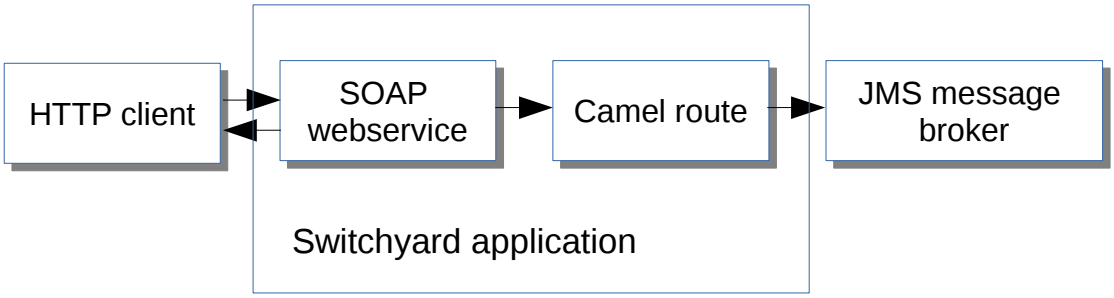
Dissecting the `soaptest` Switchyard example

Kevin Boone, July 2017

This document explains how the simple soaptest Switchyard example works. It isn't intended to be a tutorial on how to create the application using, for example, JBDS tooling; rather it shows how the various elements of the application, as specified in the switchyard.xml file, are interconnected.

Overview

The diagram below shows the application in outline. It implements a SOAP webservice in Java code, which passes data to a Camel route, which in turn produces a message to a JMS message queue. No significant processing occurs at any stage -- the purpose of the example is to show how Switchyard components, services, and gateways work together.

