CaptainCasa Enterprise Client

Web Service Integration Kit

This document shows how to integrate applications into the CaptainCasa Enterprise Client environment using web services.

The documentation is targeted to: developers, software architects, development leads, product managers.

Inhalt

CaptainCasa Enterprise Client and Web Services....3
- CaptainCasa Enterprise Client....3
- Java Binding on Server Side....3
- Java calls Web Services – That's it! ...?....4
- “No Development” Approach – Hmmm....5

Bringing Web Services into the Interaction Layer....6
- Purpose....6
- Usage....6
  - WSDL Import....7
  - Stub Generation....7
  - Operation Template Generation....7
- Example....7
- Summary....10

Declarative Approach for Structuring the Interaction Layer....11
- Typical Interaction Tasks – Enriching a Web Service API....11
- Declarative Approach....11
  - Bean Wrapper....11
  - Interface Configuration....12
- Using the Declarative Approach....13
  - Generating a Screen for a Web Service Operation....13
  - Enriching the Screen Interaction Logic....14
  - Enriching the Screen....16
- Summary....16

Installing the Web Service Integration Kit....18
- Installation of the Kit....18
- Project Injection....18
- Eclipse Plug-in....19

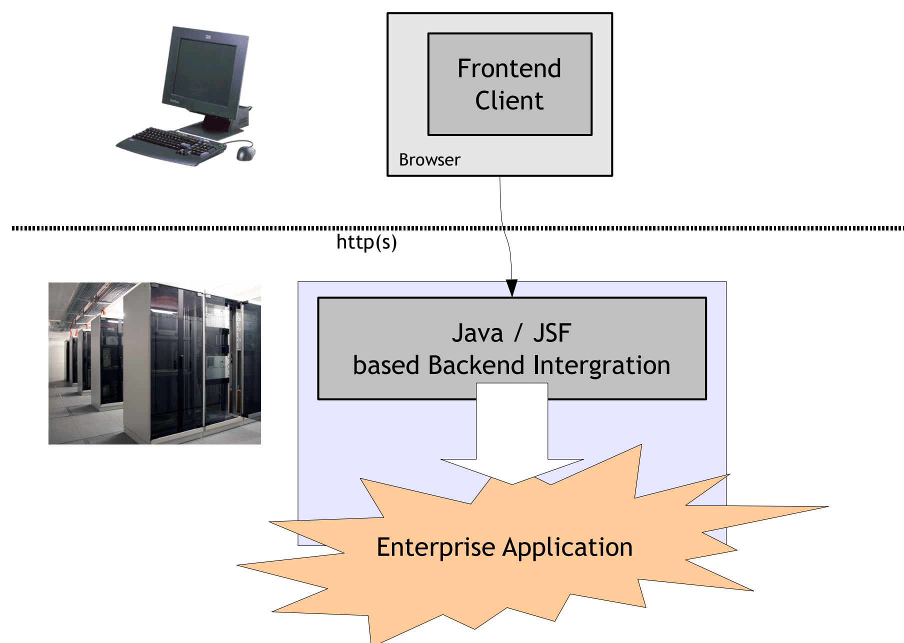
CaptainCasa Enterprise Client and Web Services

CaptainCasa Enterprise Client

CaptainCasa Enterprise Client is a rich client technology solution: the user interface of an application is running in the frontend client, the application logic is running on backend side. The client itself is a generic, Java-based client, that is connected to the backend by using http(s). The server side is based on Java Server Faces (JSF) – the J2EE standard for user interface processing in a client-server environment.

CaptainCasa Enterprise Client is designed to reflect frontend requirements of large, operationally used, demanding applications:

- The client is fast and robust. The interaction quality of components is high.
- A standard component library contains a large set of high quality components covering a wide range of applications needs, from processing complex forms to direct data input via touch screens.
- The server processing is scalable and based on approved technology (JSF).
- The development model is simple and efficient, in order to server applications with >100 screens. There is no client side application development required – all application development including the definition of user interfaces is done on server side.



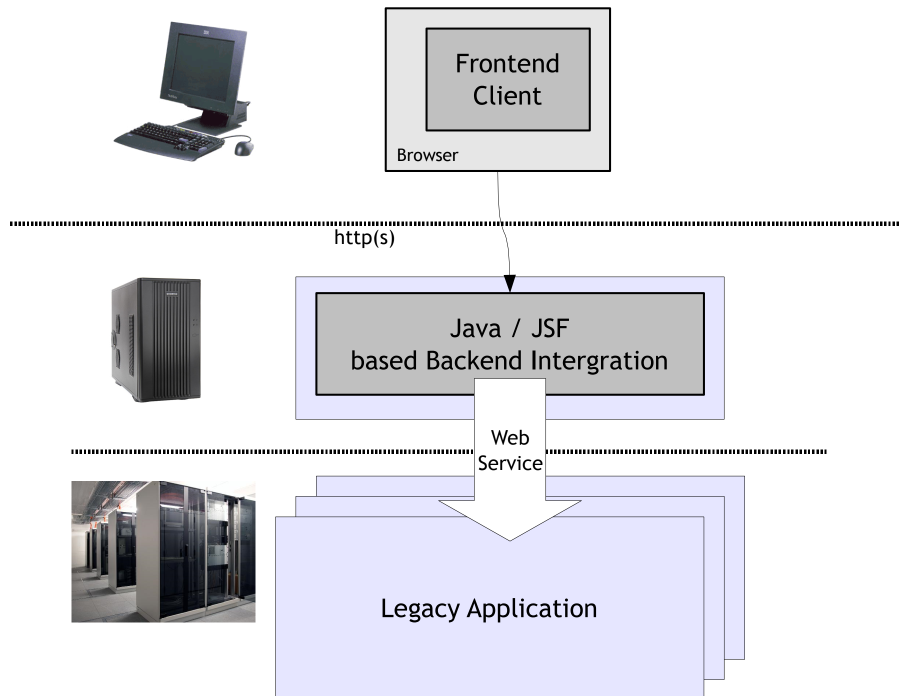
Java Binding on Server Side

Developing user interfaces with CaptainCasa Enterprise Client typically consists of working with two objects:

- The XML page definition, that is designed by using a WYSIWYG Layout Editor.
- The Java code, that binds the page to application processing on server side. The Java code is a logical reflection of the page. In the page you have fields, check boxes, buttons – in the Java code you have properties and methods.

