Web service simulator framework solution using Spring

Anthony Allegri
Sri Ramanathan
Mark Sehorne
Matthew Terry
Matthew B. Trevathan

Large Web development initiatives generally comprise multiple development projects executed simultaneously by geographically distributed teams who work around the clock. Specific components developed by these teams are expected to interoperate as seamlessly as possible. One key requirement for such development is the definition of the contracts between the teams charted with building the different components. A good simulator framework provides the best possible team isolation and potentially accelerates productivity by supporting the full suite of requests and responses supported by each interface contract. This article describes using a simulator framework in fast-paced Web development environments, and shows, step by step, how to create one in short order using Rational® Application Developer, along with examples and sample code.

Introduction

Complex development projects often require that development streams progress as independently from one another as possible. Assuming that the interfaces (contracts between these different subsystems) have been finalized as early as possible in the software development life cycle (SDLC), a good simulator-based development strategy can yield significant optimization during development.

This is particularly germane during the deployment of new applications and systems into existing environments, such as in the telecommunications domain, where a large number of Independent Software Vendors (ISVs) are in the mix, and where non-functional requirements are of critical importance, especially to manage systems latency and throughput considerations. In these types of complex environments, there are a number of factors that complicate the actual delivery of the new application into the existing environment, some of which are listed below:

1. The need to develop and deliver functions without the existence of all systems and service implementations that the new application depends on.
2. The need to parallelize development activities associated with different streams to the maximum extent possible, while minimizing the cross-stream dependencies
3. The need to deliver functions in such a way that the new application is able to prove it meets throughput and latency requirements end to end
4. The need to insulate development teams from changes that occur to subsystems that they depend on during the code and unit test phase of the SDLC
5. The need to easily provision various test data scenarios in a repeatable, reliable, and stable process

In all of these areas, a robust simulator framework provides a tremendous amount of value. This article describes such a simulator framework, and explains how to construct one using WebSphere® Application Server. While the focus here is on an application using a Service Oriented Architecture (SOA), it is possible to extend this to other types of environments and architectures as well.

Simulators as a best practice

Simulator overview

Webster's Dictionary defines a simulator as "a device that enables the operator to reproduce or represent under test conditions phenomena likely to occur in actual performance." Simulators can be utilized as part of the SDLC to enable a more agile development environment by providing an endpoint for back-end service calls that appear real to the calling application, but in actuality are only mimicking the real system. Simulators respond in the same format as the live back-end services and use the same request/response contract, but simply return preset data from configurable data. This lets you model functional and non-functional characteristics of a supporting system, even if you do not have access to the system itself.

A similar and more common programming practice is to utilize stub interfaces that bypass the local client interface call, and instead exercise a different local code path to return preset data. In many cases, stubs speed up the initial development, but stubs do not provide the same level of functionality that you can get by exercising the full code path and calling a working interface. With the stub method, the application has to be aware it is in "stub mode" so it can exercise the different code path. By using simulators, the application only changes endpoints and is not aware that the responses are from simulated back-end systems.

Figures 1 and Figure 2 show the differences between the two approaches.
In most cases, applications talk to various Web services, and your initial thought may be to utilize the stub approach when a live interface is not available. This may result in quickly producing the desired visual flow on the user interface, but stubbing at the client does not test the interconnectivity or the marshaling of data. Therefore, the further back you can simulate, the greater the benefit because it exercises more of the system. Figure 2 illustrates the optimal point to create a simulator.

To further illustrate this concept, take a look at a Web 2.0 example that includes an Asynchronous JavaScript and XML (Ajax) request from the client and a JavaScript Object Notation (JSON) response from the server that depends on a back-end system to build the response.

The first contract that is established is the JSON object between the client and the application. The second contract established is between the application and the back-end server using Web Service Definition Language (WSDL). During initial development, the JSON contract and the Web Service contract represent common points where you can create a stubbed interface, letting you develop independent of the developer on the other side by maintaining this contract.
In theory, when each group completes their development according to the established contracts, the application can be integrated with little or no problems. In reality, however, this is often not the case when integrating to multiple external systems. It is common during integration for interfaces to return unexpected results or to have a sequence of dependent calls from one service to another that requires a level of orchestration. A single broken or unavailable interface can block multiple development teams downstream.

