Core: Developer's Guide

Overview
APIs
Programmer's Tutorial
>Java Programmer's Guide: Getting Started
Java Programmer's Guide: Additional APIs
C Programmer's Guide
Performance Guide
Security Support
Test Framework

Java Programming Guide: Getting Started

This guide describes programming with Grid services in Java. It covers both client- and server-side programming, including example code and use cases. It does not provide a complete reference to all of the APIs. Please refer to the Javadocs for such information.

This guide only covers the Core framework. For more information on EJB, security, and higher-level services (such as Managed Job Service, RFT, and the Index Service) support, see separate documentation. For installation, deployment, and development environment documentation see the Installation Guide.

This guide is structured into two pages. The first page, Getting Started, describes how to write, deploy, and access a simple Grid service in the framework. The second part, Additional APIs, describes some additional APIs and features provided by our framework for more advanced service development.

Basic knowledge of Java and Ant (http://jakarta.apache.org/ant) is assumed in this guide. We also assume that you are familiar with the basic OGSA environment.

This page contains the following topics:

- Writing a Service
- Writing a Client
Testing the Service

Writing a Service

The following steps are involved in writing a Grid service:

- Step 1. Provide a Service Interface
- Step 2. Generate Grid Service Support Code
- Step 3. Implement the Service
- Step 4. Deploy the Service

The complete source code for this example is available in the guide directory in your framework distribution.

**Step 1. Provide a Service Interface**

There are two approaches to providing an interface for your service that is to be exposed to remote clients. You can either write the interface in Java, and generate the WSDL interface, or you can provide the interface using a WSDL port type definition. We will look closer at the two approaches next.

**Java Interface**

We define the following Java interface:

```java
package org.globus.ogsa.guide.impl.guide;

public interface Counter {
    public int add(int value);
    public int subtract(int value);
    public int getValue();
}
```

See `guide/src/org/globus/ogsa/guide/impl/Counter.java` for the full example.

This approach should be used with care because some complex Java types do not map very well into WSDL and could thus impair the interoperability of your service. It is however useful if you want to expose legacy code written in Java as Grid services, without requiring any programmatic effort.

**WSDL PortType Interface**

When providing a WSDL interface you only need to provide the abstract definition of the service including the types, message, and portType parts of WSDL. The binding, and service part will be generated for you by our tools. This approach may at first glance look more complicated and verbose than the Java Interface approach. But the more complicated your service gets the likelier it is that you will come across types and constructs in Java that don't have a clear mapping to XML. Defining the interface in WSDL also makes it easier to reuse designs in a language neutral way. Someone might for
instance write a Grid service in C, and define a set of types for that service in XML Schema. Using the WSDL PortType Interface approach will make it very easy to embed all these types and even extend the interfaces from this service in your new service to be developed in Java.

Note in the example below that we define port types in the gwsdl namespace, this is done to circumvent the fact that WSDL 1.1 does not allow portType inheritance nor embedded extensions. WSDL 1.2, however, does allow these constructs, and when that specification is released and the OGSI specification is fully based on it, the gwsdl namespace will go away and be replaced by the wsdl 1.2 namespace.

Here are some excerpts from the wsdl definition:

```xml
<types>
  ...
  <xsd:element name="add">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:int"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  ...
</types>
...
<message name="AddInputMessage">
  <part name="parameters" element="tns:add"/>
</message>
...
<gwsdl:portType name="CounterPortType" extends="ogsi:GridService">
  <operation name="add">
    <input message="tns:AddInputMessage"/>
    <output message="tns:AddOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
  </operation>
</gwsdl:portType>
```

The types section defines the add operation to have one parameter called value of type int. The message section ties the input message to the concrete xml element to be used as payload of the operation. Finally the portType section defines that this portType should also expose all the operations defined in the ogsi:GridService portType. Also note that the standard osgi fault can be thrown from this operation, which is good practice to declare even if you don't have any application specific faults.

See guide/schema/counter_port_type.gwsdl for the full example.

