JAX-WS - 1.1

On this page:

Target audience:

Users of the extension providing **JAX-WS** support for SOAP Web Services.

Summary: This document provides information about the extension providing **JAX-WS** support for Web Services.

Extension ID

com.castsoftware.jaxws

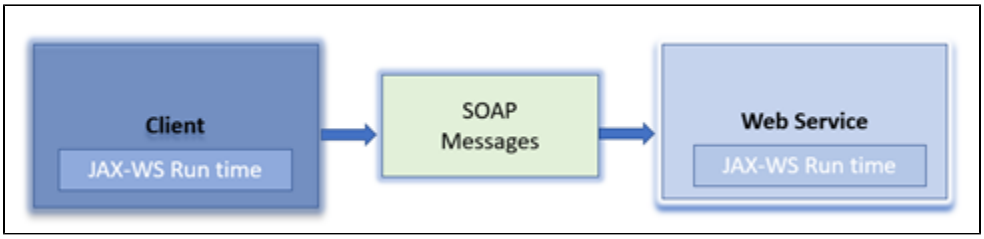
What's new?

Please see JAX-WS - 1.1 - Release Notes for more information.

Description

In what situation should you install this extension?

The main purpose of this extension is to enable linking client side requests to the server side services that use JAX-WS. If your JEE application contains source code which uses **JAX-WS** (**JSR 224)** and you want to view these object types and their links with other objects, then you should install this extension.



Features

Annotations

This extension handles JAX-WS web services (particularly SOAP services) used in JEE applications. JAX-WS contains a collection of annotations that enables the definition of the web service contract directly inside the java code:

- JAX-WS Service is basically defined by a **javax.jws.WebService** (**@WebService**) annotation set on top of a class. This annotation may also be set on top of an interface, in this case, the interface will be the "Service Endpoint Interface" and no new web service will be created during the analysis
- An operation represents an action that can be triggered by a client application. It is represented by an object called "**SOAP Java Operation**". One operation represents a Java method of a **@WebService** class that is an annotation with **@WebMethod**.
- A port type represents a collection of operations, it is represented by an object called "**SOAP Java Port Type**" which is a child of the java file containing the class annotated by **@WebService**.
- The JAX-WS Extension also handles the annotations **@WebServiceClient** and **@WebEndpoint**. Two different types of object are created to represent these items: "**SOAP Java Client**" and "**Soap Client end point"**. Each web end point contains a list of operations called "**SOAP Java Client Operation**" and they represent the operations that can be remotely invoked on the server offering the web service.

For example, the following code:

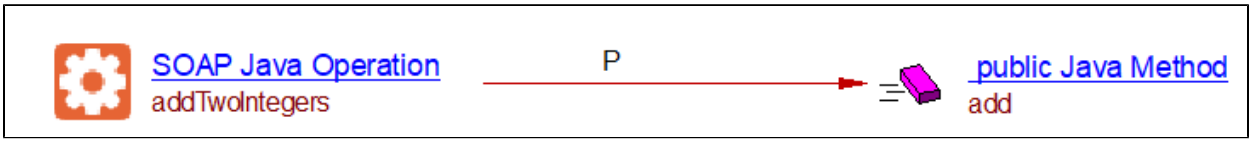
CalculatorImplCeiling.java

```
@WebService(
 serviceName="CalculatorServiceCeiling",
 portName="CalculatorPort",
 name="CalculatorCeiling",
 endpointInterface="com.castsoftware.ws.Calculator",
 targetNamespace="http://ws.castsoftware.com/")
public class CalculatorImplCeiling implements Calculator {
@Override
 public float add(int x, int y) {
 return x+y;
 }
}
```

Calculator.java

```
@WebService()
public interface Calculator {
 @WebMethod(operationName="addTwoIntegers")
 float add(int x, int y);
}
```

Will generate:



Support for Apache CXF

The Apache CXF API offers a mechanism where the web service is implemented without using annotations. It is instead specified inside an XML file (cxf-context.xml file). This is in fact a kind of integration with Spring. This extension will handle JAX-WS seb services generated using the Apache CXF framework.

