cid:image001.png@01CCB995.7A4B6990

Transactional Business Services

Developers Handbook

Version 3.0

1. Contact Information 3

2. Goals 3

3. Prerequisites 3

4. Software setup 5

5. Sample Projects 8

6. Development tasks 19

7. Maven Archetype for TBS Web Service 22

8. MessageInformation 25

9. Logging 29

10. TBS Versioning 33

Revision History 41

Developers Handbook

# Contact Information

Project Delivery Leader: Tim Kelley – timothy.kelley@wnco.com

Document Contact Information

Prepared By: Suneetha Veeravalli – suneetha.veeravalli@wnco.com

# Goals

The Transactional Business Services (TBS) project is the replacement for the existing Southwest Airline Java Service Oriented Architecture frameworks. The four main goals for the new service stack are:

1. Design and implement a software stack to create Southwest Airlines Transactional Business Services (TBS) based on Web Services technologies and standards to replace today’s CORBA and Mule based transactional business services
2. Optimize the solution to enable loose coupling between the services and their client applications
3. Integrate the solution with the Complex Event Processing stack through the TIBCO Event Bus
4. Accelerate Developer Velocity using Industry Standards Tools and Patterns

This handbook is targeted at the developer’s interaction with the TBS stack. It is envisioned that the bulk of the individual developer experience with TBS will be within the Eclipse IDE. Developers are able to compile, test, and deliver TBS services within Eclipse.

# Prerequisites

## Software and releases for your team

Most development teams use a certain development process and decide or standardize on a particular set of development software, for example, Eclipse, TortoiseSVN, SoapUI… Please check with your tech lead or build administrator for the list of supported tools and their releases. This document refers to the most recent release versions of the described tools that were available at the writing. However, it is not updated when a newer version of the tools is released (unless the process described here is no longer supported by these tools). The development teams within SWA decide what version of the tools they use.

Example development tools:

|  |  |  |
| --- | --- | --- |
| **Tool/application** | **Version used by your team** | **Version used in this document** |
| STS/Eclipse |  | 2.9.1 |
| Subclipse (Eclipse plugin for Subversion) |  | 1.8.9 (Subversion 1.7.4 client) |
| TortoiseSVN |  | 1.7.6 (Subversion 1.7.4 client) |
| soapUI |  | 4.5 |
| Java JDK |  | jdk1.6.0\_xx |

Note the list is not exclusive. Your team may use more tools, like Jenkins, Nexus, Sonar…

## Use matching version of Subversion clients

Most teams use more than one Subversion client application, for example:

1. Integrated with Eclipse (typically, Subclipse)
2. Standalone client – command-line or TortoiseSVN or other GUI tool

Each Subversion client application is compiled against the “subversion client code/library” that communicates with the server. It is somewhat confusing because the application version may differ from the client library version. But in all cases there is some indication about which Subversion client library version the application uses.

It is important to keep the client releases in sync with each other. For example, the binary format of the working copy (files on the local PC) changed between Subversion 1.6 and 1.7. **You should not mix client releases that work on the same working copy**. The details are at <http://tortoisesvn.tigris.org/compatibility.html> and <http://subversion.apache.org/docs/release-notes/1.7.html>. In summary, the rules are:

* Use the latest version of the client available at the time
* It’s OK to use newer version of the client against an older server release
* Make sure the Subversion client applications use the same/compatible subversion client library if you plan to use them on the same working copy of the files.

## Subversion location for your team

Source code repository (most typically, Subversion) is specific to the development team. **You will be unable to complete the setup described below if you do not know the specific Subversion location/URL for your team.** You need to create a Remedy Service Request for “Config/Build Mgmt-Client Srvr” team to set up a Subversion repository if your team does not have one. Note the examples and screenshots in this document may refer to an example repository used by the TBS team internally. You will not be able to use that Subversion repository (only TBS developers have access to that repository).

## settings.xml for Maven

Maven (a build tool) needs to be configured to point to a location where TBS artifacts (e.g. release component JAR files) can be located. As with a Subversion repository (for source code), the **Maven** **repository URL may be specific to each team**. Please get the a copy of settings.xml from your tech lead or build administrator.

This document shows a minimal settings.xml file (on page 4) to build and run the example projects. However, you should use it only as a last-resort default.

## Other resources and repositories for your team

If your team has implemented any Continuous Integration or automated process you need to get the locations and resources from your tech lead. Many teams maintain some wiki/Confluence page to list the resources. Use them as an addendum to this document to enter the proper location and follow the process for your team.

# Software setup

TBS utilizes the SpringSource Tool Suite (STS) which provides various tools that will be helpful for creating TBS services. This includes but is not limited to support for tcServer (the Spring version of Tomcat which is used by TBS), bean and XML wizards, graphical Spring configuration editors, Spring 3.0 support.

SpringSourceTool Suite includes the Eclipse IDE distribution for building enterprise applications.



## Install SpringSourceTool Suite

* Download SpringSourceTool Suite from: <http://www.springsource.com/downloads/sts>
* SpringSourceTool Suite installation guide is available in PDF from the download page above.
* Download installer, for example springsource-tool-suite-2.9.1.RELEASE-e3.7.2-win32-installer.exe
  + Windows XP/2003 desktop should use 32-bit version
  + To determine if you have 32-bit or 64-bit version check <http://www.funduc.com/comp_ver_info.htm>

### Notes for installing the IDE:

* In Step 4 of 8 of the install dialog (Select Installation Package) make sure to check all options (Spring Source Tool Suite, Maven, and tcServer)
* Spring Source Tool Suite is rather large set of tools. Sometimes during the installation and configuration process Eclipse displays errors. In general the errors go away after all tools, plugins, and configuration settings are completed.

### Keep your STS updated

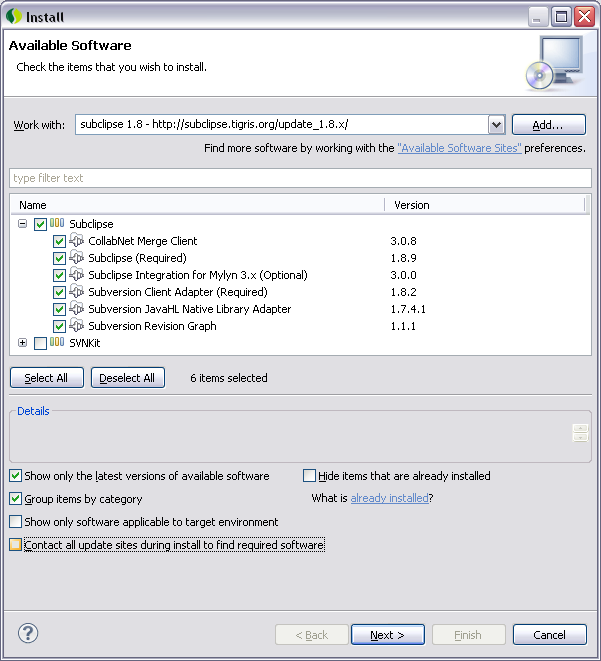
STS periodically checks for updates. In general, it is a good idea to let it update all features and plugins to the most recent available. Check with your tech or dev lead on your team’s update policy. Note that even if your STS updates to a newer version, the initial installation directory does not change. So in a few months you may end up with STS 3.0.2 that still resides in C:\Program Files\springsource-2.9.1\sts-2.9.1.RELEASE\ directory. Use Help -> About and Help -> Check for Updates to manually check for the latest version.

## Install Subclipse plugin for Subversion

Ask your tech lead or admin which version to get. TBS recommends the latest released verion.

Go to <http://subclipse.tigris.org>

* Click “Download and Install” link on the page
* Follow the Installation Instructions on the page
* Enter the desired update site (for example, <http://subclipse.tigris.org/update_1.8.x>)



## Install standalone Subversion client

Some developers prefer to use a standalone Subversion client, for example, TortoiseSVN. Download and install the version that your team uses. Make sure to use version of standalone client that matches your Subclipse version – see note on page 2.

## Install soapUI

soapUI is a popular tool for testing web services. In the context of this document it is used only for ad-hoc testing, sending a SOAP request to sample services.

## Maven setup

### Add Maven binaries to your computer’s PATH

If you want to use Maven from the command line you need to update your PC’s PATH variable. The SpringSourceTool Suite installer comes with Maven (currently version 3.0.3) bundled inside. After installing STS, the Maven installation directory can be located within the SpringSourceTool Suite installation directory. Add the bin directory of the Maven installation (for example, C:\Program Files\springsource\apache-maven-3.0.3\bin) into the PATH environment variable so that the 'mvn' command can be run from the command line. Run mvn –version from a command line to verify Maven is correctly configured.

### Configure Maven for TBS and Enterprise Repositories

Before you can build any sample projects (within or outside of Eclipse) you need to configure Maven repository locations. Your Maven instance needs to know where to get SWA-specific code. Typically, **your tech lead or admin** will provide **settings.xml** to use. Copy the file to your default Maven configuration directory, typically C:\Documents and Settings\<your user directory>\.m2

**If your team does not have a preferred/shared settings.xml** you can paste the following content into settings.xml. Note: this configuration is only good to compile and run examples; it will not work with the projects your team/group develops.

<!-- Minimal configuration that is only good to compile tbs examples. Get the real configuration from your tech lead. -->

<settings>

  <mirrors>

    <mirror>

      <!-- Forces all repositories to be rerouted to Nexus. -->

      <id>nexus</id>

      <mirrorOf>\*</mirrorOf>

      <url>http://repositories.swacorp.com:9092/nexus/content/groups/public</url>

    </mirror>

  </mirrors>

</settings>

When you change settings.xml content you should remove your local repository (typically, under C:\Documents and Settings\<your user directory>\.m2\**repository** ) and rebuild your projects (run mvn clean). Also, you need to restart Eclipse to refresh the dependencies.