Due to the usage of Java and Java Server Faces, the server side processing is adaptable to many application scenarios:

- The application may be a Java program that directly is invoked. For example it is possible to directly invoke Java applications that are based on Hibernate persistence management.
- The application may be a Java program that runs on a different server and that is couple with certain communication protocol (e.g. Enterprise Java Beans based applications).
- **OR: The application may be a program that is connect-able by using Web Services!**



Well, the last option is the one this documentation is about. Web Services are a powerful way to define interfaces to service endpoints. Web Services decouple the processing behind the web service interface completely from the interface itself. This means: the caller of a Web Service does not need to know any implementation details of the system providing the Web Service. This typically makes Web Services an integral part of Service Oriented Architectures.

There are frameworks available that simplify the calling of Web Services from Java. One of the most prominent ones is “Axis2” which transfers Web Services into plain Java based APIs.

Java calls Web Services – That's it! ...?

Well, the story sounds quite simple – and the good news: from technical point of view it finishes with “Java calling Web Services using Axis2”.

But, as usual, there's more to fight with than “just technology”, in order to develop applications which run in the very backend and provide a rich user interface in the frontend.

Let's start with the very backend:

- Existing backend systems (from now on called: legacy systems) typically do not provide

web services to the outside.

- Legacy systems typically even do not provide a proper functional access layer: as usual in many cases front end and back end logic are melted together a bit too strong...

...and continue with the interaction part:

- Bringing web services to Java requires some knowledge.
- Designing an interaction process for a web service is (much!) more, than straightly transferring the Web Service into a user interface. Web Service interfaces are “logical APIs” - which do not provide sufficient information to build a nice UI. Example: in a web service typically id's are passed (e.g. “article-id”) - the user wants to see text information for the id, an he/she wants to select the id from a list of valid values.
- The interaction may call several web services to gather all the information it requires.
- The interaction part typically couples diverse screens in order to form proper usage scenarios: from the overview you select the detail item, from the detail item you start certain reports, etc.

In many cases frontend programs and backend programs need to cooperate in a certain way in order to efficiently work together. Imagine a standardized frontend pattern, in which you maintain certain master data objects by first presenting them as list, then letting the user select one of the items and opening a detail form, in which the user maintains the attributes of the object. - In this case it is useful to always have a defined web service infrastructure for each object, providing operations like create/ read/ update/ delete (the so called CRUD-interface) with a defined data structure for each operation.

“No Development” Approach – Hmmm...

You see from the previous chapter: building your rich client application in front of a web service based application is much more than technically calling the web service, and binding the user interface to the input/output parameters of the Web Service.

There is development work both on interaction side and on legacy side. This is the bad news.

The good news: there is enough room for structuring what you do in a way, so that it is efficient and “as-less-coding-as-possible”. And so that your legacy developers and your interaction developers know how to split up work.

And, the good new: this is nothing “new”. You always need to architecturally design, structure and implement the interaction part of your application. You need to find a way to outsource the typically needed functions into declarative patterns in order to decrease the amount of interaction coding. - Now, with web services you just have a system interface between the application's interaction part ant the application's processing part.

What you need on interaction side: a flexible user interface framework in which your structural patterns can be transferred. And that's what CaptainCasa Enterprise Client is: a straight rich client framework with server side interaction management and server side Java-based application binding.

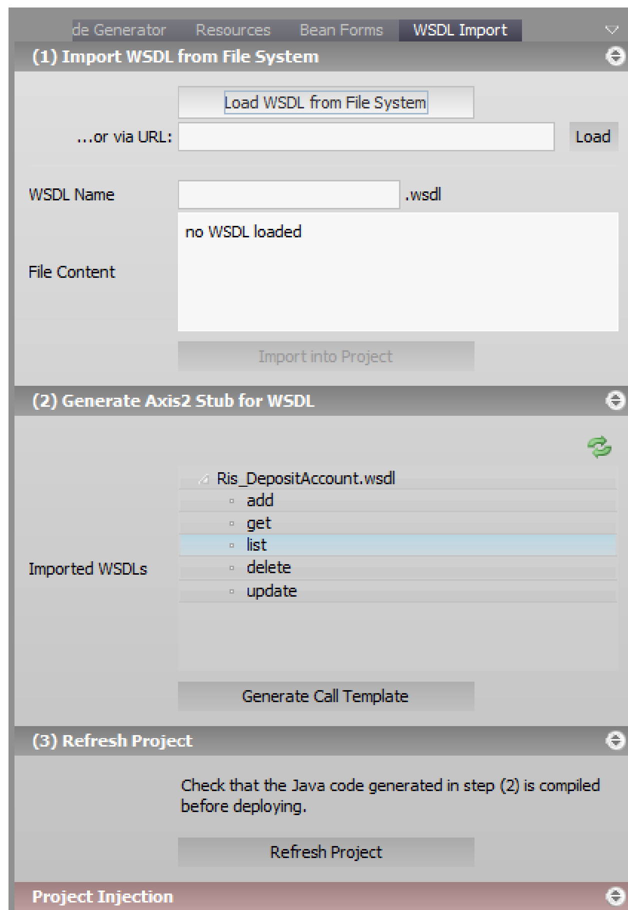
The following chapters will first show you the technical side: bringing Web Services to Java – using tools that are plug-ins to the CaptainCasa Enterprise Client toolset.

