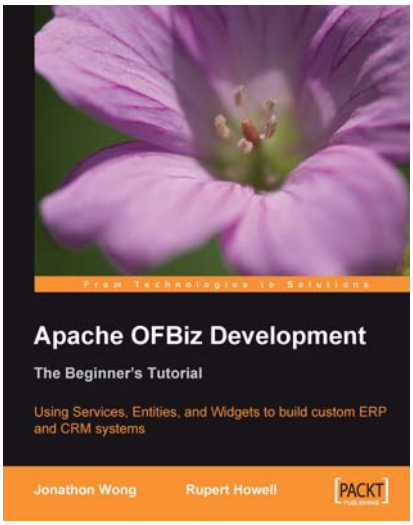
Apache OFBiz Development

The Beginner's Tutorial

Jonathon Wong

Rupert Howell



Chapter No. 10

"The Service Engine"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.10 "The Service Engine"

A synopsis of the book’s content

Information on where to buy this book

About the Authors

Jonathan Wong Jong Hann is an avid puzzle solver. He is constantly in search of new problems to take apart. He has dabbled in Rubik's Cubes, maze navigation algorithms, and various other logical and mechanical puzzles.

He had taken apart OFBiz within a month for a client who wanted to evaluate it. Having mapped out the architectural structure of OFBiz, he also embarked on documenting the functional and ERP-specific aspects of OFBiz. Within the following six months, he had completed three small-scale projects with OFBiz. He is currently using OFBiz in almost every new project, leveraging OFBiz's advantage for rapid prototyping and development.

He first delved into Java some 10 years ago. He has since specialized in clean programming structures, design patterns, and parallel computing. Since then, he also picked up the hobby of reverse engineering various open source software to equip his employers and himself with new technologies. Jonathon has also worked for clients who needed to take apart legacy systems to make corrections.

I would like to thank James Lumsden, the Acquisition Editor at Packt Publishing, for his help in keeping this book within a coherent scope and on schedule. I would also like to thank my co-author and reviewers for completing and rounding off this book when my hand injury and work schedule overwhelmed me.

Rupert Howell, while developing Java applications for the UK's Office for National Statistics, stumbled upon Open For Business and has been working with the framework ever since. Since early 2003, Rupert has been a consultant to some of the UK's largest OFBiz implementations and has helped some major retailers successfully migrate their entire ERP systems to OFBiz.

Rupert holds a Master's degree in Mechanical Engineering and is a Director of Provolve Ltd, a company specializing in OFBiz-based solutions. For more information see the Provolve website at `www.provolve.com`.

I would like to thank the reviewers for giving up their precious time—your input was invaluable.

For Sophie—who I love with all my heart and who never fails to make me happy.

Apache OFBiz Development

The Beginner's Tutorial

Apache Open For Business or OFBiz as it is more commonly known, is an open source framework designed to facilitate the building of Enterprise Resource Planning (ERP) software. ERP is a general name for any system which attempts to integrate all business processes and underlying data into one single system. Indeed the OFBiz framework not only facilitates the building of your own custom software, but also comes packaged with many tools you would expect from an ERP system, and much more. The extent to which you wish to use these applications is entirely up to you and the needs of your business. Some businesses choose to use some or all of these components virtually straight out of the box. Others may spend time and money customizing these components or building new ones to suit their own needs and their own unique business processes. Since OFBiz is licensed under the Apache License Version 2.0, organizations can use, customize, extend, modify, repackage, and even resell OFBiz completely free of charge.

OFBiz is aimed primarily at ecommerce businesses, giving easily customizable tools such as a full Warehouse Management System (WMS), an accounting system and full order and product management systems. It even has a full front end, customer facing website and shopping cart with tools and features comparable to multimillion dollar websites such as Amazon, not to mention its own set of self maintenance and administrative tools. Out of the box, OFBiz is a multi-currency system working just as well with British Pounds, Euros, or any other currency as it does with US Dollars. It is multilingual and is able to display text in different languages depending on where in the world the user or customer is looking. It is so versatile it is not even tied to one database, and fully supports most well known databases.

The main reason for its versatility and size has been its open source model. OFBiz is truly a collaborative effort with a small number of committers who have volunteered to develop and maintain a code base supplied by both themselves and a growing community. Although documentation on the tools is often thin on the ground (this is mainly because of the speed at which the project and components evolve), there are free and active mailing lists set up that will become an invaluable learning tool and source of information as you progress with OFBiz. The OFBiz project employs the use of the well known JIRA application (a bug and issue tracking and project management tool – which is using the OFBiz Entity Engine, a major part of the framework). This allows developers and users to tell the community about any bugs they find in the software or request new features that they would find handy but perhaps don't have the resources to develop for themselves. Who knows? Once you have read this book you may even want to have a go at developing an outstanding issue or fixing a bug for the project yourself!

What This Book Covers

Chapter 1, using a Windows machine, guides us through downloading and installing the necessary software we need to obtain and run the OFBiz framework. We are shown how to create an Eclipse project and once running we are shown a few of the components as we place an order on the applications customer facing website and fulfill the order using the back office's Order Manager component.

In *Chapter 2* we learn the structure of OFBiz. We are introduced to the concepts of the framework, applications, and hot-deploy directories. We perform our first customization on an existing OFBiz component and finally create the structure of our own bespoke application.

In *Chapter 3* we take a look at how the output to the screen is constructed using the screen widgets. We start by creating a simple screen in our learning component, showing a basic output. By the end of the chapter we have learned how to create complex screens, made up of different sections.

In *Chapter 4* we study form widgets. We learn how they are used within screen widgets and can save us development time and effort by quickly producing XHTML forms so we can input information to the application.

In *Chapter 5* we complete our investigation into the presentation layer of OFBiz by learning how to use Menu-Widgets to navigate around our component. We also take more of a look at how FreeMarker can help us display more complicated screens.

In *Chapter 6* we re-visit the Control, learning more about how OFBiz makes use of the Front Controller pattern to configure the flow through our component in just one place. We learn how OFBiz handles different types of requests and we are introduced to the concept of security. By the end of the chapter, we have added "log in" functionality to our bespoke application and have seen how easy it is to force a request to be "secure".

In *Chapter 7* we move on to the concepts of the Entity Engine and learn how OFBiz employs the use of the Delegation Pattern to give us easy access to methods to persist data. We learn how OFBiz creates the database structure, adding fields, tables, constraints and indexes from definitions in XML files. We see how, by using View Entities, we can perform joins across tables allowing us to create complex queries. We are also introduced to the Webtools administration component of OFBiz and discover how to access the raw data through these screens.

In *Chapter 8* we are led through a series of examples designed to showcase data lookup and persistence techniques. We learn how to use the GenericDelegator's methods to lookup and manipulate the underlying data. We discover how using the Entity Engine Cache can massively improve performance by cutting down the number of database queries and learn how complex queries can be created on the fl y by using Dynamic View Entity. Finally we learn how to use the EntityListIterator to efficiently paginate through large record sets.

In *Chapter 9* we take a closer look at Java events by learning a number of techniques vital to programming the flow of the application. We also take a look at how we can assign users permissions and how these permissions are checked within the Java methods.