A JAX-WS web service implemented by a Spring bean defined in a Spring XML configuration file and declared via the Apache CXF should be correctly detected and created with this extension. All expected operations and links will be created.

For example, the following code:

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
>

        <display-name>Product Service Web Service</display-name>

        <context-param>
                <param-name>contextConfigLocation</param-name>
                <param-value>/WEB-INF/beans-cxf.xml</param-value>
                <!-- <param-value>/WEB-INF/mvc-dispatcher-servlet.xml</param-value> -->
        </context-param>
```

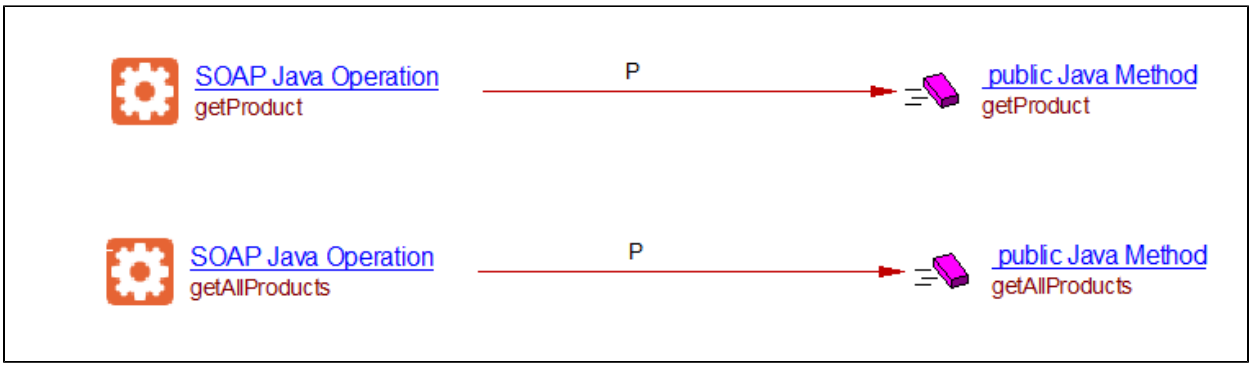
beans-cxf.xml

```
<jaxws:endpoint id="productEndpoint"
                implementor="com.spring_cxf.ProductServiceImpl"
                address="/products" />
```

ProductServiceImpl.java

```
package com.spring_cxf;

import java.util.List;

public class ProductServiceImpl implements ProductService {

        private ProductServiceMockDaoImpl productServiceMockImpl;

        public void setProductServiceMockImpl(ProductServiceMockDaoImpl productServiceMockImpl) {
                this.productServiceMockImpl = productServiceMockImpl;
        }
        public Product getProduct(int productId) {

                return productServiceMockImpl.getProduct(productId);
        }

        public List<Product> getAllProducts() {
                return productServiceMockImpl.getAllProducts();
        }