And then we will present what we think about structuring what you do.

Bringing Web Services into the Interaction Layer

CaptainCasa Enterprise Client provides a set of add-on-kits to its default rich client processing. Each add-on-kit may provide additional tools that are available within the CaptainCasa toolset and they may provide additional libraries or resources that need to be loaded into your project to be called at runtime.

After having installed the Web Service Integration Kit you will see a new tool being part of the Enterprise Client toolset:



Purpose

The purpose of this tool is:

- You can generate the Axis2 Java stub code that is required to call a certain web Service.
- You can generate template code for an operation of a web service. This simplifies the calling of the web service significantly.

Usage

The steps you need to follow are:

WSDL Import

You import a WSDL file into the project. Importing can be done either by selecting the file from the file system or by referencing a URL.

As result of the import the corresponding WSDL file will be stored in a /wsdl-folder of your project.

Stub Generation

Select one of the WSLD-files that are already imported and press the button “Generate Stub Code”. The Axis2 generator will be called so that a Java program will be generated. The Java program contains a so called stub, i.e. a class that represents the web service structure so that it can be easily called from a Java program.

Operation Template Generation

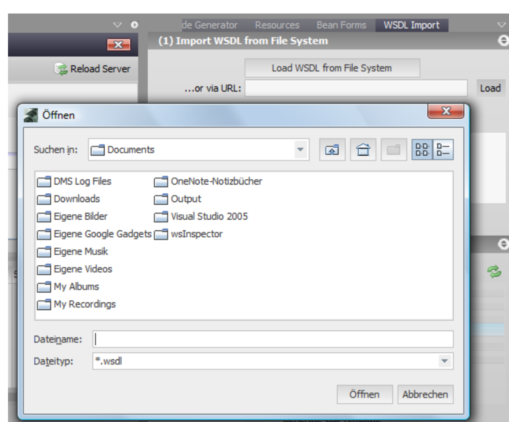
The stub is a reflection of a web service with all of its operations. The XML structure of the input and output parameters of the web service operations are transferred into classes. Web service interfaces typically have a quite deep hierarchical nesting of XML segments so that it is not a trivial job to assemble all classes in order to properly call a web service operation.

The template generation reduces this work of assembling significantly. After having generated the stub code (and refreshing) you now can select an operation of the web service and generate a template code that calls this operation.

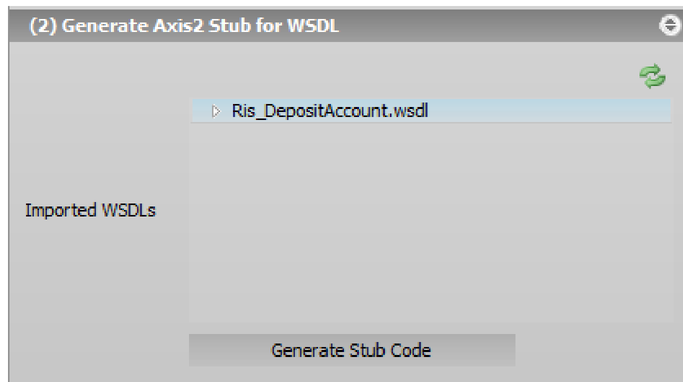
Example

As example we use a public demo web service provided by the company Risaris Ltd., Ireland. Risaris provides tools and frameworks for transferring legacy system interfaces (such as RPC protocols) into web services – and provided a publically reachable site in which you can access their demo systems.

You can download WSDL from their site and store it in your local file system. Using the import function of the WSDL import tool, you can transfer the WSDL into you project:

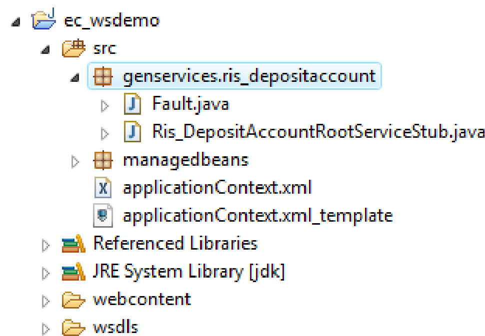


Press the “Load WSDL from File System” button, select the wsdl-file and press the “Import into Project” button. After the import the wsdl file is stored in the /wsdl directory of your project and the wsld file will be added to the list of imoprted wsdl's within the tool.



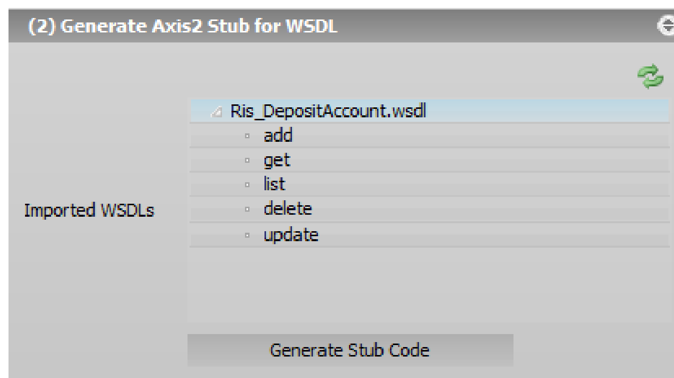
Now you can select the wsdl-file and press the “Generate Stub Code” button. Axis2 will be invoked internally to create the stub code. The stub code will be generated into a package with the name “genservices.<nameOfWsdl in lowercase>”, in this case “genservices.ris_depositaccount”.

