

## Bulletproofing Web Services

Web services are gaining industry-wide acceptance and usage. They are moving from proof-of-concept deployments to actual usage in mission-critical enterprise applications. While Web services allow businesses to connect to partners and customers, this same flexibility and connectivity provide an increased opportunity for errors.

As companies and consumers rely more on Web services, it becomes increasingly important for Web services developers to know how to properly design, develop, deploy, and ultimately manage a Web services system. However, because of the inherent complexities that can arise with a Web service implementation, it can be difficult to grasp practical fundamentals and devise a step-by-step plan for Web services development.

Therefore, we will take a look at the nuts and bolts of implementing and deploying a reliable, high-quality integration system – or rather, a bulletproof Web service. By implementing the Parasoft Automated Error Prevention (AEP) Methodology into your development process, you can be sure that fundamental development practices are established for a reliable Web service. The AEP Methodology provides clear and practical guidelines for every Web service implementation detail — from the required practices and infrastructure elements, to a plan for introducing each practice and element, to a team workflow that makes the most efficient use of the prescribed practices and elements.

This paper will explain issues specific to Web services and will illustrate the solid engineering and testing practices required to ensure complete Web service functionality, interoperability, and security. Whether creating Web services from scratch or integrating legacy back-end servers via Web services, the practices and principles outlined in this paper will be of great benefit.

## ***Web Service Creation: Planning and Design***

To make the discussion as concrete and pragmatic as possible, we will walk you through a sample Web service implementation. The example used will be a service for a large realtor with office branches across the country. This realtor is in need of implementing a Web services initiative that supports the following requirements:

- Potential and existing customers will submit contact information, desired living location, and desired price range of a home via the Web service. These users should receive a response from the server that gives them the location of the branch closest to them as well as an estimate of the monthly mortgage. This will enable users to contact a real estate agent and begin the process of finding a home.
- Real estate agents from different branches will submit a request for a list of potential customers who are looking for homes in the local area. This will enable the real estate agents to earn business and establish contact with interested customers.

## **Target Requirements**

To build the example Web service, we begin with two target requirements. As the name suggests, these targets are landmarks within the development process that we are aiming to reach. These targets will help to drive the feature set of our Web service and enable us to measure our progress. When we hit these targets, we know we are on the right path.

**Target 1:** A use case scenario shall pass after meeting these requirements:

1. A dummy request is sent to the Web service for customers.
2. A SOAP response is received.

3. The SOAP response contains a SOAP fault.

**Target 2:** A use case scenario shall pass after meeting these requirements:

1. A valid request is sent to the Web service for agents.
2. A SOAP response is received.
3. The SOAP response contains a list of customers that may be of length zero.

For each target, a test case is created to verify that each requirement is met. At the beginning, each of these test cases should return an "incomplete" failure message to clearly indicate that the related feature has not yet been implemented. These test cases will continue to fail until the feature is implemented. We start with two targets, but this number is arbitrary. For a bigger project, you may want to have 10 targets, each one measuring an incremental step. You can track how close you are to completing your project by monitoring the test cases for these targets.

You may use various frameworks or tools to create test cases for the targets, but for this example we will use Parasoft SOAPtest to verify that the targets are met. Whatever framework or tools you decide to use, the use case scenario involves a SOAP Client sending a message, waiting for a response, and then verifying the responses.

## Robustness Requirements

When test cases for our targets succeed, we can begin to flesh them out and verify that the new feature is implemented completely and correctly. As seen in Figure 1, robustness requirements should be met before moving on to the next target feature.

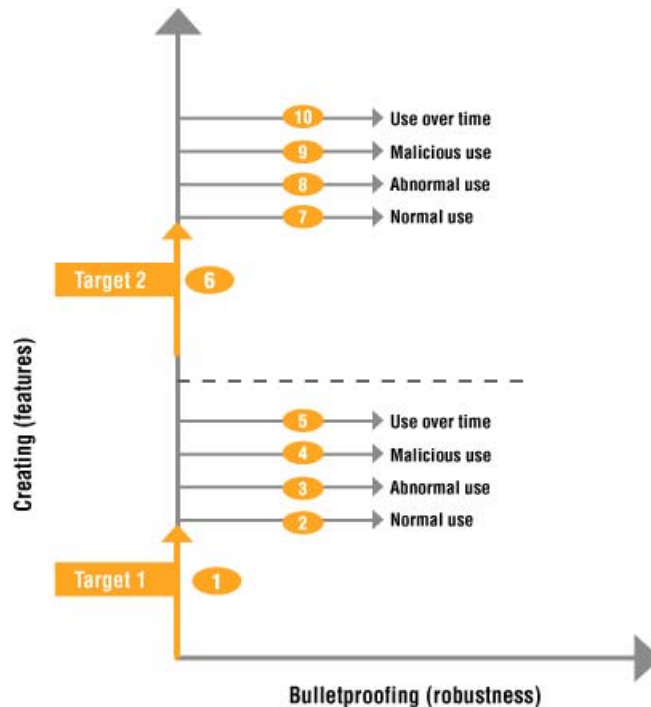


Figure 1: Target and Robustness Requirements

Whereas the target requirements drive the features set, these additional requirements ensure the robustness of the Web service:

**Normal Use:** The Web service must function in the manner for which it was designed. For each operation exposed through the Web service, the request and response pair should adhere to the binding, and the XML should conform to the message description. In short, the server and client send and receive what is expected.

**Abnormal Use:** The Web service must function even when it is being consumed outside the lines of its intended use. An abnormal use case would involve sending a value other than those expected or not sending a value at all. For example, one application may send an XML instance document based on an older version of schema, and the receiving application may be using a newer version of schema. In any case, a Web service should alert the consumer appropriately without any malfunctions.

**Malicious Use:** The Web service must function even when it is deliberately and maliciously being consumed outside the lines of its intended use. For example, hackers may try to gain access to privileged information from a Web service transaction without authorization, or they may attempt to undermine the availability of the Web service. To be able to function under— or even prevent— malicious use, a Web service should have security measures in place.

**Use Over Time:** A Web service implementation is likely to change over time. For example, perhaps a Web service exposes an application that is undergoing an iterative development process. Any Web service must continue to function properly during its entire lifespan, even as it is evolving.

## Initial Architecture

Before jumping into the necessary steps needed to fulfill the target and robustness requirements above, we'll take a brief look at the parts of the Web service that will need to be developed, tested, and verified. The initial architecture of the Web service will be comprised of the following components:

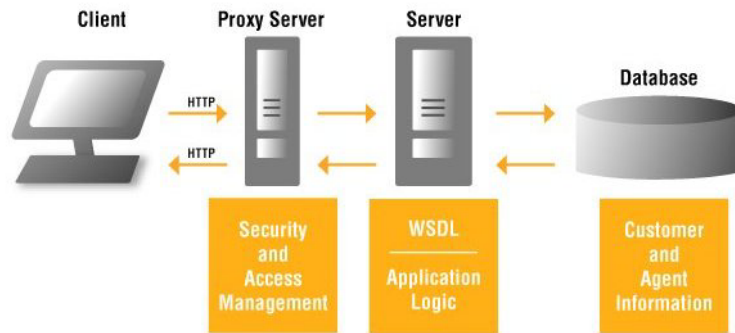


Figure 2: Initial Architecture

- Application Logic (or business logic) – handles requests from customers and agents, makes necessary connection to the database, and returns response to customers and agents. For our example, we will use Java for the application logic. Another common option is to use C# for the .NET platform.
- Database – stores relevant information about customers and agents. We will use MySQL.

