Service Engine Guide

Service Engine Guide

Written by: Andy Zeneski, jaz@apache.org
Edited by: Les Austin

Table of Contents

Related Documents:

- Service Engine Configuration Guide
- XML File Definitions

Introduction

Services are independent pieces of logic which when placed together process many different types of business requirements. Services can be of many different types: Workflow, Rules, Java, SOAP, Groovy, etc. A service with the type Java is much like an event where it is a static method, however with the Services Framework we do not limit to web based applications. Services require input parameters to be in a Map and the results are returned in a Map as well. This is nice since a Map can be serialized and stored or passed via HTTP (SOAP).

Services are defined through the Service Definition and are assigned to a specific Service Engine. Each Service Engine is responsible for invoking the defined service in an appropriate way. Since services are not tied to web based applications this allows services to run when there is no response object available. This allows services to be scheduled to run at specific times to run in the background via the Job Scheduler.

Services have the ability to call other services. So, chaining small services together to accomplish a larger task makes reusing existing services much easier.

Services which are declared in a component (service-resource in ofbiz-component.xml file) are reachable from anywhere in OFBiz, and even outside using the export feature. Though this possibility is not commonly used (no examples OOTB), it's also possible to create services which are specific to an application: restricted to be available only in that application. For that, you put the service definition and implementation files under the WEB-INF directory. You can also override a service by using the same name down in the deployment context (which is first framework, then themes, then applications, then specialpurpose, then hot-deploy). This is handy, but beware if not wanted...

When used in a web application services are available to web events, which allow events to stay small and reuse existing logic in the Services Framework. Also, services can be defined as 'exportable' which means they are allowed to be accessed by outside parties. Currently there is a SOAP EventHandler which allows services to be made available via SOAP. Other forms of remote invocation may be added to the framework in the future.

Service Dispatcher

The Service Dispatcher handles dispatching services to the appropriate Service Engine where it is then invoked. There is exactly one ServiceDispatcher for each Entity Delegator. If there are multiple delegators in an application there will also be multiple dispatchers. The ServiceDispatcher is accessed via a LocalDispatcher. There can be many LocalDispatchers associated with a ServiceDispatcher. Each LocalDispatcher is uniquely named and contains its own list of service definitions. When creating an instance of a LocalDispatcher, a DispatchContext is also created and passed to the ServiceEngine.

A LocalDispatcher is associated with an application. Applications never talk directly to the ServiceDispatcher. The LocalDispatcher contains an API for invoking services, which are routed through the ServiceDispather. However, applications may be running in different threads than the actual ServiceDispatcher, so it is left to the LocalDispatcher to keep a DispatchContext which among other things keeps a reference to the applications classloader.

Dispatch Context

The DispatchContext is created by the LocalDispatcher upon instantiation. This is the runtime dispatcher context. It contains necessary information to process services for each dispatcher. This context contains the reference to each of the service definition files, the classloader which should be used for invocation, a reference to the delegator and its dispatcher along with a 'bag' of user defined attributes. This context is passed on to each service when invoked and is used by the dispatcher to determine the service's model.

Service Engine

This is where the service is actually invoked. Each service has an engine name assigned in its definition. This engine name is mapped via the serviceengine.xml file and is instantiated by the GenericEngineFactory when called upon. Third-party engines are supported and must follow the GenericEngine interface when implemented. See the Service Engine Configuration Guide for details on defining engines.

It is the job of the engine to handle invocation of both synchronous and asynchronous services. Engines which use the Job Scheduler for asynchronous services can extend the GenericAsyncEngine.

Job Scheduler

The overhauled job scheduler is now integrated with the services framework. This is the most appropriate place for a scheduler. Since it cannot be guarantee that an HttpServletRequest and HttpServletResponse object will be available when a job is ready to run, it does not make sense to integrate with the web controller. Plus, this feature is most useful when not limited to web environments.

The scheduler is a multi-threaded component with a single thread used for job managing/scheduling and separate threads used for invocation of each service. When a job is scheduled to run, the scheduler will call the service dispatcher associated with the job to invoke the service in its own thread. This will prevent long or time consuming jobs from slowing down other jobs in the queue.

The scheduler now supports the iCalendar rule structure for recurrence. The jobs are no longer stored in an XML file and each is part of a ServiceDispatcher. There is one Job Scheduler for each ServiceDispatcher (which means there is only one per GenericDelegator as well).