Benefits of using simulators for development

When developing complex systems with multiple development groups that require numerous dependencies, the risk probability of interface delays reaches a level of near certainty. Service simulation is a critical mitigation tool that can be proactively included during the development of complex systems to significantly reduce risk and delays.

There are key factors to consider when determining the level of simulation. In many cases, the simplest level of simulation is to return the same response regardless of the input. This is useful to complete initial unit testing, but rarely sufficient to ensure quality code. Typically, the simulator is extended further by returning more complex responses based on the request data. In some cases, the request data is used to look up the response in an XML file. In other cases, the request data is reused or manipulated in a calculation for use in the response.

To enable these more complex responses, a simulator pattern needs to be established that is simple, flexible, and lightweight so that it can easily be adopted by developers, but also provide a thorough level of simulation.

Figure 4. Data available to the simulator will affect its usefulness to the development team and its overall success

Benefits of using simulators for functional testing

The benefits of simulation in the SLDC extends beyond the development phase. Newer methodologies, like scrum, extreme programming, and feature-driven development promote iterative development, but may clash with conventional waterfall-driven methodologies where the sequential gates need to be met to move to the next phase. Iterative testing is difficult to achieve if quickly developed components are dependent on key interfaces with longer development cycles. In these situations, simulators can be used to test components before the live interface is available and will typically identify the majority of issues early. It is also common to find challenges when attempting to provision test data in back-end systems to support all possible error scenarios. Simulators provide an efficient alternative for testing rare scenarios or negative test cases that are difficult to produce using live back-end systems.
Simulators can also provide a great risk mitigation strategy. As systems interfaces count and complexity increase, the risks of outage and schedule slip also increase. Situations always arise where an interface is unavailable because it needs a patch to resolve an issue, or developers must schedule downtime to work with an interface within an active testing window. In these cases, the developers and test team can lose precious time waiting for a fully integrated and functional system. By including simulators in the test environment, the application can avoid the outage by reconfiguring the endpoint for a particular interface from the unavailable service to the working simulated service. The test system can continue to utilize live interfaces for all working systems and only point to the simulator for services that are unavailable.

A service dashboard is typically developed along with the simulators to enable developers and testers to test the requests and responses of the simulated services. This enables quick verification of response data configuration. Because simulators mimic the live back-end services, the service dashboard can also be used to test and troubleshoot the live back-end services when available. In some cases, the service dashboard will reveal a unique response scenario that is not available in the simulator. The service dashboard can include the ability to log the raw request and response that enables the enhancement of the simulator to include this scenario in its simulation set.

Figure 5 illustrates a logical view of a system supported by a combination of live back-end services and simulators. It shows an external application that is connecting to various services through the Enterprise Service Bus (ESB). The yellow services (Services 1, 2, and 7) indicate services that are not available at the moment and are provided by the simulator. The red services (Services 3 and 4) represent services that may be under development or otherwise unavailable. The service requests to these red services are rerouted by the ESB to the simulated service.

**Figure 5. Logical view of a system utilizing a combination of simulated services and true back-end services**
Benefits of a simulator for non-functional testing

Performance testing is an important aspect of the SDLC. In many cases, the environments created for testing do not mimic the performance characteristics of the production environment. This is often the case when companies have costly systems that are too expensive to fully duplicate in a test environment. Simulators can be an important part of performance testing in these situations because they can simulate the expected production environment.

Understanding why simulators can be a valuable tool for performance testing requires some understanding of load. Little's Law of Load states that average load (L) is equal to the arrival rate (λ) multiplied by the average time (T) it takes for a transaction to complete. So, L = λT.

If an application takes 1 second to respond to a request, then the number of requests per second must be 1 for the load to be in equilibrium. Figure 6 demonstrates what happens when the arrival rate is increased while the duration is constant. Most front-end load testing tools emphasize increasing the arrival rate to produce load.

![Figure 6. Increasing requests per second with constant response time](image)

The inverse is also true, and demonstrated when the arrival rate does not change at the register, but the time it takes for the application to respond decreases. Figure 7 shows the effect when the response rate decreases with a constant request rate.

![Figure 7. Constant requests per second with decreasing response time](image)

Both Figure 6 and Figure 7 are common in a complex system. It is important to note that both the request rate and the response time can have significant effects on the load.