### Streamline Maven

Make sure there are no other Maven configuration settings that may conflict. For example, M2 and M2\_HOME environment variables do not need to be defined or at least they should not conflict with the above setup.

Verify your Maven settings inside Eclipse by going to Window -> Preferences -> Maven -> Installations (use Embedded) and Window -> Preferences -> Maven -> User Settings (should point to your location described above).

# Sample Projects

Two sample services are provided to demonstrate how a complete TBS service can be built. The samples are very similar to a new service generated from the archetype, but several important components have been added. This document describes each sample and points out a few of the most important features.

## Overview of sample projects

### Functionality and dependencies

* customer-management – a simple TBS service with no external dependencies (calls no other service or database). It retrieves customer data (including account number, for example “11111111”) for a customer, for example “Joe Doe”)
* customer-flight-availability – a TBS service that calls (integrates) three other services:
  + a TBS service (a deployed version of customer-management TBS service described above)
  + a framework 2 service – a made-up service called “loyalty” that retrieves loyalty status (for example, “gold” for a customer account “11111111”)
  + a framework 1 service - a modified version of Firebird’s FlightAvailability that retrieves flights given the date and loyalty status

### Demonstrated features

The projects share the following features with the generated archetype:

* contract project – the WSDL and XSD files are in a separate contract project that is a dependency of the implementation project
* file and subproject structure – each project has two subprojects:
  + service – domain, external façade, and downstream dependency files
  + it – integration testing project – a test project that accesses a locally deployed service (running in a local tcServer) via a web service infrastructure
* TBS “stack” - a set of tools and libraries used to build and test the service, such as CXF, JAXB, Spring, Maven, JUnit…

### Differences

The differences between samples and/or the archetype are:

* No downstream dependency: customer-management does not depend on any other service. That makes it easier to follow (it is close to “Hello, World” example). However, customer-flight-availability is a better example of real-world dependency on legacy services.
* Downstream dependency mapping: TBS advocates that downstream dependencies should be mapped. For example, customer-flight-availability service depends on customer-management. But instead of using JAXB-generated classes from customer-management’s contract directly, it builds CustomerManagementAdapter that acts as a façade/isolation layer between customer-flight-availability domain and customer-management downstream dependency.
* Server side mapping: customer-management service uses a similar façade/isolation layer between its domain and its external interface. For example, CustomerManagementEndpoint acts as an adapter to CustomerManagementImpl. In contrast, customer-flight-availability and the archetype use the JAXB generated classes in its implementation. TBS does not mandate server-side mapping – it is up to service developer (and architect) to decide if it is needed.

## How to install the samples

Download and expand the projects from the following two ZIP files (both are required to run the samples):

* contracts: <https://repositories.swacorp.com:8443/nexus/content/groups/public/com/swacorp/tbs/contracts/tbs-samples-contracts/3.0/tbs-samples-contracts-3.0.zip>
* implementation: <https://repositories.swacorp.com:8443/nexus/content/groups/public/com/swacorp/tbs/tbs-samples/3.0.0/tbs-samples-3.0.0.zip>

The contract projects contain schema (WSDL and XSDs) for the samples; the implementation projects contain code to implement the sample. Expand both ZIP files to some temporary location (where they’ll be imported to Eclipse from).

## Customer-management

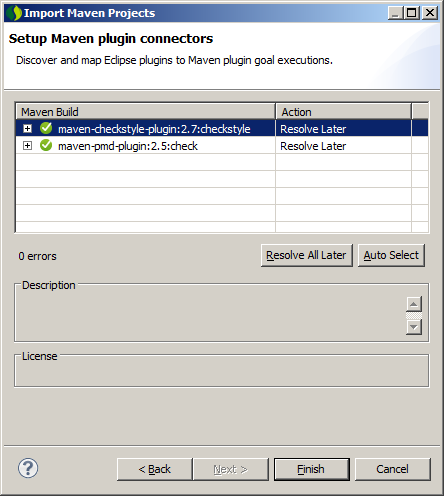
The project has no downstream dependencies; it is a good first sample to study.

Import the project into Eclipse with the following steps:

* Go to File -> Import… -> Maven -> Existing Maven Projects
* Import the customer-management **contract** project (a single pom), then the customer-management **implementation** projects (with all its sub-projects):

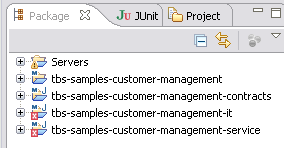
|  |  |
| --- | --- |
|  |  |

* If prompted to install any Maven Plugin Connectors, click “Auto Select” to allow Eclipse to choose the best action, and then click finish. Click through any additional dialogs and restart Eclipse when asked (when new m2e connectors were installed).

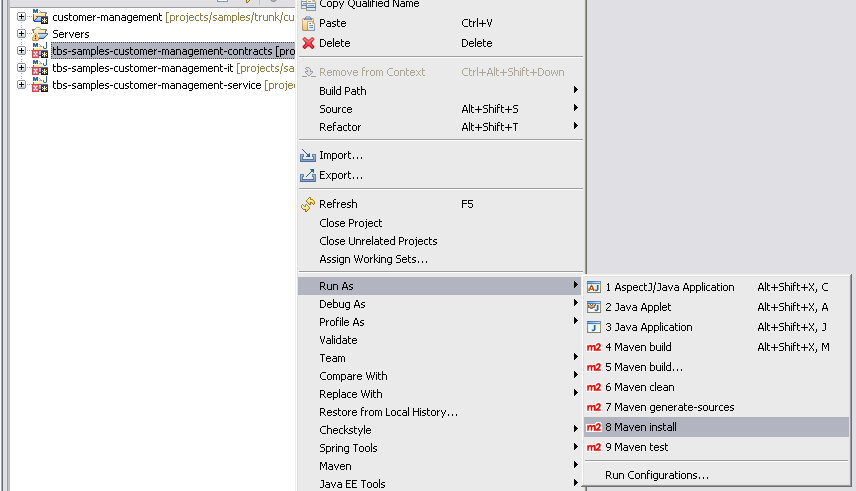


Some users experienced errors after Eclipse restart. Ignore them; they seem to be harmless and do not come back.

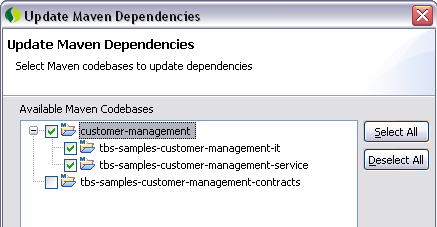
* After importing you’ll end up with four top-level projects in your Eclipse (one for each pom.xml). Ignore errors after the initial import; focus on the project content:



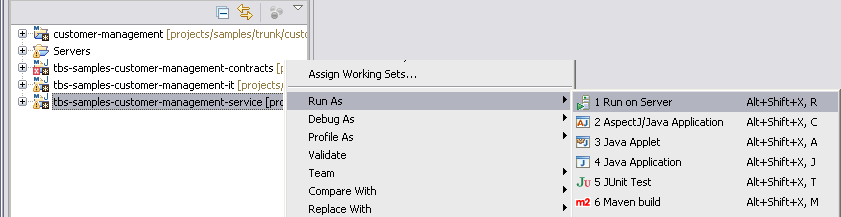
* + tbs-samples-customer-management top-level project – **ignore it**. None of the Maven functionality can be applied to it.
  + tbs-samples-customer-management-**contracts**: you’ll work with contract project only when WSDL or XSD changes (rather rare).
  + tbs-samples-customer-management-**service**: you’ll spend most of time in the implementation project working on service implementation and unit tests.
  + tbs-samples-customer-management-**it**: integration test project (a logical equivalent of “NG tests”). It is used to write and run integration tests that access the deployed service through a web service layer.
* The contract needs to be installed in your local Maven repository. Do it by selecting the contract project and choosing Run As -> Maven install. Make sure the console displays BUILD SUCCESS upon completion. This is a functional equivalent of running mvn install command from the command line.



* Both tbs-samples-customer-management-service and tbs-samples-customer-management-it projects use CXF Maven Plugin for code generation, but this plugin does not work inside Eclipse. In order to correct hundreds of “cannot be resolved to a type” errors, right click on the project (tbs-samples-customer-management-service and tbs-samples-customer-management-it) and choose Run As -> Maven generate- sources. This is a functional equivalent of running mvn generate-sources command on the command line from customer-management/service and customer-management/it directory.
* After the source code generation completes, right click on the project again and choose Maven -> Update Project Configuration. The project will be selected by default in the pop up window. Right click on the parent project (customer-management) and choose Select Tree from the pop-up menu, then click OK to update all subprojects:



* In order to test the service it must first be deployed to a server. During development the preferred way to run a service is to use the Eclipse “Run on Server” action:



The initial server startup may take 1-2 minutes. The service will start on port 8080 with WSDL available at http://localhost:8080/tbs-samples-customer-management-service/customerManagement?wsdl. If the service starts correctly the address above should load a WSDL file in your browser.

* After the server has started, the integration tests can be run by right clicking on the tbs-samples-customer-management-it project and clicking Run As -> JUnit Test.
* Once the service is started it can be tested by using soapUI.
* launch soapUI
* select File -> New soapUI Project
  + enter a project name, for example customer-management
  + WSDL address from above, for example http://localhost:8080/tbs-samples-customer-management-service/customerManagement?wsdl
  + ensure that “Create sample requests for all operations?” is checked

SoapUI will download the WSDL and create sample requests for all operations found in the WSDL. Navigate to retrieveCustomerByName and its child, “Request 1” in the Projects tree and double click on it. The request will be only partially complete (see next screenshot) and will need to be updated before the operation can be run.

