.framework.xsd.ComponentDescInfo();
		cxsd.setComponentAuthor(c.getComponentAuthor());
		cxsd.setComponentName(c.getComponentName());
		cxsd.setComponentVersion(c.getComponentVersion());
		response.set_return(cxsd);

		return response;
	}
```

Axis2 artifact deploy

During the server-side code generation, by using Axis Codegen plug-in, a `build.xml` is created. By executing the ant task `jar.server` an axis archive is generated. An axis archive has extension `aar`. It contains all classes and library, and also `services.xml` and `SampleService.wsdl`.

At this point Axis publish the service. It could be viewed on http://[host]:[port]/axis2/services/SampleService?wsdl

2.3.2 Client Side

Client Side Code

Again, Axis2 can be used to generate client side code in order to invoke service's operation, in a similar way it has been used for the server side code.

The plug-in generates:

- Stub Class
- CallBack Class
- Test Class

It's difficult to work directly with the generated Stub Class, so a new implementation Class is used, and now it invokes the stub's methods in order to talk with the provided service.

```
public class SampleClient implements Sample {

	private SampleStub stub;

	public SampleClient(String url) {
		try {
			stub = new SampleStub(url);
```

```
            } catch (AxisFault e) {
                  System.err.println("Constructor Axis Fault");
                  e.printStackTrace();
            }

      }


      public ComponentDescInfo getComponentDescInfo() {
            ComponentDescInfo c = new ComponentDescInfo();
            SampleStub.GetComponentDescInfoResponse response =
                                    null;
            try {
                  response = stub.getComponentDescInfo();
            } catch (RemoteException e) {

                  e.printStackTrace();
            }
            SampleStub.ComponentDescInfo c_stub = response
                        .get_return();

            c.setComponentAuthor(c_stub.getComponentAuthor());

            c.setComponentVersion(c_stub.getComponentVersion());

            c.setComponentName(c_stub.getComponentName());

            return c;

      }
```

It's interesting to remark that in the Skeleton case, each Stub method return a *Response object, so a translation from the Response object to the business object is requested, in order to provide a more readable code.

Service invocation

As explained in “The Client Side” paragraph, it could be difficult to work directly with the generated Stub. So, a client can use directly the `SampleServiceClient` class by only specifying the url location of the service to be invoked. In this way client application doesn't deal with marshalling and unmarshalling problem, but simply works as it could invoke business method.

```
public class SampleServiceClient {

          private static String URL =
"http://localhost:8080/axis2/services/SampleService";

          public static void main(String[] args) {

            SampleClient client = new SampleClient(URL);
            ComponentDescInfo info = client.getComponentDescInfo();

            System.out.println("Component name: " +
                        info.getComponentName());}

}
```

3. Developing Web Services with JAX-WS

3.1 Development Environment

- JDK 5 - http://java.sun.com
- Eclipse Europa 3.3 for JEE Developers - http://www.eclipse.org/downloads/moreinfo/jee.php
- GlassFish V2 - https://glassfish.dev.java.net/
- Plug-In SoapUI - http://www.soapui.org/eclipse/update
- JAX_WS 2.1.2 - https://jax-ws.dev.java.net/2.1.2/

3.2 Development Process

3.2.1 Approach

JAX-WS approach is mainly based on annotations (i.e. @WebService, @WebMethod) in order to develop, deploy and publish web services and their methods.

3.2.2 Deployment overview

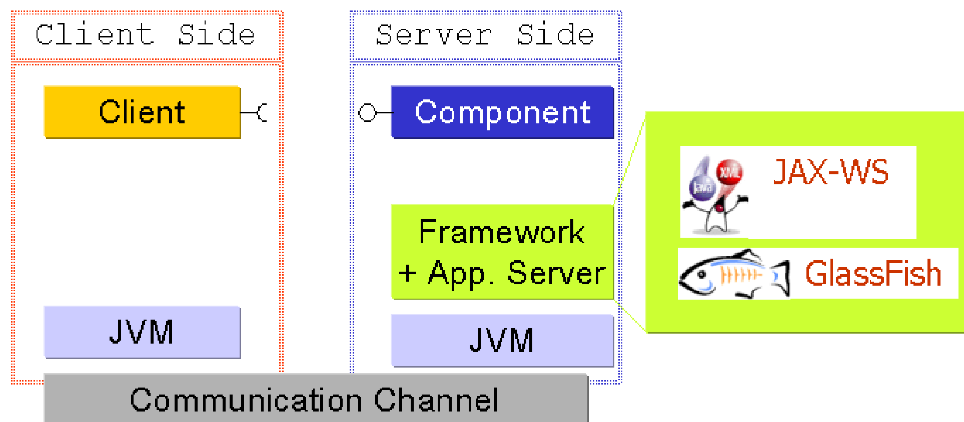


Figure 4 Web Service Deployment overview with JAX-WS

3.2.3 Development process workflow

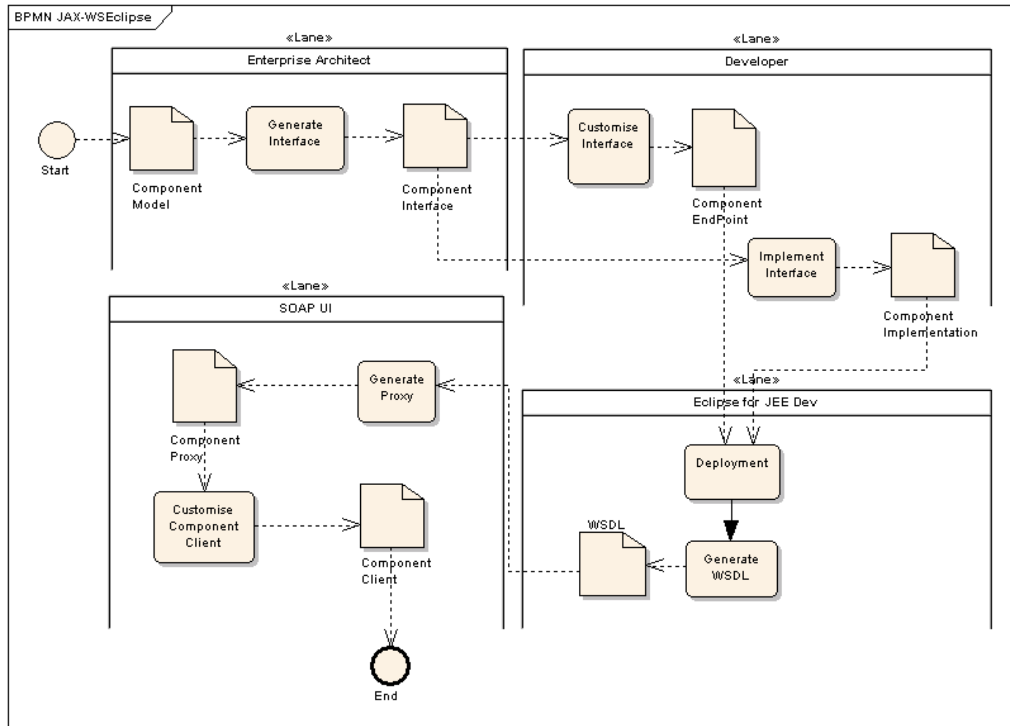


Figure 5 - JAX-WS Development Workflow

3.3 How-To step by step

Prepare you IDE for developing web services

1. Open Eclipse and select J2EE perspective: Windows->Open Perspective->Others->J2EE;
2. Select, in the lower window, the Tab Server (if not present Windows-> Show View and select Servers), right click in the window and select new->Server;
3. Download the GlassFish server (select Download additional server adapters. Accept the license and wait for Eclipse to restart);
4. After Eclipse is restarted, create a new GlassFish V2 javaEE5 server
5. Select, in the creation dialog, Installed Runtimes and select the directory where your GlassFish installation resides

3.3.1 Server Side

1. To create a Sample service, create a new dynamic Web project. Give it a name (ServerSample) and select as target runtime GlassFish
2. Create an Endpoint Web Service

Interface Sample.java