Have a look into your Eclipse project:

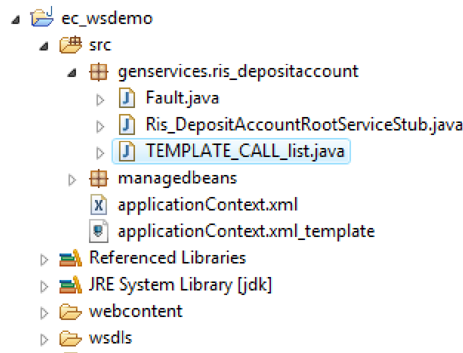


The stub code was generated as class “Ris_DepositAccountRootServiceStub.java”. Eclipse will refresh automatically when using the CaptainCasa Eclipse plugin – so we definitely recommend to install it.

Now you can refresh the “Imported WSDLs” section of the CaptainCasa toolset:



You see the web service with its operations. You now can click one of the operations, e.g. the “list”-operation and press the button “Generate Call Template” - as result a template class will be added:



The Java code of the class looks as follows:

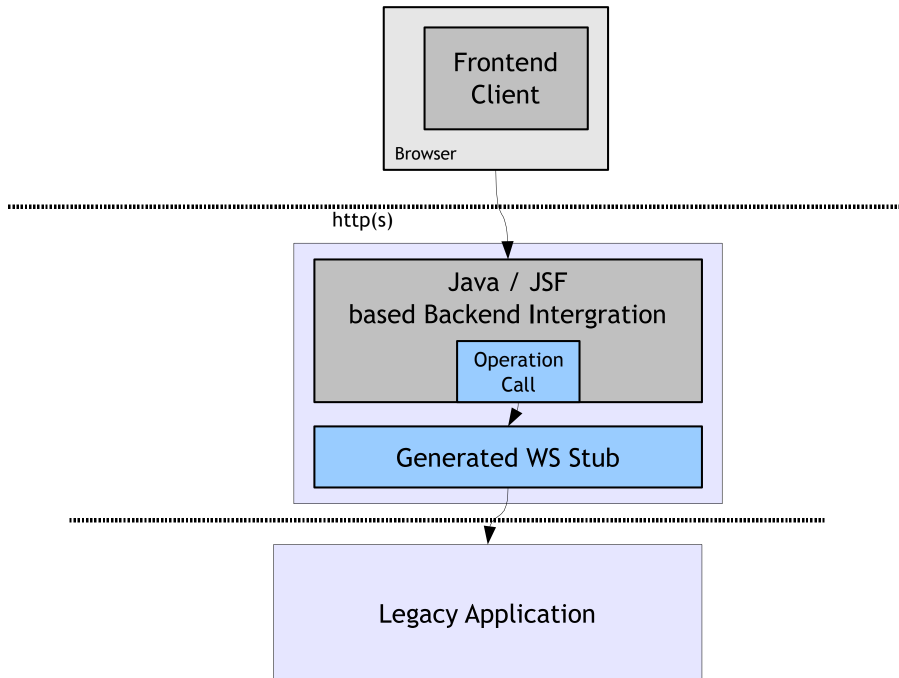
```
package genservices.ris_depositaccount;

....
....
....

public class TEMPLATE_CALL_list
{
    public void list() throws Exception
    {
        try
        {
            // ---------- parameters ----------
            Ris_DepositAccountGroupListElement p0 = new
Ris_DepositAccountGroupListElement();
            {
                Ris_DepositAccountGroupKeyType p0_0 = new Ris_DepositAccountGroupKeyType();
                p0.setRis_DepositAccountGroupListElement(p0_0);
                {
                    //p0_0.setAccountNumber(...);
                    //p0_0.setCustomerNumber(...);
                    //p0_0.setSortCode(...);
                }
            }
            Security p1 = new Security();
            {
                UsernameToken_type0 p1_0 = new UsernameToken_type0();
                p1.setUsernameToken(p1_0);
                {
                    //p1_0.setUsername(...);
                    //p1_0.setPassword(...);
                }
            }
            Ris_DepositAccountGroupHeader p2 = new Ris_DepositAccountGroupHeader();
            {
                //p2.setSOAGateway_Internal_AutoCommit(...);
                //p2.setTransactionState(...);
                //p2.setTransactionId(...);
                //p2.setConversationId(...);
                //p2.setConversationState(...);
                //p2.setVersion(...);
            }
            // ---------- invoke ----------
            Ris_DepositAccountRootElement r;
            Ris_DepositAccountRootServiceStub stub = new
Ris_DepositAccountRootServiceStub();
            // stub._getServiceClient().getOptions(). ...
            r = stub.list(p0,p1,p2);
            // ---------- result ----------
            Ris_DepositAccountRootElementType r_0 = r.getRis_DepositAccountRootElement();
            {
                Ris_DepositAccountRootType r_0_0 = r_0.getRis_DepositAccountRoot();
                {
                    Ris_DepositAccountGroupType[] r_0_0_0 =
r_0_0.getRis_DepositAccountGroup();
                    for (Ris_DepositAccountGroupType item: r_0_0_0)
                    {
                        //String ... = item.getBalance();
                        //String ... = item.getAccountNumber();
                        //String ... = item.getCustomerNumber();
                        //String ... = item.getAccountType();
                        //String ... = item.getSortCode();
                    }
                }
            }
        }
        catch (Exception exc)
```

```
            {
                org.eclnt.jsfserver.defaultscreens.Statusbar.outputError(exc.toString());
            }
        }
}
```

You see: the Java code is generated in a way so that it is easy to take over into existing Java coding.