Copy in the content (or fill in highlighted data values) and then run the operation by pressing the green arrow to the upper left of the request entry text area:

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:v1="http://tbs.swacorp.com/messageinformation/v1" xmlns:v3="http://tbs.swacorp.com/samples/customerManagement/v3">

<soapenv:Header>

<v1:messageInformation>

<v1:commonInformation>

<v1:callingApplication>myTestApp</v1:callingApplication>

<v1:transactionId>myTxId</v1:transactionId>

<v1:contractVersion>

<v1:major>3</v1:major>

<v1:minor>0</v1:minor>

</v1:contractVersion>

</v1:commonInformation>

</v1:messageInformation>

</soapenv:Header>

<soapenv:Body>

<v3:retrieveCustomerByNameRequest>

<v3:customerName>

<v3:firstName>John</v3:firstName>

<v3:lastName>Doe</v3:lastName>

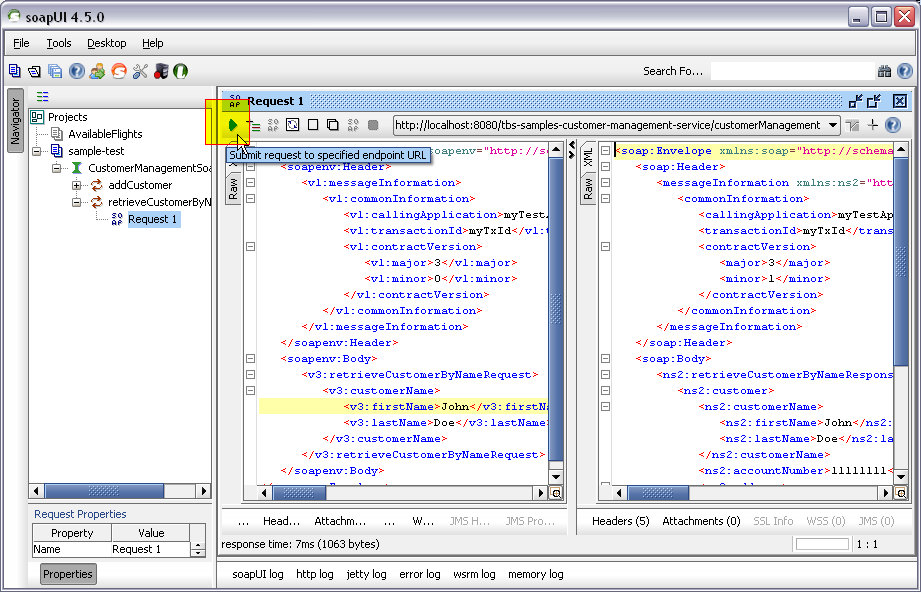
</v3:customerName>

</v3:retrieveCustomerByNameRequest>

</soapenv:Body>

</soapenv:Envelope>

The result should look similar to the window shown below. In particular the response should contain a <accountNumber> element with the content "11111111".



If you encounter an error try to “massage” the request to fill out only highlighted data into empty request that soapUI generated from the WSDL.

The sample is distributed with hardwired data in customer-management/service/src/main/resources/com/swacorp/tbs/customermanagement/domain/customer1.xml. You can change the values or create a similar file, customer2.xml to add your test data. Note that service needs to be restarted to use the new seed values.

Now that you have run the service locally you should step through all the project modules in order to understand how the service is built.

## Contract project

The project contains the WSDL and XSD for the service. These files specify the request and response formats that were seen in the earlier example. Review the WSDL and XSD before continuing so that you understand how they map to the XML request and response objects we saw earlier.

The implementation project depends on this project. In real development, the contract project is released before the implementation projects so that the client applications can use the contract to program their implementation and express it as their dependency (thus they would not need to install or run the contract project locally). Note the contract project uses only 2-digit major.minor versions (since there is no implementation/build version).

## Implementation project

### service

The service module defines the business logic for the service. While the contract project defines the “external” interface (also called the service contract) using the WSDL and XSD, the service module implements the business logic.

customer-management service uses server side mapping – a façade/isolation layer between its domain and its external interface. For example, CustomerManagementEndpoint acts as an adapter to CustomerManagementImpl. In contrast, customer-flight-availability and the archetype use the JAXB generated classes in its implementation. TBS does not mandate server-side mapping – it is up to service developer (and architect) to decide if it is needed.

The service module builds a .war file that includes all wiring to enable a SOAP HTTP endpoint. The service/src/main/webapp/WEB-INF/web.xml file starts CXF and tells it to load its configuration from the serviceContext.xml file in the same directory.

The service/src/main/environment directory stores resources only (no Java classes) that are deployment environment specific. The resources directory is structured to allow the placement of resources needed in specific environments, as well as providing a common area for resources used across all environments. Customer-management contains no environment-specific values; see the discussion of customer-flight-availability section below for explanation of how the resources work.

service/src/test contains unit tests. They test domain classes, adapters/transformers, etc as a unit (not as an integrated whole). These tests should not access the web service infrastructure– only direct Java calls. They may use mocks, stubs, or other unit-testing techniques to remove dependency on the downstream dependencies.

### **it**

The “**it”** (integration testing) module is used to perform end-to-end integration testing. In order to perform a true end-to-end test, the *it* module implements a complete SOAP web service client starting from only the WSDL generated source code from the contract. The client code (used to access the service) and configuration is maintained entirely in the *it* project.

When Maven is used to build the *it* module (when run from the command line or from any of the Eclipse “Run as -> Maven install” menu options) it automatically starts a **separate** Tomcat server (different from the embedded tcServer in Eclipse) on a random port and deploys the *service* module. The tests then communicate with the server over SOAP exactly as a real client would, and when the tests are finished the server is automatically stopped.

The tests in the “*it”* module can also run inside Eclipse, but the *service* module must first be deployed to a Tomcat server on port 8080. As shown earlier, the Eclipse “Run on Server” action can be used to automatically start a server with the appropriate configuration.



## customer-flight-availability

This sample demonstrates a TBS service talking to a framework 1.x CORBA service, a framework 2.x service, and another TBS service. The discussion here focuses on the differences between the customer-flight-availability and the customer-management service.

### Downstream dependencies

All external dependencies for the sample service run on xldpod02:



* customer-management service was deployed so that the CustomerFlightAvailabilityIT (and customer-flight-availability implementation) can run standalone – without a need to run both samples inside Eclipse. The configuration (location of the service) is hard-wired in customer-flight-availability/service/src/main/environment/filter.properties. So even **if you make changes locally to the customer-management service sample, it will not be reflected in the already deployed copy on xldpod02**. Functionality-wise, it retrieves customer data, including account number (for example, 11111111) for a customer (for example, “John Doe”).
* CustomerLoyalty is a sample framework 2.5 project exclusively developed and deployed to be a downstream dependency of this sample. It runs on xldpod02:54927; the configuration is at customer-flight-availability/service/src/main/environment/common/customerloyalty-fw25-config.xml. The service retrieves loyalty status (for example, GOLD) for an account number (for example, 11111111). If you need to see all seed data, the source code for the service implementation is at http://repositories.swacorp.com:9092/nexus/content/groups/public/com/swacorp/tbs/fw25/CustomerLoyaltyImpl/1.0.0/CustomerLoyaltyImpl-1.0.0-sources.jar
* FlightAvailability is a stolen/tweaked version of CEBS’ Firebird service. The service was tweaked by:
  + Deployed as a locally run instance with isolation prefix tbs-fw1-flight-availability-1.0.0
  + Frozen in version (36.3); not meant to be updated for the current trunk version
  + Gutted out implementation to not make SAAS calls. It returns 14 hardwired daily flights for any origin/destination combination. It also ignores the otherwise required VPAR, connection pool… parameters from STI.
  + The dependencies in customer-flight-availability/service/pom.xml contain an important warning:

<!-- Note these are FAKE ARTIFACTS\*. You need to to talk to EBS build team (Tim, Johnny, Chris, and gang) to get group, artifact, and version of the real Firebird files.

Note2: the artifacts are real in enterprise Nexis. However, 36.1.290 is the only version in

there. Again, if you need current trunk Firebird version send email to "CEBS Platform Build-DG"

-->

The downstream dependencies are mapped. For example, customer-flight-availability service depends on customer-management. But instead of using JAXB-generated classes from customer-management’s contract directly, it builds CustomerManagementAdapter that acts as a façade/isolation layer between customer-flight-availability domain and customer-management downstream dependency. Browse the code in com.swacorp.tbs.customerflightavailability.**loyalty**, com.swacorp.tbs.customerflightavailability.**flightavailability**, and com.swacorp.tbs.customerflightavailability.**customermanagement** packages to see how are the adapters and mappers implemented (they vary in how they use adapters and transformers).

The sample uses hand-written mapping code. The developers may choose other mapping tools (for example, Dozer) to map objects and fields.

### Stubs used in unit tests

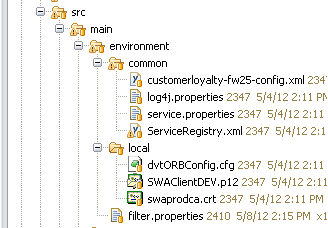
Using mapping for downstream dependencies makes it easier to replace real adapters with test doubles (<http://en.wikipedia.org/wiki/Test_double>). See classes like CustomerManagementStub, FlightAvailabilityStub, and CustomerLoyaltyStub for an example of stubbing out real downstream dependencies.

### No server-side mapping

Unlike customer-management, the customer-flight-availability uses no server side mapping. The JAXB generated classes are used in the service implementation and domain. For example, you can find classes from the com.swacorp.tbs.samples.customerflightavailability.v3 package used in the domain code. TBS does not mandate server-side mapping – it is up to service developer (and architect) to decide if it should be used.

### Environment-specific resources

The image below shows an example of environment-specific resources.



The required resources (CORBA configuration files in this example) are placed in the relevant folders based on the needs of the service. The filter.properties file, also located in the environment folder, provides the ability to set resource variables that change based on the hosting environment - local, dev, sat, and prod.