```
package eu.casparpreserves.imp.sample
import javax.jws.WebService;
import javax.jws.WebMethod;
import eu.caspar.framework.*;


@WebService(name="Sample")
public interface Sample extends CASPAR{
      @WebMethod
      public ComponentDescInfo getComponentDescInfo();
}
```

3. Create an implementation class

SampleImp.java

```
package eu.casparpreserves.impl.sample;
import javax.jws.*;
import eu.caspar.framework.*;
import eu.casparpreserves.api.sample.Sample;


@WebService(serviceName="SampleService", name="Sample",
portName="SamplePort")
public class SampleImp extends CASPARKeyComponent implements
Sample{
      public SampleImp(){
            componentDescInfo = new ComponentDescInfo();
            componentDescInfo.setComponentAuthor("Versienti    CNR-
ISTI Italy");
            componentDescInfo.setComponentName("Finding     Manager
Service");
            componentDescInfo.setComponentType(ComponentType.FIND);
            componentDescInfo.setComponentVersion("vers. 0.1");
      }


      public ComponentDescInfo getComponentDescInfo () {
            return componentDescInfo;
      }}
```

4. Deploy the service by selecting the project and select Run as->Run on the server

5. Check in the server Window that the Sample project has a status of Synchronized. If this is not the case, right-click in the server window and select

publish

6. You can check that the GlassFish server is started and contains the Web service by going to the GlassFish admin console (http://localhost:4848)

Creating Web Service Client using Soap UI Plug-in

1. Install SOAP UI Eclipse Plug-in
2. Select "Help->Software Updates->Find and Install…"
3. Press the "New Remote Site" button and add http://www.soapui.org/eclipse/update/site.xml as the plug-in URL
4. Select Finish and follow the dialogs to install the soapUI feature
5. Create a new (Java) project for the ServerSample client
6. Select Add GlassFish v2 as Server Runtime in Build Path
7. Right click->BuildPath->Add Library->ServerRuntime->GlassFish v2
8. Select the project and Right Click->Soap UI-> Add SOAPUI Nature, SOAP UI WebService item will be added in Project Explorer
9. Switch prospective Windows->Open Prospective->SoapUI
10. Select SamplePortBinding and Right Click->GenerateCode->JAX_WS Artifact
11. Enter the appropriate info in the JAX_WS Artifacts window

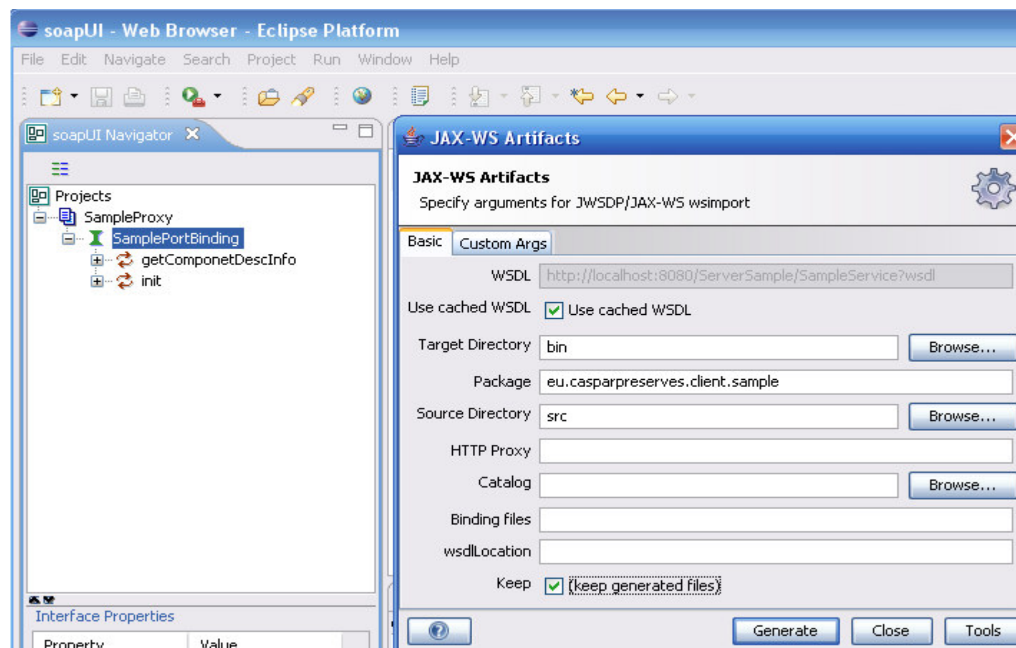


Figure 6- JAX-WS Artifacts window

12. Click ok
13. Then click Generate on JAX-WS Artifacts window, it will display a dialog box that the operation was successful. Switch back to Java Perspective, then refresh the src folder and you can see the wsimport generated classes

3.3.2 Client Side

The last step is to develop the client side

You can follow two way

First way:

1. Uses the `javax.xml.ws.WebServiceRef` annotation to declare a reference to a web service. WebServiceRef uses `wsdlLocation` element to specify the `URI` of the deployed service's `WSDL` file.

```
@WebServiceRef(wsdlLocation="
http://localhost:8080/ServerSample/SampleService?wsdl")
static SampleImpl service;
```

2. Retrieve a proxy to the service, also known as a port, by invoking `getSamplePort` on the service.

```
Sample port = service. getSamplePort();
ComponentDescInfo  response = port. getComponentDescInfo();
```

Second way:

1. Create the jar file of your proxy, point on the `ClientSample` project and create the jar file

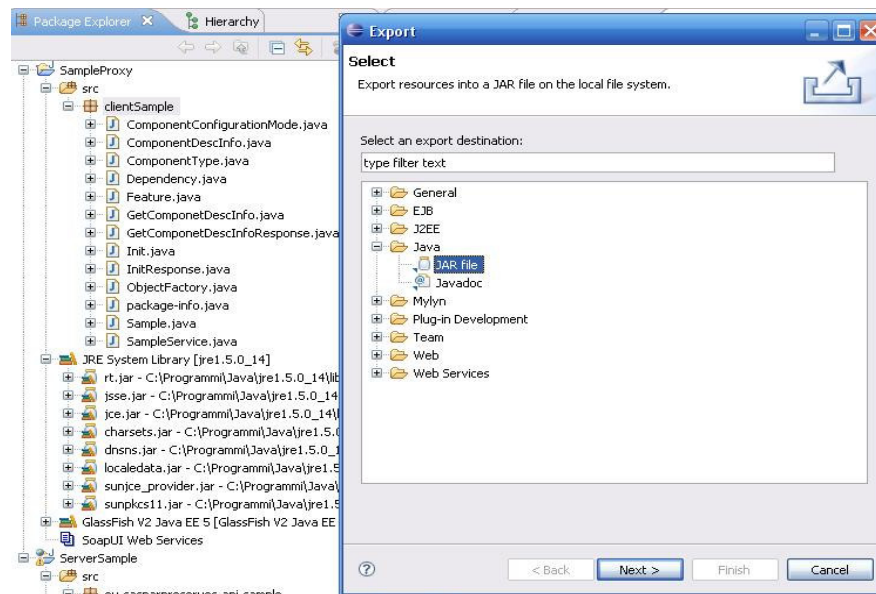


Figure 7 - Create jar file from proxy

2. Create a Dynamic project, set GlassFish as Application Server
3. Put the jar file in the GlassFish's lib folder
4. Now implement your client code, in this case it been implemented in the `jsp` file directly

SampleClient.jsp