Summary

The result: you now can embed web services directly into any Java interaction processing:



Declarative Approach for Structuring the Interaction Layer

The previous chapter told you about the “Java only” way: the interaction processing of the CaptainCasa Enterprise Client is implemented using simple Java classes which are the counterparts of pages, that are rendered on client side. From the Java classes (e.g. as reaction on a certain event) any further Java API can be called – in the case of web services this may be a calling of a generated stub.

Typical Interaction Tasks – Enriching a Web Service API

A web service API contains data segments that are passed into logical processing or that are the result of logical processing. When transferring a web service directly into a screen representation then the screen will not be usable for “normal” users. Examples:

- Web services contain “id-data” while the user always wants to see “text-for-ids”.
- The user wants to receive a certain guidance when entering data, e.g. the user wants to be informed about mandatory fields. The user wants to see if a field contains a value that contains wrong data, etc.
- The user wants to access valid-value information, e.g. instead of entering a certain value into a text field, the user wants to select the value from a list of valid values by using a combo box.

It is now the task of the interaction layer to “enrich” the data of the web service so that is can be presented to the “normal” user.

The enriching on the one hand is important for building usable user interfaces, on the other side is useful for pre-validating data before it is sent via web service. A web-service call is still an expansive call from system's perspective so it makes sense to have some pre-validation of data before calling it.

Declarative Approach

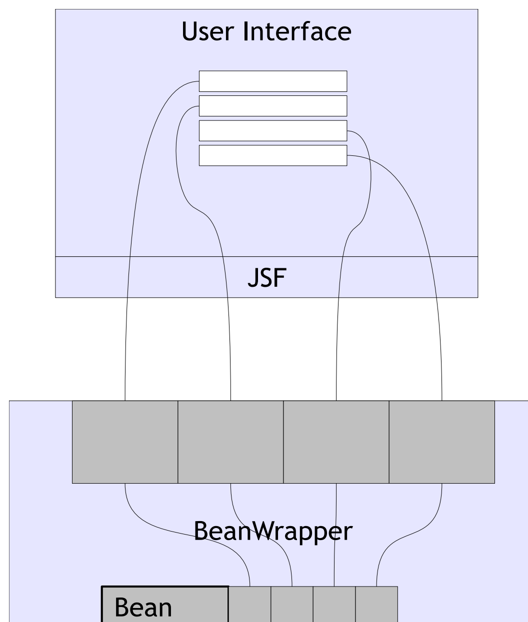
The goal now is, to enrich web services in a way that no (better: as less as possible) coding is required for the typical tasks when enriching a web service. - CaptainCasa provides a kit (that is part of the Web Service Integration Kit) that allows to enrich Java beans in general. The framework of this kit is also used in the Web Service environment.

Bean Wrapper

When using Axis2 then a web service call is represented by a method into which certain beans are passed (web service parameters) and from which a certain bean as result is received (web service result). Each bean typically represents a certain XML segment.

The framework wraps each bean by a “bean wrapper” object. The bean wrapper exposes each property of the bean as complex object. The complex object is providing a lot of extra information that is referenced from the user interface components. Per property the following information is added:

- Coloring: e.g. colored as mandatory input, colored so that error is indicated
- Text support: a text is loaded for a certain id where required
- Valid value support: a list of valid values is supported where required
- ...and some more information...

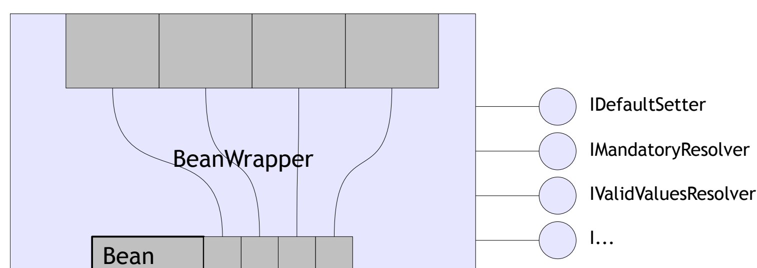


The bean wrapper internally introspects the wrapped bean, and creates the enriched property objects dynamically.

Interface Configuration

Now that we have an enriched bean object, it's the next question where the bean wrapper takes the information from in order to e.g. decide if a certain input is mandatory. The approach that is taken is as follows:

Any question the bean wrapper has in order to get to know details about how to enrich the contained bean is asked to an outside interface. Example: when being created the bean wrapper wants to know about which properties are mandatory and which are not. This information is requested by the bean wrapper calling an interface "IMandatoryResolver".



When working with interfaces then there needs to be instance to bind the correct implementation of an interface into the right object. There is a nice framework to take over this job: Spring.

Spring allows to configure the assembling of objects in a declarative, powerful way. Spring passes the right interface implementations into the bean wrapper objects. Spring serves as object factory binding the user of interfaces to its interface implementations.

Example: the question "What are my valid values for a certain value?" which is abstracted in the interface "IValidValuesResolver" can have multiple implementations:

- The valid values may be "hard coded".

- The valid values may be kept in a property file on interaction layer.
- The valid values may be picked by calling a certain web service. - Maybe the result of the web service will be buffered on interaction layer level, so that the next request for valid values will not trigger a call of the web service but will directly respond with data that was buffered.

You see: the question “which are my valid values for a certain property” can be implemented in multiple ways – each making sense for a certain scenario.

The configuration of spring is kept as XML file. You will see an example later on.