Load testing often emphasizes increasing the load by increasing the arrival rate (λ) of clients accessing the application. This is commonly accomplished through load generation tools like...
Mercury Load Runner or IBM Rational Performance Tester, but it does not address the average transaction time (T) that a back-end system is expected to return a result. As a recommended best practice, simulators can be combined with a performance model to create performance tests that better resemble a production environment by determining average responses for each interface using a performance model, and programming these response times into the simulators.

A performance model document can be used to document the Service Level Agreements (SLA) for each production system, based on existing or expected average response times in production. This can include response times for different levels of request rates. Typically, the response time per request will decrease as the transactions per second (TPS) increase. These varying response times can be included in the simulators to better model the expected production environment.

Simulators can also enable negative performance test scenarios by simulating a temporary "hiccup" or pause in the service or by simulating responses outside the expected SLAs so the effect on the dependent systems can be measured. This additional level of testing allows issues to be uncovered and addressed earlier in the SDLC.

**Technical implementation overview**

The following information provides a concrete implementation of a simulator framework. It leverages the Spring 2.0 Framework to transform response data stored in an XML format into Plain Old Java Objects (POJO) representing the associated service responses. The framework supports the use of both a simulator and a dashboard for unit testing services. The simulator framework consists of the following defined layers:

**Web services**

The underlying framework of the simulator solution consists of Web clients and Web services. Both the Web clients that run inside the main application and the receiving Web service for the simulator are generated off a provided WSDL. The WSDL enables the automatic creation of the request and response objects necessary to test the business functions.

**Dashboard layer**

The dashboard layer contains the user interface components used to gather the data that the client service layer uses to generate the request objects. Along with the gathering of information, the dashboard will likewise display the results associated with the invoked simulator service. The dashboard can also be used as a health check tool to verify connectivity and results when testing the actual Web services.

**Client service layer**

The client service layer defines the implementation classes responsible for taking the data passed from the dashboard, creating the request objects, and invoking a specified simulator service.

**Simulator service layer**

The simulator service is responsible for generating the response object that will be returned to the calling client service. This layer extracts key values provided in the request that will be passed to
the process layer responsible for retrieving the associated test data XML files. It is also within this layer that any business logic necessary to generate the correct response can be applied prior to returning the response.

**Processing layer**

The processing layer will use the key provided by the simulator service to identify the Spring XML file to retrieve. After the file is retrieved, the Spring BeanFactory consumes the XML file and transforms the data into the defined POJO that will be returned to the simulator service layer. Another facet of the process layer is the introduction of a processing delay that can emulate different types of transaction latency.

**Data layer**

The data layer is a repository of the XML files that represent the Web service response objects. This XML defines a simple and consistent way of creating JavaBean objects that can be managed by the Spring BeanFactory that is compliant with the spring-beans-2.0.dtd. Retrieval of the XML file is handled by the Spring File System Resource and transformed into the defined POJO using the Spring XmlBeanFactory class.

**WebSphere resource provider**

The WebSphere resource provider will be used as a storage solution for maintaining dynamic configuration values used within the simulator solution. Values can be added, modified, and deployed using the Admin Console, thus providing easy maintenance for a stand-alone or clustered environment.

**Figure 8. Simulator framework diagram**

**Technical implementation steps**

The following steps detail the process of generating a sample simulator application:

1. Generate Web service artifacts from the provided WSDL.
2. Create the dashboard UI component to capture inputs.
3. Map the service implementation, methods, and input parameters in the RPC configuration.
4. Create client implementation to gather and transfer the request object to the simulator service.
5. Create the simulator service that will apply the business rules to the generated response object.
6. Create Spring XML data files that will be transformed into the response objects.
7. Configure WebSphere Environment Provider values to support the application.

Web clients and services

Web services are the foundation for the simulator framework. Previously defined WSDLs are used to generate the Web service artifacts that will be built upon to provide the functions needed to exercise the simulator business cases. When you have implemented this with clients, you commonly establish a baseline WSDL and create a Web service client based on the WSDL. The following steps can be used to guide you through the Web service generation process using an example WSDL.

Use the Rational Application Developer (RAD) platform to quickly generate the Web services and clients through the wizard tool.