To produce a complete wsdl 1.1 definition from this gwsdl port type definition you would need to run it through the GWSDL2WSDL and generateBinding tools as follows:
GWSDL2WSDL creates a WSDL 1.1 portType containing all the operations inherited from the gwsdl definition. The generateBinding tool generates the wsdl:binding and wsdl:service parts for the portType definition. Currently we only support doc/literal SOAP 1.1 bindings.

**Step 2. Generate Grid Service Support Code**

We provide high level ant task and xml batch file based tools to simplify the generation of the required stub and support code for hosting your service as an OGSI compliant Grid service. Although you may not use these tools directly, all the tools are centered around two tools primitives generateWSDL and generateStubs, which are used to generate WSDL from a Java interface and Java stubs for a WSDL interface respectively. However in order to understand better how the higher level tools work we first briefly explain these two basic tools. For the best tools support we recommend that you use the GWSDL2WSDL, generateBinding, and generateStubs primitives when generating a service from a WSDL file. However, if you start from a java interface we recommend that you use the higher level bottomUp tool described below. The generateWSDL tool is described below for completeness.

**Generate WSDL from Java**

An existing Java interface can be run through a Java to WSDL tool using the following ant command:

```
<ant antfile="${build.services}" target="generateWSDL">
    <property name="interface.package" value="org.globus.ogsa.guide.impl"/>
    <property name="interface.name" value="Counter"/>
    <property name="generated.dir" value="guide"/>
</ant>
```

See guide/build.xml for the full example.

Note that the ${build.services} property has to point to the location of build-services.xml shipped with the framework. This command will generate a WSDL file and populate it with a binding supporting the required GridService PortType.

**Generate Stubs form WSDL**
After you have obtained a WSDL interface either by generating it from a java interface or from a wsdl portType definition (Step 2 above), the next step is to generate Java stubs to handle all the serialization/deserialization of your data to/from XML.

See guide/build.xml for the full example.

This command will generate the JAX-RPC compliant interfaces to be used both on the client and on the server side. Client side stub implementations of these interfaces will also be generated.

**Bottom Up vs Top Down**

We provide two higher level tools based on these primitives called *Bottom Up* and *Top Down*. Bottom Up refers to an approach when you start out from a legacy java application and want to generate a Grid service layer on top of it. Top Down refers to an approach when you either get the WSDL from an implementation in another environment or standards community, or you write the WSDL yourself, and want to generate the java mapping for this interface. For a detailed tutorial with examples of how to use these tools please see the
**Tools Guide**. Note that if you use these high level tools you can skip step 3 and 4 below. It is all generated for you.

**Step 3. Implement the Service**

**Inheritance Approach**

```java
public class CounterImpl extends GridServiceImpl implements CounterPortType {
    private int val = 0;

    public CounterImpl() {
        super("Guide Counter");
    }

    public int add(int val) throws RemoteException {
        this.val = this.val + val;
        return this.val;
    }

    public int subtract(int val) throws RemoteException {
        this.val = this.val - val;
        return this.val;
    }

    public int getValue() throws RemoteException {
        return this.val;
    }
}
```

See guide/src/org/globus/ogsa/guide/impl/CounterImpl.java and guide/src/org/globus/ogsa/guide/impl/WSDLCounterImpl.java for the full examples of implementations of the *Java Interface* approach and the *WSDL PortType Interface* approach respectively. Note the only thing that differs between the two implementations is the namespace from where you pick up the generated counter interface to implement.

Note that the CounterPortType is the endpoint interface generated in the previous step. All remotely available operations must be public and throw java.rmi.RemoteException as defined in the PortType interface. The GridServiceImpl class may be inherited from. It is provided by our framework and implements the OGSI defined GridService interface, along with other core Grid service behavior.

**If your service inherits from GridServiceImpl and overrides any of the standard GridServiceCallback methods such as postCreate(), activate(), deactivate(), and preDestroy() always make sure to invoke the super method of the overridden method, e.g. in your postCreate() implementation make sure to call super.postCreate()**.