```
<% page import="clientSample.*"%>
<%
SampleService service = new SampleService();
Sample proxy = service.getSamplePort();
 ComponentDescInfo descInfo =  proxy. getComponentDescInfo();
out.println("<h4>"+"Author:"+descInfo.getComponentAuthor()+"</h4>");
%>
```

A build file for Ant or Maven are going to be provided.

4. Developing Web Services with Spring-WS

4.1 Development Environment

The following tools and libraries have been used to realise the 'Sample' web service. Some of them are mandatory (M), others are optional (O) as they depend on implementation choices and can be replaced or not used.

JDK 5 *(M)*

The Spring Framework 2.0.7 *(M)*

Spring-ws libraries 1.0.2 *(M)*

Maven 2.0.7 *(M)*

Tomcat 5.5.25 *(O)*

Eclipse Europa 3.3 *(O)*

Altova XML Spy (optional, just for automatic xsd generation from xml) *(O)*

Castor XML (chosen as XML Marshalling libraries) *(O)*

4.2 Development Process

4.2.1 Approach

Spring-WS uses a contract-first approach, which consists in developing web services by starting with the XML Schema/WSDL contract first followed by the Java code implementation.

The most important thing when doing contract-first Web service development is to think in terms of XML; it is the XML that is sent across the wire, and Spring-WS focuses on that. The fact that Java is used to implement the Web service is an implementation detail.

4.2.2 Deployment overview

The following diagram gives an overview of the elements that build up the deployment system.

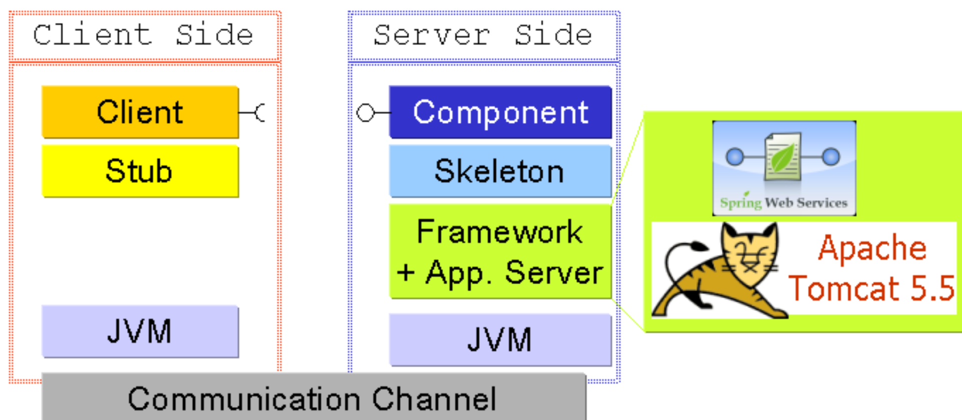


Figure 8 - Web Service Deployment overview with Spring-WS

4.2.3 Development process workflow

The following diagram summarises the steps that have been followed to realise the 'Sample' web service with Spring-WS.

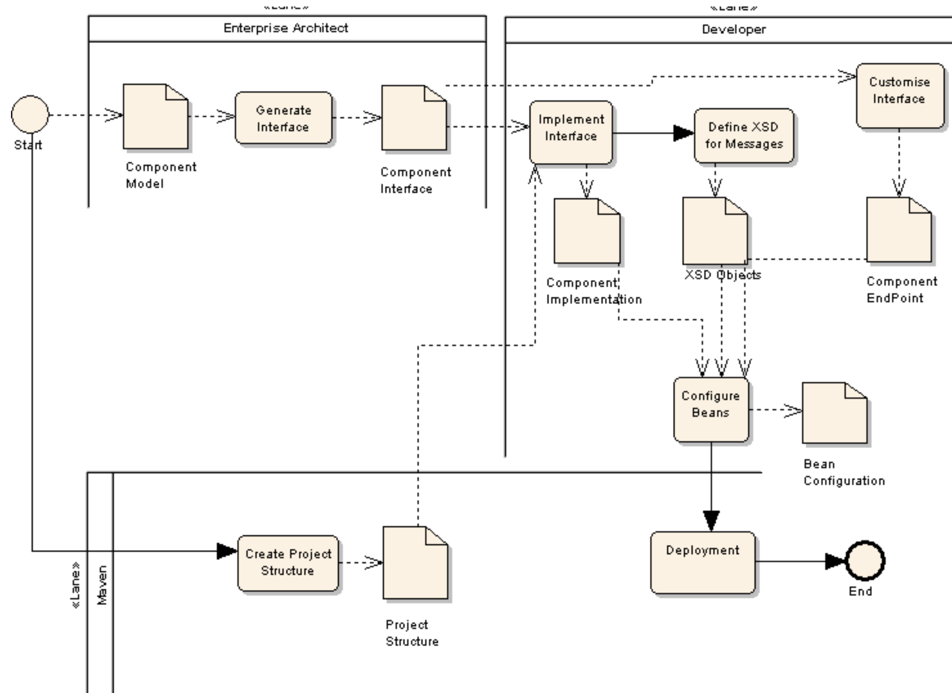


Figure 9 - Spring-WS Development Workflow

1. Generate the template code from Enterprise Architect (interfaces and class templates)
2. Create the project structure for the web application (using Maven)
3. Implement the business logic of the component and configure in the servlet beans configuration file
4. Define the xsd for the exchanged web service messages (starting from the definition of the messages) and configure the dynamic creation of wsdl in the servlet beans configuration file
5. Implement the Endpoint (including serialization) and configure in the servlet beans configuration file
6. Complete the servlet beans configuration file with the EndpointMapping information
7. Create the WAR (using Maven) and deploy the service

4.3 How-To step by step

4.3.1 Server Side

Sample Interface

```
package eu.casparpreserves.api.sample;
import eu.casparpreserves.framework.CASPAR;
public interface Sample extends CASPAR {}
```

Creating the project structure

In this section, we will be using Maven2 to create the initial project structure for us. Doing so is not required, but greatly reduces the amount of code we have to write to setup our `Sample` service.

The following command creates a Maven2 web application project for us, using the Spring-WS archetype (that is, project template):

```
mvn archetype:create -DarchetypeGroupId=org.springframework.ws \
-DarchetypeArtifactId=spring-ws-archetype \
-DarchetypeVersion=1.0.2 \
-DgroupId=eu.casparpreserves \
-DartifactId=sample
```

This command will create a new directory called `sample`.

Even if it is possible to change the default directory structure created by Maven, it is recommended not to do it.

In this directory, there is a `'src/main/webapp'` directory, which will contain the root of the WAR file. You will find the standard web application deployment descriptor `'WEB-INF/web.xml'` here, which defines a Spring-WS `MessageDispatcherServlet` and maps all incoming requests to this servlet:

```
<web-app xmlns=http://java.sun.com/xml/ns/j2ee
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation=http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd version="2.4">
    <display-name>Archetype Created Web Application</display-name>
    <servlet>
        <servlet-name>spring-ws</servlet-name>
        <servlet-
class>org.springframework.ws.transport.http.MessageDispatcherServle
t</servletclass>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring-ws</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

In addition to the above `'WEB-INF/web.xml'` file, you will also need another, Spring-WS-specific configuration file, named `'WEB-INF/spring-ws-servlet.xml'`. This file contains all of the Spring-WS-specific beans and is used to create a new Spring container. The name of this file is derived from the name of the attendant servlet (in this case `'spring-ws'`) with `'-servlet.xml'` appended to it.

```
<beans xmlns=http://www.springframework.org/schema/beans
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
```

```
</beans>
```

Web Service Descriptors

XML Messages

In this section, we will focus on the actual XML messages that are sent to and from the Web service. We will start out by determining what these messages look like.