Web service generation

1. Within RAD, create a new Web Project that will contain the Web services and simulator support artifacts.

   Figure 9. Create dynamic Web project

   ![Dynamic Web Project](insert_image)

2. Add a "wsdl" folder under the project's "src" tree. Add the WSDLs that represent the Web services to simulate. Add the associated namespace-to-package properties files provided with the WSDL that will be used to generate the services and clients.
3. Generate the Web Service by right-clicking on the WSDL and selecting **Web Services > Generate Java bean skeleton**. This will open the Web Service Wizard.

4. Verify the Web Service wizard settings. The defaults are typically used.
5. Select the option to map the namespace to packages, which provides a more user-friendly package structure in the generated objects.

**Figure 12. Validate Server, Service Project, and EAR**

6. Import the namespace and packages by clicking **Import** before clicking **Finish**.

**Figure 13. Verify source folder and select custom mappings**

**Figure 14. Import namespace to package definition**
After you click **Finish**, the objects required for the client and service side of the Web service are generated. These new objects will appear in the project created under the package name selected.

**Simulator dashboard**

The simulator dashboard provides the UI components used to test the simulator services. For each Web service to be tested, a JSP will be created to capture all required data necessary to generate the SOAP request object within the client service layer.

The dashboard layer leverages several tools that provide fast out-of-the-box functions that aid in the gathering, retrieval, and displaying of data within the simulator solution. You'll combine two elements to form the dashboard:

- **Ajax**— Ajax components are used to gather information, communicate to service layer, and present the response data.
- **IBM Web 2.0 Feature Pack – RPC Adapter Servlet**— The RPC Adapter Servlet maps the Ajax request to the desired service implementation and method managing the Http request as well as the data contained within. The resulting bean is converted by the RPC Adapter Servlet to a JSON object that can then be rendered in the JSP using Ajax. (See [Related topics](#) for the download.)

**Figure 15. The relationship between the RPC Adapter and the JSP to Web client callflow**

---

**Simulator-specific JSP**

The JSP will define the inputs needed by the client service implementation to compose the request object passed to the simulator. Along with capturing the necessary input values, the JSP will use a **callback** function to interact with the specified implementation defined in the RPC Adapter configuration. The data returned by the simulator will be converted to a JSON object using the RPC Adapter servlet where it will be displayed.

The JSP includes the following:

1. Retrieves the default simulator endpoint from the WebSphere Resource Provider. (The Resource Provider configuration is discussed in [WebSphere Resource Environment Provider](#).)
2. Define all HTML input fields needed by the client service to generate the SOAP request.
3. Create the Ajax callback function mapped to the desired process service implementation and method.

```html
<% String acctEndPoint=(String) EnvironmentProviderUtility.getWebServiceEnvironmentAttribute("AccountSearch.Endpoint"); %>
```

```java
getRpcDataAccountSearch ('AccountSearchImpl','getAccountsByEquipment', $('accountEndPoint').value, $('equipId').value,'getAccountsByEquipment')
```

**Figure 16. Simple dashboard JSP**

![Simple dashboard JSP](image)

**RPC Adapter servlet configuration**

The RPC Adapter Servlet must be configured in the simulator project using the following steps along with the files provided in the IBM Web 2.0 Feature Pack.

1. **IBM Web 2.0 Feature Pack – RPC Adapter Servlet**— Register the IBM WEB 2.0 Feature Pack – RPC Adapter Servlet in the simulator project by editing the web.xml file and adding the RPX Adapter Servlet definition. Listing 1 shows the registration of the IBM RPC Adapter servlet in the web.xml file.

   **Listing 1. Add RPC Adapter Servlet definition to web.xml**

   ```xml
   <servlet>
     <description> IBM WEB 2.0 Feature Pack - RPC Adapter </description>
     <display-name>RPCAdapter</display-name>
     <servlet-name>RPCAdapter</servlet-name>
     <servlet-class>com.ibm.websphere.rpcadapter.RPCAdapter</servlet-class>
   </servlet>
   <servlet-mapping>
     <servlet-name>RPCAdapter</servlet-name>
     <url-pattern>/RPCAdapter/*</url-pattern>
   </servlet-mapping>
   ```

2. Add the RpcAdapterConfig.xml file to the WEB-INF folder. The RpcAdapterConfig.xml file is provided within the IBM Web 2.0 Feature Pack.
Figure 17. Add RpcAdapterConfig.xml to WEB-INF folder


It is through this configuration that the RPCAdapter knows which implementation to invoke from the calling JSP. In Listing 2, the <name> value under <pojo> registers the implementation to be referenced within the JSP; the <name> value under <method> registers the implementation method to be used by the JSP; and the <name> values under <parameter> register the input values to be passed onto the implementation method.