- Server – SOAP enabled HTTP server that handles serialization from XML to objects for the application logic. The Apache Axis SOAP engine is an open source SOAP implementation that can be deployed on any J2EE server.
- Proxy Server – Allows for security and access management, so that customers and agents have different levels of access to the Web services that are available.
- WSDL (Web Service Description Language) document– a description of the Web service
- Client – The Web Service client that the customers and agents will use to invoke the Web services.

For our example, we will choose HTTP as the transport layer. When reliable messaging is required, SOAP over JMS is a good option.

## Critical Infrastructure

Now that we have explained the basic necessities of the service to be created, we must now concentrate on the foundations from which this service can be built. For Web service development to be successful, specific practices must be implemented correctly and consistently throughout your development group. This consistent application requires you to ensure that your development group has an appropriate supporting infrastructure, then ensure that the group follows a workflow from which error prevention practices are performed appropriately. Until this critical infrastructure is in place, you cannot expect a team to begin the development of a bulletproof Web service.

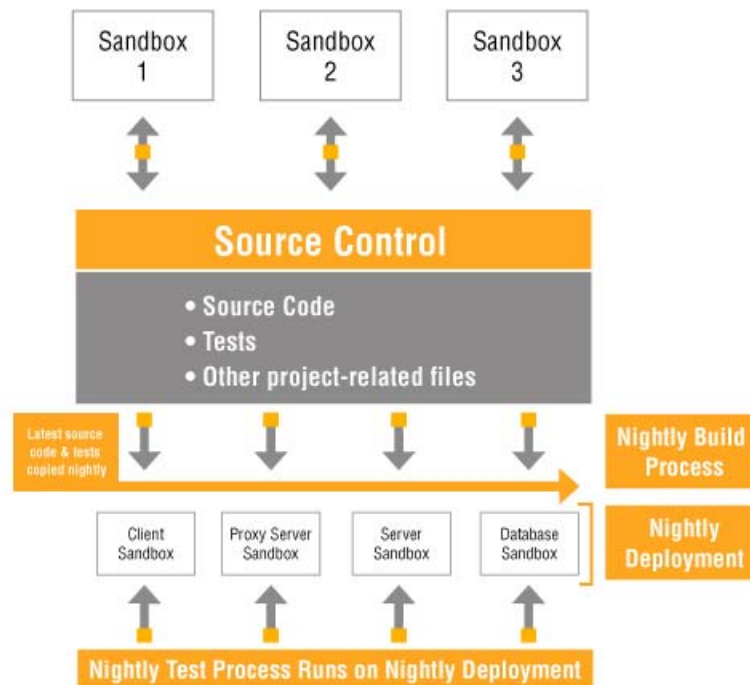


Figure 3: Critical Infrastructure

As seen in Figure 3, your development group must have a functioning source control system and automated build process before its members can begin writing any code. A source control system and an automated build process are *the* fundamental requirements needed to ensure the development of quality Web services. As you will see, establishing this infrastructure provides the necessary framework for creating reliable Web services.

## Source Control

A source control system is a database where source code is stored. Its purpose is to provide a central place where the team members can store and access the entire source base. There are two main reasons why we require a source control system.

First, source control gives each developer the freedom and safety to write, modify, and refactor code – even when it is risky on his own sandbox (we explain the concept of sandbox below). If the code change turns out to be undesirable, the developer can easily undo his changes by reverting back to the code in the source control. While he is working on his changes, the rest of the developers always have a version of the code that is working.

Secondly, having a source control system is a pre-requisite for the nightly build process. All of the files needed for the build process should be in source control. As we explain later, the nightly processes access all required files from the source control.

Most organizations do not understand how to effectively use their source control system. Many simply under use their source control system, don't require its use at all, or they have it configured wrong for the group environment. To put your source control system to proper use, it is important that you understand and establish guidelines for your developers. Source control systems are so important that without a properly configured system, quality software cannot be made.

### *Using a Sandbox*

A sandbox is an area where copies of source code and other project-related files can be stored and manipulated without affecting the master source code base. As mentioned before, the reason that each developer should have his own developer sandbox is so he can have the freedom and safety to undertake code changes even when it is risky. In addition to each developer having his own sandbox, organizations should keep one sandbox called the build sandbox. The nightly build process will take place on the build sandbox.

#### **Guidelines for Developer Sandbox:**

- None of the files in the developer sandbox should be checked out for extended periods of time.
- Once a developer is finished writing a feature, he should delete all files from his developer sandbox and then shadow a new copy. This way, the sandbox stays in synch with the source of the application.
- Before checking in any code into the source control system, it is vitally important that the code is compiled first. Developers should never check in code that has not been compiled. The moment code is written, developers should resolve all compilation errors and compile-level warnings.

### **Guidelines for Build Sandbox:**

- The build sandbox should be cleanly shadowed (i.e., receive read-only copies of the master files stored in the source control system) from the source control system and deleted on a daily basis.

These principles may sound obvious, or even naïve, but it is surprising to find that many software organizations lack a clear policy of what is allowed to be checked in and out of their source control systems.

## **Nightly Build, Deployment, and Test Process**

Once a source control system is in place, the next step is to establish an automated nightly build, deployment, and test process. The main reason for establishing these automated processes is to be able to monitor the progress of our development. The results from the nightly build process tell us whether there are any incompatible changes in the application components. A side benefit to having a nightly build process is that in cases where a build is shipped or released, we already have a tested build process that has been running all along. The nightly deployment process serves to set up a context in which a set of tests can be run that verify our Web service. This enables us to run the nightly test process, which in turn tells us whether the Web service continues to run as expected even as it continues to grow and change.

### ***Nightly Build Process***

A separate build computer (different than the computers used by developers) should be designated for the nightly build process. A separate computer is used because configurations and system settings on a developer machine can sometimes hide various dependency errors.

Each night, all the source code and related files should be shadowed from the source control system. A scheduled task should initiate the build script that compiles the necessary components and builds the application. The results of the nightly build process must be monitored each morning. If for any reason the build process fails, the failure must be investigated and resolved so that the build process succeeds and the following processes can be performed.

### ***Nightly Deployment Process***

If the nightly build process succeeds, the nightly deployment process should be launched. This process should also be automated via a script that is initiated by a scheduled task after the nightly build process succeeds. The deployment consists of the WSDL, server, database, client, and proxy server. These components comprise our staging system, an n-tiered test bed for our testing.

First, the WSDL should be created from the latest source code and exposed on a port on the same machine that the build process ran on. The WSDL should reference the most recent versions of the schemas. The application should be then exposed as a Web service on the same machine. This machine, which should be the server, should be accessible on the network and should have reliable connection to the database. The database should be set to its default configuration during this process. The Web service client, when applicable, should be created from the latest source code and deployed. The nightly deployment process should also be monitored each night so that any errors can be detected and fixed right away.

### ***Nightly Test Process***

Finally, a nightly test process must be implemented. In this process, the newly built application, WSDL, Web service, and client are automatically tested to verify that they satisfy all the

requirements and that no regressions occur in the functionality. All the test cases should be shadowed from the source control and run. Any failures must be reported and monitored the next morning. In this way, a feedback loop is established whereby errors are detected and fixed as soon as they are introduced.