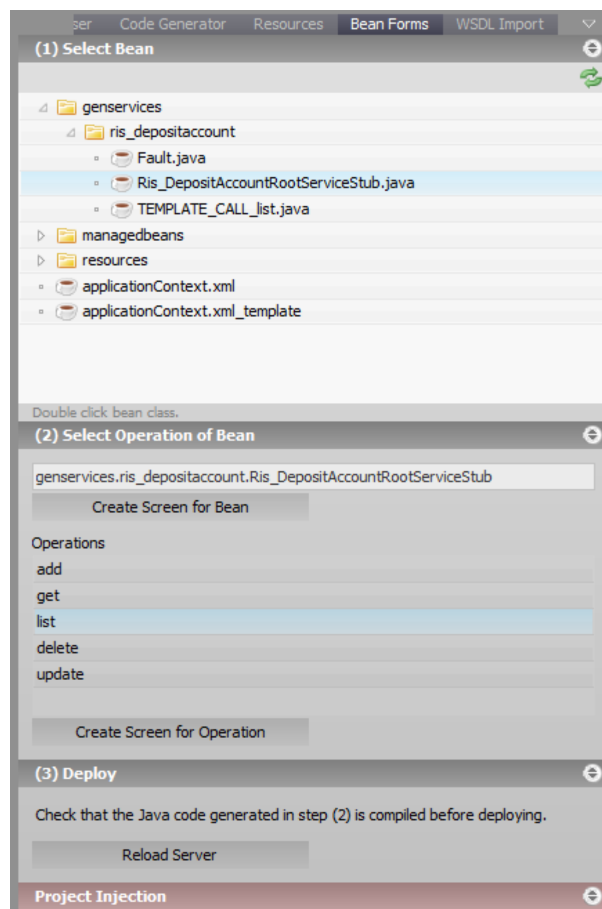
Using the Declarative Approach

Based on the declarative approach you now can build web pages for a certain web service operation. There are two steps:

- You create a page from the web service operation. The page will be a rather technical reflection of the web service structure.
- You enrich the technical reflection by adding more and more UI relevant information so that at the end the screen will become usable.

Generating a Screen for a Web Service Operation

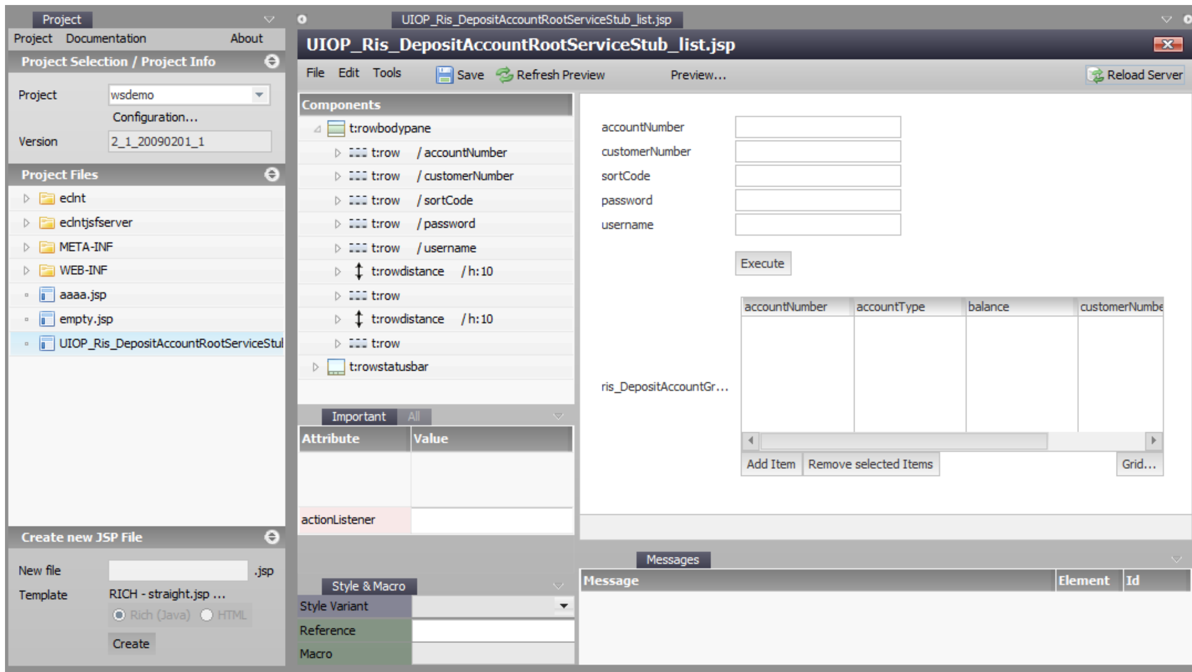
We assume that you already loaded the “RIS_DepositAccount.wsdl” web service description into your project and generated the Java stub. If not: please check the previous section of this document.



In the tool “Bean Forms” you see the generated stub for the web service. After double clicking you see the list of available operations. By selecting “list” and pressing the button “Create Screen for Operation” two artifacts will be generated:

- A default JSP page that is a technical reflection of the web service.
- A default program that serves the JSP page.

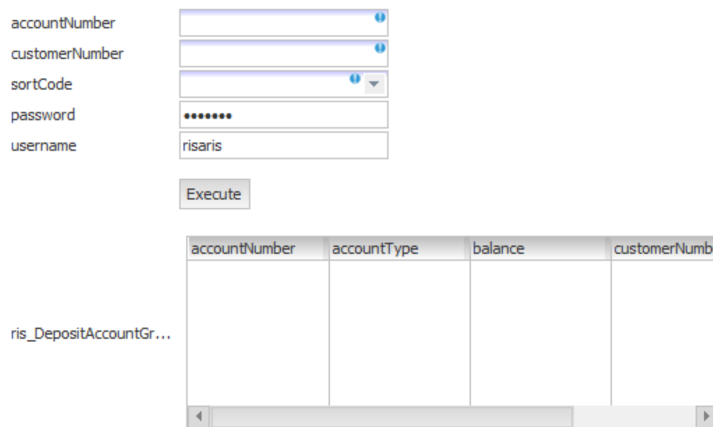
After pressing “Reload Server” the page will be visible in the project overview:



In the preview of the Layout Editor you can immediately test the web service: input “*” for the search parameters, use password “letmein” and username “risaris” and press “Execute” - and the list should be populated due to the web service call.