Listing 2. Define all pojos, implementations, methods, and parameters
<services>
  <pojo>
    <name>AccountSearchImpl</name>
    <description>com.simulator.service.AccountSearchImpl</description>
    <methods filter="whitelisting">
      <method>
        <name>getAccountsByEquipment</name>
        <description>web service method to retrieve account information by equipment id</description>
        <parameters>
          <parameter>
            <name>request</name>
            <description>end point for account search simulator service</description>
          </parameter>
          <parameter>
            <name>equipId</name>
            <description>the equipment id used as key for retrieving simulator data</description>
          </parameter>
        </parameters>
      </method>
    </methods>
  </pojo>
</services>

Client service layer (simulator)
The simulator client service layer consists of the Web service clients and implementation classes needed to interact with the simulator Web services. The implementation class builds a simulator-specific SOAP request from the data gathered within the associated JSP and forwards the information to the mapped simulator Web service determined by the endpoint retrieved as a
mapped WebSphere Environment Provider value. The response object will be converted to a JSON object by the RPC Adapter servlet and presented back to the user in the calling JSP.

In Listing 3, `equipment.setSim(equipId)` populates the request object from values entered into the JSP. The `endPoint` values are defined as WebSphere Environment Provider values.

**Listing 3. Example request object passed to simulator service**

```java
public ESIGetAccountsByEquipmentResponseType getAccountByEquipment (HttpServletRequest request, String endPoint, String equipId) {
    String method = "getAccountsByEquipment";
    ESIGetAccountsByEquipmentResponseType rc = new ESIGetAccountsByEquipmentResponseType();
    ESIGetAccountsByEquipmentResponseType req = new ESIGetAccountsByEquipmentRequestType();
    Equipment equipment = new Equipment();
    equipment.setSim(equipId);
    req.setEquipment(equipment);
    try {
        InitialContext jndiContext = new InitialContext();
        AccountSearch service = (AccountSearch) jndiContext
            .lookup("java:comp/env/service/AccountSearch");
        AccountSearchPort port = (AccountSearchPort)service.getPort(AccountSearchPort.class);
        ((Stub)port)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, endPoint);
        rc = port.getAccountsByEquipment(req);
    } catch (java.lang.Exception e) {
        String[] variables = new String[] { e.getLocalizedMessage() };
        java.text.MessageFormat form = new java.text.MessageFormat("Remote system error occurred, message is 
{0}\);"
        String formattedText = form.format(variables);
        logger.logp(WsLevel.SEVERE, CLASS_NAME, method, formattedText, e);
        return rc;
    }
    logger.exiting(CLASS_NAME, method, rc);
    return rc;
}
```

**Simulator service layer**

Within the simulator services layer, a "transaction" key is extracted from the request object, and will be used to determine the specific XML file to retrieve, consume, and return as a POJO response.

In Listing 4, the Equipment ID is extracted from the request object and passed as the XML retrieval key to the process layer. It is also within this layer that any business logic necessary to validate the use cases can be applied prior to returning the response.

Also in Listing 4, the first `try` statement passes a key from the request that will be used by the generator to locate the XML file that will be transformed into a response POJO.

**Listing 4. Build key from request object used by data generator layer to locate the Spring XML file**

```java
public ESIGetAccountsByEquipmentResponseType getAccountsByEquipment(ESIGetAccountsByEquipmentRequestType getAccountsByEquipmentInput) throws java.rmi.RemoteException {
```


String methodName = "getAccountsByEquipment";

// return object
ESIGetAccountsByEquipmentResponseType rc = new ESIGetAccountsByEquipmentResponseType();

