Dependency Manager - Service Scopes

Contents

By default service providers are registered once in the osgi registry, and all service consumers share the same service provider instance. Now, you can control the scope of the provided services: From the provider side, a "scope" parameter is available for all types of DM components and allows to define the scope of the registered service.

The `scope` attribute has three enum values: SINGLETON, BUNDLE, PROTOTYPE

- SINGLETON: it's as before: your registered service is a singleton, meaning that the service must be instantiated and registered once, and all using services will share the same service instance of your component.
- BUNDLE: the service will be registered as a ServiceFactory, meaning an instance of the component must be created for each bundle using the service.
- PROTOTYPE: the service will be registered as a PrototypeServiceFactory, meaning the service is registered as a prototype scope service and an instance of the component must be created for each distinct service requester.

Scoped Services are supported by all kind of DM service components:

- Component
- Aspects
- Adapters
- Bundle Adapters
- Resource Adapters

When a consumer requests a service (using ServiceDependency), then DM will automatically dereference the service like this:

- if the service has a SERVICE_SCOPE service property matching SCOPE_PROTOTYPE, the DM will internally derefence the service using the ServiceObject API: so, the consumer will get its own copy of the requested service instance.

- else, DM will internally dereference the service, as before. When defining scoped component implementation, you can optionally define two special class fields in order to get injected with the client Bundle requesting the service, and the ServiceRegisgtration Object. Just use @Inject annotations in order to get injected with the client Bundle or the ServiceRegistration. You can also define a constructor which takes as argument the client bundle as well as the service registration.

Examples

So, here is a MyService component with PROTOTYPE scope, and each requester will get its own copy of MyService instance (the MyServiceImpl.start() method will be called for each MyServiceImpl instance):

```java
public class Activator extends DependencyActivatorBase {
    @Override
    public void init(BundleContext context, DependencyManager dm) throws Exception
{
        dm.add(createComponent()
            .setScope(ServiceScope.PROTOTYPE)
            .setInterface(MyService.class.getName(), null)
            .setImplementation(MyServiceImpl.class));
    }
}

public class MyServiceImpl implements MyService {
    void start() {
        // called on each MyService instance
    }
}
```

The MyServiceImpl, like with annotations, can define a constructor in order to be injected with the client bundle invoking the service and also the service Registration:

```java
 public class MyServiceImpl implements MyService {
     public MyServiceImpl(Bundle clientBundle, ServiceRegistration reg) { ... }
     void start() {
         // called on each MyService instance
     }
 }
```

(if you want to auto configure the client Bundle, and the ServiceRegistration, then simply define class fields, they will be auto injected, unless you disable auto configuraiton for Bundle/ServiceRegistration using Component.setAutoConfig(Class, boolean) methods.

Here is a Client component which simply depends on the MyService service using a basic DM activator (nothing special to do):

```java
public class Activator extends DependencyActivatorBase {
    @Override
    public void init(BundleContext context, DependencyManager dm) throws Exception
{
        dm.add(createComponent()
            .setImplementation(Client.class)
            .add(createServiceDependency()
                 .setService(MyService.class,
null).setRequired(true).setCallbacks("bind", "unbind"));
    }
}

public class Client {
    void bind(MyService service) {
        // our client is injected with a specific instance of the MyService
component
        // that is created for our Client.
        // If another component defines a service dependency on MyService, then the
other
        // component will get its own private copy of MyService component instance
    }
}
```

Example using ServiceObjects API

If now you want to control the creation of the MyService using raw OSGI ServiceObjects API, you can also do it like this:

```java
 public class Client {
     void bind(ServiceObjects<MyService> so) {
         MyService s1 = so.getService();
         MyService s2 = so.getService();
         ...
         so.ungetService(s1); // will deactivate the MyService s1 instance
         so.ungetService(s2); // will deactivate the MyService s2 instance
     }
 }
```

Scoped services and init/destroy callbacks

When you need to specify dynamic required dependencies from your component.init() method, the following mechanism will be used:

First, if your component defines an init callback, then one single component prototype instance singleton is created, as if the component is declared with SCOPE=SINGLETON. So, the prototype instance will be invoked in the init callback, but won't be called in start()/stop(). And when all dependencies are satisfied and injected (including the dynamic dependencies defined in the init method), then the ServiceFactory (or PrototypeServiceFactory) is registered. And when one client will request a component instance, then a clone will be created and returned.

Example of a scoped component which defines an init method:

```java
public class Activator extends DependencyActivatorBase {
    @Override
    public void init(BundleContext context, DependencyManager dm) throws Exception
{
        dm.add(createComponent()
            .setScope(ServiceScope.PROTOTYPE)
            .setInterface(MyService.class.getName(), null)
            .setImplementation(MyServiceImpl.class));
    }
}

public static class MyServiceImpl implements MyService {
    void init(Component comp) {
        // add required dependencies dynamically
    }

        void start() {
           // only called on clone, not on the prototype instance singleton
        }

        void stop() {
           // called on each clone, not on the prototype instance singleton
        }
}
```

So, if you don't specify an init callback then the prototype instance singleton won't be instantiated. Also,

Limitation when using DM ServiceDependency from API and ServiceObjects

When using DependencyManager ServiceDependency from the DM API (not using annotations), you have to know that the ServiceDependency always internally dereferences the service dependency, even if you specify a ServiceObjecs parameter in your bind method. If now you really want to disable the auto-deref ServiceDependency (because you want to directly use the ServiceObjects API), you must then use the "setDereference(false") method on your ServiceDependency: in this way, you tell DM to never dereference internally the scoped service. Here is an example:

```java
public class Activator extends DependencyActivatorBase {
    @Override
    public void init(BundleContext context, DependencyManager dm) throws Exception
{
        dm.add(createComponent()
            .setImplementation(Client.class)
            .add(createServiceDependency()
                .setService(MyService.class,
null).setRequired(true).setCallbacks("bind", "unbind")
                .setDereference(false));
    }
}

public class Client {
    void bind(ServiceObjects<MyService> so) {
        MyService s1 = so.getService();
        MyService s2 = so.getService();
        ...
        so.ungetService(s1); // will deactivate the MyService s1 instance
        so.ungetService(s2); // will deactivate the MyService s2 instance
    }
}
```

In the above example, the Activator defines the ServiceDependency using the ServiceDependency.setDereference(false) method because it's the Client.bind method which will create the MyService instances manually.

In the future, I will try to auto detect the signatures of the Client.bind method in order to never auto-dereference the injected service in case the bind method takes as argument a ServiceObjects (or a ServiceReference) method.

Contents