The combinations of the environment folders along with the filter.properties file allow the Maven build process to create the required resource modules ready for deployment. At deployment time, the correct resource module will get packaged with the service based on the targeted environment - see the TBS Build Handbook for more information regarding deploying a TBS service.

Because customer-flight-availability uses the CORBA FlightAvailability service, most of the files are used to configure CORBA.

The TBS Maven environment plugin is used to filter the resources into environment specific configuration .jar files. It is configured in the pom.xml:

<configuration>

<filters>

<filter>src/main/environment/filter.properties</filter>

</filters>

<resources>

<resource>

<name>common</name>

<directory>src/main/environment/common</directory>

<filtering>true</filtering>

</resource>

<resource>

<name>local</name>

<directory>src/main/environment/local</directory>

<filtering>true</filtering>

</resource>

</resources>

<environments>

<environment>

<name>local</name>

<resources>

<resource>common</resource>

<resource>local</resource>

</resources>

<patterns>

<pattern>\*.local</pattern>

<pattern>\*</pattern>

</patterns>

</environment>

<environment>

<name>dev</name>

<resources>

<resource>common</resource>

</resources>

<patterns>

<pattern>\*.dev</pattern>

<pattern>\*</pattern>

</patterns>

</environment>

</environments>

</configuration>

The configuration has three important sections. The <filters/> section lists files that contain property values that will be substituted into resource files in order to make them environment specific. The <resources/> section lists directories that contain resource files. In order to apply filtering the <filtering/> element must be set to true. Finally, the <environments/> section defines how the resources and filters are combined to build a set of environment-specific resources.

In this case two environments are defined: local and dev. When the build is run the target/environments/local and target/environment/dev directories will be populated with environment specific resources and two Maven classified artifacts (${artifactId}-${version}-**local**.jar and ${artifactId}-${version}-**dev**.jar) will be created.

The local environment uses the resources from both the common and local resource directories. As files are copied from the resource directories to target/environments/local any ant style variables (for example ${client.orb}) are found and replaced. Variable replacement is a multistep process. First, the <patterns/> defined for the <environment/> are used to find candidate property names. In this case the patterns are \*.local and \*, so the ${client.orb} variable can be replaced with either the value of the client.orb.local property or the client.orb property. The next step is to find the appropriate property in one of the configured filter files. The src/main/environment/filter.properties contains a definition for client.orb.local, so the value of that property is used in place of the variable reference. The first pattern match found wins. The \*.local pattern was listed above the \* pattern, so client.orb.local is used in preference to client.orb.

The common environment uses only the common resource directory, so after processing it will only contain a single file, ServiceRegistry.xml. Those familiar with CORBA service configuration will immediately notice that the local environment is properly configured for testing on Windows, while the dev environment is configured for testing on a Linux machine with an existing Orbix installation. Production services may need more advanced property substitution, but the sample provides a good starting point for further customization.

# Development tasks

## Import and Run Projects in SpringSourceTool Suite

### Pre-requisites

* A TBS Service project already exists. The project has already been stored on a file system from either
* Version Control System (VCS)
* TBS webservice archetype
* Downloaded Sample Projects
* IDE (SpringSourceTool Suite) already installed and configured
* Maven already configured
* Subversion is a preferred Source Control System and Subclipse is the recommended Eclipse plugin
* knowledge on how modules are laid out for a TBS Service



**IDE Workflow**

Most project code resides in a Subversion repository. However, Eclipse works the same with local files or checked out version of files from Subversion. After checking out a copy of a project from Subversion you can point Eclipse to the local directory just as if it were local files (Eclipse will recognize .svn files and will offer Subversion operations on them). From then on the operations are the same on simple files or a Subversion checkout copy.

## Code Quality

TBS service project relies on static code analysis tools – PMD, FindBug and Checkstyle. Eclipse plugins for these tools should have already been installed as part of the IDE setup. Any code that fails the rules defined in these tools will show up as an error or a warning. It is expected that the developer fixes any such errors/warnings in the code since the same rules are applied in the CI (continuous integration) build and such errors/warnings left alone will also fail the CI build.

## Building within an IDE

The IDE is already equipped with a Maven plugin. In most cases, Eclipse automatically builds and copies the compiled code and resources into appropriate target folders. Therefore, there is no need to run Maven builds within an IDE. However, in some cases (mostly when files are referred to from a file system directly) a user must run a Maven command (especially, mvn install) to make the module available from the local repository. (mvn install for a module can be run within an IDE by right clicking on the project and then selecting the “Run as” option from the drop down and then selecting the “Maven install” option.)

For a Maven imported project, all the project related dependencies show up in a “Maven Dependencies” folder including the jars defined in the pom.xml (Maven build file). The m2e plugin (Maven plugin for Eclipse) resolves dependencies of open projects and uses already opened projects as dependencies instead of the jars installed in a local repository.

## Testing a Service within an IDE

### Unit tests

* Functional test cases reside in service/src/test/java folder. Test related resources reside in src/test/resources folder. For a stable service, proper test cases must exist and they all should run green all the time.
* Test cases contain mock/stub classes for all touch points to downstream dependencies (external services). The mock/stub classes should be part of the src/test/java folder so that they don't get bundled into the service at deployment time. Run the JUnit tests by right clicking on the Test case class and selecting Run as -> JUnit Tests. The test cases must run green.

### Integration testing

* End to end Integration testing is done in “it” module (A client and a server on two separate JVMs). A service must be deployed and running on a tcServer (when running within an IDE) for integration test cases to run successfully. Unless modified, as can be seen in the “Servers” view pane of an IDE, the IDE already has a tcServer instance configured with http port 8080. A service (an integration service project in this case) can be deployed to the tcServer instance by right clicking on the tcServer instance and selecting the “add and remove…” menu option from the pop-up menu. Deploying a service does not start the tcServer instance. The Servers view has the buttons to start a tcServer instance.
* Please note that if you make code/configuration changes to a service (and service dependent) projects/modules, the changes are hot deployed to the running service. At this time, it is not recommended to do hot deployments since we have experienced that the hot deployment causes some memory leak and after a few iterations of hot deployment, the server runs out of memory and crashes.
* Once the service is running, the JUnit test cases in the integration test project should run green.
* The test cases can also be run as mvn test from within IDE if all the modules are already installed in a local repository. When running mvn test within the integration test module of the service, the build process runs a **different** tcServer instance using the Maven Cargo plugin and then runs the test cases against the tcServer instance. Upon completion of running the test cases, the tcServer is shut down. In rare cases, if tcServer was not brought down (for example, if the test cases were abandoned in the middle of running), you will have to manually kill the process for the tcServer forked by the Cargo plugin.

## Working with Source Control System

Subversion (SVN) is the preferred source control system for TBS Services. If you’ve not already done so, please install and configure the Subclipse plugin for the IDE.

If an imported TBS Service is already maintained through Subversion, each of the projects within an IDE will have the location of the service within the repository right next to the project name and each of the versioned files within projects will show a revision number followed by a date/time stamp of the last change in the repository. The versioned files are your own writable copies; there is no need to check out files as you do in a ClearCase dynamic view. A project may have a new file that is versioned, a versioned file that was modified, and an unmodified versioned file. Subclipse modifies the file icons (in front of the file) to indicate such differences in the state of a file.

For an Subversion maintained shared project, a “Team Submenu” (when right clicking on any of the Subversion file/directory) shows all the commands/options available for a selected file/directory (and subdirectories recursively). One must execute the “update” option to sync up with the repository version. The new and modified files can be checked in using the “commit” option. Any conflicts between local files and repository files can be best resolved by switching to the “Team Synchronizing” perspective. Refer to Eclipse tutorials and best practices document for more on working with Subversion.

## Bug fixing a TBS service - a sample scenario

The following steps define the activities from a developer perspective:

* Check out a TBS Service project from Subversion. As a result, a directory (for example, MyTBSService-trunk) is created with the contents from a project.
* Import the project into the IDE. As a result, all the modules of a service will show up as Eclipse projects. Use Run As -> Maven generate-sources on service and it modules. If a service is correctly imported, none of the projects should show problems/errors.
* Create test case(s) in an appropriate module/project that help re-create the behavior the bug exposes. The test case fails as the bug still exists.
* Make the code/configuration changes that fix the problem following best practices and coding guidelines. If there are identifiable code quality problems in the changed code, PMD/FindBug/Checkstyle should show them as errors/warnings in the IDE. Fix errors/warnings.
* Run all the test cases. Make sure that they all pass (run green). If a test case requires integration testing, deploy and run the service and then run the IT test cases.
* Once you are satisfied with the changes, run mvn clean install on the top level module. This will ensure that the service can run consistently not just in an IDE but on the command line as well.
* As part of a bug fix, if you are required to change a build process, make sure that the process does not break on Linux or in other environments (all possible Maven profiles/settings are covered). Follow Maven best practices on defining dependencies. You may run the mvn dependency:analyze goal and resolve some of the warnings (e.g., remove a dependency not used in a module).
* Update the changes from the Subversion repository before checking in. If there are conflicts, merge the files to resolve them and then check the files in.
* Once checked in, the CI (Continuous Integration) process for the service should trigger the build in due time. If the CI build fails, verify that your changes have not broken the build.

## Note on changing the contract

There are extra steps needed when you change the contract (.wsdl or .xsd file) in the contract project:

* Use Run As -> “Maven install” on the contract project to put the updated contract into your local Maven repository (that resides on your PC).
* Delete the codegen marker to trigger CXF code generation from the contract files in “-service” and “-it” project: delete <your-service>**-service**/target/cxf-codegen-plugin-markers and <your-service>**-it**/target/cxf-codegen-plugin-markers directories.
  + Alternatively, you could run Run As -> Maven clean. However, that seems to confuse Eclipse since it deletes generated sources and compiled classes from underneath it. It may require running Refresh (F5) on both projects and Maven -> Update Project Configuration… to clear all (what Eclipse perceives as) errors.
* Run As -> Maven generate-sources on both “-service” and “-it” projects.

