

# DOSGi DS Demo page

This page describes the CXF Distributed OSGi with Declarative Services demo.

The Declarative Services demo uses a DS implementation to create a remotd OSGi service from a DS component. The consumer side uses DS to create a component that consumes the remote OSGi service. By using Declarative Services, you don't need to write code to interact with the OSGi Service Registry. That's all handled through injection which hugely simplifies the code.

Declarative Services is similar to Spring-DM/OSGi Blueprint in that service dependencies are satisfied through injection. There are a few differences as well. DS is lighter weight than Spring-DM but also has less features. Declarative Services have been part of the OSGi specifications since version 4.0.

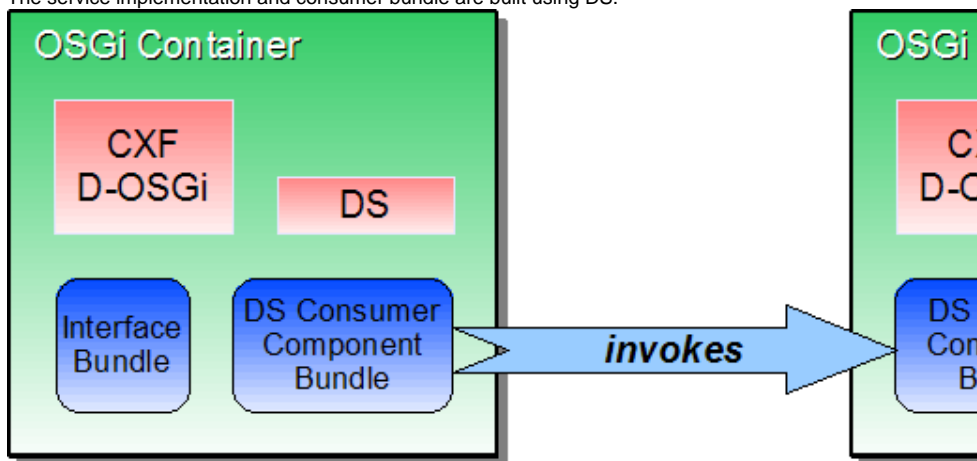
This demo can be used with any DOSGi distribution, in this document the single-bundle distribution is used with the Equinox implementation of DS.

## DEMO DESIGN

This demo is quite similar to the Spring-DM demo and the Greeter demo in structure. It consists of 3 bundles:

- An interface bundle defining the Adder Service interface.
- An Adder Service implementation bundle.
- An Adder Service consumer bundle.

The service implementation and consumer bundle are built using DS.



The Adder Service interface is as follows:

```
public interface AdderService {  
    int add(int a, int b);  
}
```

## THE ADDER SERVICE IMPLEMENTATION

The service implementation provides a simplistic implementation of the AdderService interface, which is instantiated as a DS component.

In the `OSGI-INF/component.xml` file the AdderServiceImpl is instantiated and registered with the OSGi service registry with the distribution properties. These properties instruct. Distributed OSGi into making the service available on `http://localhost:9090/adder`.