In the scenario, we have to deal with `ComponentDescInfo` objects, so it makes sense to determine what a `ComponentDescInfo` looks like in XML:

```
<ComponentDescInfo>
    <ComponentVersion>0.1</ComponentVersion>
    <ComponentName>Sample</ComponentName>
    <ComponentAuthor>Metaware Spa</ComponentAuthor>
</ComponentDescInfo>
```

Starting from this we have for our Sample service the following request XML message:

```
<GetComponentDescInfoRequest/>
```

and response XML message:

```
<GetComponentDescInfoResponse>
    <ComponentDescInfo>
        <ComponentVersion>0.1</ComponentVersion>
        <ComponentName>Sample</ComponentName>
        <ComponentAuthor>Metaware Spa</ComponentAuthor>
    </ComponentDescInfo>
</GetComponentDescInfoResponse>
```

XSD

By far the easiest way to create an XSD is to infer it from sample documents. Any good XML editor or Java IDE offers this functionality. Basically, these tools use some sample XML documents, and generate a schema from it that validates them all.

Using the sample described above, we end up with the following generated schema:

```
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema
targetNamespace="http://casparpreserves.eu/sample/schemas">
    <xs:element name="GetComponentDescInfoRequest"/>
    <xs:element name="GetComponentDescInfoResponse">
        <xs:complexType>
      <xs:sequence>
          <xs:element name="ComponentDescInfo">
              <xs:complexType>
            <xs:sequence>
                <xs:element name="ComponentVersion"
```

```
type="xs:string"/>
                <xs:element name="ComponentName" type="xs:string"/>
                <xs:element name="ComponentAuthor"
type="xs:string"/>
            </xs:sequence>
              </xs:complexType>
          </xs:element>
      </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

We store this file as sample.xsd.

Don't forget to specify the namespace!

WSDL

Finally, we need to publish the WSDL. We don't need to write a WSDL ourselves; Spring-WS can generate one for us based on some conventions. Here is how we configure the dynamic WSDL generation in the spring-ws-servlet.xml file:

```
<bean id="sample"
class="org.springframework.ws.wsdl.wsdl11.DynamicWsdl11Definition">
    <property name="builder">
        <bean
class="org.springframework.ws.wsdl.wsdl11.builder.XsdBasedSoap11Wsd
l4jDefinitionBuilder">
            <property name="schema" value="/WEB-INF/sample.xsd"/>
            <property name="portTypeName" value="SampleService"/>
            <property name="locationUri"
value="http://localhost:8080/sample/services"/>
            <property name="targetNamespace"
value="http://casparpreserves.eu/sample/definitions"/>
        </bean>
    </property>
</bean>
```

- The bean id determines the URL where the WSDL can be retrieved. In this case, the bean id is sample, which means that the WSDL can be retrieved as sample.wsdl in the servlet context. The full URL will typically be http://localhost:8080/sample/sample.wsdl.
- The DynamicWsdl11Definition uses a Wsdl11DefinitionBuilder implementation to generate a WSDL the first time it is requested. Typically, we use a XsdBasedSoap11Wsdl4jDefinitionBuilder, which builds a WSDL from a XSD

schema. This builder iterates over all element elements found in the schema, and creates a message for elements that end with the defined request or response suffix. The default request suffix is `Request`; the default response suffix is `Response`, though these can be changed by setting the `requestSuffix` and `responseSuffix` properties, respectively. Next, the builder combines the request and response messages into a `WSDL` operation, and builds a `portType` based on the operations.

- The schema property is set to the `XSD` we defined in the previous section: we simply placed the schema in the `WEB-INF` directory of the application.
- Next, we define the `WSDL` port type to be `SampleService`.
- We set the location where the service can be reached (note: it must be in the servlet context): `http://localhost:8080/sample/services`.
- Finally, we define the target namespace for the `WSDL` definition itself. Setting these is not required. If not set, we give the `WSDL` the same namespace as the schema.

Sample Service Implementation

```
package eu.casparpreserves.impl.sample;
import eu.casparpreserves.api.sample.Sample;
import eu.casparpreserves.framework.CASPARKeyComponent;
import eu.casparpreserves.framework.CasparException;
import eu.casparpreserves.framework.ComponentDescInfo;


public class SampleImpl extends CASPARKeyComponent implements
Sample {
  public SampleImpl(){
    //setting component descriptive information
    componentDescInfo = new ComponentDescInfo();
    componentDescInfo.setComponentName("DAMS - User Manager");
    componentDescInfo.setComponentVersion("vers. 0.1 build 1");
    componentDescInfo.setComponentAuthor("Metaware SpA - Pisa
(Italy)");
  }
  public void init() throws CasparException {}
}
```

Here is how we configure the Business Logic in the `spring-ws-servlet.xml` file:

```
<bean id="sampleService"
class="eu.casparpreserves.impl.sample.SampleImpl"/>
```

Endpoint and Endpoint Mapping

In Spring-WS, you will implement Endpoints to handle incoming `XML` messages. Endpoints are the central concept in Spring-WS's server-side support. They provide access to the application behaviour which is typically defined by a business service interface. An endpoint interprets the `XML` request message and uses that input to invoke a method on the business logic service. The result of that service invocation is represented as a response message. Spring-WS has a wide variety of endpoints, using various ways to handle the

XML message, and to create a response.

In our implementation we used an endpoint which allows to handle multiple requests in one single class you, thus grouping functionality together. This model is based on annotations, so you can use it only with Java 5 and higher.

```
package eu.casparpreserves.impl.sample.endpoint;

import org.springframework.ws.server.endpoint.annotation.Endpoint;

import
org.springframework.ws.server.endpoint.annotation.PayloadRoot;

import eu.casparpreserves.api.sample.Sample;

import eu.casparpreserves.framework.ComponentDescInfo;


// Endpoint main class

@Endpoint

public class SampleEndpoint {

  private Sample sample;

  public SampleEndpoint(Sample sample) {

    this.sample = sample;

  }

  @PayloadRoot(localPart = "GetComponentDescInfoRequest",
namespace = "http://casparpreserves.eu/sample/schemas")

  public GetComponentDescInfoResponse

        getComponentDescInfo(GetComponentDescInfoRequest req) {

    ComponentDescInfo i = sample.getComponentDescInfo();

    GetComponentDescInfoResponse res = new
GetComponentDescInfoResponse(i);

    return res;