In *Chapter 10* we next see another type of event and a very important one—the services. We learn about the advantages of the Service Engine and how it works, learning how to define and write services in Java. We learn the difference between invoking these services synchronously and asynchronously and how services can be scheduled using the Job Scheduler. Finally we learn how to trigger these services using ECAs (Event Condition Actions).

In *Chapter 11* we move on to study complex permissions, learning how to assign users granular permissions, and how simply these permissions can be checked from our services. We learn by example how to restrict users from viewing or inputting data depending on access rights whilst building up our bespoke application.

In *Chapter 12* we learn about the OFBiz Mini-Language. We see how we can write simple services and events in Minilang, and we learn when it should be ideally used. We see its versatility and see how widespread its concepts are used throughout the framework.

In *Chapter 13* we come towards the end of learning about the framework, we see how easy it is to change the look and feel of the component and study the structure of the existing screens. The chapter moves on to some more advanced FreeMarker techniques that are commonly used throughout all of the components. Finally using the production of a PDF as an example we see how to output different formats.

In *Chapter 14* we learn some real world developing techniques, including how to debug through the different parts and languages found within the framework. We see how to connect a remote debugger to the application and step through the Java code line by line using the Eclipse IDE. We next learn the concepts behind getting the latest bug fixes and features and merging these into our project using Windows tools enabling us to successfully work with the latest and greatest version on OFBiz. Finally we see learn how to run OFBiz behind the Apache HTTP Server allowing us to create a scalable architecture.

10
The Service Engine

In this chapter, we will be exploring the Service Engine. Services in OFBiz operate in a **Service Oriented Architecture** (**SOA**). These services not only have the ability to invoke other services internally, but can also be 'opened up' and invoked by remote applications using, amongst other methods, the widely adopted messaging protocol SOAP.

Besides serving as a platform for interoperability, OFBiz services also offer us additional capability to organize our code. The traditional organizational strategies in object-oriented Java were a great improvement over the procedural paradigm. Wrapping both methods and variables together into objects to form a powerful "behavioral model" for code organization (where object's methods and variables define their behavior). Similarly with OFBiz services we are able to bundle groups of behavior together to form a coherent "service". We can say that OFBiz services, in terms of code or software organization, operate at a higher level than Java object-oriented organizational strategies.

In this chapter, we will be looking at:

- Defining and creating a Java service
- Service parameters
- Special unchecked (unmatched) IN/OUT parameters
- Security-related programming
- Calling services from code (using dispatcher).
- IN/OUT parameter mismatch when calling services
- Sending feedback; standard return codes `success`, `error` and `fail`
- Implementing Service Interfaces
- Synchronous and asynchronous services
- Using the Service Engine tools
- ECAs: Event Condition Actions

Defining a Service

We first need to define a service. Our first service will be named `learningFirstService`.

In the folder `${component:learning}`, create a new folder called `servicedef`. In that folder, create a new file called `services.xml` and enter into it this:

```
<?xml version="1.0" encoding="UTF-8" ?>

<services xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
                            "http://www.ofbiz.org/dtds/services.xsd">
  <description>Learning Component Services</description>

  <service name="learningFirstService" engine="java"
          location="org.ofbiz.learning.learning.LearningServices"
          invoke="learningFirstService">
    <description>Our First Service</description>
    <attribute name="firstName" type="String" mode="IN"
               optional="true"/>
    <attribute name="lastName" type="String" mode="IN"
               optional="true"/>
  </service>
</services>
```

In the file `${component:learning}\ofbiz-component.xml`, add after the last `<entity-resource>` element this:

```
<service-resource type="model" loader="main"
                  location="servicedef/services.xml"/>
```

That tells our component `learning` to look for service definitions in the file `${component:learning}\servicedef\services.xml`.

It is important to note that all service definitions are loaded at startup; therefore any changes to any of the service definition files will require a restart!

Creating the Java Code for the Service

In the package `org.ofbiz.learning.learning`, create a new class called `LearningServices` with one static method `learningFirstService`:

```
package org.ofbiz.learning.learning;

import java.util.Map;
```

```
import org.ofbiz.service.DispatchContext;
import org.ofbiz.service.ServiceUtil;

public class LearningServices {

   public static final String module =
                                    LearningServices.class.getName();

   public static Map learningFirstService(DispatchContext dctx,
        Map context){

      Map resultMap = ServiceUtil.returnSuccess("You have called on
         service 'learningFirstService' successfully!");
      return resultMap;
   }

}
```

Services must return a map. This map must contain at least one entry. This entry must have the key `responseMessage` (see `org.ofbiz.service.ModelService.RESPONSE_MESSAGE`), having a value of one of the following:

- `success` or `ModelService.RESPOND_SUCCESS`
- `error` or `ModelService.RESPOND_ERROR`
- `fail` or `ModelService.RESPOND_FAIL`

By using `ServiceUtil.returnSuccess()` to construct the minimal return map, we do not need to bother adding the `responseMessage` key and value pair.

Another entry that is often used is that with the key `successMessage` (`ModelService.SUCCESS_MESSAGE`). By doing `ServiceUtil.returnSuccess("Some message")`, we will get a return map with entry `successMessage` of value "Some message". Again, `ServiceUtil` insulates us from having to learn the convention in key names.

Testing Our First Service

Stop OFBiz, recompile our learning component and restart OFBiz so that the modified `ofbiz-component.xml` and the new `services.xml` can be loaded.

In `${component:learning}\widget\learning\LearningScreens.xml`, insert a new Screen Widget:

```
<screen name="TestFirstService">
  <section>
    <widgets>
      <section>
        <condition><if-empty field-name="formTarget"/></condition>
```

```
          <actions>
             <set field="formTarget" value="TestFirstService"/>
             <set field="title" value="Testing Our First Service"/>
          </actions>
          <widgets/>
        </section>
        <decorator-screen name="main-decorator"
                        location="${parameters.mainDecoratorLocation}">
          <decorator-section name="body">
            <include-form name="TestingServices"
                        location="component://learning/widget/learning/
                                  LearningForms.xml"/>
<label text="Full Name: ${parameters.fullName}"/>
          </decorator-section>
        </decorator-screen>
      </widgets>
    </section>
</screen>
```

In the file `${component:learning}\widget\learning\LearningForms.xml`, insert a new Form Widget:

```
<form name="TestingServices" type="single" target="${formTarget}">
  <field name="firstName"><text/></field>
  <field name="lastName"><text/></field>
  <field name="planetId"><text/></field>
  <field name="submit"><submit/></field>
</form>
```

Notice how the `formTarget` field is being set in the screen and used in the form. For now don't worry about the **Full Name** label we are setting from the screen. Our service will eventually set that.