# Maven Archetype for TBS Web Service

A Maven archetype for development of TBS Web Services has been created to assist in setting up a template project hierarchy based on a few parameters. See the sample project for a description of the structure and organization of the generated template project hierarchy. A major benefit of the generated archetype is the set of working pom.xml files for managing the project hierarchy with Maven. Skeleton implementations of important Java and configuration files are also generated. These generated files provide a useful starting point for service development, since there are many text fragments that are repeated in multiple files that are best kept in sync for the sake of clarity, consistency, and correctness.

## Maven Command

Before running the command obtain the most current archetypeVersion that your team uses. To find the most recent released archetype version browse <https://repositories.swacorp.com:8443/nexus/content/groups/public/com/swacorp/tbs/tbs-webservice-archetype/>

There are no prerequisites to running the command below, other than Maven binaries and Maven configuration (in settings.xml) that contain the enterprise Nexus repository (Maven will fetch all binaries based on supplied archetypeGroupId, archetypeArtifactId, and archetypeVersion).

Here is an example Maven command that generates a project template from the archetype (the single command line is broken up with backslashes for ease of reading). The command line parameters should be customized appropriately to fit the requirements of the new service. Namely, the values in gray should be supplied for your service. The **bold** parameters are required.

mvn archetype:generate

-D**archetypeGroupId**=com.swacorp.tbs \

-D**archetypeArtifactId**=tbs-webservice-archetype \

-D**archetypeVersion**=3.0.0 \

-D**groupId**=com.swacorp.finance \

-D**artifactId**=finance-reports \

-DinterfaceName=ReportGenerator

-DmethodName=generateReports

The project files will be generated in a subdirectory of the current directory (in this example, in finance-reports directory). Each “-D” argument defines a property that affects the archetype generation. The individual properties are described below. **Bold** properties are required.

|  |  |  |
| --- | --- | --- |
| Property Name | Description | Example Value |
| **archetypeGroupId** | Used to identify this archetype. Values are *static* (same for everyone) | **com.swacorp.tbs** |
| **archetypeArtifactId** | **tbs-webservice-archetype** |
| **archetypeVersion** | Archetype version. Use most recent available at <https://repositories.swacorp.com:8443/nexus/content/groups/public/com/swacorp/tbs/tbs-webservice-archetype/> | **3.0.0** |
| **groupId** | Group identifier of the service generated by the archetype. Usually in Java package name format. | com.swacorp.finance |
| **artifactId** | Service name. Use lowercase letters and hypens | finance-reports |
| interfaceName | The name of the service generated by the archetype. Must be a valid Java class name. Usually similar to the artifactId. Defaults to HelloService | ReportGenerator |
| methodName | The name of a single operation to add to the service. Must be a valid Java method name. Defaults to sayHello | generateReports |
| version | Initial service version. Defaults to 1.0.0-SNAPSHOT | 1.0.0-SNAPSHOT |
| contractGroupId | Name of group identifier for the contract. Defaults to <groupId>.contract | com.swacorp.finance .contract |
| contractArtifactId | Name of contract project for the service. Defaults to <artifactId>-contracts | finance-reports-contracts |
| contractVersion | Version of the contract project. Defaults to 1.0-SNAPSHOT | 1.0-SNAPSHOT |
| package | The root package name for all Java code. Defaults to <groupId> | com.swacorp.finance |
| interactiveMode | if true: confirm every default parameter suggestion  if false: use defaults (no chance to override) | true |

## Key Files

Some of the key files generated by the archetype are described below using the files generated by the above Maven command as an example.

### Project Management Files

These files help manage the project and include information regarding project hierarchy, dependencies, common names, versions, and build tools:

finance-reports

| pom.xml

+---finance-reports-contracts

| pom.xml

\---finance-reports

| pom.xml

+---it

| pom.xml

\---service

pom.xml

### Top-level project (finance-reports) and pom file

### The top level directory pom.xml (marked in grey) is created because it is required by the Maven archetype plugin. **The top-level pom.xml file should be deleted immediately after archetype generation.** The contracts and the implementation (second-level finance-reports) projects should be built separately and checked into Subversion separately.

### Service Contract

The contract between the consumers and implementers of the service are contained in these documents (these are only skeletons):

finance-reports/finance-reports-contracts/src/main/resources/ReportGenerator.wsdl

finance-reports/finance-reports-contracts/src/main/resources/ReportGenerator.xsd

Service contracts live in a separate project; they have a different (less volatile) lifecycle. They are released on a different schedule compared to the implementation project.

### Service files

A skeleton java file for the CXF endpoint is generated:

finance-reports/finance-reports/service/src/main/java/com/swacorp/finance/ ReportGeneratorImpl.java

Unit test files are placed in finance-reports/finance-reports/service/src/test/java/com/swacorp/finance/ReportGeneratorTest.java

### Integration Test files The integration test driver is at finance-reports/finance-reports/it/src/test/java/com/swacorp/finance/ReportGeneratorIT.java

## Note on very first use of “-it” project

When loading the newly archetype-generated project into Eclipse for the first time, you need to perform an extra step: run mvn install on service project. This is because the CXF codegen plugin is explicitly written to require all dependencies (system, provided, compile, runtime, and test) to be resolved before any code is generated in the “-it” project. This step is only required the very first time you load a new project.

In summary, the steps needed to load the archetype-generated projects to Eclipse are:

* Remove the top-level pom.xml (highlighted in grey above)
* Load all projects into Eclipse. It should generate 4 projects
* Run As -> Maven install on finance-reports-contracts project
* Run As -> Maven clean on finance-reports-service project. This is necessary because Eclipse attempted to compile the just imported Java files (which failed because CXF-generated sources were not yet generated). Running clean wipes all .class that Eclipse used and make the next step succeed.
* Run As -> Maven install on finance-reports-service project
* Run As -> Maven generate-sources on finance-reports-it project
  + Verify that all three projects have no errors at this point
* Run As -> Run on Server on finance-reports-service project
* Run As -> JUnit test on finance-reports-it project

Alternatively, you can bootstrap the process from a command line, for example::

cd \work\finance-reports\finance-reports-contracts

mvn install

cd ..\finance-reports

mvn install

At this time the projects are ready to be imported to Eclipse. All generated code will work in Eclipse.

# MessageInformation

MessageInformation is a mechanism for relaying “transactional” and bookkeeping data throughout the service. For example, many legacy back end services need to know VPAR and other SAAS connection parameters. MessageInformation is a way to receive this data from web service clients and pass it to downstream dependencies (such as CEBS framework 1 CORBA services).



Out of the two components within the MessageInformation, only CommonInformation is mandatory. ServiceInformation is optional and can contain the information specific to a given service.

<sequence>

<element name="commonInformation" type="CommonInformationType"/>

<element name="serviceInformation" type="tmi:ServiceInformationType" **minOccurs="0"**/>

</sequence>

saasInformation within serviceInformation is also optional

<sequence>

<element name="saasInformation" type="SaasInformationType" **minOccurs="0"**/>

</sequence>

The MessageInformation namespace is <http://tbs.swacorp.com/messageinformation/v1>.

## MessageInformation definition in WSDL

MessageInformation is part of the service contract, for example in customerManagement.wsdl:

<wsdl:definitions name="customerManagement" ...

...

<wsdl:types>

<xsd:schema>

<xsd:import namespace= "<http://tbs.swacorp.com/messageinformation/v1>" s chemaLocation="http://repositories.swacorp.com:9092/nexus/content/groups/public/com/swacorp/tbs/contracts/tbs-message-information-contracts/1.1/tbs-message-information-contracts-1.1.xsd"/>

</xsd:schema>

</wsdl:types>

...

<wsdl:message name="CustomerLookupByNameRequestMessage">

<wsdl:part name="parameters" element="sch:retrieveCustomerByNameRequest" />

<wsdl:part name="messageInformation" element="tmi:messageInformation"/>

</wsdl:message>

...

<wsdl:binding name="CustomerManagementSoapBinding" type="tns:CustomerManagementPortType">

<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />

<wsdl:operation name="retrieveCustomerByName">

<soap:operation soapAction="…" style="document" />

<wsdl:input name="CustomerLookupByNameRequestMessage">

<soap:header use="literal" part="messageInformation"

message="tns:CustomerLookupByNameRequestMessage" wsdl:required="true" />

<soap:body use="literal" parts="parameters" />

</wsdl:input>

...

</wsdl:binding>

...

## MessageInformation Project Organization

The MessageInformation schema that defines the SOAP header format is defined in the tbs-message-information-contract project, while tools for manipulating MessageInformation objects are defined in the tbs-message-information project hierarchy. The tbs-message-information project hierarchy contains two projects that developers will use: domain and interceptor.

The*domain* module contains the files which are used by the interceptor module. This module contains the domain classes and utility classes. This module has no webservice stack dependencies (no CXF dependencies).

The*interceptor* contains classes supporting validators and helper classes. This module depends upon the CXF stack for its functioning.

## Including MessageInformation in the project

MessageInformation can be pulled into a project as a Maven dependency in the project’s **pom.xml**

<dependency>

<groupId>com.swacorp.tbs.contracts</groupId>

<artifactId>tbs-message-information-contracts</artifactId>

<version>1.0</version>

</dependency>

<dependency>

<groupId>com.swacorp.tbs</groupId>

<artifactId>tbs-message-information-domain</artifactId>

<version>1.1.0</version>

</dependency>

<dependency>

<groupId>com.swacorp.tbs</groupId>

<artifactId>tbs-message-information-interceptor</artifactId>

<version>1.1.0</version>

</dependency>

The tbs-message-information-contract contains only the XSD, and it should only be referenced as a dependency in a service contract project. All other projects that use message information should import either tbs-message-information-domain or tbs-message-information-interceptor.