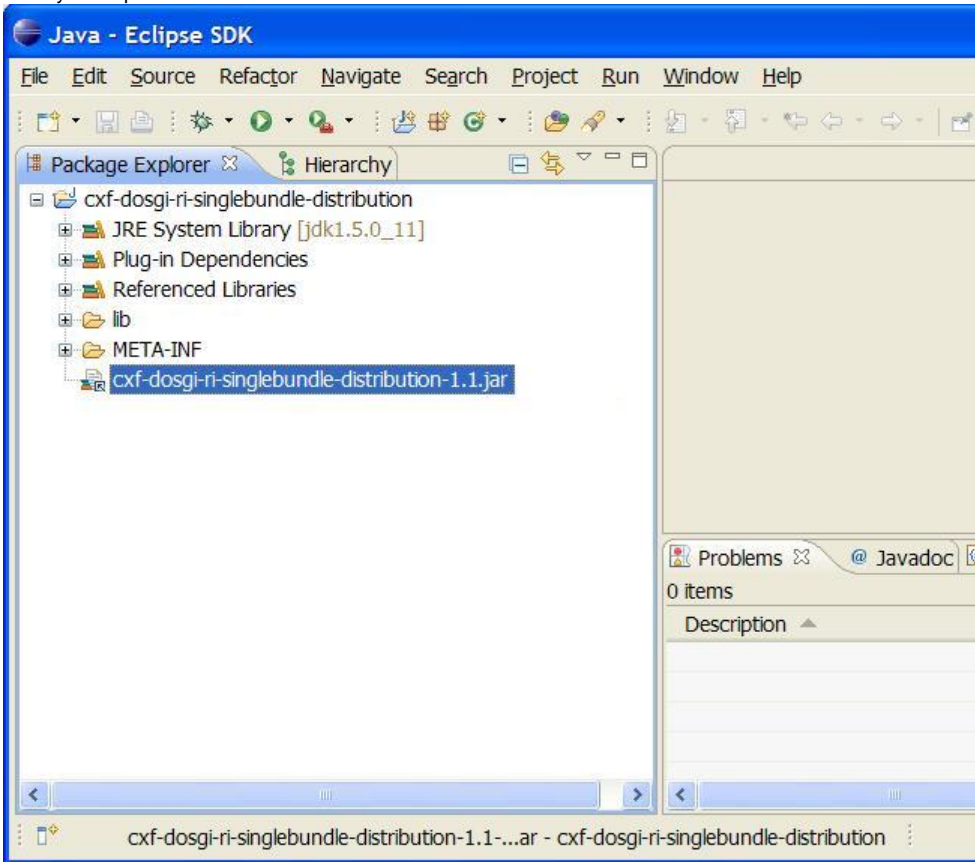
```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="DS  
Service Sample">  
    <implementation class="org.apache.cxf.dosgi.samples.ds.impl.AdderServiceImpl"  
/>  
  
    <property name="service.exported.interfaces" value="*" />  
    <property name="service.exported.configs" value="org.apache.cxf.ws" />  
    <property name="org.apache.cxf.ws.address" value="http://localhost:9090  
/adder" />  
  
    <service>  
        <provide interface="org.apache.cxf.dosgi.samples.ds.AdderService"/>  
    </service>  
</scr:component>
```

Note that the `META-INF/MANIFEST.MF` file needs to contain a special DS header that tells the system where to find this file. In case of this demo, this header is added by the Maven build system. The header used by the demo is:

```
Service-Component: OSGI-INF/component.xml
```