  }

}


// Request class

public class GetComponentDescInfoRequest  implements
java.io.Serializable {

    public GetComponentDescInfoRequest() {}

}


// Response  class

public class GetComponentDescInfoResponse  implements
java.io.Serializable {

    private ComponentDescInfo componentDescInfo;

    public GetComponentDescInfoResponse() {}
```

```
    public GetComponentDescInfoResponse(ComponentDescInfo info) {
            this.componentDescInfo = info;
    }
    public ComponentDescInfo getComponentDescInfo() {
        return componentDescInfo;
    }
    public void setComponentDescInfo(ComponentDescInfo info) {
        this.componentDescInfo = info;
    }
}
```

By annotating the `UserManagerEndpoint` class with @`Endpoint`, we mark it as a Spring-WS endpoint. Because the endpoint class can have multiple request handling methods, we need to instruct Spring-WS which method to invoke for which request. This is done using the @`PayloadRoot` annotation: the `getComponentDescInfo` method will be invoked for requests with a `GetComponentDescInfoRequest` local name and a `http://casparpreserves.eu/sample/schemas` namespace `URI`.

We also need to configure Spring-WS to support the marshalling/unmarshalling of the objects returned by/passed to the `getComponentDescInfo` method, in our case `GetComponentDescInfoRequest` and `GetComponentDescInfoResponse`. This is done using a Castor Marshaller.

Here is how we configure the Endpoint in the spring-ws-`servlet.xml` file:

```
<bean id="sampleEndpoint"
class="eu.casparpreserves.impl.sample.endpoint.SampleEndpoint">
    <constructor-arg ref="sampleService"/>
</bean>
<bean
class="org.springframework.ws.server.endpoint.adapter.GenericMarsha
llingMethodEndpointAdapter">
    <constructor-arg ref="castorMarshaller"/>
</bean>
<bean id="castorMarshaller"
class="org.springframework.oxm.castor.CastorMarshaller">
    <property name="mappingLocation"
value="classpath:eu/casparpreserves/impl/sample/endpoint/mapping.xm
l"/>
</bean>
<bean
class="org.springframework.ws.server.endpoint.mapping.PayloadRootAn
notationMethodEndpointMapping"/>
```

- The `GenericMarshallingMethodEndpointAdapter` converts the incoming `XML` messages to marshalled objects used as parameters and return value.
- The `PayloadRootAnnotationMethodEndpointMapping` is responsible for routing incoming messages to our Endpoint. This mapping uses the @`PayloadRoot`

annotation, with the localPart and namespace elements, to mark methods with a particular qualified name. Whenever a message comes in which has this qualified name for the payload root element, the method will be invoked.

- CastorMarshaller relies on Castor XML mapping, which is an open source XML binding framework. It allows you to transform the data contained in a java object model into/from an XML document. By default, it does not require any further configuration, though a mapping file can be used to have more control over the behaviour of Castor. The mapping can be set using the mappingLocation resource property, indicated below with a classpath resource. Here is our mapping.xml file:

```
<mapping>
    <class
name="eu.casparpreserves.impl.sample.endpoint.GetComponentDescInfoR
equest">
        <map-to xml="GetComponentDescInfoRequest" ns-
uri="http://casparpreserves.eu/sample/schemas" />
    </class>
    <class
name="eu.casparpreserves.impl.sample.endpoint.GetComponentDescInfoR
esponse">
        <map-to xml="GetComponentDescInfoResponse" ns-uri="
http://casparpreserves.eu/sample/schemas" />
        <field name="componentDescInfo"
type="eu.casparpreserves.framework.ComponentDescInfo">
            <bind-xml name="ComponentDescInfo" node="element" />
        </field>
    </class>
    <class name="eu.casparpreserves.framework.ComponentDescInfo">
        <map-to xml="ComponentDescInfo" ns-uri="
http://casparpreserves.eu/sample/schemas" />
        <field name="componentVersion" type="string">
            <bind-xml name="ComponentVersion" node="element" />
        </field>
        <field name="componentName" type="string">
      <bind-xml name="ComponentName" node="element" />
        </field>
        <field name="componentAuthor" type="string">
      <bind-xml name="ComponentAuthor" node="element" />
        </field>
    </class>
</mapping>
```

Deploying the WAR

You can create a WAR file using mvn install. If you deploy the application and point

your browser at `http://localhost:8080/sample/sample.wsdl`, you will see the generated `WSDL`.

Because we use `Annotations` we need to add a plug-in to Maven in order to compile the Java code. Here is the relevant section of the `pom.xml`:

```
<build>
    <finalName>sample</finalName>
    <pluginManagement>
        <plugins>
            <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
              <source>1.5</source>
              <target>1.5</target>
          </configuration>
      </plugin>
        </plugins>
    </pluginManagement>
</build>
```

Because we use Castor Marshaller, we must install Castor libraries and add the dependencies to the Maven `pom.xml`, which is in the root of our project directory. Here is the relevant section of the `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.ws</groupId>
        <artifactId>spring-oxm</artifactId>
        <version>1.0.2</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.ws</groupId>
        <artifactId>spring-ws-core</artifactId>
        <version>1.0.2</version>
    </dependency>
    <dependency>
        <groupId>jdom</groupId>
```

```
            <artifactId>jdom</artifactId>
            <version>1.0</version>
        </dependency>
        <dependency>
            <groupId>jaxen</groupId>
            <artifactId>jaxen</artifactId>
            <version>1.1</version>
        </dependency>
        <dependency>
            <groupId>javax.xml.soap</groupId>
            <artifactId>saaj-api</artifactId>
            <version>1.3</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>com.sun.xml.messaging.saaj</groupId>
            <artifactId>saaj-impl</artifactId>
            <version>1.3</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>castor</groupId>
            <artifactId>castor</artifactId>
            <version>1.1.2.1</version>
        </dependency>
            <dependency>
            <groupId>org.springframework.ws</groupId>
            <artifactId>spring-oxm-tiger</artifactId>
            <version>1.0.2</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.ws</groupId>
            <artifactId>spring-ws-core-tiger</artifactId>
            <version>1.0.2</version>
        </dependency>
    </dependencies>
```

4.3.2 Client Side

The `WebServiceTemplate` is the core class for client-side Web service access in Spring-WS. It contains methods for sending `Source` objects, and receiving response messages as either `Source` or `Result`. Additionally, it can marshal objects to XML before sending them across a transport, and unmarshal any response XML into an object again.

It uses an URI as the message destination. You can either set a `defaultUri` property on the template itself, or supply an URI explicitly when calling a method on the template.

In order to facilitate the sending of plain Java objects, the `WebServiceTemplate` has a number of `send(..)` methods that take an `Object` as an argument for a message's data content. The method `marshalSendAndReceive(..)` in the `WebServiceTemplate` class delegates the conversion of the request object to XML to a `Marshaller`, and the conversion of the response XML to an object to an `Unmarshaller`. By using the marshallers, your application code can focus on the business object that is being sent or received and not be concerned with the details of how it is represented as XML. In order to use the marshalling functionality, you have to set a marshaller and unmarshaller with the marshaller/unmarshaller properties of the `WebServiceTemplate` class (we again use Castor Marshaller).