## ***Stages of Development***

In this section, we describe the progression of developing a bulletproof Web service. The process is broken down into stages, each with its own set of requirements and practices. The order of the stages is not absolute. In this paper, the stages are presented in the following order, however, in practice, it makes sense to work on multiple stages in parallel. And as we will reiterate in the load testing stage, load testing should not be left until the end, but used throughout the entire development lifecycle.

### **Stage One: Application Logic**

#### **a) Divide Application Logic into modules that provide needed functionality**

First, we begin by writing the code that makes up the application logic. From a high level, the logic can be divided into several modules. This allows for parallel development of individual units that can later be integrated. In our example, we can divide the code into the following modules:

- **Module 1:** From the customer request, process contact information, desired living location, price range, then return the location of the closest branch and mortgage information.
  - Sub-module A: Submit contact information into the database
  - Sub-module B: Calculate mortgage information from the price range
  - Sub-module C: Retrieve closest branch(es) from the contact information and desired living location
  
- **Module 2:** From the agent, process agent information, then return a list of potential customers in the area.
  - Sub-module A: verify agent information
  - Sub-module B: based on agent location, query the database for potential customers around that area.

#### **b) Define functionality requirements for each module**

For each sub-module, the functionality requirements should be decided on. For instance, we define the following requirements for Sub-module 2 of Module 1:

- **Inputs:** cost of home, down payment, estimated interest rate, length of loan in years
- **Output:** monthly payment
- **Requirements:** should identify bad inputs
  - a) Cost of home is less than the down payment

- b) Interest rate is less than 0 or greater than 100%
- c) Length of loan that is not one of the following: 10, 15, 20, or 30
- For valid inputs, return correct monthly payment rounded to the nearest dollar.

***Begin by writing classes that represent the objects.***

Suppose we've been assigned to write the code for Module1: sub-module B. We begin by writing the class that encapsulates the input for this module.

```
public class PaymentInfo {
    private double homePrice, downPayment, interestRate;
    private int years;
    public PaymentInfo(double homePrice, double downPayment,
        double interestRate, int years) {
        this.homePrice = homePrice;
        this.downPayment = downPayment;
        this.interestRate = interestRate;
        this.years = years;
    }
    ... get and set methods...
}
```

***Write the methods that provide required functionality***

Next, we write the method that takes PaymentInfo as the input and returns the monthly payment.

```
Public class RealtorServiceImpl {
    ... other methods ...

    protected int getMortgageResponse(PaymentInfo paymentInfo) {
        // TODO: implement with help of JUnit test cases
        return 0;
    }

    ... other methods ...
}
```

**c) Create unit tests that verify the requirements are met. Whenever possible, create unit tests prior to writing the code.**

When writing code, it is tempting to take a large step. The problem with taking large steps is that if something does not work properly after the large step, you have to backtrack through that large step to find the problem. It is better to think of the next small step to accomplish and validate that step before moving on to the next step. By taking many validated small steps, we will have made the equivalent of one large step, except that now we are a lot more confident about that large step.

We use unit tests to take these small steps that ultimately result in requirements being met and validated. Unit tests serve a two-fold purpose. First, they frame the functionality that is needed. Second, they validate that the functionality works.

Each module should be developed along with unit tests that verify its functionality. By requiring the development of unit tests before or during, and not after, the development of the code, we will ensure that the code actually provides the functionality. The span of the unit tests should be such that every functionality requirement of the module is verified. In addition, by using unit tests to drive development, we avoid writing unnecessary code that does not need to be written.

From an architect or project manager's point of view, once an entire array of unit tests are written, this gives a progress monitor of how the code is being developed. As more and more unit tests succeed, we know that code is closer and closer to completion.

**Write JUnit test cases that verify that the requirements are met.**

Note that the previous method currently returns 0 for all inputs. This is okay. We now create JUnit tests that verify that the requirements are met. One easy way to do this is to use Parasoft Jtest. Jtest automatically creates JUnit test cases for every method in the source code. In this case, Jtest creates the following class with the JUnit test case to test getMortgageResponse() method.

```
public class RealtorServiceImplTest extends PackageTestCase {
    /**
     * Test for method: getMortgageResponse(PaymentInfo)
     * @see RealtorServiceImpl#getMortgageResponse(PaymentInfo)
     * @author Jtest
     * @throws junit.framework.AssertionFailedError
     */
    public void testGetMortgageResponse() {
        PaymentInfo t0 = new PaymentInfo(7.0, 7.0, 7.0, 7);
        RealtorServiceImpl THIS = new RealtorServiceImpl();
        // jtest_tested_method
        double RETVAL = THIS.getMortgageResponse(t0);
        assertEquals(0.0, RETVAL, 0.0); // jtest_unverified
    }
    ... other test cases ...
}
```

Using the JUnit test case that was created for us, we now write positive test cases. By referencing another source, we know that for a 30 year loan of \$100,000 at 6.5%, the monthly payment should be \$632. So we now write the JUnit test case that will test for this.

```
public void testGetMortgageResponse() {
    PaymentInfo t0 = new PaymentInfo(100000.0, 0.0, 6.5, 30);
    RealtorServiceImpl THIS = new RealtorServiceImpl();
    // jtest_tested_method
    int RETVAL = THIS.getMortgageResponse(t0);
    assertEquals(632, RETVAL, 632);
}
```

We should write a few more positive test cases, making sure that we test the second argument of PaymentInfo object. For instance, here is another test case.

```
public void testGetMortgageResponse2() {
    PaymentInfo t0 = new PaymentInfo(120000.0, 20000.0, 6.5, 30);
    RealtorServiceImpl THIS = new RealtorServiceImpl();
    // jtest_tested_method
    int RETVAL = THIS.getMortgageResponse(t0);
    assertEquals(632, RETVAL, 632);
}
```

**Modify the code until the JUnit tests pass**

Running the JUnit test cases now will fail. Now we modify our getMortgageResponse() method until the JUnit test cases succeed. Here is our code when the JUnit test cases begin to pass.

```
/*
    P = 
$$\frac{rL(1 + r/12)^{(12N)}}{12((1 + r/12)^{(12N)} - 1)}$$

*/
```

where:

```
    r = interest rate (for 8.25% ==> r = 0.0825)
    L = loan amount
    N = loan time (years)
    P = the monthly payment
*/
protected int getMortgageResponse(PaymentInfo paymentInfo) {
    double amount = paymentInfo.getHomePrice() -
        paymentInfo.getDownPayment();
    int years = paymentInfo.getYears();
    double rate = paymentInfo.getInterestRate()/100;
    double base = (double)(1 + rate/12);
    double exponent = (double)(12 * years);
    double result = java.lang.Math.pow(base, exponent);
    double numerator = rate * amount * result;
    double denominator = 12 * (result - 1);
    int monthlyPayment = (int)(numerator / denominator);
    return monthlyPayment;
}
```

**Iteratively, add more JUnit tests and modify code until all requirements are met.**

Now that our method returns correct monthly payments for valid inputs, we now test how our method handles bad inputs. For instance, this JUnit test case makes sure that a `BadInputException` is thrown when the down payment is greater than the cost of the house.

```
public void testGetMortgageResponseWithBadInput() {
    BadInputException exception = null;
    try {
        PaymentInfo t0 = new PaymentInfo(0.0, 100000.0, 6.5, 30);
        RealtorServiceImpl THIS = new RealtorServiceImpl();
        // jtest_tested_method
        int RETVAL = THIS.getMortgageResponse(t0);
    } catch (BadInputException bie) {
        exception = bie;
    }
    assertTrue(exception != null);
    String msg = RealtorServiceImpl.NL_BAD_AMOUNT;
    assertEquals(msg, exception.getMessage());
}
```