In the file `${webapp:learning}\WEB-INF\controller.xml`, insert a new request map:

```
<request-map uri="TestFirstService">
  <event type="service" invoke="learningFirstService"/>
  <response name="success" type="view" value="TestFirstService"/>
</request-map>
```

The control servlet currently has no way of knowing how to handle an event of type service, so in `controller.xml` we must add a new handler element immediately under the other `<handler>` elements:

```
    <handler name="service" type="request"
             class="org.ofbiz.webapp.event.ServiceEventHandler"/>
    <handler name="service-multi" type="request"
            class="org.ofbiz.webapp.event.ServiceMultiEventHandler"/>
```

We will cover `service-multi` services later. Finally add a new view map:

```
<view-map name="TestFirstService" type="screen"
          page="component://learning/widget/learning/
                LearningScreens.xml#TestFirstService"/>
```

Fire to webapp `learning` an http OFBiz request `TestFirstService`, and see that we have successfully invoked our first service:

Service Parameters

Just like Java methods, OFBiz services can have input and output parameters and just like Java methods, the parameter types must be declared.

Input Parameters (IN)

Our first service is defined with two parameters:

```
<attribute name="firstName" type="String" mode="IN" optional="true"/>
<attribute name="lastName" type="String" mode="IN" optional="true"/>
```

Any parameters sent to the service by the end-user as form parameters, but not in the services list of declared input parameters, will be dropped. Other parameters are converted to a `Map` by the framework and passed into our static method as the second parameter.

Add a new method `handleInputParamaters` to our `LearningServices` class.

```
public static Map handleParameters(DispatchContext dctx, Map
   context){

   String firstName = (String)context.get("firstName");
   String lastName = (String)context.get("lastName");
   String planetId= (String)context.get("planetId");
```

```
        String message = "firstName: " + firstName + "<br/>";
        message = message + "lastName: " + lastName + "<br/>";
        message = message + "planetId: " + planetId;

        Map resultMap = ServiceUtil.returnSuccess(message);
        return resultMap;
    }
```

We can now make our service definition invoke this method instead of the `learningFirstService` method by opening our `services.xml` file and replacing:

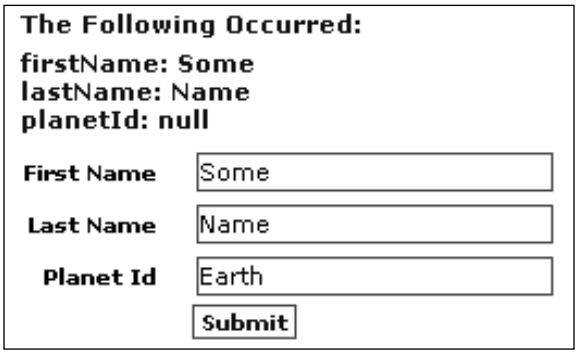
```
<service name="learningFirstService" engine="java"
         location="org.ofbiz.learning.learning.LearningServices"
         invoke="learningFirstService">
```

with:

```
<service name="learningFirstService" engine="java"
        location="org.ofbiz.learning.learning.LearningServices"
         invoke="handleParameters">
```

Once again shutdown, recompile, and restart OFBiz.

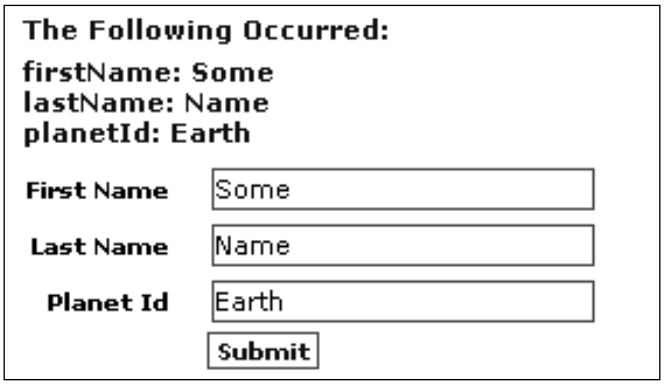
Enter for fields **First Name, Last Name**, and **Planet Id** values **Some**, **Name**, and **Earth**, respectively. Submit and notice that only the first two parameters went through to the service. Parameter `planetId` was dropped silently as it was not declared in the service definition.



Modify the service `learningFirstService` in the file `${component:learning}\servicedef\services.xml`, and add below the second parameter a third one like this:

```
<attribute name="planetId" type="String" mode="IN" optional="true"/>
```

Restart OFBiz and submit the same values for the three form fields, and see all three parameters go through to the service.



Output Parameters (OUT)

Just like Java methods have return values (although Java methods can have only one typed return value), services can be declared with output parameters. When invoked as events from the controller, parameters will be silently dropped if they are not declared in our service's definition. Add this to our service definition:

```
<attribute name="fullName" type="String" mode="OUT" optional="true"/>
```

And in the method `handleParameters` in `org.ofbiz.learning.learning.LearningServices` replace:

```
Map resultMap = ServiceUtil.returnSuccess(message);
return resultMap;
```

with

```
Map resultMap = ServiceUtil.returnSuccess(message);
resultMap.put("fullName", firstName + " " + lastName);
return resultMap;
```

We have now added the `fullName` parameter to the `resultMap`. To see this in action we need to create a new screen widget in `LearningScreens.xml`:

```
<screen name="TestFirstServiceOutput">
  <section>
    <actions><set field="formTarget"
                  value="TestFirstServiceOutput"/></actions>
```

```
      <widgets>
        <include-screen name="TestFirstService"/>
      </widgets>
    </section>
  </screen>
```

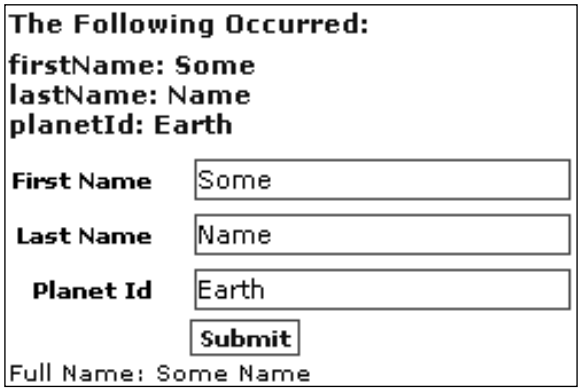
Add the `request-map` to the `controller.xml` file:

```
<request-map uri="TestFirstServiceOutput">
  <event type="service" invoke="learningFirstService"/>
  <response name="success" type="view"
            value="TestFirstServiceOutput"/>
</request-map>
```

and finally the `view-map`:

```
<view-map name="TestFirstServiceOutput" type="screen"
          page="component://learning/widget/learning/
          LearningScreens.xml#TestFirstServiceOutput"/>
```

Stop OFBiz, rebuild our Learning Component and restart, fire an OFBiz http request `TestFirstServiceOutput` to webapp `learning`. Submit your first and last names and planet and notice that now the `fullName` parameter has been populated.



Two Way Parameters (INOUT)

A service may change the value of an input parameter and we may need a calling service to be aware of this change. To save us declaring the same parameter twice, with a mode for `IN` and a mode for `OUT`, we may use the mode `INOUT`.

```
<attribute name="fullName" type="String" mode="INOUT"
           optional="true"/>
```

Special Unchecked Parameters

There are a few special cases where `IN/OUT` parameters can exist even though the service definition does not declare them. They are:

- `responseMessage`
- `errorMessage`
- `errorMessageList`
- `successMessage`
- `successMessageList`
- `userLogin`
- `locale`

The parameters `responseMessage`, `errorMessage`, `errorMessageList`, `successMessage` and `successMessageList` are necessary placeholders for feedback messages. They must be allowed through all validation checks.