How it works:

The best usage example of the scheduler is an asynchronous service call. When an asynchronous service is invoked, it is passed to the Job Scheduler to be queued to run. A recurrence entry is created (RecurrenceInfo and RecurrenceRule entities are created), the job is stored, (JobSandbox entity is created) and the context (Map) is serialized and stored (RuntimeData entity is created). The scheduler then adds the job to the top of the list of scheduled jobs (asynchronous services do not have any delay time) and invoked.

Jobs are no longer defined in an XML file. This has been moved to the JobSandbox entity. There is a web based client in planning for adding predefined jobs to the queue, but currently the entities will have to be created by hand.

Service Definition

Services are defined in *Service Definition Files.* There are global definition files used for all service dispatchers as well as individual files associated only with a single dispatcher. When a LocalDispatcher is created it is passed a Collection of Arils which point to these definition files. These files are composed using XML and define the necessary information needed to invoke a service. The XSD of this file can be found here.

Services are defined with a unique name, associated to a specific service engine and the input and output parameters are defined explicitly. Below is an example of a service definition:

```
<service name="userLogin" engine="java" location="org.ofbiz.commonapp.security.login.LoginServices" invoke="
userLogin">
    <description>Authenticate a username/password; create a UserLogin object</description>
    <attribute name="login.username" type="String" mode="IN"/>
    <attribute name="login.password" type="String" mode="IN"/>
    <attribute name="userLogin" type="org.ofbiz.entity.GenericValue" mode="OUT" optional="true"/>
</service>
```

SERVICE ELEMENT

Attribute	Required?	Description	Default Value
name	Y	The unique name of the service.	
engine	Y	The name of the engine (defined in serviceengine.xml).	
location	N	The location or package of the service's class.	
invoke	N	The method name of the service.	
auth	N	Does this service require authorization? (true/false)	true
debug	N	Enable verbose debugging when calling this service?	true
default-entity-name	N	The default Entity to use for auto-attributes	
export	N	Is this service allowed to be accessed via SOAP/HTTP/JMS? (true/false)	false
max-retry	N	Sets the max number of times this service will retry when failed (persisted async only)	-1 (unlimited)
require-new-transaction	N	Require a new transaction for this service	true
semaphore	N	Defines how concurrent calls to this service should be handled: none: multiple calls to this service may run concurrently wait: while this service is running, queue any subsequent calls fail: while this service is running, fail any subsequent calls	none

semaphore-wait-seconds	N	When semaphore="wait" how many seconds to wait before failing the service call	300
sempahore-sleep	N	When semaphore="wait" how often (in milliseconds) to check if the waiting service call can be run	500
transaction-timeout	N	Override the default transaction timeout, only works if we start the transaction	0 (Use system default)
use-transaction	N	Create a transaction for this service (if one is not already in place)	true
validate	N	Do we validate the attributes found below for name and type matching? (true/false)	true

IMPLEMENTS ELEMENT

- **service** - The name of the service which this service implements. All attributes are inherited

ATTRIBUTE ELEMENT

- **name** - The name of this attribute
- **type** - The object type (String, java.util.Date, etc.)
- **mode** - Is this an input or output parameter or both (IN/OUT/INOUT)
- **optional** - Is this parameter optional (true/false)*underlined values are defaults

Above you can see the name of this service is *userLogin* and it uses the *java* engine. This service expects two **required** IN parameters: *login.username* and *login.password*. Required parameters are tested before the service is invoked. If the parameters does not match by name and object type the service is not invoked. Parameters which may or may not be sent to the service should be defined as **optional**. After the service is invoked, the OUT parameters are tested. Only required parameters are tested, however if a parameter is passed which is not defined as optional or required it will cause the service to fail. This service has not required OUT parameters, so the result is simply returned.

Automatic Entity Maintenance Service Definition

OFBiz can automatically define services for simple Create, Update and Delete (CrUD) operations for many entities.

Set the engine attribute to "entity-auto" and the invoke attribute to "create", "update", or "delete".

For example:

```
<service name="createInvoiceContactMech" engine="entity-auto" invoke="create" default-entity-name="
InvoiceContactMech">
    <description>Create a ContactMech for an invoice</description>
    <permission-service service-name="acctgInvoicePermissionCheck" main-action="CREATE"/>
    <auto-attributes include="pk" mode="IN" optional="false"/>
</service>
```

The entity-auto engine can implement the following Create operations:

1. a single OUT primary key for primary auto-sequencing
2. a single INOUT primary key for primary auto-sequencing with optional override
3. a multi-part primary key with all parts part IN except the last which is OUT only (the missing primary key is a sub-sequence mainly for entity association
4. all primary key fields IN for a manually specified primary key

For any more complex situation, write code for your own service instead of relying on an automatically defined one.