After many, iterations, our modified code looks like this:

```
protected int getMortgageResponse(PaymentInfo paymentInfo)
    throws BadInputException {
    double homePrice = paymentInfo.getHomePrice();
    if (homePrice < 0) {
        throw new BadInputException(NL_BAD_HOME_PRICE); } Added as result of JUnit test
    }
    double downPayment = paymentInfo.getDownPayment();
    if (downPayment < 0) {
        throw new BadInputException(NL_BAD_DOWN_PAYMENT); } Added as result of JUnit test
    }
    int years = paymentInfo.getYears();
    if (years != 10 || years != 15 || years != 20 || years != 30) {
        throw new BadInputException(NL_BAD_YEARS); } Added as result of JUnit test
    }
    double rate = paymentInfo.getInterestRate()/100;
    if (rate < 0 || rate >= 1) {
        throw new BadInputException(NL_BAD_RATE); } Added as result of JUnit test
    }
}
```

```

double amount = paymentInfo.getHomePrice() -
                paymentInfo.getDownPayment();
if (amount < 0) {
    throw new BadInputException(NL_BAD_AMOUNT); } Added as result of JUnit test
}
double base = (double)(1 + rate/12);
double exponent = (double)(12 * years);
double result = java.lang.Math.pow(base, exponent);
double numerator = rate * amount * result;
double denominator = 12 * (result - 1);
int monthlyPayment = (int)(numerator / denominator);
return monthlyPayment;
}

```

Notice that the code is more robust now. This is because the JUnit test cases ensured that all the requirements were met.

#### **d) Unit tests are checked into source control and run nightly as regression tests**

Once a requirement is met, it is frozen, and we can move on to the next requirement. The way to make sure that the requirement stays frozen is to check the unit tests into source control and run them as regression tests during the nightly test process. This means that once a unit test passes, it should always pass. There will be times when a change to a seemingly unrelated part of the code causes undesirable behavior in another part of the code. By having these unit tests that serve as regression tests, we will be able to detect these undesirable changes.

The unit tests and coding standards applied in this stage should always pass even as the application logic changes over time. As the functionality becomes increasingly complex, previous tests should continue to pass and new tests that test new functionality should be added.

#### ***Check code and JUnit tests into source control***

The source code can now be checked into source control. As part of each nightly build process, the checked-in source code will be compiled with the rest of the source code to make sure everything builds correctly.

The JUnit tests are now checked into source control as well and set up so that they are run nightly as part of the nightly test process. When using Jtest to run JUnit tests, a Jtest project can be created from the source code and run nightly using its scheduled task feature

#### **e) Apply coding standards as part of the nightly test process**

There are two main reasons for applying coding standards. First, we want our code to be written in a consistent style by all the developers. We want to avoid styles and paradigms that have been found to lead to bugs. A consistent style leads to better readability, maintainability, and in the end, reusability of the code. The coding standards will enforce a certain coding style among both experienced and new developers on the team.

Secondly, coding standards allow us to catch errors and bugs long before they appear in the application during run-time. There are many bugs that can be found by running coding standards and we do not want to waste development time chasing these bugs. We should be sure to enforce coding standards that apply to database connections. In particular, we want to make sure that we do not leave unnecessary database connections open. This leads to an instable Web service that can take a long time to debug.

Coding standards are most effective when they are added incrementally and as unobtrusively as possible upon the development process. The enforcement of coding standards should be incorporated into the nightly test process and run on the latest source code each night.

The best way to apply coding standards is to enforce them on recently checked in source code. Using some of Jtest's built-in coding standards, we make the following changes:

```
if (years != 10 || years != 15 || years != 20 || years != 30) {  
becomes if ((years != 10) || (years != 15) || (years != 20) || (years != 30)) {  
due to coding convention standard  
and double base = (double)(1 + rate/12); becomes double base = 1 + rate/12;  
due to unnecessary cast.
```

#### **f) When bugs are found, tests must be created that verify that the bug is fixed**

Whenever a bug is found, it should be fixed, and a unit test should be added for that case. If possible, a coding standard should be created that will catch similar patterns in the future. We show an example of this in Stage Three.

## **Stage Two: The WSDL**

### **a) Create and Deploy WSDL as early as possible on the staging server environment as part of the Nightly Deployment Process**

Before moving on to the creation of the WSDL, it must be noted that the creation of WSDL does not have to follow writing of the application code. There are cases in which the WSDL is created before any code is ever written. This "WSDL-first" approach is useful when you're already dealing with XML in your applications and you already have schemas. Various industries are promoting communication via the Internet – as such, the use of XML and common schemas that will allow them to work as one global supermarket of products and services. (One example can be seen from the travel industry in which the Open Travel Alliance has created and distributed schemas for business-to-business information exchange).

In our example, it makes sense for the WSDL to be generated from the code. This "code-first" approach makes sense when applications that operate at the level of objects are being developed or are being exposed. On the J2EE platform, Axis provides a Java2WSDL tool that automatically generates a WSDL from the Java code base. The .NET platform supports similar features for code written in C#. In either case, the latest WSDL should be deployed nightly on the staging server environment.

Using Parasoft SOAPtest, the WSDL can be automatically created from the source code implementation of the following interface:

```
public interface RealtorService {  
    CustomerResponse submitRequest(CustomerInfo customerInfo,  
                                   DesiredLocation desiredLocation,  
                                   PaymentInfo paymentInfo);  
    PotentialCustomers queryPotentialCustomers(AgentInfo agentInfo,  
                                               DesiredLocation location);  
}
```

The WSDL looks like this:

```
<?xml version="1.0" encoding="UTF-8" ?>  
<wsdl:definitions targetNamespace="..."  
xmlns="http://schemas.xmlsoap.org/wsdl/"  
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"  
xmlns:apachesoap="http://xml.apache.org/xml-soap"  
... other namespaces declared ...
```

```

xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<wsdl:types>
  <schema targetNamespace="http://DefaultNamespace"
    xmlns="http://www.w3.org/2001/XMLSchema">
    <import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
    <complexType name="CustomerInfo">
      <sequence>
        <element name="city" nillable="true" type="xsd:string" />
        <element name="firstName" nillable="true" type="xsd:string" />
        <element name="lastName" nillable="true" type="xsd:string" />
        ... other types defined here ...
      </sequence>
    </complexType>
  </schema>
</wsdl:types>
<wsdl:message name="submitRequestRequest">
  <wsdl:part name="in0" type="tns1:CustomerInfo" />
  <wsdl:part name="in1" type="tns1:DesiredLocation" />
  <wsdl:part name="in2" type="tns1:PaymentInfo" />
</wsdl:message>
<wsdl:message name="submitRequestResponse">
  <wsdl:part name="submitRequestReturn" type="tns1:CustomerResponse" />
</wsdl:message>
<wsdl:portType name="RealtorServiceImpl">
  <wsdl:operation name="submitRequest" parameterOrder="in0 in1 in2">
    <wsdl:input message="impl:submitRequestRequest" name="submitRequestRequest"
  />
    <wsdl:output message="impl:submitRequestResponse"
name="submitRequestResponse" />
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="realtorSoapBinding" type="impl:RealtorServiceImpl">
  <wsdlsoap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="submitRequest">
    <wsdlsoap:operation soapAction="" />
    <wsdl:input name="submitRequestRequest">
      <wsdlsoap:body namespace="http://DefaultNamespace" use="literal" />
    </wsdl:input>
    <wsdl:output name="submitRequestResponse">
      <wsdlsoap:body
namespace="http://trout.parasoft.com:8080/axis/servlet/AxisServlet/realtor"
use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="RealtorServiceImplService">
  <wsdl:port binding="impl:realtorSoapBinding" name="realtor">
    <wsdlsoap:address
location="http://trout.parasoft.com:8080/axis/servlet/AxisServlet/realtor"
  />
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

Inline Schema

## b) Inline schemas should be avoided when XML Validation is required.

Notice in the WSDL that the `<wsdl:types>` element has a `<schema>` child element. This is called an “inline schema” because the schema is found inside the WSDL. Most SOAP/WSDL toolkits generate WSDLs that have inline schemas. Although this is perfectly legal, it is preferable to have separate files for each schema and to import the schemas using the schema import mechanism. The main reason for this is that during XML validation of the SOAP request and the SOAP response, the XML parser will not parse the inline schema. Unless the XML parser is

specially configured to parse schemas inside a WSDL, it will not be able to validate elements that are defined in the inline schema.

### **c) When using inline schemas, avoid cyclical referencing.**

When it is not required to validate the SOAP request and the SOAP responses, it is okay to use inline schemas. However, care should be taken to make sure that the inline schemas do not reference each other. The following example illustrates how two inline schemas reference each other by importing from each other.