Enriching the Screen Interaction Logic

Now the enrichment is to be done. Instead of explaining each step we directly confront you with the result:



You see: some fields have changed to indicate mandatory input, there are valid values available for the “sortCode” and “password” and “username” are per-set with reasonable

values.

Now let's take a look into the Spring configuration file which is kept in “src/applicationContext.xml” of your project:

```
<?xml version="1.0" encoding="UTF-8"?>
<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.xsd">

    <bean id="GenericWrapper" class="org.eclnt.beanmgmt.wrapper.BeanWrapper"
scope="prototype">
    </bean>

    <bean id="VVSortCode" class="org.eclnt.beanmgmt.wrapper.StaticValidValuesResolver"
scope="singleton">
        <property name="validValues">
            <list>
                <value>*</value><value>All sortcodes (*)</value>
                <value>1</value><value>Prio A</value>
                <value>2</value><value>Prio B</value>
            </list>
        </property>
    </bean>

    <bean id="VVAccount" class="org.eclnt.beanmgmt.wrapper.PropertyFileValidValuesResolver"
scope="singleton">
        <property name="resourceName">
            <value>resources/accounts</value>
        </property>
    </bean>

    <bean id="Ris_DepositAccountRootServiceStub_UsernameToken_type0Wrapper"
class="org.eclnt.beanmgmt.wrapper.BeanWrapper" scope="prototype">
        <property name="defaultSetters">
            <list>
                <bean class="org.eclnt.beanmgmt.wrapper.StaticDefaultSetterString">
                    <property name="property"><value>username</value></property>
                    <property name="propertyValue"><value>risaris</value></property>
                </bean>
                <bean class="org.eclnt.beanmgmt.wrapper.StaticDefaultSetterString">
                    <property name="property"><value>password</value></property>
                    <property name="propertyValue"><value>letmein</value></property>
                </bean>
            </list>
        </property>
        <property name="mandatoryResolver">
            <bean class="org.eclnt.beanmgmt.wrapper.StaticMandatoryResolver">
                <property name="mandatories">
                    <set>
                        <value>username</value>
                        <value>password</value>
                    </set>
                </property>
            </bean>
        </property>
    </bean>

    <bean id="Ris_DepositAccountRootServiceStub_Ris_DepositAccountGroupKeyTypeWrapper"
class="org.eclnt.beanmgmt.wrapper.BeanWrapper" scope="prototype">
        <property name="mandatoryResolver">
            <bean class="org.eclnt.beanmgmt.wrapper.StaticMandatoryResolver">
                <property name="mandatories">
                    <set>
                        <value>accountNumber</value>
                        <value>customerNumber</value>
                        <value>sortCode</value>
                    </set>
                </property>
            </bean>
        </property>
        <property name="validValuesResolvers">
            <map>
                <entry><key><value>accountNumber</value></key><ref
bean="VVAccount"/></entry>
                <entry><key><value>sortCode</value></key><ref bean="VVSortCode"/></entry>
            </map>
        </property>
    </bean>

</beans>
```

If you know Spring already then you will not have problems to de-code this file...

If you do not know Spring then you may see that for each web-service-XML-segment

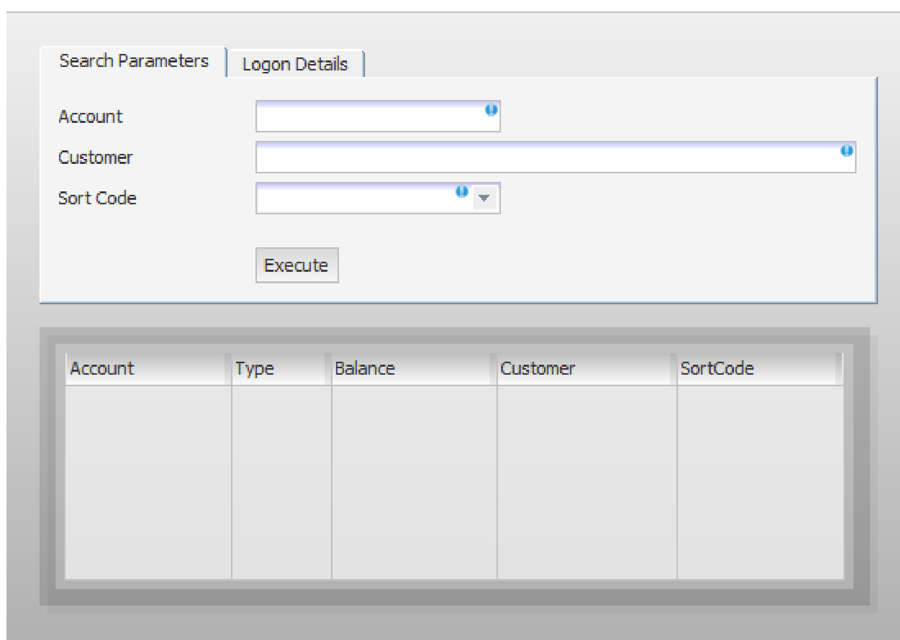
(represented by a corresponding class in the stub) there is a certain declaration. In the declaration properties (e.g. "mandatoryResolver") are set – e.g. with instances of class "StaticMandatoryResolver".