Usage

Internal Usage of the Services Framework is quite simple. In a Web Application the LocalDispatcher is stored in the ServletContext which can be accessed via the Session object in an Event. For non-web based applications you simply create a GenericDispatcher: GenericDelegator delegator = GenericDelegator.getGenericDelegator("default");
LocalDispatcher dispatcher = new GenericDispatcher("UniqueName",delegator);
Now we have a dispatcher which we can use to invoke services. To invoke the test service create a Map for the context which contains the IN parameter *message* *then invoke the service:*

```
Map context = UtilMisc.toMap("message","This is a test.");
Map result = null;

try {
    result = dispatcher.runSync("testScv", context);
} catch (GenericServiceException e) {
    e.printStackTrace();
}

if (result != null) {
    System.out.println("Result from service: " + (String) result.get("resp"));
}
```

Now look at the console and see what the test service has echoed.
The test service is located in core/docs/examples/ServiceTest.java You must compile this and place it in the classpath.

To schedule a service to run at a later time or to repeat use this:

```
// This example will schedule a job to run now.
Map context = UtilMisc.toMap("message","This is a test.");
try {
    long startTime = (new Date()).getTime();
    dispatcher.schedule("testScv", context, startTime);
} catch (GenericServiceException e) {     e.printStackTrace();


// This example will schedule a service to run now and repeat once every 5 seconds a total of 10 times.
Map context = UtilMisc.toMap("message","This is a test.");
try {
    long startTime = (new Date()).getTime();
    int frequency = RecurrenceRule.SECONDLY;
    int interval = 5;
    int count = 10;
    dispatcher.schedule("testScv", context, startTime, frequency, interval, count);
} catch (GenericServiceException e) {
    e.printStackTrace();
}
```

Advanced

There have been a number of 'Advanced' features added to the service engine. You will find examples, definitions and information on each below.

Interfaces

The interface service engine has been implemented to help with defining services which share a number of the same parameters. An interface service cannot be invoked, but rather is a defined service which other services inherit from. Each interface service will be defined using the interface engine, for example:

```
<service name="testInterface" engine="interface" location="" invoke="">
    <description>A test interface service</description>
    <attribute name="partyId" type="String" mode="IN"/>
    <attribute name="partyTypeId" type="String" mode="IN"/>
    <attribute name="userLoginId" type="org.ofbiz.entity.GenericValue" mode="OUT" optional="true"/>
</service>
```

**Note that the location and the invokefields are required in the DTD, so we simply leave these as empty strings when used as an interface.

Now that we have an interface we need to define a service which implements this interface

```
<service name="testExample1" engine="simple" location="org/ofbiz/commonapp/common/SomeTestFile.xml" invoke="
testExample1">
    <description>A test service which implements testInterface</description>
    <implements service="testInterface"/>
</service>
```

The testExample1 service will have the exact same required and optional attributes as the testInterface service. Any service which implements testInterface will also inherit the parameters/attributes. If needed, additional attributes can be added to a specific service by including the attribute tag along with the implementstag. You may also override an attribute by re-defining it after the implements tag.

ECAs

ECA (Event Condition Action) is much like a trigger. When a service is called, a lookup is performed to see if any ECAs are defined for this event. Events include before authentication, before IN parameter validation, before actual service invocation, before OUT parameter validation, before transaction commit, or before the service returns. Next each condition in the ECA definition is evaluated and if all come back as true, each action is performed. An action is just a service which must be defined to work with the parameters already in the service's context. There are no limit to the number of conditions or actions each ECA may define.

```
<service-eca>
    <eca service="testScv" event="commit">
        <condition field-name="message" operator="equals" value="12345"/>
        <action service="testBsh" mode="sync"/>
    </eca>
</service-eca>
```

Tip: Note that you may use the set operation to rename a parameter for the triggered service or insert values for instance...

```
    <eca service="setCustRequestStatus" event="commit">
        <condition field-name="oldStatusId" operator="not-equals" value="CRQ_ACCEPTED"/>
        <condition field-name="statusId" operator="equals" value="CRQ_ACCEPTED"/>
        <set field-name="bodyParameters.custRequestId" env-name="custRequestId"/>
        <set field-name="bodyParameters.custRequestName" env-name="custRequestName"/>
        <set field-name="partyIdTo" env-name="fromPartyId"/>
        <set field-name="emailTemplateSettingId" value="CUST_REQ_ACCEPTED"/>
        <action service="sendMailFromTemplateSetting" mode="sync" />
    </eca>
```

ECA tag

Attribute Name	Required?	Description
service	Y	The name of the service this ECA is attached to.
event	Y	The event on which this ECA will run can be (before): auth, in-validate, out-validate, invoke, commit, or return.
run-on-error	N	Should this ECA run if there is an error in the service (default false)

The eca element should also have 0 or more condition or condition-field elements and 1 or more action elements.