**Operation Providers**

If you do not want any code to depend on implementation classes in our container you can also implement the service using the operation provider (aka delegation) approach. This design makes it easy to plug in various implementations of wsdl operations at deployment time. As reference points, the OGSI defined NotificationSource and Factory interfaces have been
implemented as operation providers in our framework. So to add in factory, or notification behavior to your service you only need to change the deployment descriptor of your service (described in more detail in the next section).

Below is the counter example above implemented as an operation provider:

```java
public class CounterProvider implements OperationProvider, GridServiceCallback {
    // Use the double wild card with care - all operations not in the OGSI
    // namespace will be delegated to this class by default
    private static final QName[] operations = new QName[]{new QName("", "+")
    private GridServiceBase base;
    private int val = 0;

    // Operation Provider methods
    public void initialize(GridServiceBase base) throws GridServiceException {
        this.base = base;
    }
    public QName[] getOperations() {
        return operations;
    }
    // Counter PortType methods
    public int add(int val) throws RemoteException {
        this.val = this.val + val;
        return this.val;
    }
    public int subtract(int val) throws RemoteException {
        this.val = this.val - val;
        return this.val;
    }
    public int getValue() throws RemoteException {
        return this.val;
    }
    // GridServiceCallback methods (optional)
    public void preCreate(GridServiceBase base) throws GridServiceException {
    }
    public void postCreate(GridContext context) throws GridServiceException {
    }
    public void activate(GridContext context) throws GridServiceException {
    }
    public void deactivate(GridContext context) throws GridServiceException {
    }
    public void preDestroy(GridContext context) throws GridServiceException {
    }
}
```

A provider needs to specify all the operation QNames (namespace and local name as defined in wsdl) to implement. Note that we allow the wildcard '*' to be used to specify that all operations from a certain namespace are implemented. An empty string namespace means that operations in all namespaces apart from the built-in OGSI namespace are implemented. These operations need to be returned in a getOperations() callback. The Provider also needs to provide an implementation of the initialize method which is used to bootstrap the provider and to associate it with a GridServiceBase. The GridServiceBase object is an implementation of the core Grid service
behaviors. GridServiceImpl in the previous example is an example of a GridServiceBase implementation.

See guide/src/org/globus/ogsa/guide/impl/CounterProvider.java for the complete example.

Step 4. Deploy the Service

This step consists of three sub tasks 1) write a deployment descriptor configuring your service, and 2) create a gar package of the configuration along with your implementation, 3) deploy the gar package into a Grid service hosting environment

1) Write a deployment descriptor

```xml
<?xml version="1.0" encoding="UTF-8"?>
<deployment name="defaultServerConfig" xmlns="http://xml.apache.org/axis/wsdd/
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
<service name="guide/counter/CounterProviderFactoryService" provider="Han"
<parameter name="name" value="Guide Counter Provider Factory"/>
<parameter name="instance-name" value="Guide Counter Proviider Counter"/>
<parameter name="instance-schemaPath" value="schema/guide/Counter/counter_"
<parameter name="instance-baseClassName" value="org.globus.ogsa.guide.Counter"
<parameter name="instantiation-operationProviders" value="org.globus.ogsa.impl.ogsi"
<parameter name="instance-persistent" value="true"/>
<parameter name="baseClassName" value="org.globus.ogsa.impl.ogsi.PersistentGridServiceImpl"
<parameter name="handlerClass" value="org.globus.ogsa.handlers.RPCURIProvider"
<parameter name="className" value="org.gridforum.ogsi.NotificationFactory"
<parameter name="allowedMethods" value="*"/
<parameter name="factoryCallback" value="org.globus.ogsa.impl.ogsi.DynamicFactoryCallbackImpl"
<parameter name="operationProviders" value="org.globus.ogsa.impl.ogsi.FactoryProvider
org.globus.ogsa.impl.ogsi.NotificationSourceProvider"/>
</service
</deployment>
```

See guide/guide-config.wsdd for the complete example.