}
```

will generate:



Support for Server Tag in CXF web service

Apache CXF allows the creation of **jaxws:server** endpoints based on **Server** and **ServerBeans** tags defined in the **applicationContext.xml** file. The **appli cationContext.xml** file contains the mapping to the **implementor** class. The implementor class contains the web service operations definitions. This extension is capable of handling these scenarios and is able to create the expected links and objects.

In the example below, the **serviceBean** is **"#administration"** and **administration** is mapped to the implementing class **cpp.administrationGestionnaire. Administration**. The required objects and links for the class **Administration** should be created during an analysis.

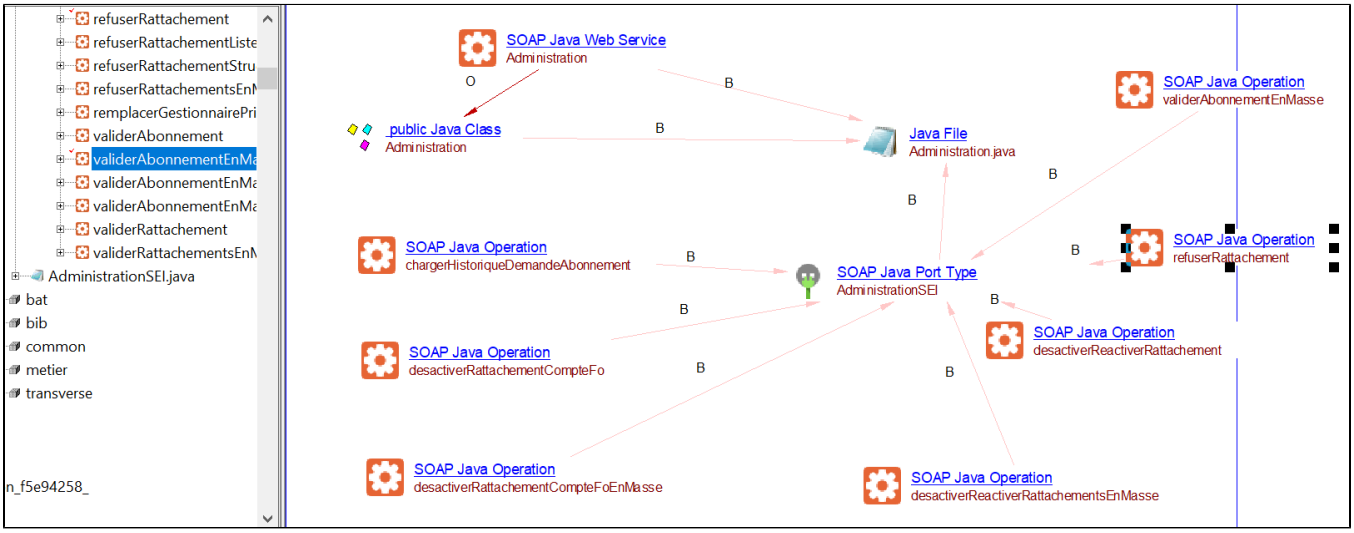
applicationContext.xml

```
<jaxws:server serviceClass="company.com.cpp.administrationGestionnaire.AdministrationSEI"
    address="/Administration" serviceBean="#administration">
</jaxws:server>

<!-- liaison avec Spring l'implémentation -->
<bean id="administration" class="company.com.cpp.administrationGestionnaire.Administration">
</bean>
```

Administration.java

```
@WebService(endpointInterface = "company.com.cpp.administrationGestionnaire.AdministrationSEI", serviceName = "Administration")
public class Administration implements AdministrationSEI
{
```

This will generate the following objects and links:



Support for Spring-WS

WebServiceTemplate

Spring-WS provides a client-side web service API that allows for consistent, XML-driven access to web service. 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. This extension will identify client-side services using WebServiceTemplate Methods that are used to invoke the web service.

For example, the following code:

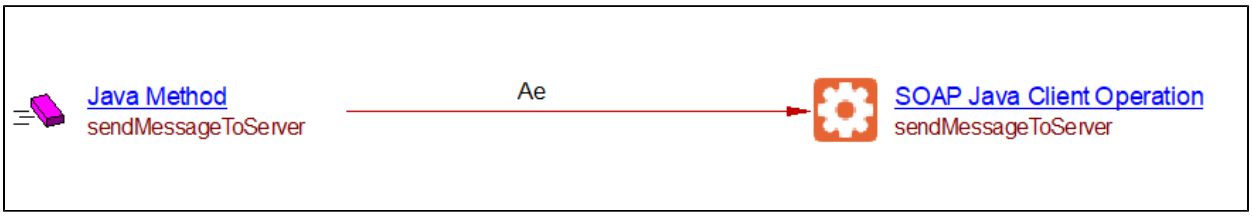
TempClass.java