```
<types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://soaptest.parasoft.com/schemaMath">
    <xsd:import namespace="http://soaptest.parasoft.com/schemaPhysics">
    ...
  </xsd:schema>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://soaptest.parasoft.com/schemaPhysics">
    <xsd:import namespace="http://soaptest.parasoft.com/schemaMath">
    ...
  </xsd:schema>
</types>
```

Although the above example is not explicitly forbidden by WSDL or XML Schema specs, cyclical referencing presents a tough problem for most XML schema parsers. In most instances, one of the inline schemas has an unnecessary `<import>` element and that element should be removed.

### **d) XML Validity should be checked nightly**

Each night, the WSDL should be checked for XML validity. This insures that the WSDL conforms to all the specs and that WSDL parsers will not have trouble parsing the WSDL. This check should be incorporated into the nightly test process described earlier.

### **e) Interoperability should be checked nightly - Avoid "encoded" coding style**

The WSDL must also be checked for conformance to the WS-I Basic Profile 1.0 for interoperability. This check should be incorporated into the nightly test process as well.

Although the WSDL specification allows the use of the "Encoded" encoding style, for interoperability reasons, it is wise to stay away from "Encoded" style. In fact, WS-I Basic Profile 1.0 says that in order for Web services to be interoperable, a WSDL should use either document-literal or rpc-literal bindings.

### **f) Create a regression test on the WSDL and schemas to track undesired changes**

Just like the application logic code, the WSDL may undergo many iterations. Even after it is published, the WSDL may undergo changes. At each iteration, the WSDL must be tested for validity and interoperability conformance.

### **Available Parasoft Solution**

With Parasoft SOAPtest, all of the above can be done by merely supplying the URL to the WSDL and using SOAPtest's "WSDL Tests" feature. SOAPtest automatically generates all the test cases and creates a test suite that can be run nightly as part of the nightly test process.

## Stage Three: Server Deployment

### a) Deploy the Web service as early as possible on a staging server environment as part of the Nightly Deployment Process

Once we have enough application logic written, the application should be deployed as a Web service on a staging server environment during the nightly deployment process. This makes the latest version of the Web service available for testing. We can then start consuming the Web service under the various use conditions: normal, abnormal, and malicious use.

### b) Create Web Service "Unit" Tests that verify the functionality of the Web service

For each operation exposed as a Web service, we create test cases that verify the functionality. We call these "unit" tests because they verify each operation individually. Later on, we will test the operations together in what we call "scenario" testing.

These "unit" tests are not the traditional Java unit tests. Instead, these tests are performed using a "driver" client that can generate various SOAP requests, send it to an endpoint, and then process the SOAP responses. As seen in Figure 4, we used Parasoft SOAPtest to create these tests.

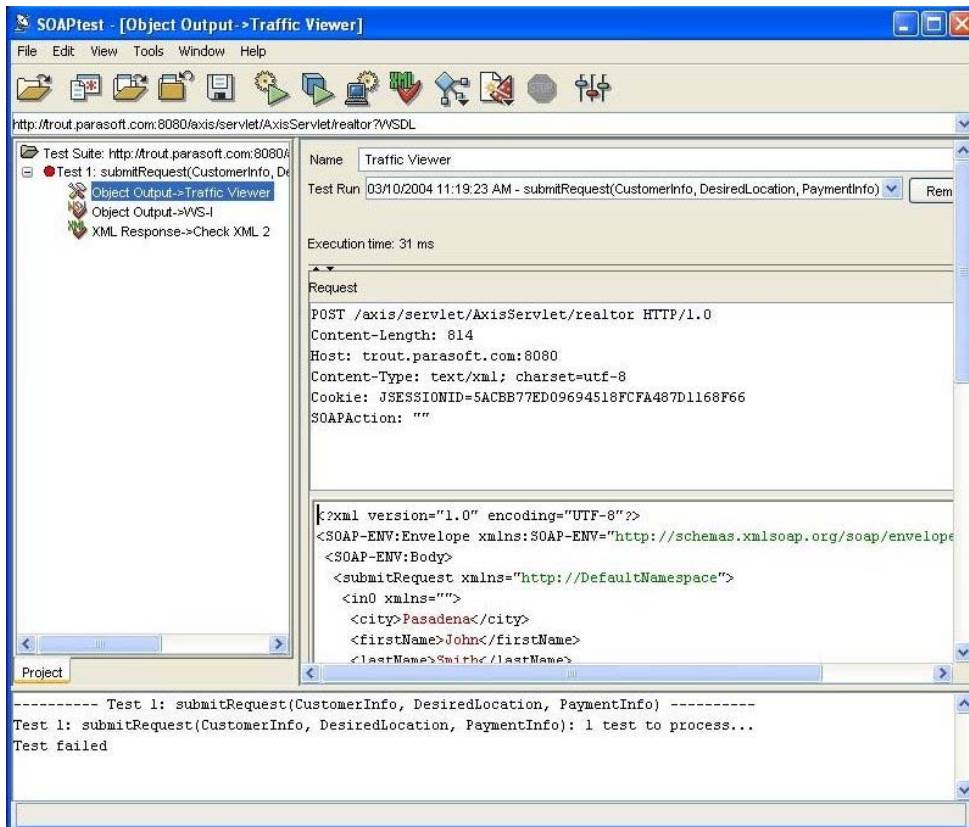


Figure 4: SOAPtest as a "driver" client

### **Positive test cases should return expected responses.**

Positive test cases should return expected values. For example, a potential customer in Pasadena may want to find a home in Pasadena. If the realtor has an office branch in Pasadena, the customer should receive a response with the location of this branch. In this instance, a

number of different positive test cases should be constructed to flush out any bugs in the Web service. The SOAP responses from the Web service should be captured and compared against the expected responses. These comparisons should be saved as regression tests. A positive test case is said to succeed when the comparisons succeed.

**Responses must be XML valid and pass interoperability checks.**

The captured responses should be verified by an XML validator. They should also be conformant to Basic Profile 1.0.

**Web service should gracefully handle negative test cases that simulate abnormal and malicious uses.**

Negative test cases can be divided into several categories. The first set of negative test cases contains transport level errors. The Web service should be able to handle various HTTP Header errors, such as wrong SOAPAction values (when applicable), wrong Content-Types, and wrong Content-Lengths. In each case, the Web service should not crash, but instead return appropriate HTTP level error messages or ignore the errors and process the XML request. These return values should be captured and compared against expected results. A negative test case is said to succeed when the comparison succeeds.