// request value used for simulator xml retrieval key
String sim = getAccountsByEquipmentInput.getEquipment().getSim();

if (log.isLoggable(WsLevel.FINER)) {
    log.entering(className, methodName);
}

try {
    rc = (ESIGetAccountsByEquipmentResponseType) com.simulator.DataGenerator.getDataByBeanId(
        "ESIGetAccountByEquipmentResponseType",
        ACTION_KEY_EQUIPMENT, sim);
    return rc;
} catch (org.springframework.beans.factory.BeanDefinitionStoreException ioe) {
    // return default equipment response
    rc = (ESIGetAccountsByEquipmentResponseType) com.simulator.DataGenerator.getDataByDefault("ESIGetAccountsByEquipmentResponseType",
        ACTION_KEY_EQUIPMENT, DEFAULT_EQUIPMENT_CODE);
    log.logp(WsLevel.SEVERE, className, methodName, ioe.getResourceDescription());
    return rc;
} catch (Exception e) {
    rc.setStatus(getCommonStatusType(e));
    return rc;
} finally {
    if (log.isLoggable(WsLevel.FINER)) {
        log.exiting(className, methodName);
    }
}

### Processing layer

The processing layer takes the inbound key that will be used to identify which Spring XML file to retrieve. After the file is retrieved, the SpringBeanFactory consumes the XML file and transforms the data into the defined POJO. Spring will throw a wrapped IO Exception in the event the requested XML file cannot be located. In this event, a default XML file can be defined and returned, a useful process when a "default" behavior is defined in a scenario. Another facet is the introduction of a processing delay to emulate transaction latency. In this case, the constant latency value is stored as a WebSphere Environment Provider value.

In Listing 5, the `bf.getBean(clazzName)` retrieves the data XML based on the key provided from the simulator service. Notice there is also a latency injection check, with values defined by the WebSphere Envirnoment Provider.

### Listing 5. Example retrieval of data XML and latency injection

```java
// Method used for retrieving spring xml
public static Object getDataByBeanId(String clazzName, String action, String key) throws Exception {
    String dataFile = null;
    Object response = null;
```
// check if service is "enabled"
if ((ACCOUNT_SEARCH_ENABLED_KEY != null) && (ACCOUNT_SEARCH_ENABLED_KEY != "")
&& (ACCOUNT_SEARCH_ENABLED_KEY.equalsIgnoreCase("FALSE"))) {
    throw new Exception("service is disabled");
}
dataFile = getFile(action, key);
com.simulator.SpringContext stsc = new com.simulator.SpringContext(dataFile);
org.springframework.beans.factory.BeanFactory bf = stsc.getBeanFactory();

// retrieve pojo
response = bf.getBean(clazzName);

// check if latency has been injected
if ((ACCOUNT_SEARCH_LATENCY_KEY != NULL) && (ACCOUNT_SEARCH_LATENCY_KEY.length() != 0)) {
    sleep(ACCOUNT_SEARCH_LATENCY_KEY);
}
return response;

Data layer

The data layer consists of defined XML files that will be consumed by the Spring framework and returned as POJOs to the simulator service layer. The POJO in Listing 6 is an example of a typical JavaBean that can be used to connect the data layer to the XML within the Spring framework. The POJO leverages simple getters and setters that bind the XML to the Java bean using the framework.

Listing 6. Example of a simple JavaBean definition

* AccountSummaryType.java*

```java
package com.simulator.eo.account.accountsearch;

public class AccountSummaryType {
    private java.lang.String ssn;

    publicAccountSummaryType() {
    }

    public java.lang.String getSsn() {
        return ssn;
    }

    public void setSsn(java.lang.String ssn) {
        this.ssn = ssn;
    }
}
```

Listing 7 provides an example Spring XML representing the JavaBean shown in Listing 6.
Listing 7. Example Spring XML representing JavaBean