```
public class TempClass {

         private final WebServiceTemplate webServiceTemplate = new WebServiceTemplate();
         private static final String MESSAGE =
                        "<message xmlns=\"http://tempuri.org\">Hello Web Service World</message>";
         void myFunc(String str)
         {
                 System.out.println(str);
         }
         void sendMessageToServer(ClciUsoReadRequest123 request)
         {
                 StreamSource source = new StreamSource(new StringReader(MESSAGE));
             StreamResult result = new StreamResult(System.out);
             myFunc("This to test");
                 webServiceTemplate.marshalSendAndReceive(request);
         }
}
```

TestFile.wsdl

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:tns="http://sam.att.com/dns/samservice" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        targetNamespace="http://sam.att.com/dns/samservice">
        <wsdl:portType name="CLCIQueryPortType123">
                <wsdl:operation name="getClciUso123">
                        <wsdl:input message="tns:ClciUsoReadRequest123" />
                        <wsdl:output message="tns:ClciUsoReadResponse123" />
                        <wsdl:fault name="WSException" message="tns:WSException" />
                </wsdl:operation>
        </wsdl:portType>
</wsdl:definitions>
```

Will generate:



@Endpoint and @PayloadRoot annotations

The classes where SOAP web service requests are handled are annotated by @Endpoint. The analyzer inspects these classes in search of further @PayloadRoot-annotated methods. This annotation maps the method to the root element of the request payload. It should be noted that typical setup of Spring Web Service configurations is performed from initial .xsd schema files to generate .java files and stubs as well as contract .wsdl files. These .wsdl files are generated following some customizable naming conventions (using for example *jsxb2-maven-plugin*). The analyzer directly searches the corresponding SOAP web service operations exposed by directly inspecting the nearby .wsdl files in the analysis unit. Thus the presence of .wsdl files (as it is usually the case) in the source code is necessary for the analyzer to correctly interpret SOAP web service operations based on the @PayloadRoot annotation.

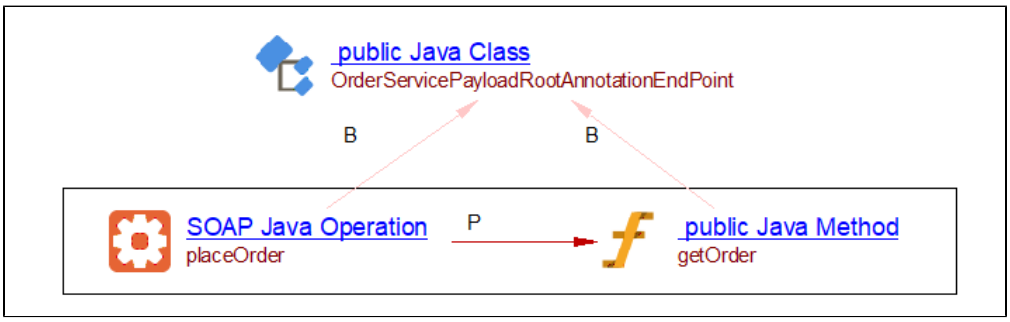
Consider the below example code:

```
// Source: http://justcompiled.blogspot.com/2010/09/building-web-service-with-spring-ws.html
package com.live.order.service.endpoint;

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

@Endpoint
public class OrderServicePayloadRootAnnotationEndPoint {

    @PayloadRoot(localPart = "placeOrderRequest", namespace = "http://www.liverestaurant.com/OrderService
/schema")
    public JAXBElement<PlaceOrderResponse> getOrder(...) {
}
```

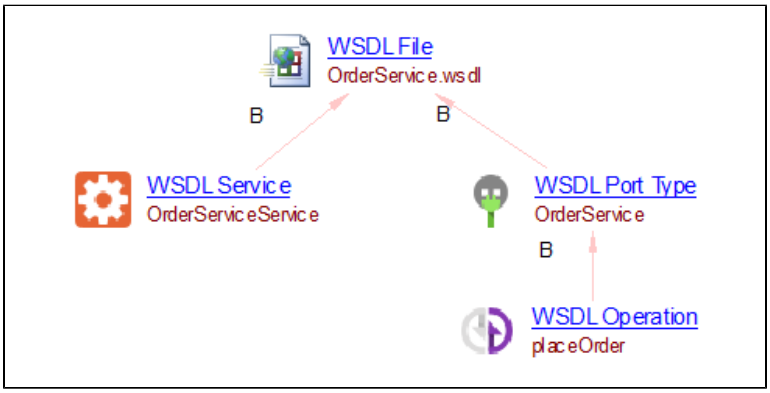
Together with an excerpt of the .wsdl file, note the connection between the localPart and the name of the input message

```
    <wsdl:portType name="OrderService">
        <wsdl:operation name="placeOrder">
            <wsdl:input message="tns:placeOrderRequest" name="placeOrderRequest">
            </wsdl:input>
            <wsdl:output message="tns:placeOrderResponse" name="placeOrderResponse">
            </wsdl:output>
        ...
    </wsdl:portType>
```

The analyzer will create a SOAP Java Operation object (with fullname *OrderService.placeOrder*) together with a link to the handler method *getOrder*.