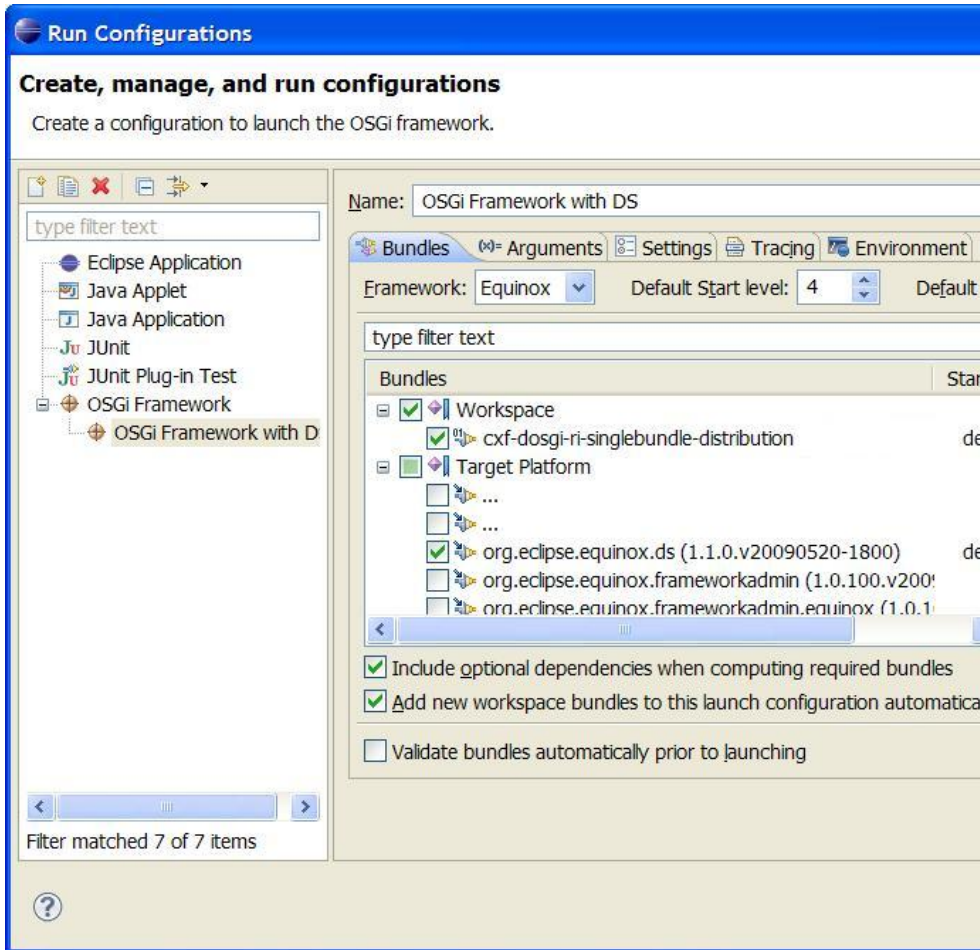
So let's install the server side in Equinox, together with the Equinox DS implementation. You can do this from the Equinox command line, but in this document I'll launch Equinox from within Eclipse (last tried with Eclipse 3.6).

I'm starting off by importing the Single Bundle Distribution as a binary project in Eclipse by going *File -> Import | Plug-ins and Fragments* and then select the directory that contains the single bundle distribution JAR file. My workspace now looks like this:



Next I'll create an OSGi Framework launch configuration that includes DS. To do this

1. *deselect* the 'Target Platform' tickbox in the Eclipse Launch configuration screen
2. select org.eclipse.equinox.ds
3. hit the 'Add Required Bundles' button



Now run the OSGi container, you will get a setup like this:

```
osgi> ss

Framework is launched.

id      State      Bundle
0       ACTIVE     org.eclipse.osgi_3.5.0.v20090520
1       ACTIVE     org.eclipse.equinox.util_1.0.100.v20090520-1800
2       ACTIVE     org.eclipse.osgi.services_3.2.0.v20090520-1800
3       ACTIVE     cxf-dosgi-ri-singlebundle-distribution
4       ACTIVE     org.eclipse.equinox.ds_1.1.0.v20090520-1800
```