In the area of property "ValidValuesResolvers" you may see that referenced to objects are made that are defined centrally ("VVSortCode" and "VVAccount"). These central definitions are defined in a different way: "VVSortCode" is an implementation of "IValidValuesResolver" in which the valid values are directly specified, "VVAccount" is an implementation of "IValidValuesResolver" in which the valid values are taken from a resource file.

....hmmm, all this may sound complex, if you are not familiar with Spring already. But it's a matter of digging deeper and deeper into the area of integrating a rich client framework into web service processing. - A deeper explanation will be too big to be continued as part of this documentation. Please contact CaptainCasa for detailed information.

Enriching the Screen

Now, fun starts...: you take the screen that was generated and update its elements in the Layout Editor. And you may put some styling behind. The result is a page that may look like:



This user interface now can be directly used inside a browser or it can be embedded into workplace scenarios in which a couple of user interfaces that are available for a certain user/ role are arranged, so that the user can pick his/her functions.

Summary

Enriching the logic-oriented web service API in order to make it usable from a user-friendly page requires some effort. Instead of individually coding you can build up interaction frameworks which standardize and simplify the process of enriching.

By using wrappers each object that is part of a web service call can be enriched by default. The configuration of enrichment consists out of assigning interface implementations to the wrapper's interface requirements. Spring is a very usable

framework to do this.

...and, you see: there is still some thinking and structuring left on interaction side!

And this is “obvious”, because what you basically do, is: you push your legacy system into the role of a “processing system”. It's your “working horse”. You refactor the logic of this system, so that it does not contain any UI logic anymore.

Well, where does this UI logic go to? It is moved into the interaction layer in front of the web services. And that's the right place: it is close to the rich client processing, e.g. it is close to the JSF layer. It is using Java – an environment where you might doubt, if it's the right one for the “heavy processing” - the one that you still do within your legacy environment! But: it is the right environment for flexible, pattern oriented interaction implementations.

Installing the Web Service Integration Kit

Installation of the Kit

Installation is simple:

- Close your development tools.
- Extract the zip file of the Web-Service-Integration-Kit into the toolset's web application. In default installations this is:

```
<installdir>
    /tools
        /embeddedserver
            /webapps
                /editor     <== directory in which to extract
```

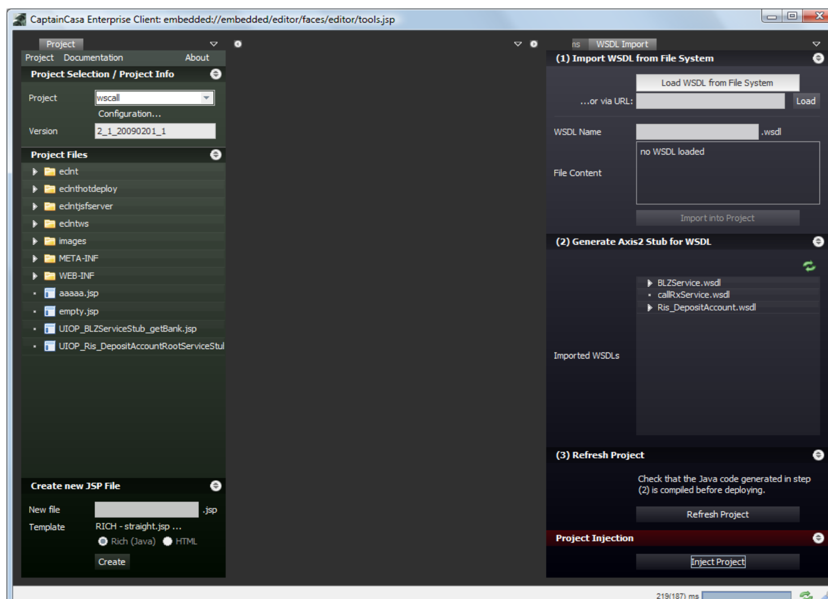
That's it! After the installation the /editor-directory should be updated so that it looks like:

```
<installdir>
    /tools
        /embeddedserver
            /webapps
                /editor
                    /tools
                        /ccbean
                        /ccwscall
                   /WEB-INF
                       /lib
                           eclnteditortoolccbean.jar
                           eclnteditortoolccwscall.jar
```

Project Injection

If using the Web Service Integration Kit within an project then certain information (libraries) need to be added to the project. This step is called “project injection”.

The project injection is available from the toolset: use the corresponding buttons on the bottom of the tool “WSDL Import” and “Bean Forms”:



Eclipse Plug-in

Part of the Web Service Integration Kit is the generation of Java code. As consequence you required a development environment for Java coding purposes.

The default environment that we reference to is Eclipse. For Eclipse there is a CaptainCasa plug-in that automates the file synchronization: whenever files are updated within the CaptainCasa toolset then a refresh of the corresponding project is executed within the Eclipse environment, so that the Eclipse view is kept up to date.

The Eclipse plug-in is part of the normal CaptainCasa Enterprise Client environment. Please read the “Installation and Configuration Guide” for getting information how to install.

Folgende Handelsmarken oder eingetragenen Markenzeichen werden innerhalb dieser Dokumentation referenziert:

Java - Sun Microsystems Inc.; UNIX – The Open Group; SAP, R/3, ABAP – SAP AG; Microsoft, Windows – Microsoft Corporation; BEA, WebLogic – Oracle Corporation; JBoss – Red Hat Middleware; IBM, Websphere – IBM Corporation; Tomcat – Apache Group

Alle anderen genannten Handelsmarken und Produkte sind Handelsmarken oder eingetragene Warenzeichen ihrer Eigentümer.

CaptainCasa Enterprise Client

CaptainCasa GmbH

Hindemithweg 13
69245 Bammental

Tel +49 6223 484147

http://www.CaptainCasa.com
info@CaptainCasa.com