A warning note: the .wsdl file itself will contain a WSDL Operation object that can lead to confusion:



This WSDL Operation, though related, does not belong to com.castsoftware.jaxws, and in contrast to SOAP Java Operation objects, it is not expected to participate in transactions, and thus only inter-techno links are expected to be found for the latter (via the web-service linker at application analysis level).

Support for WS call using QName

The web service can be published in a server and the wsdl can be used on the client side referring to the hosted URL and web service. The publisher of the web service uses the URL and the web service to publish the web service. For example "http://<hostname>:port/service". The client can use the web service by referring to the wsdl and creating a service. Client needs to create a Qualified Name to refer to the service. Once the service is referred it can use any port and request the service. For example:

CalWebServiceImpl.java

```
package com.cast;

import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

@WebService(endpointInterface = "com.castwork.CalWebService")
@SOAPBinding(style = Style.RPC)
public class CalWebServiceImpl implements CalWebService{
        @Override
        public int add(int val1, int val2){
                return val1+val2;
        }
}
```

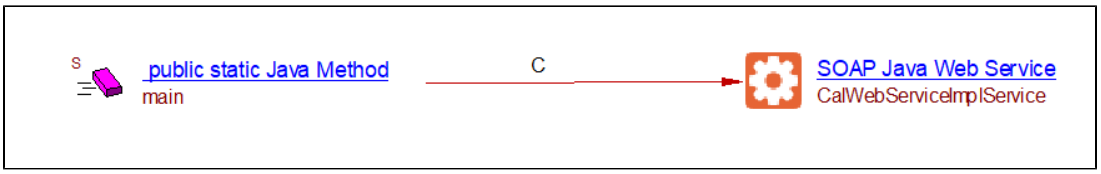
Publish.java

```
package com.cast;

import javax.xml.ws.Endpoint;

public class Publish {
        public static void main(String []args)
        {
                Endpoint endpoint = Endpoint.publish("http://localhost:8080/cal",new CalWebServiceImpl());
                System.out.println(endpoint.isPublished());
        }
}
```

CalClient.java

```
package com.cast;

import java.net.MalformedURLException;
import java.net.URL;

import javax.xml.ws.Service;

import javax.xml.namespace.QName;

public class CalClient {

        public static void main(String[] args) throws MalformedURLException {
                // TODO Auto-generated method stub
                URL url = new URL("http://localhost:8080/cal?wsdl");
                javax.xml.namespace.QName qName = new QName("http://rajeshwork.com/","
CalWebServiceImplService");
                Service service = Service.create(url,qName);
                CalWebService calService = service.getPort(CalWebService.class);
                System.out.println(calService.add(100, 400));

        }

}
```

In the above code **CalClient** is using the service. So there should be a link between client and server. Using this the JAX-WS extension the link shown below can be obtained:



Support of Handlerchain

This extension handles the links to the **close**, **handleMessage** and **handleFault** methods which will be called for the request or response. @HandlerChain annotation receives as argument the path to the xml file where the handle class name is specified (we don't support paths given as URLs). The class containing the handler methods is identified in the handler file and the respective call-links are created from the operations to the methods.