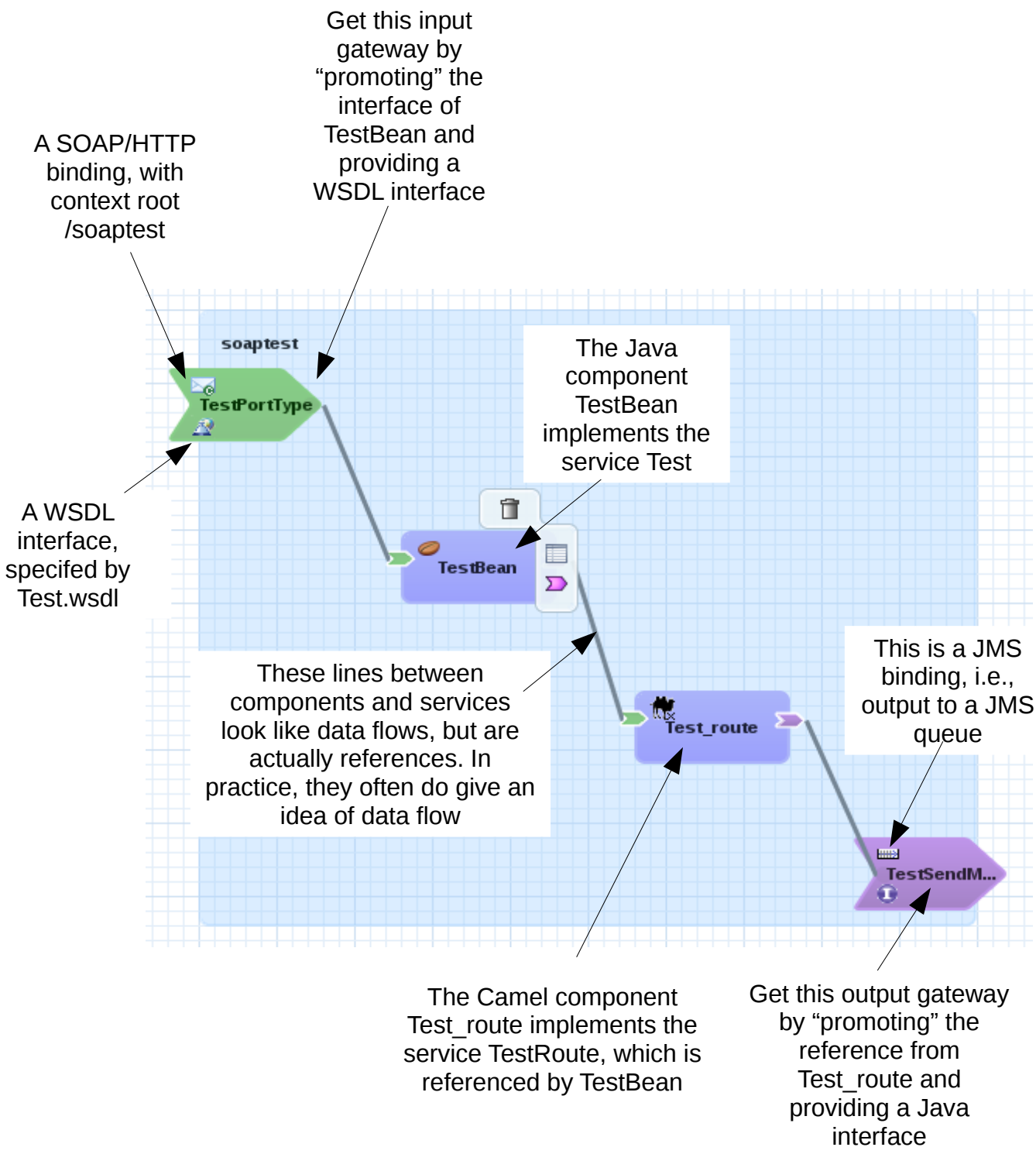


Development and testing

`soaptest` was created using JBoss Developer Studio (JBDS) version 9.1, and tested using JBoss Fuse 6.3, running on JBoss EAP 6.4.

The switchyard.xml file

The diagram below shows how `switchyard.xml` is represented graphically in JBDS, annotated to show some of the relationships between the elements.


A SOAP/HTTP binding, with context root /soaptest
Get this input gateway by “promoting” the interface of TestBean and providing a WSDL interface
soaptest
TestPortType
A WSDL interface, specifed by Test.wsdl
The Java component TestBean implements the service Test
TestBean
These lines between components and services look like data flows, but are actually references. In practice, they often do give an idea of data flow
Test_route
This is a JMS binding, i.e., output to a JMS queue
TestSendM...
The Camel component Test_route implements the service TestRoute, which is referenced by TestBean
Get this output gateway by “promoting” the reference from Test_route and providing a Java interface

The TestBean component

`TestBean` is a Java class, specified by an interface `Test`. It will be exposed as a SOAP-based webservice, specified by a WSDL file.

`TestBean` takes an object of class `TestIn` as input, a returns an object of class `TestOut`. These objects are converted to and from XML using JAXB, as controlled by the Switchyard transformer specification. In `switchyard.xml` we have (among others):

```
<transform:transform.jaxb
  from="java:com.example.switchyard.soaptest.TestOut"
  to="{urn:com.example.switchyard:soaptest:1.0}testOut"
  contextPath="com.example.switchyard.soaptest"/>
```

That is, we will use JAXB to transform the object `TestOut` to an an XML element `<testOut>` in the specified namespace. This transformation is largely automatic -- JAXB defaults are used for the mappings between Java property names and XML elements. All that is necessary to make this happen is to add the necessary JAXB annotations to the `TestIn` and `TestOut` classes:

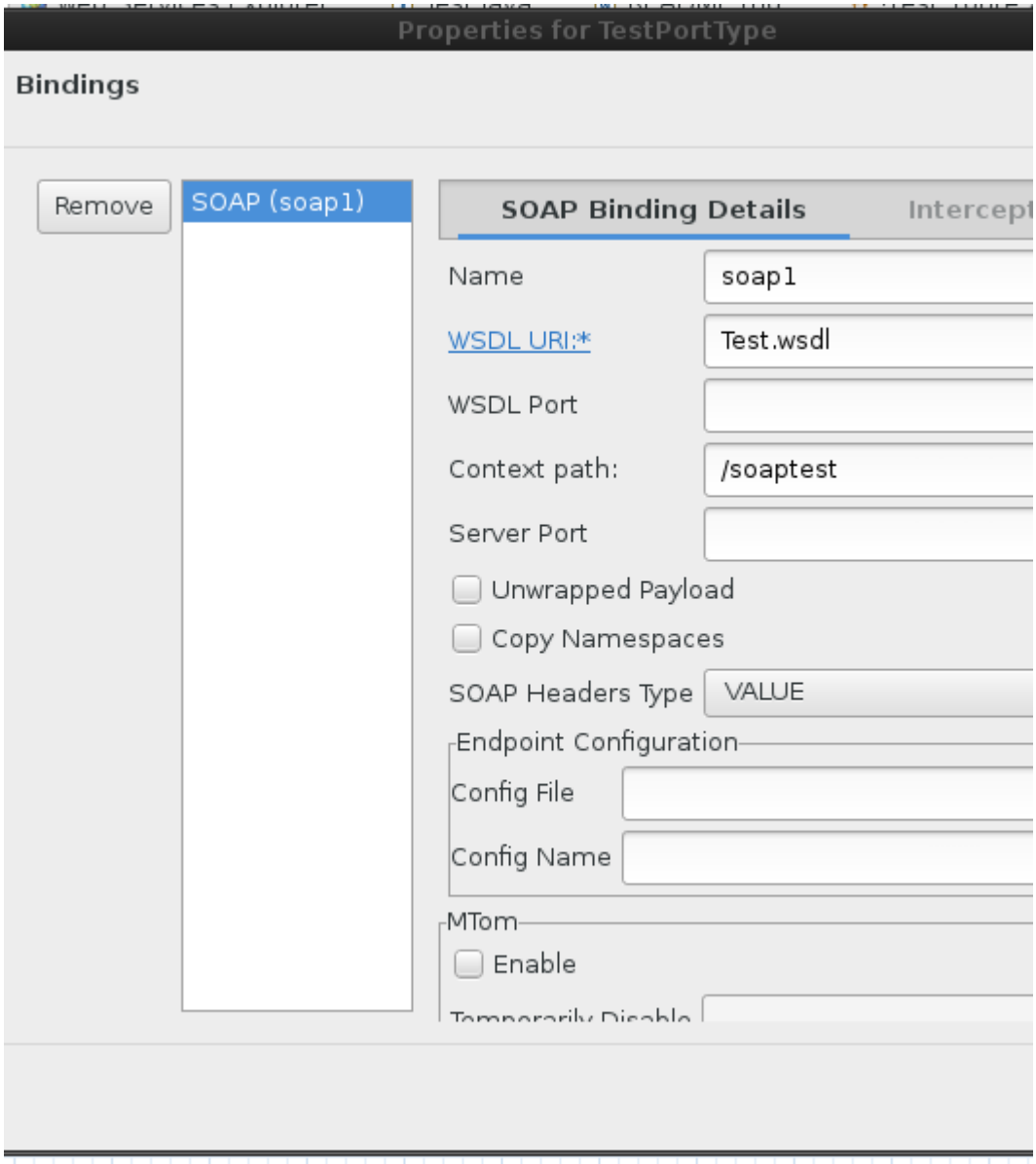
```
@XmlRootElement(name ="testIn")
public class TestIn {
```

The `TestPortType` input gateway

This gateway is obtained by “promoting” the interface (green arrow in JBDS) on `TestBean`. As part of the promotion we specify an interface, which is of type WSDL in this case, as this gateway will have a SOAP binding. It is the use of WSDL that has led to the name “TestPortType” -- this is actually the name of a `<wsdl:portType>` definition in the WSDL.