Here is a more detailed description of the various parts of the deployment descriptor:

(parameters prefixed with `instance-` concern instances created by a factory, other parameters belong to the factory configuration)

Note that for factories almost all the non-instance scoped parameters are boiler-plate configuration and will be the same for most of your factories.

<table>
<thead>
<tr>
<th><code>&lt;service name=xxx...</code></th>
<th>Defines the remotely accessible name for your service</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Your service factory URL will have the form of</code></td>
<td></td>
</tr>
</tbody>
</table>
A Grid service can be declared with the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hosting environment URL</td>
<td><code>&lt;hosting environment URL&gt;/xxx, where hosting environment URL typically is http://&lt;host&gt;:&lt;port&gt;/ogsa/services</code></td>
</tr>
<tr>
<td>name</td>
<td>Optional.</td>
</tr>
<tr>
<td></td>
<td>Defines the human readable description of this service to be exposed to e.g. admin clients.</td>
</tr>
<tr>
<td>className</td>
<td>Specifies a class or an interface that has public methods corresponding to all wsdl operations.</td>
</tr>
<tr>
<td></td>
<td><strong>Note that all the operations available in operation providers and base implementations must be exposed in this class or interface.</strong></td>
</tr>
<tr>
<td>baseClassName</td>
<td>Optional.</td>
</tr>
<tr>
<td></td>
<td>Specifies what class implements this service. If it is the same as className it can be left out.</td>
</tr>
<tr>
<td>operationProviders</td>
<td>Optional.</td>
</tr>
<tr>
<td></td>
<td>Defines the list of classes that are to be loaded into this service as operation providers.</td>
</tr>
<tr>
<td></td>
<td>The list items are separated by spaces.</td>
</tr>
<tr>
<td></td>
<td>The order of the providers is the order in which they will be initialized.</td>
</tr>
<tr>
<td>persistent</td>
<td>This sets the Grid service apart from other web services you may have configured in your hosting environment.</td>
</tr>
<tr>
<td></td>
<td>If this flag is not present, your service will be treated as a regular Web service.</td>
</tr>
<tr>
<td>schemaPath</td>
<td>Points the framework to a place where a template WSDL description of the service exists (in this case the standard factory description provided by the framework.)</td>
</tr>
<tr>
<td>instance-</td>
<td>If deploying a factory (like in this example), sets the</td>
</tr>
<tr>
<td><strong>schemaPath</strong></td>
<td>instance-schemaPath to a location describing the interface of the service that the factory can create. In our case this WSDL description was generated in step 1 above</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>handlerClass</strong></td>
<td>Specifies what dispatcher to use on the server side. The default one we provide in our framework dispatches into the local hosting environment based on the URI of the incoming request and is called org.globus.ogsa.handlers.RPCURIProvider</td>
</tr>
<tr>
<td><strong>factoryCallback</strong></td>
<td>Points to a class that implements the FactoryCallback interface used to create service instances.</td>
</tr>
<tr>
<td><strong>handleProtocol</strong></td>
<td>Optional. Indicates the protocol used to publish handles. Only https and http are allowed in the current OGSA specification. The default value is http.</td>
</tr>
<tr>
<td><code>&lt;protocol&gt;</code></td>
<td><strong>Note:</strong> If a non default protocol is selected, it is recommended to accompany this configuration with a global configuration of <code>&lt;protocol&gt;</code>Port. See above.</td>
</tr>
<tr>
<td><strong>lifecycle</strong></td>
<td>Optional. Indicates whether instances are to be checkpointed into the deployment descriptor to maintain state between server lifecycles.</td>
</tr>
</tbody>
</table>

2) Package your configuration, schemas and code into a gar package

```xml
<ant antfile="${build.packages}" target="makeGar">
  <property name="gar.name" value="${build.lib}/guide.gar"/>
  <property name="garlib.dir" value="${build.lib}"/>
  <property name="garserverdeployment.file" value="guide-config.wsdd"/>
  <property name="garschema.origin" value="${build.schema}/guide"/>
  <property name="garschema.path" value="guide"/>
</ant>
```

See guide/build.xml for the full example.