For example, the following code:

handler-chain.xml

```
<handler-chains xmlns:javaee="http://java.sun.com/xml/ns/javaee">
        <handler-chain>
                <handler>
                        <handler-name>com.cast.handler.MacAddressValidatorHandler</handler-name>
                        <handler-class>com.cast.handler.MacAddressValidatorHandler</handler-class>
                </handler>
        </handler-chain>
</handler-chains>
```

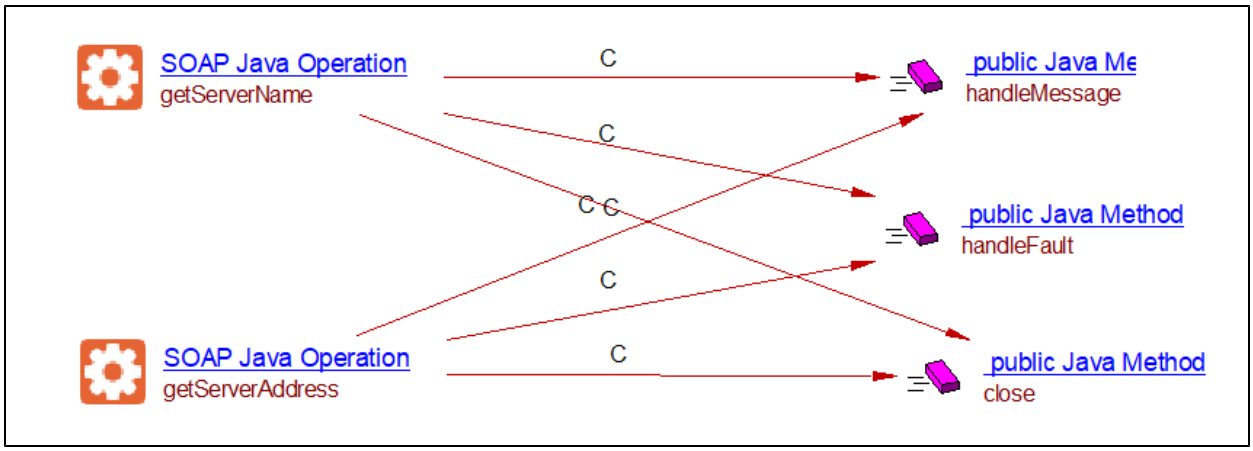
MacAddressValidatorHandler.java

```
public class MacAddressValidatorHandler implements SOAPHandler<SOAPMessageContext>{

        @Override
        public boolean handleMessage(SOAPMessageContext context) {
                return true;
        }
        @Override
        public boolean handleFault(SOAPMessageContext context) {
                System.out.println("Server : handleFault()......");
                return true;
        }
        @Override
        public void close(MessageContext context) {
                System.out.println("Server : close()......");
        }
}
```

ServerInfo.java

```
@WebService
@HandlerChain(file="handler-chain.xml")
public class ServerInfo{

        @WebMethod
        public String getServerName() {

                return "Cast server";

        }
        @WebMethod
        public String getServerAddress() {

                return "Cast server Address";
        }
}
```

will generate:



Support for Jax-RPC

The analyzer inspects *webservices.xml* and *ejb_jar.xml* in search of SOAP operations configured by Jax-RPC framework.

Function Point, Quality and Sizing support

This extension provides the following support:

Function Points (transactions)	Quality and Sizing
✅	❌

CAST AIP compatibility

This extension is compatible with:

CAST AIP release	Supported
8.3.x	✓
8.2.x	✓

Supported DBMS servers

This extension is compatible with the following DBMS servers:

CAST AIP release	CSS	Oracle	Microsoft
All supported releases (see above)	✓	✓	✗

Prerequisites

✓	An installation of any compatible release of CAST AIP (see table above)

Dependencies with other extensions

Some CAST extensions require the presence of other CAST extensions in order to function correctly. The JAX-WS extension requires that the following other CAST extensions are also installed:

- **Web services linker service** (internal technical extension).

> Note that when using the **CAST Extension Downloader** to download the extension and the **Manage Extensions** interface in **CAST Server Manager** to install the extension, any dependent extensions are **automatically** downloaded and installed for you. You do not need to do anything.

Download and installation instructions

Please see:

- Download an extension
- Install an extension

> The latest release status of this extension can be seen when downloading it from the CAST Extend server.