The JBDS tooling can create the WSDL file from the Java interface (`Test`) and its associated classes. There are various options that can be applied to this generation, not all of which work -- that is, not all options result in a WSDL file that is compatible with automatic JAXB transformation. In particular, selecting “wrapped” SOAP messages, which is the default, is problematic.

Note that the interface is distinct from the binding -- creating the WSDL interface does not, by iteself, create an HTTP endpoint in the application server. We need a SOAP binding for that.



The binding specifies the WSDL file, and the web context root for the application server -- `/soaptest` in this case. The full URL of the service will be made from the context root, and the name of the service corresponding to the `TestBean` component. The service name is `Test`, as can be seen from `switchyard.xml`:

```
<sca:component name="TestBean">
  <bean:implementation.bean
     class="com.example.switchyard.soaptest.TestBean"/>
  <sca:service name="Test">

So the full URL of this service will be

/soaptest/Test
```

The Test_route camel component

Switchyard allows for components to be implement as ordinary Camel routes, specified in Java or XML DSL. In this example we have the XML definition in the file `test_route.xml`. The route

doesnt't do anything useful: the complete definition is as follows:

```
<route id="_route1">
      <from id="_from1" uri="switchyard://TestRoute" />
      <log id="_log1" message="TestRoute - message received: ${body}" />
      <to uri="switchyard://TestSendMessage"/>
</route>
```

Note that the “from” and “to” endpoints are both Switchyard endpoints. The “to” endpoint refers to the `TestSendMessage` gateway (see below). The “from” endpoint is a reference to the service that the Camel route implements; that is, in effect, a reference to itself. The input to the Camel route will be a method call from the Java code in TestBean, but that isn't something that can be specified in the Camel route -- the reference to the route itself creates a free-floating consumer that will accept messages from any other component that holds a reference to this service.

Invoking the Camel route from the Java class TestBean

We use CDI annotations in the TestBean to create a reference to the Camel route. Note that the Camel route has a Java interface -- TestRoute, which specifies a single method `doMessage(String)`. The name of the method is arbitrary, because Camel accepts message bodies as input, not formal parameters. However, we need something for the Java code to call.

The CDI annotations are as follows:

```
@Inject
@Reference("TestRoute")
private TestRoute _testRoute;
```

This creates a reference to the service `TestRoute` (implemented by `test_route.xml`). Note that nothing in the application actually implements the interface `TestRoute` -- the implementation is a dynamically-generated proxy. When we call the interface's only method:

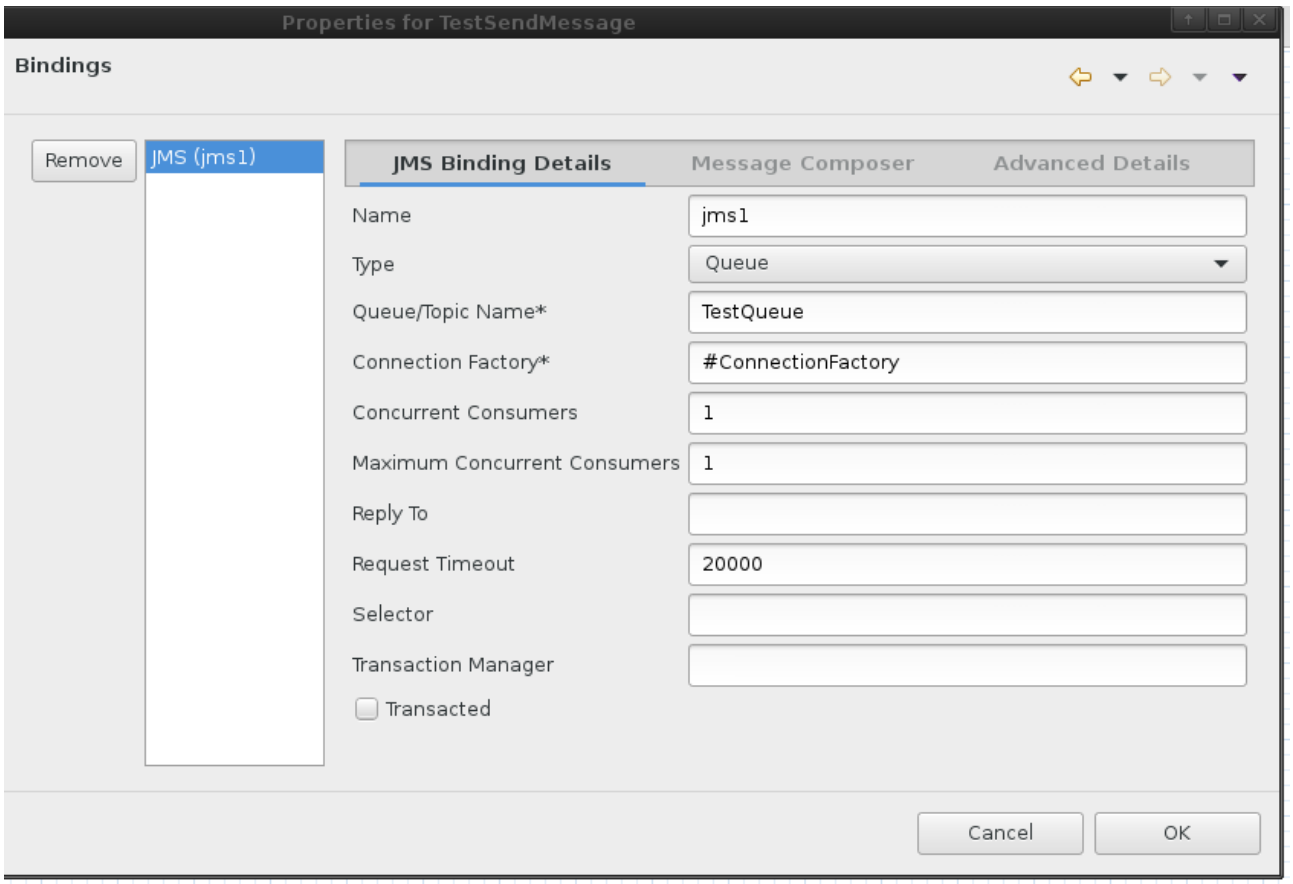
```
_testRoute.doMessage (...);
```

the argument is converted to a Camel exchange and passed into the route.

The `TestSendMessage` gateway

The output gateway `TestSendMessage` is actually a “promoted” reference. it is obtained in JBDS by selecting the reference (purple arrow) on `Test_route`, and promoting it. As part of the promotion we provide a new interface, which can be Java or something else. In this simple example, the interface is irrelevant, and JMS does not take arguments, and the provider of data will be a Camel route. However, Switchyard requires all service to have interfaces.

The gateway has a binding (connection) using JMS. The properties of this binding are shown in the following diagram:



The name of the queue and the connection factory are dependent on the platform, but Switchyard does some magic to try to find compatible references on the various supported platforms. On EAP, the name TestQueue refers to the `<entry>` element in a jms-queue definition in the configuration file (without the `java:/` prefix:

```
  <jms-queue name="AnythingAtAll">
      <entry name="java:/TestQueue"/>
      <durable>true</durable>
 </jms-queue>
```

The "#" sign at the beginning of "`#ConnectionFactory`" is common notation in Spring or Blueprint XML for an object in the Spring/Blueprint registry. On Karaf we would typically use a `<bean>` definition to provide an object based on the particular JMS client runtime. On EAP, this token refers to a connectionfactory definition in the configuration file:

```
<jms-connection-factories
  <connection-factory name="InVmConnectionFactory">
    <connectors>
      <connector-ref connector-name="in-vm"/>
    </connectors>
    <entries>
      <entry name="java:/ConnectionFactory"/>
     ...
```

(This connection factory name is provided by default in `standalone-full.xml`.) The effect of the JMS binding is that any invocation of the TestSendMessage gateway will result in the message body being posted to the specified JMS queue. Some data conversion might be required, but this is transparent in this simple example.