Here is our client Class:

```
package eu.casparpreserves.client.sample;

import org.springframework.context.ApplicationContext;

import
org.springframework.context.support.ClassPathXmlApplicationContext;

import
org.springframework.ws.client.core.support.WebServiceGatewaySupport
;

import eu.casparpreserves.framework.ComponentDescInfo;


// Client main class

public class SampleClient extends WebServiceGatewaySupport {

  public void getComponentDescInfo() {

    GetComponentDescInfoRequest req = new
GetComponentDescInfoRequest();

    GetComponentDescInfoResponse res =
(GetComponentDescInfoResponse)
getWebServiceTemplate().marshalSendAndReceive(req);

    ComponentDescInfo info = response.getComponentDescInfo();

    System.out.println(info.getComponentVersion());

    System.out.println(info.getComponentName());

    System.out.println(info.getComponentAuthor());

  }

  public static void main(String[] args) {

    ApplicationContext ctx = new ClassPathXmlApplicationContext(
"applicationContext.xml", SampleClient.class);
```

```
    SampleClient sampleClient = (SampleClient)
applicationContext.getBean("sampleClient", SampleClient.class);

    sampleClient.getComponentDescInfo();

  }

// Request class

public class GetComponentDescInfoRequest  implements
java.io.Serializable {

    public GetComponentDescInfoRequest() {}

}

// Response class

public class GetComponentDescInfoResponse  implements
java.io.Serializable {

    private ComponentDescInfo componentDescInfo;

    public GetComponentDescInfoResponse() {}

    public GetComponentDescInfoResponse(ComponentDescInfo info) {

           this.componentDescInfo = info;

    }

    public ComponentDescInfo getComponentDescInfo() {

        return componentDescInfo;    }

    public void setComponentDescInfo(ComponentDescInfo info) {

        this.componentDescInfo = info;

    }

}
```

And here is how to configure the client in the `applicationContext.xml` file:

```
<beans xmlns=http://www.springframework.org/schema/beans
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://www.springframework.org/schema/beanshttp
://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean id="sampleClient"
class="eu.casparpreserves.client.sample.SampleClient">

        <property name="defaultUri"
value="http://localhost:8080/sample/services"/>

        <property name="marshaller" ref="marshaller"/>

        <property name="unmarshaller" ref="marshaller"/>

    </bean>

    <bean id="marshaller"
class="org.springframework.oxm.castor.CastorMarshaller">

        <property name="mappingLocation"
value="classpath:eu/casparpreserves/client/sample/mapping.xml"/>

    </bean>

</beans>
```

5. Comparing the 3 ways

In this section the CASPAR Architecture Team evaluates the three ways analysed during the development experience.

	Framework + Application Server		
Evaluated Features	**(1) Axis2 + Tomcat5.5**	**(2) JAX-WS + Glassfish**	**(3) Spring-WS + Tomcat5.5**
Layering and independence from business logic implementation	++ (g)	++ (g)	++ (g)
Serialization and coverage of datatypes	+ (c)	+++ (a)	+ (b)
Automation tools and effort-reduction	++ (e)	+++ (e)	+ (e)
(Low-level) control and customization	++	+	+++ (d)
Easiness in overall generating of “skeleton and stub”	+	+++	+ (f)
Possibility to intercept messages at framework level (for Access Control)	(h)	(h)	(h)
Error handling	(h)	(h)	(h)
Degree of task (demo) completion/time	++	+++	++
Overall Evaluation	**1.6**	**2.5**	**1.6**

Legend

(a) many complex types are mapped automatically (e.g. Collection); otherwise just add a new mapping through annotation mechanism

(b) not completely automated, since serialization might need to be customised to comply with the XSD (WSDL) (as messages come first!); but more control

(c) no customization required, but some effort to map the Java types to XSD types

(d) (more low - level) -> more control and customization possibilities; but more effort to understand

(e) in (1) xml messages (and XSD / WSDL) come first -> less automation, for instance with regard to (1) and (2) where WSDL is generated from code

(f) “skeleton” part generation: need to understand bean configuration mechanism and language, and to create endpoint and mapping files “manually” (even if most is straightforward)

(g) independent from the underlying framework

(h) to be investigated

6. The CASPAR Best Practice

In this section the **CASPAR** Architecture Team defines the chosen best practice for developing the **CASPAR** Key Components.

The best practice comes from the experiences analysed in the above sections and mainly it's based on the JAX-WS standard.

But it's important to remark that the **CASPAR** Key Components interfaces and implementations will have no dependency with the chosen development technology. In fact, in order to guarantee the "preservability" of the models and of the implementation (even if in part), the **CASPAR** Key Components interfaces will be just the "code generated by the UML models" and the implementations will be just the business logic implementations.

Dependencies to technology and underlying framework are moved to skeleton and stub, as shown in the schema below.

In this case it's possible to assert that **CASPAR** Key Components implementation is "Technology Agnostic".

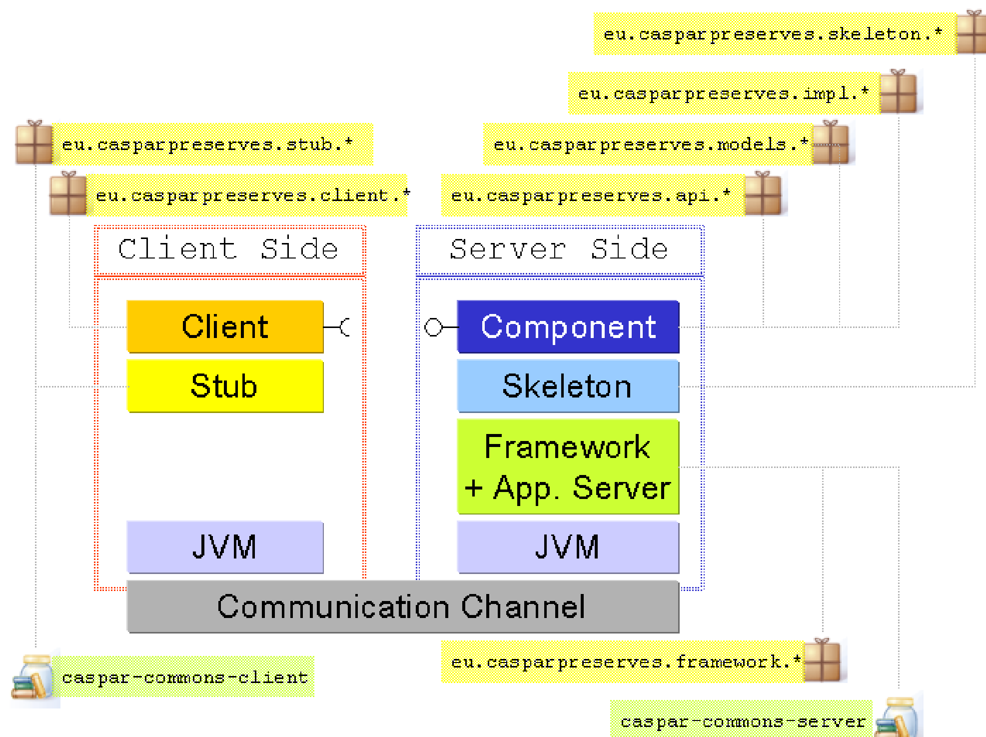


Figure 10 CASPAR Best Practice Layers

In this perspective, **CASPAR** Architecture Team identifies the following main packages for the **CASPAR** Key Components development:

- `eu.casparpreserves.api.*` - package for the **CASPAR** Key Components interfaces, specified in [D1301] and their refined specifications. The code is generated by Enterprise Architect tool;

- `eu.casparpreserves.models.*` - package for the **CASPAR** Key Components conceptual models (i.e. classes for parameters/data of operations of **CASPAR** Key

Components interfaces), specified in [D1301] [D1201] and their refined specifications. The code is generated by Enterprise Architect tool;

- `eu.casparpreserves.impl.*` - package for the implementation of the business logic of the **CASPAR** Key Components. No dependencies with the development technology and the underlying framework;
- `eu.casparpreserves.skeleton.*` - package for the part of **CASPAR** Key Component implementation which interacts with the underlying framework and chosen technology;
- `eu.casparpreserves.framework.*` - package for the common **CASPAR** functionalities which deal with registration, description and underlying framework interaction;
- `eu.casparpreserves.stub.*` - package for the part of the client of the **CASPAR** Key Component which interacts with the underlying framework and chosen technology;
- `eu.casparpreserves.test.*` - package for the **CASPAR** Key Component test, independent from the underlying framework and chosen technology;
- `eu.casparpreserves.client.*` - package for the **CASPAR** Key Component which can be invoked from client applications. This package uses the generated `eu.casparpreserves.stub.*` artifacts.