Another set of negative test cases contains XML level errors. These may involve sending invalid XML or XML with deliberately bad values. In such cases, the expected response is a SOAP fault with a reason for the fault.

**Use feedback from the test cases to modify/fix/upgrade all components of the Web service as well as the development process.**

By testing the submitRequest operation with SOAPtest, we were able to find a few problems with our Web service. As the errors show in Figure 5, there were basically two problems.

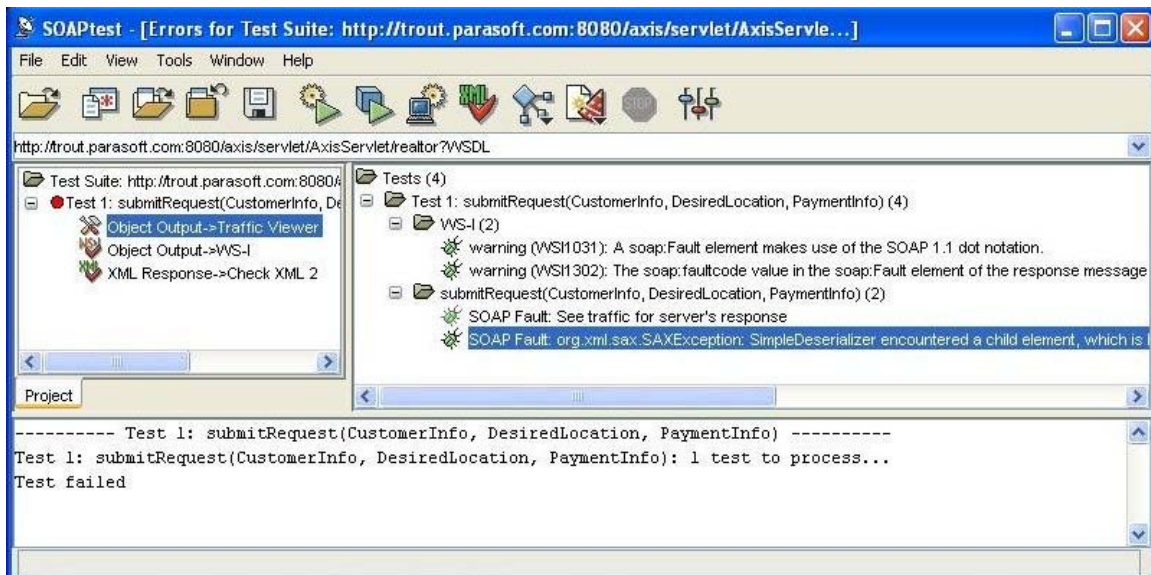


Figure 5: Test Errors found with SOAPtest

The first problem is that the server could not deserialize the XML request into the appropriate Java objects. As a result, the Web service returned a SOAP fault. This often happens when the Java object does not implement all the methods that a typical "bean" object should – that is, it does not have all the "get" and "set" methods for all of its fields. The obvious fix is to change the Java class that is responsible for this problem. However, a further step should be taken so that this problem can be avoided. This step is to add a coding standard that checks to make sure that the Java classes are properly implemented as a bean.

The second problem results directly from the first problem. The WS-I Analyzer complains about the soap:Fault element in the SOAP response. One reason for this is because the Web service was deployed on a server using Apache Axis version 1.1, which was released before WS-I Basic Profile 1.0 was completed. Because Web services standards are rapidly changing, it is important to use the latest stable versions of Web services toolkits from the vendors. As of this writing, Apache has released an alpha version of 1.2 that should conform more to WS-I Basic Profile 1.0.

***Both positive and negative test cases should be checked into source control and run as part of the Nightly Test Process.***

As the functionality of the Web service grows in complexity, having regression tests will insure that new functionality is added without breaking the existing functionality. It is very important to build up these test cases and continue to use them throughout the development cycle.

**c) Create "scenario-based" tests. Check them into source control and incorporate them into the Nightly Test Process.**

Finally, the Web service should be tested with various scenarios that simulate how real users will actually consume the Web service. A typical scenario would involve a call to the Web service by a new customer that submits a request followed by a call to the Web service by an agent who should then be able to retrieve the customer's information. Various scenarios should be composed and used to verify that the various operations of the Web service work together properly. Again, regression tests should be created from these scenarios and checked into source control.

**d) Make sure that the database is tested.**

This can be accomplished in several ways. One way is to add database queries to the positive test cases. For instance, a database query should be made before and after a customer request to verify that the customer contact info is correctly stored in the database.

Another way is through the "scenario-based" tests. A proper combination of agent requests and customer requests can be monitored to make sure that the Web service is correctly storing and retrieving from the database.

## **Stage Four: Client Development**

**a) Create client to WSDL specs**

A Web service client can be created from an existing client application or created from scratch. The client should be written to the WSDL interface, and not to the server implementation. This guards against the possibility of having client and server implementations that work great together, but are not inline with the WSDL.

## b) Use server stubs to test client functionality – deploy the server stub as part of the Nightly Deployment Process

Server stubs should be used to assist in creating and testing the client. Server stubs act as endpoints to which clients can send SOAP requests and from which clients can receive SOAP responses. It is important to not only test the client against the server that is being developed because when you encounter an error, you will not know whether the error is due to a bug in the server or due to a bug in the client. By using a server stub, you guarantee that the error is due to a bug in the client.