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <bean name="ESIGetAccountsByEquipmentResponseType" class="ESIGetAccountsByEquipmentResponseType">
    <property name="accountSummary">
      <bean class="com.simulator.eo.account.accountsearch.AccountSummaryType">
        <property name="ssn" value="ssn"></property>
      </bean>
    </property>
  </bean>
</beans>
```

WebSphere Application Server configuration

WebSphere Application Server configuration changes are required in support of the simulator framework. The simulator leverages WebSphere's Resource Environment Provider to store application values used within the simulator application. The advantage of the Resource Environment Provider is that it allows environment-specific variables to be stored within the Application server, not in a properties file. The following example demonstrates the configuration of the account search service properties for the simulator. The steps define the simulator resource environment provider and its property values. You can read more about this in "Using resource environment providers in WebSphere Application Server" (see Related topics for a link).

WebSphere Resource Environment Provider

To set up the WebSphere Resource Environment Provider, perform the following steps:

1. Log in to the WebSphere Admin console and navigate using the left navigation to Resources > Resource environment providers.
2. Click New and enter the name for the Environment Provider (that is, Simulator Environment Provider).

Figure 18. Define Simulator Environment Provider

Use the following steps to set up the WebSphere Resource Environment Provider – Referenceables:
1. Click on the name of the new Environment Provider to Navigate to Resources > Resource environment providers > Simulator Environment Provider > Referenceables.
2. Click **New** to enter a new Factory and Class Name. Enter the values:
   - com.simulator.EnvironmentProviderFactory as the factory class name
   - com.simulator.EnvironmentProvider as the class name

   **Figure 19. Define factory and class within Environment Provider Referenceables**

Use the following steps to set up the WebSphere Resource Environment Provider - Resource env entries:

1. Click the breadcrumb entry for the Simulator Environment Provider and navigate to Resources > Resource environment providers > Simulator Environment Provider > Resource env entries.
2. Click **New** and enter the following information.
   - Simulator Configuration as the name
   - rep/SimulatorConfig as the JNDI name
   - Simulator configuration properties as the description

   **Figure 20. Define resource environment entries**

Use the following steps to configure WebSphere Resource Environment Provider – Custom Properties:
1. Click the title of the newly created Resource env entry "Simulator Configuration" and navigate to Resources > Resource environment providers > Simulator Environment Provider > Resource env entries > Simulator Configuration >Custom properties.
2. Click New and enter the information as shown in Table 1. Your results will look similar to the window in Figure 21.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccountSearch.Endpoint</td>
<td><a href="http://localhost:10000/Simulator/services/AccountSearch">http://localhost:10000/Simulator/services/AccountSearch</a></td>
<td>simulator endpoint</td>
<td>false</td>
</tr>
<tr>
<td>Simulator.Directory</td>
<td>C:\simulator\xml\</td>
<td>simulator data directory</td>
<td>false</td>
</tr>
<tr>
<td>AccountSearch.Latency</td>
<td>500</td>
<td>latency value</td>
<td>false</td>
</tr>
<tr>
<td>AccountSearch.Folder</td>
<td>Account\</td>
<td>data folder</td>
<td>false</td>
</tr>
<tr>
<td>AccountSearch.Prefix</td>
<td>ACT</td>
<td>file prefix</td>
<td>false</td>
</tr>
<tr>
<td>AccountSearch.Enabled</td>
<td>true</td>
<td>enabled status</td>
<td>false</td>
</tr>
</tbody>
</table>

Figure 21. Define custom properties used by the application


Figure 29. Resource Environment Reference within web.xml
These configuration values are now retrievable from the code in the Java objects or the JSP. This is used instead of property files in this example because the values can be managed through the admin console and synchronized across the cluster from the console.

Conclusion

Simulators provide a convenient way to develop and test against systems that are not always available or are unable to support the full testing requirements of the system. The simulator framework provides a mechanism for quickly adopting the use of the simulator best practice, including the use of the service dashboard to monitor simulated and live back-end services. Simulators are an obvious choice during development when live services are unavailable or unreliable, but are also a valuable testing mechanism for functional and performance testing, including the ability to model SLAs using latency injection. The simulator framework may appear to be too complex and time consuming at first, but it has been proven over numerous project life cycles that this investment during development significantly reduces overall project effort and risk while increasing the quality of unit, functional, and performance testing.
Related topics

- See other perspectives on testing in application development in the article "Traceability from Use Cases to Test Cases" (developerWorks, February 2006).
- Read more about Little's Law of Load.
- Test drive IBM Rational Performance Tester.
- Find ways to speed your development with Rational Application Developer.
- Learn more about Ajax in the Ajax Resource Center on developerWorks.
- Download the IBM Web 2.0 Feature Pack, including the RPC Adapter Servlet.
- Read the Spring - Java/J2EE Application Framework Reference Documentation.
- Read more about resource environmental providers in the article "Using resource environment providers in WebSphere Application Server" (developerWorks, November 2006).

© Copyright IBM Corporation 2010
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)