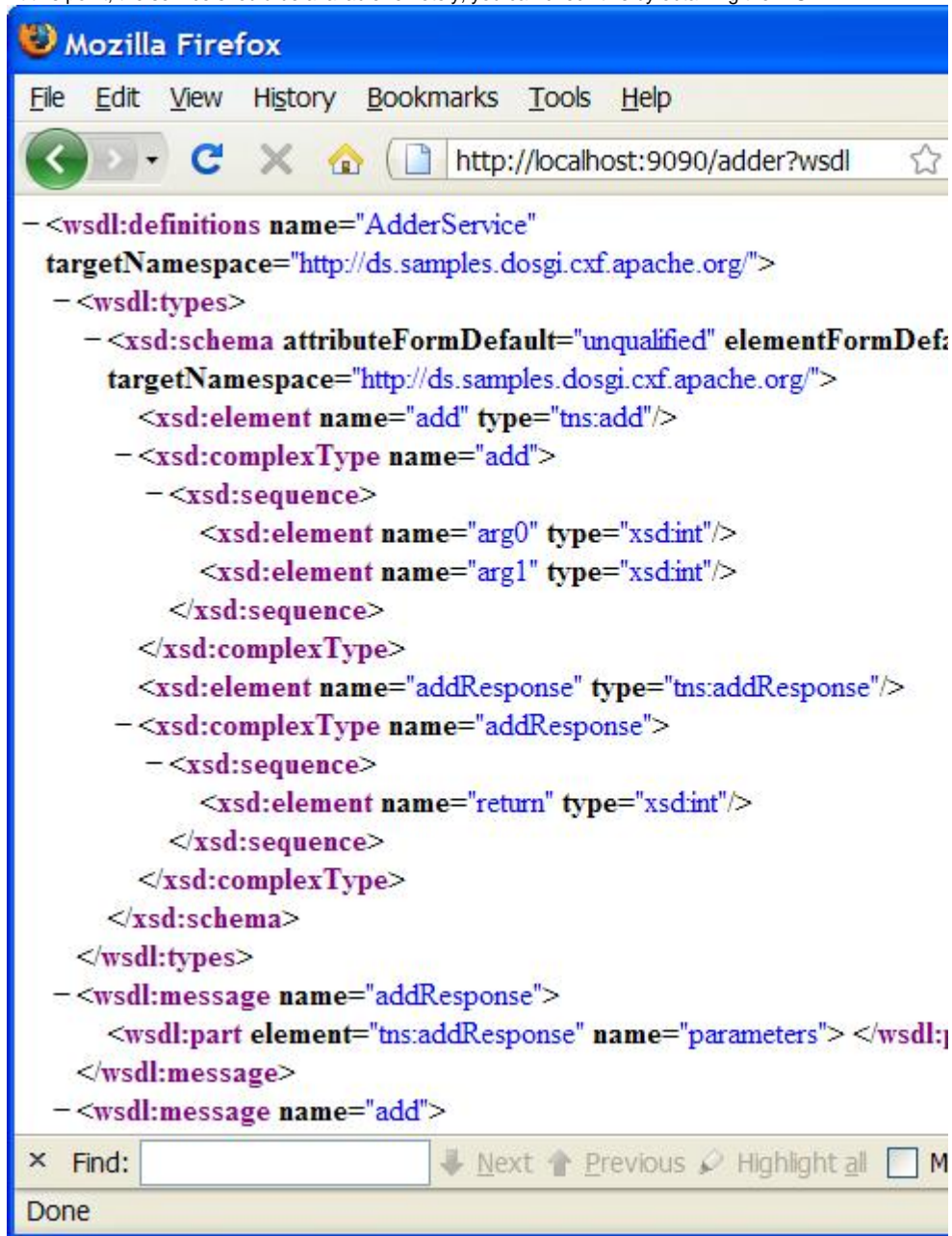
Now I can install the DOSGi DS bundles in the OSGi container directly from the maven repository.

```
osgi> install http://repo1.maven.org/maven2/org/apache/cxf/dosgi/samples/cxf-  
dosgi-ri-samples-ds-interface/1.2/cxf-dosgi-ri-samples-ds-interface-1.2.jar  
Bundle id is 5  
  
osgi> install http://repo1.maven.org/maven2/org/apache/cxf/dosgi/samples/cxf-  
dosgi-ri-samples-ds-impl/1.2/cxf-dosgi-ri-samples-ds-impl-1.2.jar  
Bundle id is 6  
  
osgi> start 6  
... log messages may appear ...
```



You can also import the maven projects of the DS demo into Eclipse, this would save you from installing it with a URL as above. To do this, check out the CXF/DOSGi source from SVN (<http://svn.apache.org/repos/asf/cxf/dosgi/trunk>), run `mvn eclipse:eclipse` and import all Eclipse projects under the `samples/ds` directory in Eclipse with *File -> Import | Existing Projects into Workspace*.