The Server Stubs should have the following properties. They should be able to return expected SOAP responses as well as intentionally bad SOAP responses. They should timeout at the HTTP level and also refuse connections.

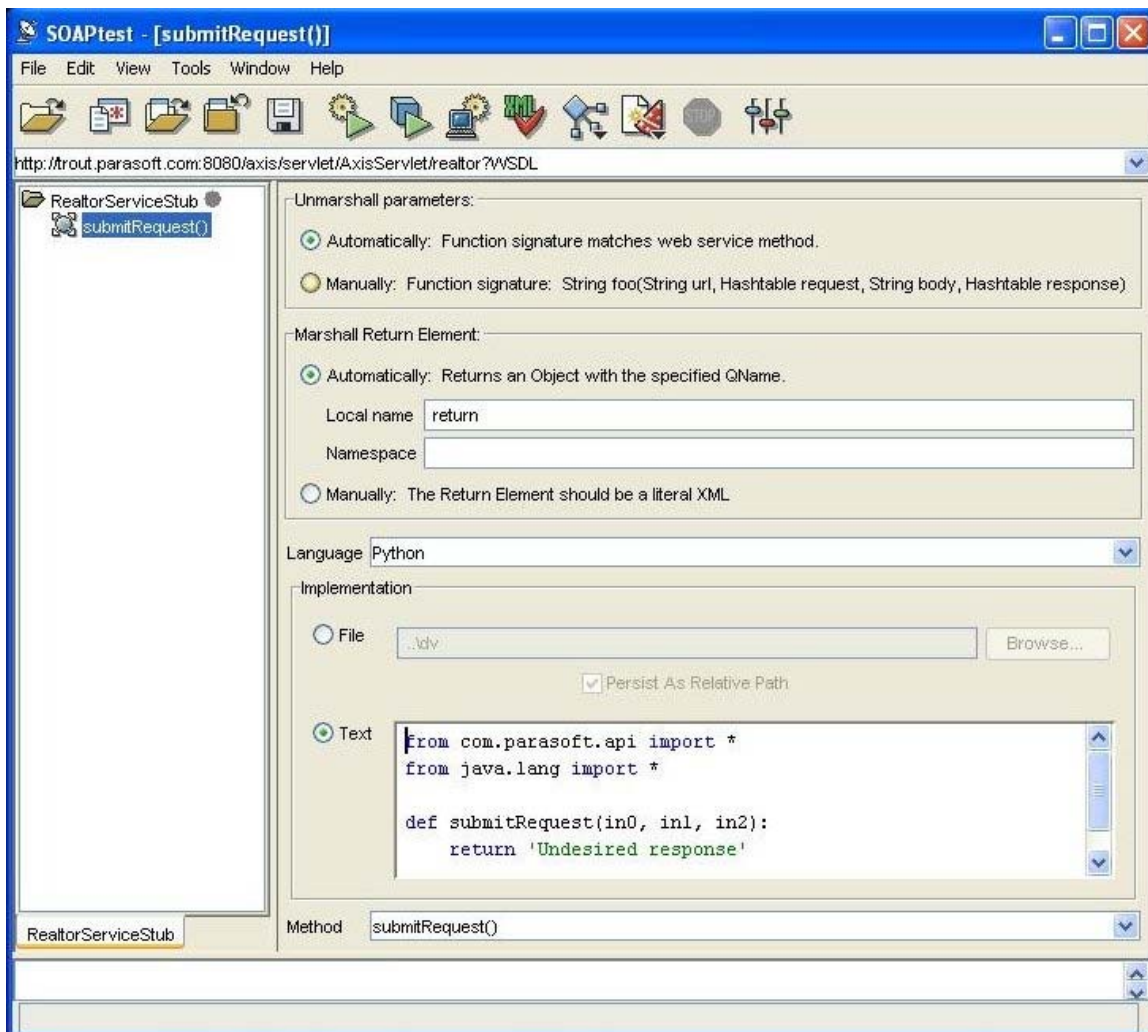


Figure 6: An example server stub using SOAPtest

The server stub shown in Figure 6 simply returns a string inside the soap:Body back to the client using a Python script. The script and the server stub can be modified to return various SOAP responses. In this way, the client can be tested against various SOAP responses. Additionally, the server stub can be configured to process the SOAP requests as needed.

### **c) Create client tests that can be incorporated into the Nightly Test Process**

The clients should be written in such a way that they can be invoked from a command-line script. Scripts can then be written to invoke the client against the Server Stub to test the client behavior when it receives expected SOAP responses as well as unexpected SOAP responses. Clients should also be tested to see how they handle connection timeouts and various errors that can happen at the transport level. Finally, the client should be tested for interoperability against the WS-I Basic Profile 1.0 specs. All of these tests should be checked into source control and run as part of the Nightly Test Process using the latest Client implementation built from the latest source code.

## **Stage Five: Adding Security**

### **a) Determine level of security needed**

Depending on the type of Web service you are creating, an appropriate level of security will need to be implemented. The gamut can range from firewalls, to simple transport level security such as basic authentication and SSL, to a variety of XML level security mechanisms such as SAML and XML Digital Signature.

For our example, we require that the operation for customer request not have any security layer. For the agent request operation, we require a username/password token with timestamp per WS-Security specs. We require that the password digest be sent (with Type="wsse:PasswordDigest") instead of the plain text version of the password. As long as the agent's username and actual password are not compromised, our Web service will be protected from unauthorized users. We also require that the timestamp be at most 5 minutes old to account for small differences in time between the client and the server. Beyond that, we do not expect much of a security threat.

### **b) Deploy security enabled Web service on another port on the staging server**

As part of the Nightly Deployment Process, a security enabled Web service should be deployed. This allows us to begin testing the security layer. Having both a security-enabled Web service and a non-security-enabled Web service running on the staging server helps us to pinpoint issues with the security layer. For instance, if the security-enabled Web service fails for a certain SOAP request, we can try the same SOAP request on the non-security-enabled Web service to see where the problem lies, whether it lies in the security layer or in the general XML processing layer.

Microsoft offers WSE (Web Service Extension) for .NET Platform that provides an implementation of the WS-Security specification. For the Java Platform, Apache has recently introduced WSS4J (Web Service Security for Java) library that provides WS-Security implementation. The WSS4J works well with the Apache Axis SOAP library.

### **c) Leverage tests already created – modify them to create positive and negative test cases – add them to source control and run them as part of the Nightly Test Process**

Instead of re-inventing the wheel, we use the functional and regression tests that were created in Stage Three to test the Web service with the security layer added. All we have to do is add extra steps of processing for security. This processing will be performed by the WS-Security implementations described above. In our case, only the SOAP Header should be affected.

First, the Web service should be tested with "positive security" test cases. For the agent request operation, this means that we test the Web service with a good pair of username/password tokens. All the functional and regression tests from Stage Three must pass. This insures that the

security layer on the server correctly processes the security information in the SOAP request without inadvertently modifying the SOAP request.