The parameter `userLogin` is often required for authentication and permissions checks.

The parameter `locale` is needed just about everywhere in OFBiz. For locale-specificity in certain operations like retrieving template feedback messages, or like formatting numbers and currency figures.

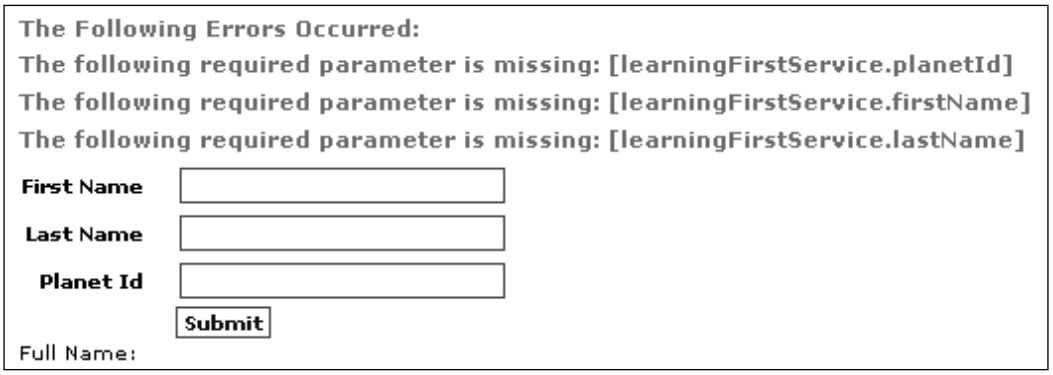
Optional and Compulsory Parameters

The Service Engine checks the validity of the input and the output to ensure that what is coming into the service and is leaving adheres to the service definition. If the `optional` attribute is set to `false` and an expected parameter is missing, then the validation will fail and the service will return an error. This transaction will now be marked for rollback, meaning any changes to the database made during this transaction will never be committed. This could include any changes made to the database by calling services. For example:

```
<attribute name="fullName" type="String" mode="INOUT"
           optional="false"/>
```

Here the parameter `fullName` must be passed into the service and the service must also add this parameter to the `resultMap` and pass it out or validation will fail and an error will be thrown.

Try changing all of the optional flags on our newly created service to `false`. After a restart we should see:



The DispatchContext

We have already seen how parameters are passed into our Java method as a `Map`. Just as the `userLogin` object of type `GenericValue` and the `locale` object of type `Locale` were added as attributes to the request for the Java events, both are now automatically added to this context map when the service is invoked in this way.

The first parameter, the `DispatchContext`, contains the remaining tools we need to access the database, or to invoke other services.

From our Java code we can get access to the following handy objects like this:

```
GenericValue userLogin = (GenericValue)context.get("userLogin");

Locale locale = (Locale)context.get("locale");

GenericDelegator delegator = dctx.getDelegator();

LocalDispatcher dispatcher = dctx.getDelegator();

Security security = dctx.getSecurity();
```

For a full list of objects that are available from the `DispatchContext`, take a look through the code in `org.ofbiz.service.DispatchContext`.

The service engine is in no way reliant on there being `HTTPServletRequest` or `HTTPServletResponse` objects available. Because of this we are able to invoke services outside of the web environment and they can be invoked remotely or scheduled to run "offline".

Service Security and Access Control

Security-related programming in services is exactly like that in events.

In the class `org.ofbiz.learning.learning.LearningServices`, create a new static method `serviceWithAuth`:

```
    public static Map serviceWithAuth(DispatchContext dctx, Map
       context){
       Security security = dctx.getSecurity();
       Map resultMap = null;
       if (context.get("userLogin") == null ||
           !security.hasPermission("LEARN_VIEW",
                       (GenericValue)context.get("userLogin"))) {
           resultMap = ServiceUtil.returnError("You have no access
                        here. You're not welcome!");
       }
       else {
           resultMap = ServiceUtil.returnSuccess("Welcome! You have
access!");
       }
       return resultMap;
    }
```

Ensure that the correct imports have been added to the class:

```
import java.util.Map;
import org.ofbiz.entity.GenericValue;
import org.ofbiz.security.Security;
```

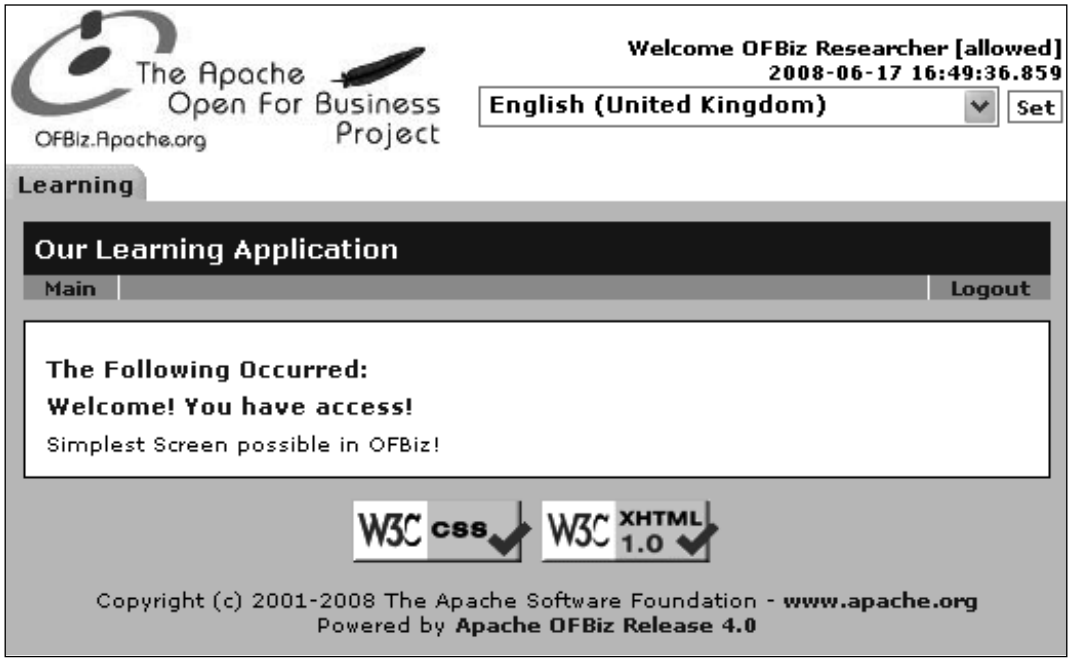
In the file `${component:learning}\servicedef\services.xml`, add a new service definition:

```
    <service name="learningServiceWithAuth" engine="java"
        location="org.ofbiz.learning.learning.LearningServices"
                  invoke="serviceWithAuth">
        <description>Service with some security-related
                     codes</description>
    </service>
```