The picture below shows the main packages, used by this best practice, with their relative classes and interfaces.

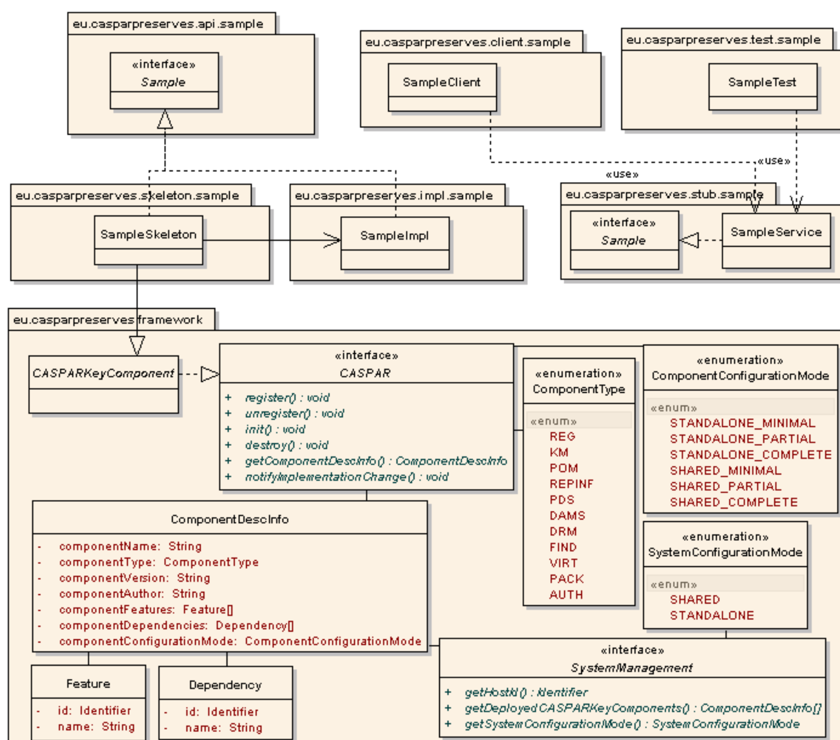


Figure 11 Main CASPAR Packages

In the following sections the **CASPAR** Architecture Team shows step by step how to develop a **CASPAR** Key Component (`Sample`). You can find the code sample at the **CASPAR** SVN repository.

6.1 Server Side

Step 1 – Define the component interface

```
package eu.casparpreserves.api.sample;

public interface Sample {

	/**
	 * hello operation
	 * @param name of the caller
	 * @return welcome to the caller
	 */
	public String hello(String name);
}
```

Step 2 – Implement the component interface

```
package eu.casparpreserves.impl.sample;

import eu.casparpreserves.api.sample;

public class SampleImpl implements Sample  {

	public SampleImpl(){

	}

	public String hello(String name) {
		return new String("Welcome to CASPAR " + name);
	}
}
```

Step 3 – Define the component skeleton

```
package eu.casparpreserves.skeleton.sample;

import javax.jws.WebService;
import javax.jws.WebMethod;

import eu.casparpreserves.api.sample.Sample;
import eu.casparpreserves.framework.*;
import eu.casparpreserves.impl.sample.SampleImpl;

@WebService(name="Sample",serviceName="SampleService")
public class SampleSkeleton extends CASPARKeyComponent implements
Sample {

	private final static Sample SAMPLE = new SampleImpl();

	public SampleSkeleton(){
		/*
		 * setting component descriptive information
```

```
             */
            componentDescInfo = new ComponentDescInfo();
            componentDescInfo.setComponentAuthor("Caspar
Architecture Team");
            componentDescInfo.setComponentName("CASPAR Sample");
            componentDescInfo.setComponentVersion("version 0.0.1");
            componentDescInfo.setComponentType(ComponentType.FIND);
      }

      @WebMethod
      public ComponentDescInfo getComponentDescInfo() {
            return super.getComponentDescInfo();
      }

      @WebMethod
      public String hello(String name) {
            return SAMPLE.hello(name);
      }
}
```

Step 4 – Deployment descriptor files

Under `sample/etc` folder there are three files in order to describe how the component can be deployed as a JAX-WS Web Service:

- **`deploy-targets.xml`** – define ant task in order to deploy on a proper application container;
- **`sun-jaxws.xml`** – define the endpoint of the service, its implementation class and its context on application container, according to JAX-WS standard;
- **`web.xml`** – standard deployment descriptor for web application.

Step 5 – Generation of server artifacts

A **`build.xml`** is provided in order to build server and client artifacts according to JAX-WS standard.

The **`server`** ant task can be executed in order to build and deploy server artifacts.

If artifacts are deployed on Apache Tomcat, it's necessary to deploy the JAX-WS libraries (available at **CASPAR** software SVN repository `software/java/framework/jaxws/lib`) on the Tomcat shared libraries folder (i.e. `CATALINA_HOME/shared/lib`).

6.2 Client Side

Step 6 – Generation of client artifacts

Client artefacts can be obtained by executing **`client-stub`** ant task. A set of classes are generated by the tool **`wsimport`** provided by JAX-WS tools and invoked in client-stub ant task. The generated classes are in eu.casparpreserves.stub.sample package.

Step 7 – Define client component

In order to facilitate interactions with the set of generated client side classes, a `SampleClient` class is provided. It implements `Sample` interface generated in client artifacts and simply delegates method implementation to `Sample` class by using the generated `SampleService`.

```
package eu.casparpreserves.client.sample;

import java.net.*;
import java.util.*;
import javax.xml.namespace.QName;
import eu.casparpreserves.stub.sample.*;
/*
 * To be compiled and executed only after generating the stub:
 * "ant client-stub" (see build.xml)
 */
public class SampleClient  implements Sample{

      private Sample port;

      public SampleClient(URL wsdlLocation){

      QName sQName =
      new QName("http://sample.skeleton.casparpreserves.eu/",
                "SampleService");
     port = new SampleService(wsdlLocation,sQName).getSamplePort();

      }
      public ComponentDescInfo getComponentDescInfo() {
            return port.getComponentDescInfo();

      }
      public String hello(String arg0) {
            return port.hello(arg0);

      }

}
```