Condition tag

Attribute Name	Required?	Description
map-name	N	The name of the service context field that contains the map that the field to be validated will come from. If not specified the field-name will be treated as a service context field name (an env-name).
field-name	Y	The name of the map field that will be compared.
operator	Y	Specified the comparison operator must be one of the following: less, greater, less-equals, greater-equals, equals, not-equals, or contains.
value	Y	The value that the field will compared to. Must be a String, but can be converted to other types.

type	N	The data type to use for the comparison. Must be one of the following: String, Double, Float, Long, Integer, Date, Time, or Timestamp. If no type is specified the default will be String.
format	N	A format specifier to use when converting String objects to other data types, mainly Date, Time and Timestamp.

Condition-field tag

Attribute Name	Required?	Description
map-name	N	The name of the service context field that contains the map that the field to be validated will come from. If not specified the field-name will be treated as a method environment field name (an env-name).
field-name	Y	The name of the map field that will be compared.
operator	Y	Specified the comparison operator must be one of the following: less, greater, less-equals, greater-equals, equals, not-equals, or contains.
to-map-name	N	The name of the service context field that contains the map that the field to be compared will come from. If left empty will default to the map-name, or the method environment if map-name is also unspecified.
to-field-name	N	The name of the to-map field that the main field will be compared to. If left empty will default to the field-name.
type	N	The data type to use for the comparison. Must be one of the following: String, Double, Float, Long, Integer, Date, Time, or Timestamp. If no type is specified the default will be String.
format	N	A format specifier to use when converting String objects to other data types, mainly Date, Time and Timestamp.

Action tag

Attribute Name	Required?	Description
service	N	The name of the service this action will invoke.
mode	Y	The mode in which this service should be invoked. Can be sync or async. Note that async actions will not update the context even when set to true.
result-to-context	N	Should the results of the action service update the main service's context. Default true.
result-to-result	N	Should the results of the action service update the main service's results. If true, the action service's messages will be appended to the main service's messages. Default false.
ignore-error	N	Ignore any errors caused by the action service. If true the error will cause the original service to fail. Default true.
persist	N	The action service store / run. Can be true or false. Only effective when mode is async. Default false.

Service Groups

Service groups are a set of services which should run when calling the initial service. You define a service using the group service engine, and include all the parameters/attributes needed for all the services in the group. The location attribute is not needed for groupservices, the invoke attribute defines the name of the groupto run. When this service is invoked the group is called and the services defined in the group are called as defined.

The group definition is very simple, it contains a group elements along with 1 or more service elements. The group element contains a name attribute and a mode attribute used to define how the group will function. The service element is much like the action element in an ECA, the difference being the default value for result-to-context.

```
<service-group>
    <group name="testGroup" send-mode="all">
        <invoke name="testScv" mode="sync"/>
        <invoke name="testBsh" mode="sync"/>
    </group>
</service-group>
```

Group tag

Attribute Name	Required?	Description
name	Y	The name of the service this action will invoke.
send-mode	N	The mode in which the service(s) should be invoked. The options are: none, all, first-available, random, or round-robin. The default is all.

Invoke tag

Attribute Name	Required?	Description
name	N	The name of the service this action will invoke.
mode	Y	The mode in which this service should be invoked. Can be sync or async. Note that async actions will not update the context even when set to true.
result-to-context	N	Should the results of the action service update the main service's context. Default false.

Route Services

Route services are defined using the routeservice engine. When a route service is called, there is no invoke performed, but all ECAs defined will run during the proper events. This type of service is not used very often, but can be used to 'route' to other services by utilizing the ECA options available to services.

HTTP Services

Using HTTP services is a way of invoking remote services defined on other systems. The local definition should match that of the remote definition, but the engine used should be http, the location should be a fully qualified URL to the httpService event running on the remote system, and the invoke should be the name of the service on the remote system you are requesting to be run. The remote system must have the httpService event mounted for HTTP services to be accepted. By default the commonapp web application has this event mounted to receive requests for services. Services on the remote system must have export set to true in order to allow the service to be run remotely. HTTP services by nature are synchronous.

JMS Services

JMS services are much like HTTP services, except the service request is sent to a JMS topic/queue. The engine should be set to jms, the location should be the name of the jms-service defined in the serviceengine.xml file (see Service Config document), and the invoke should be the name of the service on the remote system you are requesting to run. JMS services by nature are asynchronous.