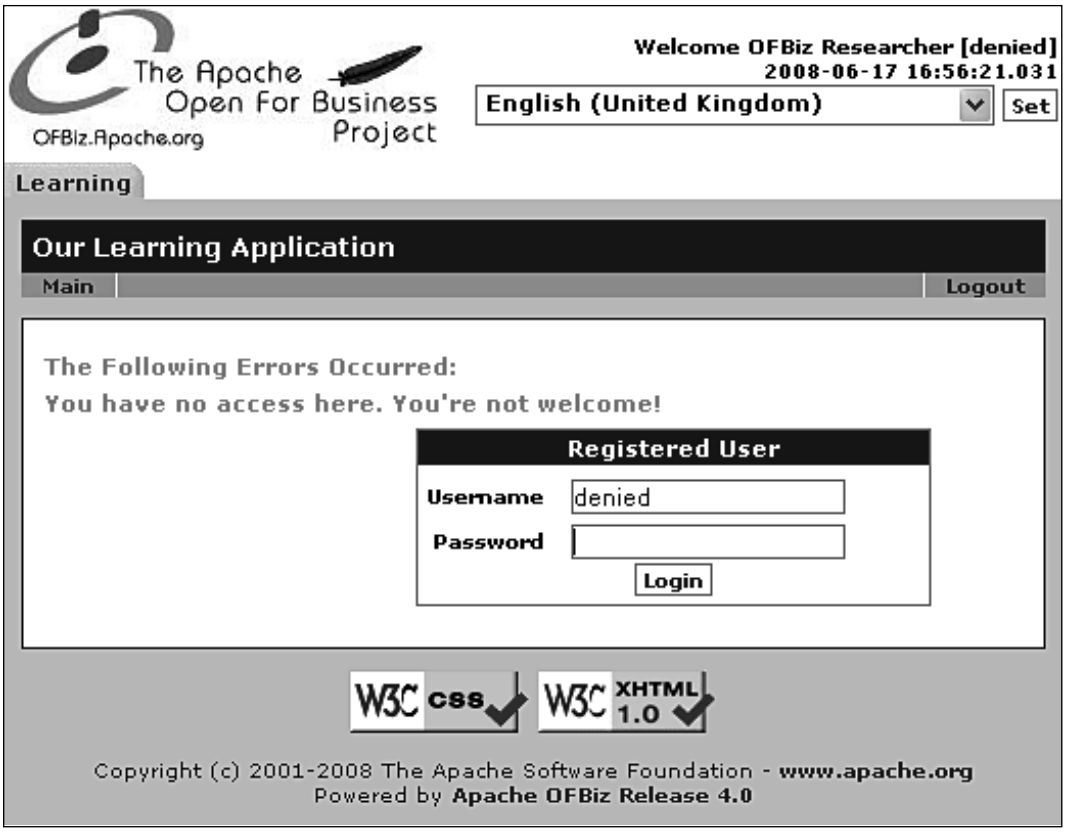
In the file `${webapp:learning}\WEB-INF\controller.xml`, add a new `request map`:

```
<request-map uri="TestServiceWithAuth">
  <security auth="true" https="true"/>
  <event type="service" invoke="learningServiceWithAuth"/>
  <response name="success" type="view" value="SimplestScreen"/>
  <response name="error" type="view" value="login"/>
</request-map>
```

Rebuild and restart and then fire to webapp `learning` an http OFBiz request `TestServiceWithAuth`, login with username **allowed** password **ofbiz**, and see the welcome message displayed:



Logging in with username **denied** password **zibfo** will show an error message. Thanks to the request-map's response element named `error` having a value of `login`, we are returned back to the login screen:



Calling Services from Java Code

So far, we have explored services invoked as events from the controller (example `<event type="service" invoke="learningFirstService"/>`). We now look at calling services explicitly from code.

To invoke services from code, we use the `dispatcher` object, which is an object of type `org.ofbiz.service.ServiceDispatcher`. Since this is obtainable from the `DispatchContext` we can invoke services from other services.

To demonstrate this we are going to create one simple service that calls another.

In our `services.xml` file in `${component:learning}\servicedef` add two new service definitions:

```
<service name="learningCallingServiceOne" engine="java"
         location="org.ofbiz.learning.learning.LearningServices"
         invoke="callingServiceOne">
```

```
        <description>First Service Called From The Controller
        </description>
        <attribute name="firstName" type="String" mode="IN"
                   optional="false"/>
        <attribute name="lastName" type="String" mode="IN"
                   optional="false"/>
        <attribute name="planetId" type="String" mode="IN"
                   optional="false"/>
        <attribute name="fullName" type="String" mode="OUT"
                   optional="true"/>
    </service>
    <service name="learningCallingServiceTwo" engine="java"
             location="org.ofbiz.learning.learning.LearningServices"
             invoke="callingServiceTwo">
        <description>Second Service Called From Service One
        </description>
        <attribute name="planetId" type="String" mode="IN"
                   optional="false"/>
    </service>
```

In this simple example it is going to be the job of `learningCallingServiceOne` to prepare the parameter map and pass in the `planetId` parameter to `learningCallingServiceTwo`. The second service will determine if the input is `EARTH`, and return an error if not.

In the class `org.ofbiz.learning.learning.LearningEvents`, add the static method that is invoked by `learningCallingServiceOne`:

```
    public static Map callingServiceOne(DispatchContext dctx, Map
context){

        LocalDispatcher dispatcher = dctx.getDispatcher();
        Map resultMap = null;
        String firstName = (String)context.get("firstName");
        String lastName = (String)context.get("lastName");
        String planetId = (String)context.get("planetId");
        GenericValue userLogin = (GenericValue)context.get("userLogin");
        Locale locale = (Locale)context.get("locale");

        Map serviceTwoCtx = UtilMisc.toMap("planetId", planetId,
"userLogin", userLogin, "locale", locale);
        try{
            resultMap = dispatcher.runSync("learningCallingServiceTwo",
serviceTwoCtx);
        }catch(GenericServiceException e){
            Debug.logError(e, module);
        }
```

```
        resultMap.put("fullName", firstName + " " + lastName);
        return resultMap;
    }
```

and also the method invoked by `learningServiceTwo`:

```
    public static Map callingServiceTwo(DispatchContext dctx, Map
        context){
        String planetId = (String)context.get("planetId");
        Map resultMap = null;
        if(planetId.equals("EARTH")){
            resultMap = ServiceUtil.returnSuccess("This planet is
                        Earth");
        }else{
            resultMap = ServiceUtil.returnError("This planet is NOT
                        Earth");
        }
        return resultMap;
    }
```

To `LearningScreens.xml`add:

```
<screen name="TestCallingServices">
  <section>
    <actions><set field="formTarget" value="TestCallingServices"/></
actions>
    <widgets>
      <include-screen name="TestFirstService"/>
    </widgets>
  </section>
</screen>
```

Finally add the request-map to the `controller.xml` file:

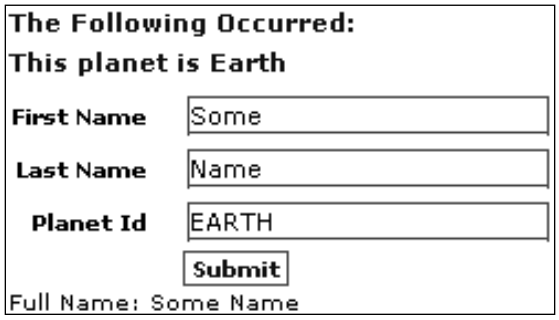
```
<request-map uri="TestCallingServices">
  <security auth="false" https="false"/>
  <event type="service" invoke="learningCallingServiceOne"/>
  <response name="success" type="view" value="TestCallingServices"/>
  <response name="error" type="view" value="TestCallingServices"/>
</request-map>
```

and also the `view-map`:

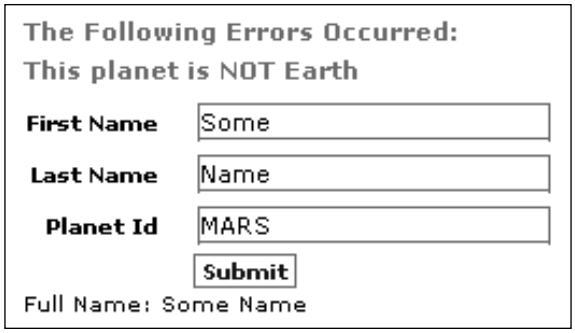
```
<view-map name="TestCallingServices" type="screen"
          page="component://learning/widget/learning/
                LearningScreens.xml#TestCallingServices"/>
```

Stop, rebuild, and restart, then fire an OFBiz http request `TestCallingServices` to webapp `learning`. Do not be alarmed if straight away you see error messages informing us that the required parameters are missing. By sending this request we have effectively called our service with none of our compulsory parameters present.

Enter your name and in the **Planet Id**, enter **EARTH**. You should see:



Try entering **MARS** as the **Planet Id**.



Notice how in the Java code for the static method `callingServiceOne` the line

```
resultMap = dispatcher.runSync("learningCallingServiceTwo",
                               serviceTwoCtx);
```

is wrapped in a `try/catch` block. Similar to how the methods on the `GenericDelegator` object that accessed the database threw a `GenericEntityException`, methods on our `dispatcher` object throw a `GenericServiceException` which must be handled.

There are three main ways of invoking a service:

- `runSync`—which runs a service synchronously and returns the result as a map.

- `runSyncIgnore`—which runs a service synchronously and ignores the result. Nothing is passed back.
- `runAsync`—which runs a service asynchronously. Again, nothing is passed back.

The difference between synchronously and asynchronously run services is discussed in more detail in the section called *Synchronous and Asynchronous Services*.

Implementing Interfaces

Open up the `services.xml` file in `${component:learning}\servicedef` and take a look at the service definitions for both `learningFirstService` and `learningCallingServiceOne`.

Do you notice that the `<attribute>` elements (parameters) are the same? To cut down on the duplication of XML code, services with similar parameters can implement an interface.

As the first service element in this file enter the following:

```
<service name="learningInterface" engine="interface">
  <description>Interface to describe base parameters for Learning
               Services</description>
  <attribute name="firstName" type="String" mode="IN"
             optional="false"/>
  <attribute name="lastName" type="String" mode="IN"
             optional="false"/>
  <attribute name="planetId" type="String" mode="IN"
             optional="false"/>
  <attribute name="fullName" type="String" mode="OUT"
            optional="true"/>
</service>
```

Notice that the `engine` attribute is set to `interface`.

Replace all of the `<attribute>` elements in the `learningFirstService` and `learningCallingServiceOne` service definitions with:

```
<implements service="learningInterface"/>
```

So the service definition for `learningServiceOne` becomes:

```
<service name="learningCallingServiceOne" engine="java"
       location="org.ofbiz.learning.learning.LearningServices"
       invoke="callingServiceOne">
```

```
        <description>First Service Called From The Controller
        </description>
          <implements service="learningInterface"/>
       </service>
```

Restart OFBiz and then fire an OFBiz http request `TestCallingServices` to webapp `learning`. Nothing should have changed—the services should run exactly as before, however our code is now somewhat tidier.

Overriding Implemented Attributes

It may be the case that the interface specifies an attribute as `optional="false"`, however, our service does not need this parameter. We can simply override the interface and add the `<attribute>` element with whatever settings we wish.

For example, if we wish to make the `planetId` optional in the above example, the `<implements>` element could remain, but a new `<attribute>` element would be added like this:

```
<service name="learningCallingServiceOne" engine="java"
         location="org.ofbiz.learning.learning.LearningServices"
         invoke="callingServiceOne">
    <description>First Service Called From The Controller
     </description>
   <implements service="learningInterface"/>
     <attribute name="planetId" type="String" mode="IN"
                optional="false"/>
</service>
```

Synchronous and Asynchronous Services.

The service engine allows us to invoke services synchronously or asynchronously. A synchronous service will be invoked in the same thread, and the thread will "wait" for the invoked service to complete before continuing. The calling service can obtain information from the synchronously run service, meaning its `OUT` parameters are accessible.

Asynchronous services run in a separate thread and the current thread will continue without waiting. The invoked service will effectively start to run in parallel to the service or event from which it was called. The current thread can therefore gain no information from a service that is run asynchronously. An error that occurs in an asynchronous service will not cause a failure or error in the service or event from which it is called.

A good example of an asynchronously called service is the `sendOrderConfirmation` service that creates and sends an order confirmation email. Once a customer has placed an order, there is no need to wait while the mail service is called and the mail sent. The mail server may be down, or busy, which may result in an error that would otherwise stop our customer form placing the order. It is much more preferable to allow the customer to continue to the **Order Confirmation** page and have our business receive the valuable order. By calling this service asynchronously, there is no delay to the customer in the checkout process, and while we log and fix any errors with the mail server, we still take the order.

Behind the scenes, an asynchronous service is actually added to the Job Scheduler. It is the Job Scheduler's task to invoke services that are waiting in the queue.

Using the Job Scheduler

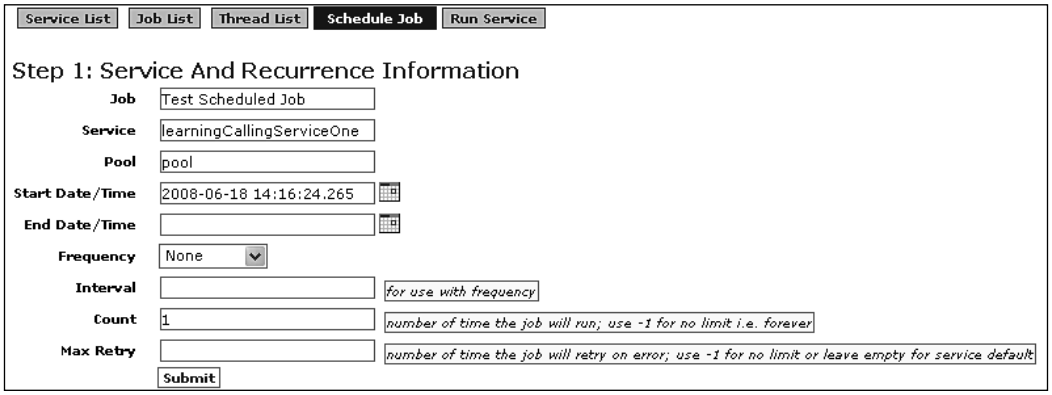
Asynchronous services are added to the Job Scheduler automatically. However, we can see which services are waiting to run and which have already been invoked through the **Webtools** console. We can even schedule services to run once only or recur as often as we like.