6.3 Advanced Client Solution: ServiceFactory

`ServiceFactory` is an utility class that creates Service objects, given the service class and the service's URL. That factory allows to simplify client application accessing to `JAX-WS` Services, and for that reason it has been promoted as a common (framework) utility for all **CASPAR** Key Components. The code below shows how to use it for invoking a `Sample` service.

```
package eu.casparpreserves.test.sample;

import java.net.*;
import eu.casparpreserves.client.ServiceFactory;
import eu.casparpreserves.stub.sample.*;

public class SampleManagerTest {

      private SampleManager port;

      public SampleManagerTest(URL wsdlLocation) {
            SampleManagerService notMan =
ServiceFactory.createService(SampleManagerService.class,
      wsdlLocation);
            port = notMan.getSampleManagerPort();
      }
```

```
	public SampleManager getPort() {
		return port;
	}


	/**
	 * @param args
	 * @throws MalformedURLException
	 */
	public static void main(String[] args) throws
MalformedURLException {
		SampleManagerTest s = new SampleManagerTest(new
		URL("http://localhost:8080/sample/SampleManager?wsdl"));
		SampleManager sampleManager = s.getPort();
		ComponentDescInfo info =
			sampleManager.getComponentDescInfo();
		System.out.println("Called the component " +
			info.getComponentName());
		System.out.println("Author of the component " +
			info.getComponentAuthor());
		System.out.println("Version of the component " +
			info.getComponentVersion());
		String welcome = sampleManager.hello("new developer");
		System.out.println(welcome);

	}

}
```

7. Future Work

The **CASPAR** Architecture Team agrees that the *Spring Framework* is a useful and practical solution to apply the Inversion of Control pattern in the **CASPAR** Key Components. And specifically, the concepts of beans and of the dependencies can be applied to manage the configuration of each **CASPAR** Key Component.

In this perspective, it'll be better investigated in the next release of this document the best practice to use *Spring Framework*.

Moreover, in the future **CASPAR** Architecture Team could investigate on the new technology proposed by Eclipse: *RAP*.

RAP will offer the ease of development of Eclipse, Eclipse extensibility and user experience by reusing Eclipse technology for distributed applications and by encapsulating *AJAX* technologies into simple-to-use Java components

Web applications remain the appropriate approach for many usage scenarios, e.g. for applications with zero-install access requirements. With centralized installation and administration, web applications offer low administration and production costs. Furthermore, they can provide high scalability and good performance.

But associated with the classic approach of HTML-based web applications is a higher expenditure of resources during the development and maintenance process due to the technology mix that is involved. With *AJAX* this becomes an even bigger problem.

By encapsulating the *AJAX* technologies in a component library one major road block to a more efficient development model can be eliminated. By offering a plug-in model on the server and providing *Eclipse* workbench UI capabilities developers can leverage their experience with *Rich Client Applications* and profit from the proven *Eclipse* development model to create Rich Internet Applications in a streamlined fashion.

References

CASPAR Deliverables referenced in this document

[D1201] Deliverable 1201 “CASPAR Conceptual Model”

[D1301] Deliverable 1301 “CASPAR Overall Component Architecture and Component Model”. Refinement Specifications are available at http://wiki.casparpreserves.eu

[D3101] Deliverable 3101 “Report on Framework Architecture”

For further details about development with Axis2, see the following references

- *Expose Your POJO-Based Domain Apps as Web Services* – http://www.devx.com/Java/Article/33839
- Axis2 Code Generator plug-in documentation http://ws.apache.org/axis2/tools/1_0/eclipse/wsdl2java-plugin.html

For further details about development with JAX-WS, see the following references

- Documentation on JAX-WS: https://jax-ws.dev.java.net/nonav/2.1.2/docs/index.html
- Sun web service tutorial: http://java.sun.com/webservices/docs/2.0/tutorial/doc/
- Developing with Eclipse:

  https://jax-ws.dev.java.net/guide/Developing_with_Eclipse.html
- GlassFish & Eclipse: https://glassfishplugins.dev.java.net/eclipse33/index.html
- SoapUI plug-in: http://www.soapui.org/eclipse/update/site.xml
- RAP: http://www.eclipse.org/proposals/rap/
- *JAX-WS 2.0 Samples on Tomcat 5.5.x:* http://weblogs.java.net/blog/arungupta/archive/2006/03/jaxws_20_sample.html

For further details about development with Spring-WS, see the following references

- *Spring Web Services - Reference Documentation*. (Chapter 3). Available at: http://static.springframework.org/spring-ws/site/reference/html/index.html
- CASTOR Website: http://castor.org/xml-framework.html
- *Castor XML Mapping*. Available at: http://castor.org/xml-mapping.html
- Maven Web site: http://maven.apache.org/
- Tomcat Web site: http://tomcat.apache.org/
- Eclipse Web site: http://www.eclipse.org/
- Spring-WS releases: http://static.springframework.org/spring-ws/site/downloads/releases.html
- Spring Framework releases: http://www.springframework.org/download

Appendix 1 – SVN Structure and Build Files

The **CASPAR** Architecture Team has defined the following structure for the SVN structure:

- `/framework` – main folder for framework and common utilities
- `/interfaces` – main folder for **CASPAR** Key Components interfaces and relative models (user defined types)
- `/implementation` – main folder for **CASPAR** Key Components implementations
- /lib – main folder for shared libraries

More, the **CASPAR** Architecture Team has defined two useful ant build files to automatically build and deploy software artifacts.

In particular, the build file `/framework/framework.xml` allows to compile framework and common utilities and provide the distribution packages for server and client side, respectively `CASPAR-Framework.jar` and `CASPAR-Client.jar`. It's enough to launch the target `dist-framework`.

The framework build file is also imported into each **CASPAR** Key Component build file `/Implementation/<KeyComponentName>/build.xml`. That build file allows to compile the component and provide the distribution packages for server and client side, respectively `<KeyComponentAcronym>.war` and `<KeyComponentAcronym>-stub.jar`. It's enough to launch the two targets server and client-stub.

In order to customise the build for each component, it's enough to use the `build.properties`.

The code below shows an example of build file for a **CASPAR** Key Component (i.e. Preservation Orchestration Manager).

```
#API relative package
COMPONENT_NAME=orchestration

#CASPAR Key Component
COMPONENT_ACRONYM=POM

#Component interfaces
#Customise targets generate-client-stub, compile-server and add
interface property in build.xml
NOTIFICATION_INTERFACE=NotificationManager

#DEPLOY PROPERTIES
GLASSFISH_HOME=C:/glassfish
CATALINA_HOME=C:/Tomcat5.5
HOST_PORT=8080

#Declare if you want to compile api and framework
compile.api=true

compile.framework=true
```

The "`COMPONENT_NAME = orchestration`" defines the name of the component package. In this case, it's

`eu.casparpreserves.[api,impl,models,skeleton,stub].orchestration`

The "`COMPONENT_ACRONYM = POM`" defines the acronym assigned to the component, and that acronym is used to set name of deployment packages. In this case, the artifacts are `POM.war` and `POM-stub.jar`.

The "`NOTIFICATION_INTERFACE = NotificationManager`" defines the names of the component interfaces. For each interface, the developer has to add a definition and customise the targets `generate-client-stub`, `compile-server` in `build.xml` file. That's used for component stub generation (i.e. client side component library).

The "`GLASSFISH_HOME=C:/glassfish`" "`CATALINA_HOME=C:/Tomcat5.5`" and "`HOST_PORT=8080`" define the installation directories for two supported application servers (i.e. Sun GlassFish and Apache Tomcat). More, it's possible to set the http port where application servers receive/response.

Before to deploy the software artifacts, the build target checks existence of those directories.

The **CASPAR** Architecture Team is analysing the possibility to extend software artifacts deployment for other existing application servers (e.g. Web Sphere Application Server – Community Edition)

Finally, "`compile.api`" and "`compile.framework`" are two flags introduced to avoid to build always api and framework code, reducing build time. In fact, usually those two parts change with not frequently.