## MessageInformation Usage

If you wish to use MessageInformation in your service, add <soap:header use="literal" part="messageInformation"… as shown above. The SOAP header is then available to the server-side code via the generated endpoint, as in customer-flight-availability/service/src/main/java/com/swacorp/tbs/customerflightavailability/CustomerFlightAvailabilityImpl.java:

public class CustomerFlightAvailabilityImpl implements CustomerFlightAvailabilityPortType {

…

public GetAvailableFlightsForCustomerResponse getAvailableFlightsForCustomer(

GetAvailableFlightsForCustomerRequest request,

Holder<MessageInformation> messageInformation)

throws NoSuchCustomer, InvalidStationCode,

CustomerFlightAvailabilityError {

…

The **client applications** can use MessageInformationBuilder to create a valid MessageInformation. For an example, see customer-flight-availability/it/src/test/java/com/swacorp/tbs/customerflightavailability/CustomerFlightAvailabilityIT.java:

Holder<MessageInformation> holder = new Holder<MessageInformation>();

holder.value = MessageInformationBuilder.empty()

.withCallingApplication("myApp")

.withTransactionId("myTx123")

.withMajorVersion(3)

.withMinorVersion(1)

.withSaasPartition("TSS1")

.withSaasPoolId("CEBS:TSS1")

.build();

## Passing MessageInformation in the service implementation

Data from the MessageInformation needs to be extracted from the incoming SOAP request and possibly passed to the request of downstream dependencies (such as CEBS fw1 request or SAAS call). The service implementation can simply transform the MessageInformation to CEBS STI, as in this simplified code from customer-flight-availability/service/src/main/java/com/swacorp/tbs/customerflightavailability/flightavailability/FlightAvailabilityServiceAdapter.java:

public AvailabilityInfo[] getAvailability(String origin,..., MessageInformation mi) throws AvailabilityException {

...

ServiceTransactionInfo sti = new ServiceTransactionInfo();

CommonInformation commonInfo = mi.getCommonInformation();

ServiceInformation svcInfo = mi.getServiceInformation();

ServiceUsage usage = new ServiceUsage();

usage.setCallingApplication(commonInfo.getCallingApplication());

sti.setServiceUsage(usage);

TransactionToken token = new TransactionToken();

token.setTransactionId(commonInfo.getTransactionId());

sti.setTransactionToken(token);

SaasInformation saasInfo = svcInfo.getSaasInformation();

SessionToken sessionToken = new SessionToken();

sessionToken.setConnectionKey(saasInfo.getConnectionKey());

sessionToken.setPartition(saasInfo.getPartition());

sessionToken.setPoolId(saasInfo.getPoolId());

sti.setSessionToken(sessionToken);

...

CheckAvailabilityResponse externalResponse = clientProxy.checkAvailability(sti, externalRequest);

## SOAP Header Validators and Interceptors

The data transmitted in the SOAP header can be used to validate requests. Using interceptors is a convenient way to enforce any validation. That way the bookkeeping tasks are done consistently via configuration, no need to do it in the Java code.

### Enforcing compatible versions (on the server)

The TBS versioning strategy calls for accepting client requests from the same major version client only. For example, 3.1 version of the service can fulfill a request from 3.0 and 3.1 clients but not from 2.5 or 3.3 or 4.0. More, the service should update the outgoing (returning) MessageInformation to put the actual service version in the response. Both tasks can be achieved by configuring the interceptor similar to the one at customer-flight-availability/service/src/main/webapp/WEB-INF/serviceContext.xml:

<jaxws:endpoint id="serviceEndpoint"

implementor="#availableFlightsImpl"

xmlns:svc="http://tbs.swacorp.com/samples/customerFlightAvailability/wsdl/v3"

serviceName="svc:CustomerFlightAvailabilityService"

endpointName="svc:CustomerFlightAvailabilityPort"

address="/customerFlightAvailability"

wsdlLocation="classpath:customerFlightAvailability.wsdl">

<jaxws:properties>

<entry key="schema-validation-enabled" value="true" />

</jaxws:properties>

<jaxws:inInterceptors>

<!-- Used to enforce contract version on incoming requests (for example, I'll refuse

2.5 request if my version is 3.2). -->

<bean

class="com.swacorp.tbs.messageinformation.interceptor.MessageInformationValidator">

<constructor-arg value="${service.version}" />

</bean>

</jaxws:inInterceptors>

<jaxws:outInterceptors>

<!-- Used to set actual contract version on the outgoing response (for example, I'll set

version to 3.2 (my actual release) even if the incoming request was for 3.1 request). -->

<bean class="com.swacorp.tbs.messageinformation.interceptor.MessageInformationVersionSetter">

<constructor-arg value="${service.version}" />

</bean>

</jaxws:outInterceptors>

</jaxws:endpoint>

### Injecting client version (on the client)

For the same reason the client application should use the interceptor to inject/set the current contract version into the outgoing request. This is again useful for consistency – if the client needs to adapt newer version, there is no need to change values hard-wired in the Java code. An example is in the same file, customer-flight-availability/service/src/main/webapp/WEB-INF/serviceContext.xml:

<!-- Required configuration for my downstream dependency, when I make a call to CustomerManagement TBS service. -->

<jaxws:client id="customerManagement"

serviceClass="com.swacorp.tbs.samples.customermanagement.wsdl.v3.CustomerManagementPortType"

xmlns:svc="http://tbs.swacorp.com/samples/customerManagement/wsdl/v3"

serviceName="svc:CustomerManagement" endpointName="svc:CustomerManagementPort"

address="${customer.management.url}"

wsdlLocation="classpath:customerManagement.wsdl">

<jaxws:properties>

<entry key="schema-validation-enabled" value="true" />

</jaxws:properties>

<jaxws:outInterceptors>

<!-- Rewrite the message information contract version number in outgoing requests to

this service. This is needed if the implementation reuses MessageInformation

(it typically does). For example, if I am 3.2 but I call CustomerManagement

2.1 I'd better set the 2.1 version for the outgoing message. Doing it here

ensures it is done all the time. -->

<bean

class="com.swacorp.tbs.messageinformation.interceptor.MessageInformationVersionSetter">

<constructor-arg value="${customer.management.version}" />

</bean>

</jaxws:outInterceptors>

</jaxws:client>

In both cases it requires defining the ${} properties in the pom.xml file or in filter.properties.

# Logging

This section of the document describes the logging implementation for a typical TBS project. The actual logging implementation code is log4j (with log4j.properties used for configuration). Projects that are created using the TBS Maven Archetype will be pre-wired with the latest approved log4j and SLF4J dependency as well as CXF interceptors designed to log SOAP requests and responses. It is up to the application team to choose Java API for logging – either log4j or SLF4J would work out of the box.

The following topics describe how a typical TBS project will be configured by default and how the deployment process is taken into account to ensure logging is flexible and dynamic. Once the TBS project is created, logging can be modified to adapt to specific needs as determined by each development team.

## log4j Properties Files

Because the logging requirements will differ between the various hosting environments, logging resources (log4j.properties) should be located and configured in the resources module. The default log4j.properties, created from the TBS Maven Archetype, is configured to log all statements out to the console with a logging level set to INFO. The default location for the log4j.properties file will be in the common folder of the service’s resources module.

To enable a robust and complete logging solution, the log4j.properties and filter.properties will need to be modified. The simple example below shows the root & logging level settings for the local and dev environments in the filter.properties file along with a snippet from the log4j.properties:

service/resources/src/main/environment/filter.properties settings for the log level and log directory location. These are referred later in log4j.properties, for example as ${logging.directory}:

logging.root.level.local=DEBUG

logging.root.level.dev=DEBUG

logging.soap.level=INFO

logging.directory=logs

logging.directory.local=target/logs

Example customer-management/service/src/main/environment/common/log4j.properties:

log4j.rootLogger=${logging.root.level}, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - <%m>%n

log4j.logger.org.apache.cxf.services=${logging.soap.level}, soap

log4j.additivity.org.apache.cxf.services=false

log4j.appender.soap=org.apache.log4j.DailyRollingFileAppender

...

Maven build process will ensure the right setting is substituted in the log4j.properties for the various deployment environments. The example is simple but illustrates the flexibility to configure logging for the various environments.

## Logging SOAP traffic (request/response/fault)

In addition to the log4j logging capabilities, all TBS projects are pre-wired to log all incoming and outgoing SOAP requests. This logging capability is implemented using Apache’s CXF web services framework in the form of CXF interceptors and is part of the TBS Common architecture. The logging is configurable for options such as logging levels. The following are key points concerning how the logging API behaves in a TBS service:

* The log levels are controlled in a log4j.properties file similar to standard log4j loggers.
* Request and response are each printed on a single line based on the log4j settings.

The figure below shows a high-level representation of how the logging interceptors work.



The Maven configuration and the CXF endpoint configuration to enable the logging is discussed below.

### CXF Configuration

The CXF logging interceptors should be configured in service/src/main/webapp/WEB-INF/serviceContext.xml.