Next, we test the security layer itself with a variation of positive and negative test cases. The agent request operation should not return anything useful if a good pair of username/password token is not included in the SOAP Header. The following is a list of the negative conditions to be tested.

- bad pair of username/password token
- password not digested, i.e., Type="wsse:PasswordText"
- no password
- no username
- no SOAP Header
- old timestamp (older than 5 minutes)

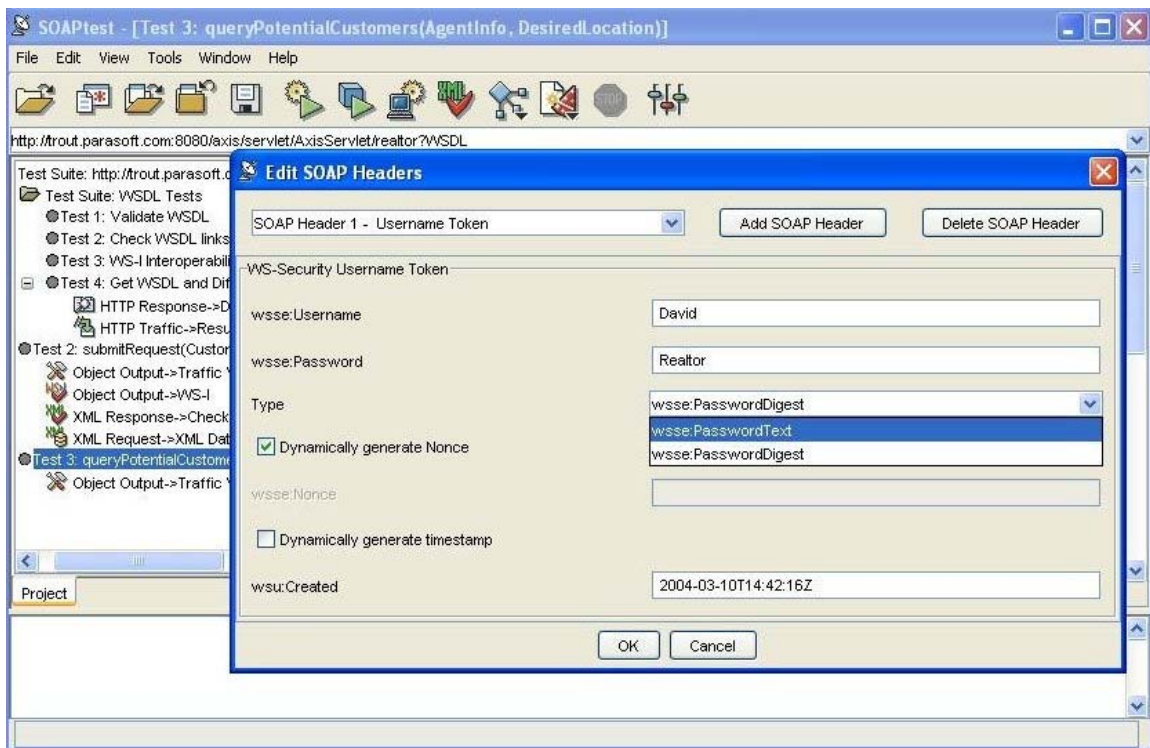


Figure 7: Creating Security Layer test cases with SOAPtest

## Stage Six: Performance/Load Testing

### a) Begin load testing as early as possible and incorporate it into the Nightly Test Process

Load testing is an important task that is frequently under-utilized when developing Web services. Often, it is left until the end to make sure that the Web service continues to work even under expected load. However, Load testing can yield more benefit if it is done as a continuous process and the results are collected over a period of time. When results are collected in this way and compared against a baseline, undesired performance drops can be detected and eliminated as soon as they are introduced. Before adding the security layer, load testing should be done and the results saved as a baseline. After adding the security layer, performance testing should be done and the results should be compared to the baseline to see how much the security layer degrades the performance. If the performance degradation is too much, the security layer may need to be modified.

### b) Use load test to determine final deployment configuration to choose from vendors

When choosing what the final production environment will consist of, such as what application server to use and what database to use, Load Test each configuration to see what configuration yields the best performance. By collecting the results and comparing them with each other, an informed decision can be made about what vendor to go with.

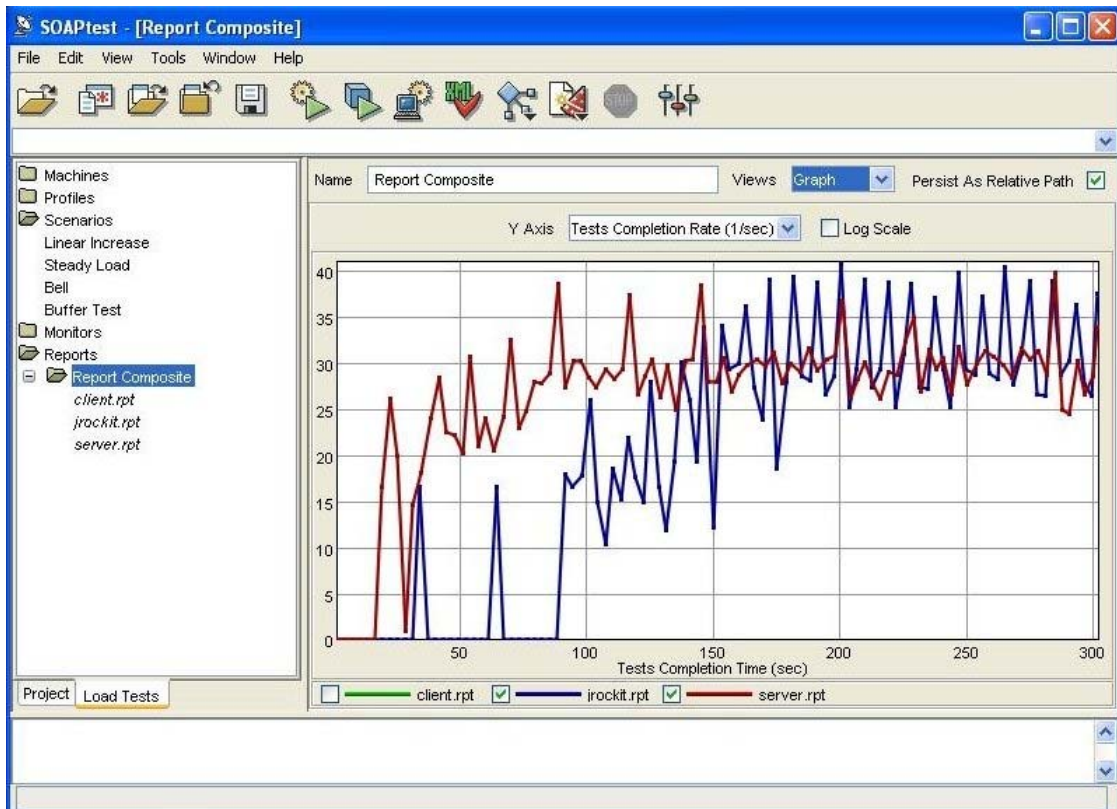


Figure 8: Performing Load Testing

The graph in Figure 8 shows a comparison of two load test runs on a Web service deployed on BEA's WebLogic Application server. The first run was conducted with the server using the default JVM (Java Virtual Machine). The second run was conducted with the server using BEA's special JRockit JVM. By performing load testing with two different configurations on the server, we were able to figure out which configuration produces the best performance.

Although the JRockit JVM configuration has a lower test completion rate for the first 150 seconds, the graph shows that it has a slightly higher rate after the initial startup time. As long as we plan to keep the server up continuously, the JRockit configuration provides better performance. However, additional parameters should be compared before a final decision is made.

## ***Conclusion***

As seen throughout this paper, there are countless opportunities for things to go wrong during Web services development—a slight mistake in any component or interface will cause problems that ripple throughout the system. Developers must ensure that each part of the system is reliable, and that all of these parts interact flawlessly and securely.

By providing a feasible way to implement error prevention throughout the Web service development cycle, the Parasoft AEP Methodology helps you to eliminate the errors that lie at the root of virtually every concern you may be struggling with— from Web service quality issues, to security weaknesses, to the need for more (or more rapid) strategic improvements. For the example used in this paper, we took this systematic approach to creating a Web service. At each step, we defined requirements and froze them once they were met. We were able to do this through creation of verification tests that were incorporated into source control and the nightly build process. By continuing this throughout the entire development cycle and following the AEP Methodology, we were able to confidently say that we created a bulletproof Web service.

## ***Contacting Parasoft***

### **USA**

101 E. Huntington Drive, 2nd Floor  
Monrovia, CA 91016  
Toll Free: (888) 305-0041  
Tel: (626) 305-0041  
Fax: (626) 305-3036  
Email: [info@parasoft.com](mailto:info@parasoft.com)  
URL: [www.parasoft.com](http://www.parasoft.com)

### **Europe**

France: Tel: +33 (1) 64 89 26 00  
UK: Tel: +44 (020) 8263 2827  
Germany: Tel: +49 7805 956 960  
Email: [info-europe@parasoft.com](mailto:info-europe@parasoft.com)

### **Asia**

Tel: +886 2 6636-8090  
Email: [info-psa@parasoft.com](mailto:info-psa@parasoft.com)