Open up the **Webtools** console at `https://localhost:8443/webtools/control/main` and take a look under the **Service Engine Tools** heading. Select **Job List** to view a full list of jobs. Jobs without a **Start Date/Time** have not started yet. Those with an **End Date/Time** have completed. The **Run Time** is the time they are scheduled to run. All of the outstanding jobs in this list were added to the `JobSandbox` Entity when the initial seed data load was performed, along with the `RecurrenceRule` (also an Entity) information specifying how often they should be run. They are all maintenance jobs that are performed "offline".

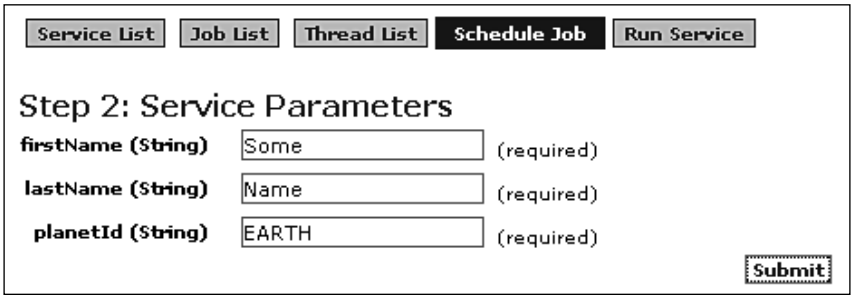
The **Pool** these jobs are run from by default is set to `pool`. In an architecture where there may be multiple OFBiz instances connecting to the same database, this can be important. One OFBiz instance can be dedicated to performing certain jobs, and even though job schedulers may be running on each instance, this setting can be changed so we know only one of our instances will run this job.

The Service Engine settings can be configured in `framework\service\config\serviceengine.xml`. By changing both the `send-to-pool` attribute and the name attribute on the `<run-from-pool>` element, we can ensure that only jobs created on an OFBiz instance are run by this OFBiz instance.

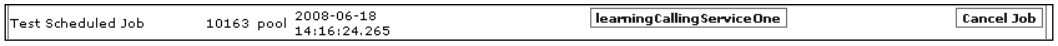
Click on the **Schedule Job** button and in the Service field enter `learningCallingServiceOne`, leave the **Pool** as **pool** and enter today's date/time by selecting the calendar icon and clicking on today's date. We will need to add 5 minutes onto this once it appears in the box. In the below example the Date appeared as **2008-06-18 14:11:24.265.** This job is only going to be scheduled to run once, although we could specify any recurrence information we wish.



Select **Submit** and notice that scheduler is already aware of the parameters that can (or must, in this case) be entered. This information has been taken from the service definition in our `services.xml` file.

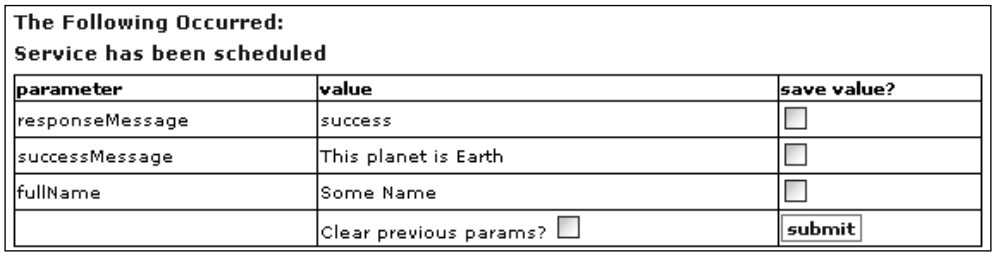


Press **Submit** to schedule the job and find the entry in the list. This list is ordered by **Run Time** so it may not be the first. Recurring maintenance jobs are imported in the seed data and are scheduled to run overnight. These will more than likely be above the job we have just scheduled since their run-time is further in the future. The entered parameters are converted to a map and then serialized to the database. They are then fed to the service at run time.



Quickly Running a Service

Using the **Webtools** console it is also possible to run a service synchronously. This is quicker than going through the scheduler should you need to test a service or debug through a service. Select the **Run Service** button from the menu and enter the same service name, **submit** then enter the same parameters again. This time the service is run straight away and the `OUT` parameters and messages are passed back to the screen:

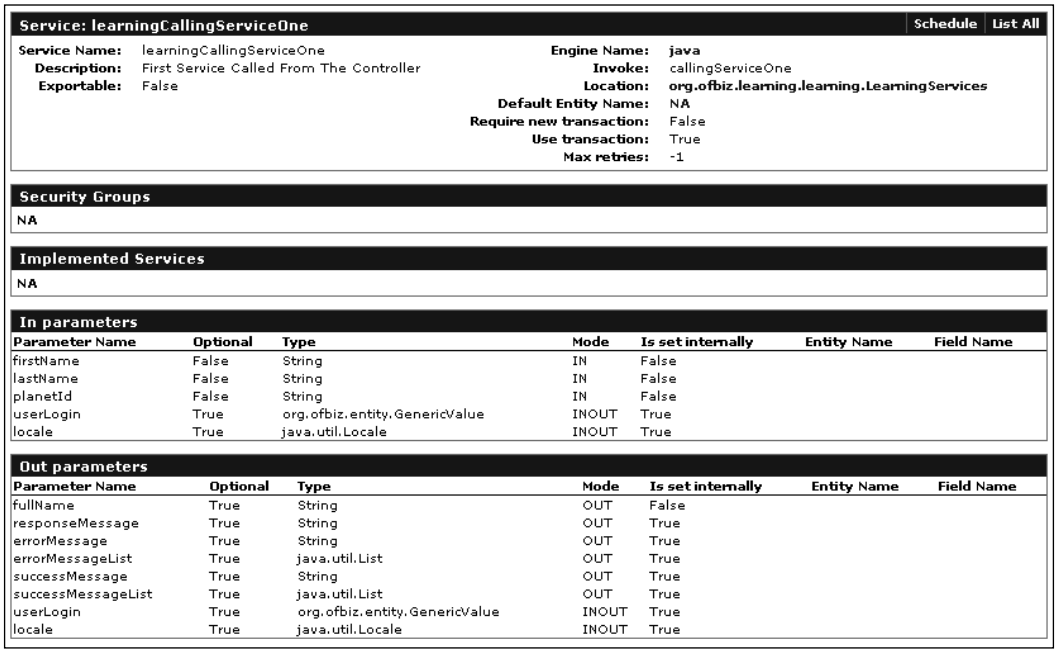


Naming a Service and the Service Reference

Service names must be unique throughout the entire application. Because we do not need to specify a location when we invoke a service, if service names were duplicated we can not guarantee that the service we want to invoke is the one that is actually invoked. OFBiz comes complete with a full service reference, which is in fact a dictionary of services that we can use to check if a service exists with the name we are about to choose, or even if there is a service already written that we are about to duplicate.