<!-- Configure CXF for XML/SOAP request/response logging. -->

<cxf:bus>

<cxf:features>

<cxf:logging limit=*"2097152"* />

</cxf:features>

</cxf:bus>

### Log file location

Since the log directory is local (see logging.directory property in filter.properties), the location of the file varies based on execution environment:

* If running in Eclipse (as in Run As -> Run On Server… from the service project), the current working directory is the Eclipse **install** folder (which differs from the workspace folder). The log file location might be C:\Program Files\springsource-2.9.1\sts-2.9.1.RELEASE\target\logs
* If running from the command line (as in running mvn install from the top-level project), the log file location is relative to current working directory. For example, if run from c:\my\_project\customer-management, the log file is at c:\my\_project\customer-management\it\target\cargo\configurations\tomcat6x\target\logs

### Sample Log Output

A sample of log output is shown below (with Log4J level set to INFO and Request and Response have been trimmed to fit in a reasonable amount of space):

2012-05-10 07:48:03,289 {tomcat-http--23} INFO [org.apache.cxf.services.CustomerManagement.CustomerManagementPort.CustomerManagementPortType] 3.1.0-SNAPSHOT - Inbound Message

----------------------------

ID: 15

Address: http://localhost:8080/tbs-samples-customer-management-service/customerManagement

Encoding: UTF-8

Http-Method: POST

Content-Type: text/xml;charset=UTF-8

Headers: {accept-encoding=[gzip,deflate], connection=[Keep-Alive], Content-Length=[936], content-type=[text/xml;charset=UTF-8], host=[localhost:8080], SOAPAction=["http://tbs.swacorp.com/samples/customerManagement/wsdl/v3/retrieveCustomerByName"], user-agent=[Apache-HttpClient/4.1.1 (java 1.5)]}

Payload: <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:v1="http://tbs.swacorp.com/messageinformation/v1" xmlns:v3="http://tbs.swacorp.com/samples/customerManagement/v3">

<soapenv:Header>

<v1:messageInformation>

<v1:commonInformation>

<v1:callingApplication>myTestApp</v1:callingApplication>

<v1:transactionId>myTxId</v1:transactionId>

<v1:contractVersion>

<v1:major>3</v1:major>

<v1:minor>0</v1:minor>

</v1:contractVersion>

</v1:commonInformation>

</v1:messageInformation>

</soapenv:Header>

<soapenv:Body>

<v3:retrieveCustomerByNameRequest>

<v3:customerName>

<v3:firstName>John</v3:firstName>

<v3:lastName>Doe</v3:lastName>

</v3:customerName>

</v3:retrieveCustomerByNameRequest>

</soapenv:Body>

</soapenv:Envelope>

--------------------------------------

2012-05-10 07:48:03,289 {tomcat-http--23} INFO [org.apache.cxf.services.CustomerManagement.CustomerManagementPort.CustomerManagementPortType] 3.1.0-SNAPSHOT - Outbound Message

---------------------------

ID: 15

Encoding: UTF-8

Content-Type: text/xml

Headers: {}

Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Header><messageInformation xmlns:ns2="http://tbs.swacorp.com/samples/customerManagement/v3" xmlns="http://tbs.swacorp.com/messageinformation/v1"><commonInformation><callingApplication>myTestApp</callingApplication><transactionId>myTxId</transactionId><contractVersion><major>3</major><minor>1</minor></contractVersion></commonInformation></messageInformation></soap:Header><soap:Body><ns2:retrieveCustomerByNameResponse xmlns:ns2="http://tbs.swacorp.com/samples/customerManagement/v3" xmlns="http://tbs.swacorp.com/messageinformation/v1"><ns2:customer><ns2:customerName><ns2:firstName>John</ns2:firstName><ns2:lastName>Doe</ns2:lastName></ns2:customerName><ns2:accountNumber>11111111</ns2:accountNumber><ns2:address><ns2:addressLine>P O BOX 537</ns2:addressLine><ns2:city>Dallas</ns2:city><ns2:state>TX</ns2:state><ns2:zipCode>75235</ns2:zipCode></ns2:address><ns2:memberSince>26 Apr, 2012</ns2:memberSince></ns2:customer></ns2:retrieveCustomerByNameResponse></soap:Body></soap:Envelope>

--------------------------------------

# TBS Versioning

This section describes versioning conventions for the new services and components developed in the new web services stack. Source-control-system branch and tag naming conventions, which might be expected to include the version number, are outside the scope of this document.

## General Versioning Conventions

In general, a full version number follows the following format:

* X.Y.ZQ

where

* X is the major version number
* Y is the minor version number
* Z is the patch version number
* Q is a qualifier (only relevant in a limited number of scenarios and locations)

and each version sub-field (X, Y, and Z) is a non-negative integer

## Major Version Number

The major version number tends to be a very strong indicator of compatibility wherever it appears. Specifically, different instances of the same entity with the same major version number may be similarly adequate for consumption, whereas different instances with different major version numbers are not similarly adequate for consumption. So if an entity with major version number X1 is being consumed, it is usually assumed that a change to consume a different instance of the entity, with major version number X2, is valid without change to the consumer only if X1 equals X2. (Note that this equality relationship alone is not sufficient to imply compatibility.) For an unequal relationship between major version numbers, say X1 < X2 (e.g., a new release of a component), the new release with the higher value of major version number is not expected to be “backward compatible” with the lower major version number.

The major version number sub-field is required any place that a version number appears.

## Minor Version Number

Minor version number can only be usefully evaluated when given in the context of a major version number. Specifically, it is guaranteed that a change in consumption from one instance of an entity to a different instance with the same major version number will be valid if the minor version number of the new instance is greater than the minor version number of the old instance. For equal major version numbers and equal minor version numbers, evaluation of the patch version number is required to establish expected compatibility. For unequal major version numbers, comparison of minor version numbers is irrelevant.

If the minor version number is present, the major version number must also be present.

## Patch Version Number

Patch version number can only be usefully evaluated when given in the context of a major version number and minor version number. Specifically, it is guaranteed that a change in consumption from one instance of an entity to a different instance with the same major and minor version numbers will be successful if the patch version number of the new instance is greater or equal to the patch version number of the old instance. For unequal major version numbers, or unequal minor version numbers, comparison of patch version numbers is irrelevant.

If the patch version number is present, the major and minor version numbers must also be present.

## Applications of Version Number

There are a number of locations in the development, release, and deployment life cycle where a version number is relevant and should be applied. The complete list of locations depends on whether the entity involved is a service or a (non-service) software component, but there are some locations that are common to both types of development. Those common locations are covered first.

## Maven

Maven is a [build automation](http://en.wikipedia.org/wiki/Build_automation) and software comprehension tool. Use of Maven is assumed, but even in scenarios where that is not the case, the non-Maven versioning conventions outlined in this document are still recommended.

## Repository

Version number is a critical part of the Maven project coordinates. The project coordinates are in turn a critical part of the Project Object Module (POM), and are included in the configuration file (pom.xml) of a Maven managed project. The project coordinates identify the artifact within the Maven repositories (local, enterprise, central, etc.), which is important during the population of the repositories and consumption from the repositories. The artifact version number complies with the general version format, includes all three version sub-fields (major, minor, and patch), and illustrates an instance where the optional suffix (the Q in the general version format shown in section ) may be included: the Maven artifact version number may include a qualifying suffix such as “-SNAPSHOT”, which is a common convention for continuous-integration builds.

Even in cases where Maven was not used to manage the project, any artifact released to a Maven repository should include the full version number.

## Packaging

A Maven-packaged build artifact that is in archive form should have the project coordinate version number included in the archive file name. Inclusion of the project coordinates version number is typical of the built-in package naming done by Maven by default. For example, a project called tbs-maven-environment-plugin with a version number of 1.0.0 might have build artifacts produced with names as follows:

* tbs-maven-environment-plugin-1.0.0-javadoc.jar
* tbs-maven-environment-plugin-1.0.0-sources.jar
* tbs-maven-environment-plugin-1.0.0.jar

Even in cases where Maven was not used to build the project/archive, the archive name should include the full version number.

## Wsdl

## Target Namespace

Any WSDL that is owned by the project should define the target namespace of the WSDL to end with the following:

* /vX

where X is the major version number described in . The “v” is included in order to accommodate java code generation from WSDL in which the package name components (generated from the namespace components) are identical to the namespace components (package names cannot start with an integer).

So, for example, a project with version 3.2.5 might have the following target namespace:

* targetNamespace=”*http://tbs.swacorp.com/services/myservice/wsdl****/v3***”

## Name

The WSDL name attribute should include the major version number in the following format:

* \_vX

where X is the major version number described in . So, for example, the name for MyService version 3.2.5 would be:

* name=”myservice**\_v3**”

## Documentation

The WSDL should contain the following documentation element:

<wsdl:documentation>

A webservice interface for myServiceName.

Version: X.Y

History: 1.0 - Initial Release.

</wsdl:documentation>

where X is the major version number, and Y is the minor version number, described in 0.

So, for example, version 3.2.5 of the MyService service would have the following documentation:

<wsdl:documentation>

A webservice interface for MyService.

Version: **3.2**

History: 1.0 - Initial Release.

* 1. - Interface re-design
  2. - No interface change. (Implementation changes only.)
  3. - Added additional data to response.
  4. - Many modifications to accommodate international travel.
  5. - Added optional parameter to request.

</wsdl:documentation>

## File Name

WSDL files do not use version in their name. For example, a project with version 3.2.5 might have the WSDL stored in a file with the following name:

* MyService.wsdl.

## Schema

## Target Namespace

Any schema definition that is owned by the project should define the target namespace of the schema to end with the following:

* /vX

where X is the major version number described in . The “v” is included in order to accommodate java code generation from WSDL in which the package name components (generated from the namespace components) are identical to the namespace components (package names cannot start with an integer).

So, for example, a project with version 3.2.5 might have schema namespace as follows:

* targetNamespace=*"* *http://tbs.swacorp.com/services/myservice****/v3****"*

## Version

The schema name attribute should include the major version number in the following format:

* X

where X is the major version number described in . So, for example, the version for the schema exposed by MyService version 3.2.5 would be:

* version=”**3**”

## Documentation

The schema should contain the following documentation element:

* X.Yk

where X is the major version number, described in , and Y is the minor version number, described in .

So, for example a project with version 3.2.5 should have a annotation element with the version number, and it might look something like:

<annotation>

<documentation>

A webservice schema with data types for messages for MyService

Version: **3.2**

History: 1.0 - Initial Release.