At this point, the service should be available remotely, you can check this by obtaining the WSDL:



The screenshot shows a Mozilla Firefox browser window with the address bar displaying `http://localhost:9090/adder?wsdl`. The main content area displays the WSDL XML document for an `AdderService`. The XML is color-coded and includes comments indicating the structure of the service definitions.

```
- <wsdl:definitions name="AdderService"
  targetNamespace="http://ds.samples.dosgi.cxf.apache.org/">
  - <wsdl:types>
    - <xsd:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
      targetNamespace="http://ds.samples.dosgi.cxf.apache.org/">
      <xsd:element name="add" type="tns:add"/>
      - <xsd:complexType name="add">
        - <xsd:sequence>
          <xsd:element name="arg0" type="xsd:int"/>
          <xsd:element name="arg1" type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:element name="addResponse" type="tns:addResponse"/>
      - <xsd:complexType name="addResponse">
        - <xsd:sequence>
          <xsd:element name="return" type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>
  - <wsdl:message name="addResponse">
    <wsdl:part element="tns:addResponse" name="parameters"></wsdl:part>
  </wsdl:message>
  - <wsdl:message name="add">
```

The bottom of the browser window shows a search bar with the text "Find:" and a "Done" button. Navigation buttons for "Next", "Previous", and "Highlight all" are also visible.

## THE ADDER SERVICE CONSUMER

The service consumer is also created using DS. DS creates an AdderConsumer component which is injected with a reference to the remote AdderService. Like in Spring, the injection is done by DS, which makes the code nice and simple. When the injection is done, the start() method is called.

```
public class AdderConsumer {
    private AdderService adder;

    public void bindAdder(AdderService a) {
        adder = a;
    }

    public void unbindAdder(AdderService a) {
        adder = null;
    }

    public void start(ComponentContext cc) {
        System.out.println("Using adder service: 1 + 1 = " + adder.add(1, 1));
    }
}
```

The client side bundle contains an `OSGI-INF/component.xml` which drives the component creation and injection:

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" activate="
start">
    <implementation class="org.apache.cxf.dosgi.samples.ds.consumer.
AdderConsumer"/>
    <reference interface="org.apache.cxf.dosgi.samples.ds.AdderService" name="
AdderService" cardinality="1..1" policy="dynamic" bind="bindAdder" unbind="
unbindAdder"/>
</scr:component>
```

As on the service provider side, the client side bundle needs to contain the DS header in the `META-INF/MANIFEST.MF`:

```
Service-Component: OSGI-INF/component.xml
```

As in the Greeter demo, the client side needs to be configured to know where the remote service actually is. This is one in the `OSGI-INF/remote-service/remote-services.xml` file:

```
<endpoint-descriptions xmlns="http://www.osgi.org/xmlns/rsa/v1.0.0">
  <endpoint-description>
    <property name="objectClass">
      <array>
        <value>org.apache.cxf.dosgi.samples.ds.AdderService</value>
      </array>
    </property>
    <property name="endpoint.id">http://localhost:9090/adder</property>
    <property name="service.imported.configs">org.apache.cxf.ws</property>
  </endpoint-description>
</endpoint-descriptions>
```

Install and run the consumer side of the demo in a separate Equinox instance (tip: you can duplicate the launch configuration used for the server side in the 'Run Configurations' dialog):

```
osgi> install http://repol.maven.org/maven2/org/apache/cxf/dosgi/samples/cxf-
dosgi-ri-samples-ds-interface/1.2/cxf-dosgi-ri-samples-ds-interface-1.2.jar
Bundle id is 5

osgi> install http://repol.maven.org/maven2/org/apache/cxf/dosgi/samples/cxf-
dosgi-ri-samples-ds-client/1.2/cxf-dosgi-ri-samples-ds-client-1.2.jar
Bundle id is 6

osgi> start 6
... log messages may appear, after a little while the following message
appears:
Using adder service: 1 + 1 = 2
```

The remote adder service has now been invoked. You will see the following line on the server side Equinox window:

```
Adder service invoked: 1 + 1 = 2
```

### Consumer Note

Some OSGi Declarative Services implementations don't explicitly register interest in the requested services with the OSGi Framework. They rather use a generic Service Tracker or Service Listener to track all available services. This doesn't provide the CXF-DOSGi implementation with the information about what services the consumer is looking for through the ListenerHook and hence it can't register the remote service on-the-fly. A simple workaround to this problem is to add an Activator to a bundle in the client-side framework (this activator could be in any bundle) which registers an explicit ServiceTracker for the remote service the DS component wants to be injected with. An example of such an Activator can be found [here](#).

In the future a more elegant solution to this problem will hopefully be provided.