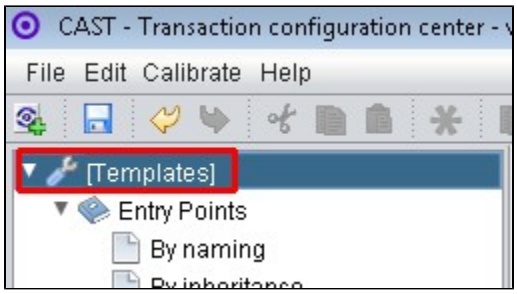
CAST Transaction Configuration Center (TCC) Entry Points

In **JAX-WS 1.0.x**, a set of JAX-WS **Transaction Entry / End Points** for use in the CAST Transaction Configuration Center is delivered in the extension via a .TCCSetup file. Therefore If you are using **JAX-WS 1.0.x**:

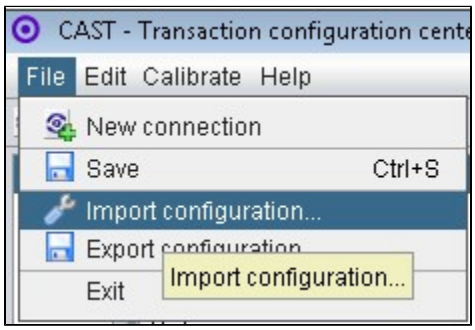
- with **CAST AIP 8.3.x**, there is nothing for you to do: these entry / end points will be automatically imported during the extension installation and will be available in the CAST Transaction Configuration Center under "Entry Points > Free Definition".
- with **CAST AIP 8.2.x**, you can manually import the file **Configuration\TCC\Base_JAXWS.TCCSetup** to obtain your entry / end points in the "**Free Definition**" section (see instructions below).

Manual import action for CAST AIP 8.2.x

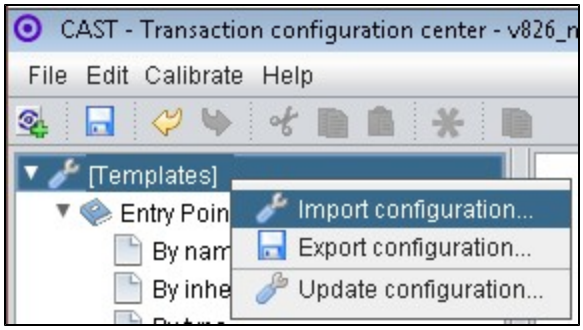
- Locate the .TCCSetup file in the extension folder: **Configuration\TCC\Base_JAXWS.TCCSetup**
- In the CAST Transaction Configuration Center, ensure you have selected the **Templates** node:



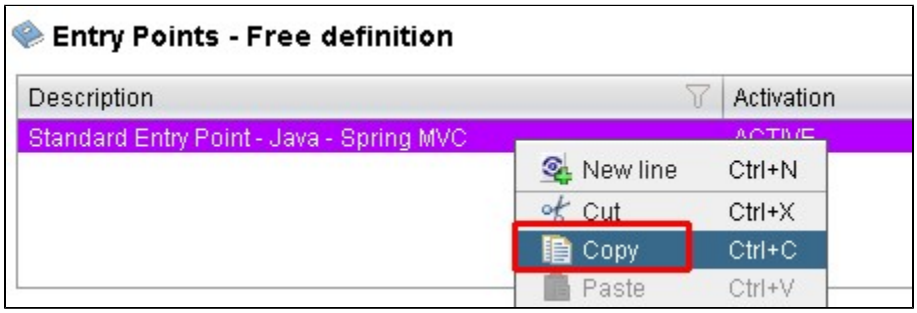
- This .TCCSetup file is to be imported into the CAST Transaction Calibration Center using either the:
  - **File** > **Import Configuration** menu option:



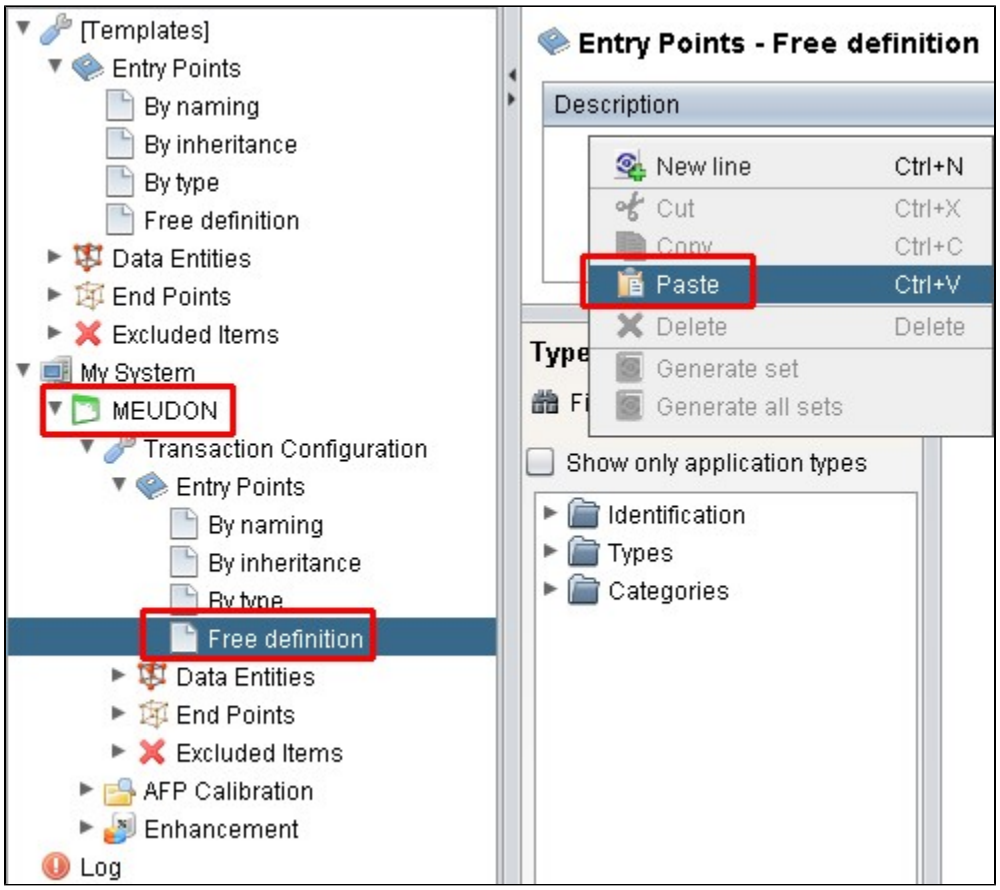
  - Or right clicking on the **Template node** and selecting **Import Configuration**:



- The import of the "**Configuration\TCC\Base_JAXWS.TCCSetup**" file will provide you with a sample Transaction Entry point in the **Free Definition** node under **Templates.**
- Now right click the "**Standard Entry Point**" item and select copy:



- Paste the item into the **equivalent node** under the **Application**, for example, below we have copied it into the **Application MEUDON**:



- Repeat for any additional items or generic sets that have been imported from the .TCCSetup file.

Packaging, delivering and analyzing your source code

Once the extension is installed, no further configuration changes are required before you can package your source code and run an analysis. The process of packaging, delivering and analyzing your source code does not change in any way:

- **Package and deliver** your application (that includes source code which uses **JAX-WS**) in the exact same way as you always have.
- **Analyze** your delivered application source code in the CAST Management Studio in the exact same way as you always have - the source code which uses **JAX-WS** will be detected and handled correctly.

What results can you expect?

- The extension is shipped with a set of CAST Transaction Configuration Center **Entry Points**, specifically related to JAX-WS Please see CAST Transaction Configuration Center (TCC) Entry Points for more information about this.
- Once the analysis/snapshot generation has completed, **HTTP API** transaction entry points will be available for use when configuring the CAST Transaction Configuration Center. In addition, you can view the results in the normal manner (for example via CAST Enlighten).

Objects

The following objects are displayed in CAST Enlighten:

Icon	Object Type
	SOAP Java Web Service
	SOAP Java Port Type
	SOAP Java Operation
	SOAP Java Client Operation

	SOAP Java Client
	SOAP Client end point