From `https://localhost:8443/webtools/control/main` select the **Service Reference** and select "**l**" for learning. Here we can see all of our learning services, what engine they use and what method they invoke. By selecting the service `learningCallingServiceOne`, we can obtain complete information about this service as was defined in the service definition file `services.xml`. It even includes information about the parameters that are passed in and out automatically.

Careful selection of intuitive service names and use of the description tags in the service definition files are good practice since this allows other developers to reuse services that already exists, rather than duplicate work unnecessarily.



Service: learningCallingServiceOne — Schedule | List All

Service Name:	learningCallingServiceOne	**Engine Name:**	java
Description:	First Service Called From The Controller	**Invoke:**	callingServiceOne
Exportable:	False	**Location:**	org.ofbiz.learning.learning.LearningServices
		Default Entity Name:	NA
		Require new transaction:	False
		Use transaction:	True
		Max retries:	-1

Security Groups

NA

Implemented Services

NA

In parameters

Parameter Name	Optional	Type	Mode	Is set internally	Entity Name	Field Name
firstName	False	String	IN	False		
lastName	False	String	IN	False		
planetId	False	String	IN	False		
userLogin	True	org.ofbiz.entity.GenericValue	INOUT	True		
locale	True	java.util.Locale	INOUT	True		

Out parameters

Parameter Name	Optional	Type	Mode	Is set internally	Entity Name	Field Name
fullName	True	String	OUT	False		
responseMessage	True	String	OUT	True		
errorMessage	True	String	OUT	True		
errorMessageList	True	java.util.List	OUT	True		
successMessage	True	String	OUT	True		
successMessageList	True	java.util.List	OUT	True		
userLogin	True	org.ofbiz.entity.GenericValue	INOUT	True		
locale	True	java.util.Locale	INOUT	True		

Event Condition Actions (ECA)

ECA refers to the structure of rules of a process. The Event is the trigger or the reason why the rule is being invoked. The condition is a check to see if we should continue and invoke the action, and the action is the final resulting change or modification. A real life example of an ECA could be "If you are leaving the house, check to see if it is raining. If so, fetch an umbrella". In this case the event is "leaving the house". The condition is "if it is raining" and the action is "fetch an umbrella".

There are two types of ECA rules in OFBiz: **Service Event Condition Actions** (**SECA**s) and **Entity Event Condition Actions** (**EECA**s).

Service Event Condition Actions (SECAs)

For SECAs the trigger (Event) is a service being invoked. A condition could be if a parameter equalled something (conditions are optional), and the action is to invoke another service.

SECAs are defined in the same directory as service definitions (`servicedef`). Inside files named `secas.xml`

Take a look at the existing SECAs in `applications\order\servicedef\secas.xml` and we can see a simple ECA:

```
<eca service="changeOrderStatus" event="commit"
     run-on-error="false">
    <condition field-name="statusId" operator="equals"
               value="ORDER_CANCELLED"/>
    <action service="releaseOrderPayments" mode="sync"/>
</eca>
```

When the `changeOrderStatus` transaction is just about to be committed, a lookup is performed by the framework to see if there are any ECAs for this event. If there are, and the parameter `statusId` is `ORDER_CANCELLED` then the `releaseOrderPayments` service is run synchronously.

Most commonly, SECAs are triggered on `commit` or `return`; however, it is possible for the event to be in any of the following stages in the service's lifecycle:

- `auth` — Before Authentication
- `in-validate` — Before `IN` parameter validation
- `out-validate` — Before `OUT` parameter validation
- `invoke` — Before service invocation
- `commit` — Just before the transaction is committed
- `return` — Before the service returns
- `global-commit`
- `global-rollback`

The variables `global-commit` and `global-rollback` are a little bit different. If the service is part of a transaction, they will only run after a rollback or between the two phases (JTA implementation) of a commit.

There are also two specific attributes whose values are false by default:

- `run-on-failure`
- `run-on-error`

You can set them to true if you want the SECA to run in spite of a failure or error. A failure is the same thing as an error, except it doesn't represent a case where a rollback is required.

It should be noted that parameters passed into the trigger service are available, if need be, to the action service. The trigger services `OUT` parameters are also available to the action service.

Before using SECAs in a component, the component must be informed of the location of the ECA service-resources:

```
<service-resource type="eca" loader="main"
                  location="servicedef/secas.xml"/>
```

This line must be added under the existing `<service-resource>` elements in the component's `ofbiz-component.xml` file.

Entity Event Condition Actions (EECAs)

For EECAs, the event is an operation on an entity and the action is a service being invoked.

EECAs are defined in the same directory as entity definitions (`entitydef`): inside files named `eecas.xml`.

They are used when it may not necessarily be a service that has initiated an operation on the entity, or you may wish that no matter what service operates on this entity, a certain course of action to be taken.

Open the `eecas.xml` file in the `applications\product\entitydef` directory and take a look at the first `<eca>` element:

```
<eca entity="Product" operation="create-store" event="return">
    <condition field-name="autoCreateKeywords"
               operator="not-equals" value="N"/>
    <action service="indexProductKeywords" mode="sync"
            value-attr="productInstance"/>
</eca>
```

This ECA ensures that once any creation or update operation on a `Product` record has been committed, so long as the `autoCreateKeywords` field of this record is not `N`, then the `indexProductKeywords` service will be automatically invoked synchronously.

The operation can be any of the following self-explanatory operations:

- `create`
- `store`
- `remove`
- `find`
- `create-store` (create or store/update)
- `create-remove`
- `store-remove`
- `create-store-remove`
- `any`

The `return` event is by far the most commonly used event in an EECA. But there are also `validate`, `run`, `cache-check`,`cache-put`, and `cache-clear` events. There is also the `run-on-error` attribute.

Before using EECAs in a component, the component must be informed of the location of the eca entity-resource:

```
<entity-resource type="eca" loader="main"
                 location="entitydef/eecas.xml"/>
```

must be added under the existing `<entity-resource>` elements in the component's `ofbiz-component.xml` file.

ECAs can often catch people out! Since there is no apparent flow from the trigger to the service in the code they can be difficult to debug. When debugging always keep an eye on the logs. When an ECA is triggered, an entry is placed into the logs to inform us of the trigger and the action.

Summary

This brings us to the end of our investigation into the OFBiz Service Engine. We have discovered how useful the Service Oriented Architecture in OFBiz can be and we have learnt how the use of some of the built in Service Engine tools, like the Service Reference, can help us when we are creating new services.

In this chapter we have looked at:

- Defining and creating services
- Service parameters
- Special unchecked (unmatched) `IN/OUT` parameters
- Security-related programming
- Calling services from code (using `dispatcher`).
- `IN/OUT` parameter mismatch when calling services
- Sending feedback; standard return codes `success`, `error` and `fail`.
- Implementing Service Interfaces
- Synchronous and asynchronous services
- Using the Service Engine tools
- ECAs: Event Condition Actions

The Service Engine is highly developed with respect to permissions and access control. In the next chapter we will be studying OFBiz Permissions and the Service Engine.

Where to buy this book

You can buy Apache OFBiz Development from the Packt Publishing website:
`http://www.packtpub.com/apache-ofbiz-development-beginners-tutorial/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/apache-ofbiz-development-beginners-tutorial/book