This sample ant task packages the code (which we assumed you have compiled, e.g with the ant javac target or manually, see full example in the
distribution for details). The garserverdeployment.file property points to the deployment descriptor file provided in the previous step.

3) Deploy the gar package into a Grid service Hosting Environment

In the distribution directory of your ogsa installation make the following command line call:

```
ant deploy -Dgar.name=<path to gar created in previous step>
```

To deploy the samples in this guide type `ant deployGuide`.

Writing a Client

The standard JAX-RPC interfaces can be used to access a Grid service. However, for convenience, since JAX-RPC currently has no knowledge of GSRs or GSHs we provide some extensions to simplify access to a Grid service.

**JAX-RPC Example**

```java
OGSIServiceLocator factoryLocator = new OGSIServiceLocator();
Factory factory = factoryLocator.getFactoryPort(new URL(url));
GridServiceFactory gridFactory = new GridServiceFactory(factory);
LocatorType locator = gridFactory.createService();
...
CounterServiceLocator counterLocator = new CounterServiceLocator();
CounterPortType counter = counterLocator.getCounterPort(new URL(instanceUrl)
int val = counter.add(2);
```

The problem with this approach is that there is no easy way of feeding the GSR that the factory returns into the instance proxy generation, to make sure you are accessing the service you just created. Note the GridServiceFactory is a wrapper utility we provide to simplify invocations on OGSI factories.

**Extended JAX-RPC Example**

```java
OGSIServiceGridLocator gridLocator = new OGSIServiceGridLocator();
Factory factory = gridLocator.getFactoryPort(handle);
GridServiceFactory gridFactory = new GridServiceFactory(factory);
LocatorType locator = gridFactory.createService();
CounterServiceGridLocator counterLocator = new CounterServiceGridLocator();
CounterPortType counter = counterLocator.getCounterPort(locator);
int val = counter.add(2);
```

See `guide/src/org/globus/ogsa/guide/impl/CounterClient.java` for the full example.

Note that the `<service>GridLocator is also generated from the WSDL definition of the service together with the standard JAX-RPC ServiceLocator, so it does not involve any extra effort for a programmer to use this approach.
The GridLocator is able to do OGSI based handle to reference resolution, and accepts both an OGSI Handle and an OGSI Locator (returned from a factory creation) as input to the proxy creation.

Testing the Service

**Command Line Client**

1. Make sure you have started a grid service container e.g. using `globus-start-container`
2. Create service instance using `ogsi-create-service <server url>/<sample factory service name> |id|`. The `<id>` is used to distinguish between instances you create under the same factory, and may be omitted in which case the server generates this id. The `<server url>` is typically `http://<host>:<port>/ogsa/services`. The `<sample factory service name>` must be the same name as defined in server-config.wsdd.
   Example: `ogsi-create-service 
   http://localhost:8080/ogsa/services/guide/counter/CounterFactoryService calc`
3. Run your command line client created as described in section 2, giving it the URL of the endpoint returned by the `ogsi-create-service` call (make sure your environment is set properly using the `setenv scripts`).
   Example: `java org.globus.ogsa.guide.impl.CounterClient 
   http://localhost:8080/ogsa/services/guide/counter/CounterFactoryService calc add 10`

**GUI client**

If you want to test your service in the ServiceBrowser GUI framework you would have to provide a GUI panel implementation for your service port type(s). See `org/globus/ogsa/gui/CounterPortTypePanel` in the ogsa distribution samples for an example. You would also need to add a mapping to your panel in the `<ogsa root>/client-gui-config.xml` file.

To test your GUI client:

1. Start the gui client by typing `globus-service-browser`
2. Locate your factory in the Service Group Entry Inspection panel, and double click on its entry
3. Create an instance in the Factory panel and now your custom Panel should be displayed if all goes well