* 1. - Interface re-design
  2. - No interface change. (Implementation changes only.)
  3. - Added additional data to response.
  4. - Many modifications to accommodate international travel.
  5. - Added optional parameter to request.

<documentation>

</annotation>

## File Name

Any schema that is owned by the service or component does not put version number in the file name. For example a project with version 3.2.5 might have a schema stored in a file with the following name:

* MyComponentName.xsd.

## Generated Code

The recommended client and service architectures require the mapping between the web service transport objects (SOAP messages) and domain objects. This mapping typically involves the conversion between the XML messages and intermediate java objects, where the intermediate java objects are instances of classes generated from the relevant schema or WSDL (e.g., using JAXB). These generated classes, by default (and this default is accepted by convention), have package names based on the schema/WSDL namespace, which means they will include some representation of the version number. This amounts to the implicit inclusion of version number in the package names of the generated classes, which in turn requires the inclusion of version numbers in the mapping code and/or configuration files.

So, for example, for a target namespace of [*http://tbs.swacorp.com/services/myservice****/v3***](http://tbs.swacorp.com/services/myservice/v3)***,*** the generated classes would have a package of

* com.swacorp.tbs.services.myservice.v3

This package name would manifest itself at a minimum in the code or configuration files that map between domain and message transport classes.

## Components

There are no versioning considerations specific to (non-service) software components.

## Services

There are some special considerations for the application of version numbers to services, and those considerations are described below.

## Service Interface

The service interface is defined in WSDL, so the service interface versioning conventions for the (web) service interface are identical to those defined for WSDL (see ). Note that within the WSDL, no further versioning is done other than that general WSDL versioning, even though there are individual elements in the WSDL that could be used to do service interface versioning.

## URL

The URL for a web service should contain only the major version number, included at the end of the URL, in the following format:

* /vX

where X is the major version number described in .

So, for example version 3.2.5 of MyService might have a URL with the following name:

* http://someHost:8080/tbs-services/MyService**/v3**

## Service Request

The MessageInformation must be passed as part of a service request, and should include values for both *majorServiceVersion* and *minorServiceVersion* as part of the *messageVersion* property - e.g., via MessageInformation.getCommonInformation().setServiceVersion(serviceVersionInstance). The *serviceVersionInstance* is defined by values for *majorServiceVersion* and *minorServiceVersion*, which indicates the *minimum acceptable service version*, as determined by the client. The values are provided as separate properties, and are both int values. For example, if the minimum acceptable service version as determined by the client is 1.3, the *majorServiceVersion* property should be set to 1, while the *minorServiceVersion* property should be set to 3.

The client can get the value of this version number, for example, from the published WSDL (major and minor version numbers). The patch version number is not considered in terms of the message information, and so is not passed in the message information as part of the service request.

## Deployed Service Description

#### Released Services

The tcServer instances for environments hosting builds of released services are managed by Hyperic. The name for each such instance is derived from the name of the single web service that is deployed to it, and this derived name is used to request the instance creation from the Web Services group. The instance name is therefore the key to determining what version of a give service is running in a given environment (a common question that a user of the Hyperic Console may want to answer).

The tcServer instance should be created with the following included in its name:

* \_vX\_Y\_Z

where X, Y, and Z are defined in section . So, for example, version 3.2.5 of the MyService service would run on a tcServer instance (managed by Hyperic) with the following name:

* MyService**\_v3\_2\_5**

#### Non-Released (Development) Services

There may be a need to deploy a non-released (development) version of a service into a tcServer instance in a development environment (not managed by Hyperic). This may be required, for example, to do proof of concept work involving a number of collaborating services. The version information available within the tcServer Manager for each deployed service, and displayed on the service console, must include the full version number:

* Version X.Y.ZQ

where X, Y, Z, and Q are defined in section . The display-name element in the web.xml file needs to appropriately populated in order to make this information available. So, for example, version 3.2.5-SNAPSHOT of the MyService service would provide the following information in the display-name element in web.xml:

* tbs-services-MyService version **3.2.5-SNAPSHOT**

#### Service Version (Runtime Behavior)

Although the service interface, as identified in , contains only the major version number, the runtime enforcement of compatibility, and the request and response versions involved in service calls, involves a more complex set of considerations, which are primarily repercussions of the rules described above.

#### Backward Compatible WSDL Modifications

Backward compatible WSDL modifications affect the version number as follows:

* the WSDL version remains unchanged (it includes only the major version number), but the WSDL documentation element includes the new major and minor version numbers.
* the runtime MessageInformation verification uses the new version number (the full version number)

Backward compatible WSDL modifications affect the service consumers as follows:

* In the event of a change to the content of an operation’s request type:
  + Clients built with the old request definition can still send the old request
  + Clients built with the new request definition can send the new request
* In the event of a change to the content of an operation’s response type:
  + All clients receive the new response type

All of the above consumer impacts are expected, since presumably the request and response types were changed in a “backward compatible” fashion.

#### Runtime Version Validation

If the complexity involved in the release schedules of a service and its consumers requires it, runtime service version validation can be performed. A request is considered valid if the messageVersion specified in the MessageInformation for the request meets the following criteria:

* the requested major version number equals the service major version number
* the requested minor version number is less than or equal to the service minor version number

If either of these criteria is not met, an exception is thrown. These checks are done before any business logic is executed in the service **but after authentication and authorization checks (if any) have passed.** If runtime service version validation is required, the client must send the following information in the messageVersion property of CommonInformation class (which is included in the MessageInformation):

* majorServiceVersion (X)
* minorServiceVersion (Y)

where X and Y are defined in section . So, for example, if a client requires runtime validation of version 3.2 of service MyService, it would pass the following as the contents of serviceVersion:

* majorServiceVersion (X) = 3
* minorServiceVersion (Y) = 2

This would pass validation of a running instance of MyService version 3.2.z, or, more generally, 3.y.z, where y>= 2. The patch version number, z, is not considered in the runtime version validation.

## Version Application Consistency

A consistent version number should be applied throughout each entity (service, component).

## Values

Consistent version number sub-field (major, minor, patch) values should be used at all points of application. Where only a truncated representation of the version number is relevant (e.g., X, X.Y) the sub-field value consistency should still be maintained for the portion included.

## Format

The version number format used at all points of application should be that specified in section unless that format is invalid at the point of application. Where only a truncated representation of the version number is relevant (e.g., X, X.Y) the format should still be maintained for the portion included. In some cases a separator prefix (e.g., “\_” or “-“), a prefix indicating the start of a version number (e.g., “v”), or a prefix accommodating code generation (e.g., “v”), is added as required.

## Web Service Example

Version number consistency within a web service requires the following criteria to be met.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Context** | **Location** | **Format** | **Example** | **Notes** |
| **General** | **General** | **X.Y.ZQ** | **4.5.2-SNAPSHOT** | The conceptual service version for this example. It can either be considered as imposed on the versioning below, or derived from it. At any rate, consistency must be maintained in the locations shown below. |
|  |  |  |  |  |
| Maven | repository | X.Y.ZQ | **4.5.2-SNAPSHOT** | Only DEV environments would have “-SNAPSHOT”. |
| Maven | packaging | X.Y.ZQ | MyService-**4.5.2-SNAPSHOT**.war |  |
| WSDL | target Namespace | /vX | targetNamespace=*"* *http://tbs.swacorp.com/services/myservice/wsdl****/v4****"* |  |
| WSDL | name | \_vX | name=”myservice**\_v4**” |  |
| WSDL | documentation | X.Y | <documentation>  A webservice interface for myServiceName.  Version: **4.5**  History: 1.0 - Initial Release.  </documentation> |  |
| WSDL | file name | \_vX | MyService**\_v4**.wsdl |  |
| schema | target Namespace | /vX | targetNamespace=*"* *http://tbs.swacorp.com/services/myservice****/v4****"* |  |
| schema | version | X | version=”**4**” | Not implemented |
| schema | documentation | X.Y | <annotation>  <documentation>  A webservice schema with data types for messages for MyService.  Version: **4.5**  History: 1.0 - Initial Release.  <documentation>  </annotation> |  |
| schema | file name | \_vX | MyService**\_v4**.xsd |  |
| runtime | URL | /vX | http://tbs.swacorp.com/myservice**/v4** |  |
| runtime | service request (client) | X.Y | MessageVersion sv = new MessageVersion();  sv.setMajorServiceVersion(4);  sv.setMinorServiceVersion(5);  messageInformation.getCommonInformation().setServiceVersion(sv); | Could be as low as 4.0 and as high as **4.5.** Optionally verified by service. Note that patch version is not considered. |
| runtime | tcServer Manager Console | X.Y.ZQ | tbs-example-SampleService version **4.5.2-SNAPSHOT** | Description in tcServer Manager console.  [Not implemented] |
| runtime | Hyperic Service Console instance name | vX\_Y\_Z | MyService**\_v** **4\_5\_2** | Name included in request to Web Services team for instance creation.  [Not implemented] |
| Java | Package name | vX | com.swacorp.sevices.myservice.**v4**.SomeGeneratedRequestOrResponseClassOrDependencyThereof | Generated code has version numbers from the namespaces.  Presumably this impact is limited to the classes and configuration files that take part in the mapping between domain and transport objects. |

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Description | Author |
| 11/17/2011 | 1.0 | Initial | Tim Kelley |
| 02/06/2012 | 2.0 | Update namespaces to v2 and remove message context interceptors. | Josh Beitelspacher |
| 02/07/2012 | 2.0 | Updated logging diagram | David Martin |
| 02/29/2012 | 2.0 | Shuffled sections, updated setup | Peter Durcansky |
| 05/16/2012 | 3.0 | Updated for 3.0 release, removed Moe | Peter Durcansky |