Invokers

Invokers allow you to customise how a particular method is executed. This is particularly useful if your underlying service objects are not plain javabeans and instead need to be created or looked up via a custom factory.

CXF does provide a number of bundled invokers to handle simple cases. One of these simple cases is when it is desirable to have a singleton for the service object. In this case, you would like to provide a single object instance that should be used for all service invocations. The provided BeanInvoker covers this functionality, and would be used as follows:

```java
Service service = ...;
service.setInvoker(new BeanInvoker(new MyCustomBean(someParams)));
```

You can access the underlying Service object in two ways. If you've created your service using a ServerFactoryBean, this will yield a Server object which can be used to gain access to the Service:

```java
ServerFactoryBean factory = new ServerFactoryBean();
....
Server server = factory.createServer();
Service service = server.getEndpoint().getService();
```

If you've created a JAX-WS Endpoint object, you can access the Service like this:

```java
EndpointImpl endpoint = (EndpointImpl) Endpoint.publish("http://host/service", new MyService());
....
Server server = endpoint.createServer();
Service service = server.getEndpoint().getService();
```

The following example illustrates how an invoker can be used to allow CXF to expose remote stateless session beans as a webservice. Given the method to invoke, this invoker will create a stateless session bean instance to invoke the method on. The same technique can be used to enable service calls to any object that requires custom creation/lookup.

The invoker implementation is as follows:

```java
public class EJBInvoker extends AbstractInvoker
{
    private EJBHome home;
    private Method createMethod;
    private static final Object[] EMPTY_OBJECT = new Object[0];

    public EJBInvoker(EJBHome home)
    {
        this.home = home;
        try
        {
            if(!home.getEJBMetaData().isSession() || !home.getEJBMetaData().isStatelessSession())
                throw new IllegalArgumentException("home must be for a stateless session bean");
            createMethod = home.getClass().getMethod("create", new Class[0]);
        }
        catch(Exception ex)
        {
            throw new IllegalArgumentException("Unable to initialize invoker: " + ex);
        }

    public Object getServiceObject(final Exchange context)
    {
        return createMethod.invoke(home, EMPTY_OBJECT);
    }
}
Invokers, once defined, need to be registered with the Service. Once a handle onto a Service object has been obtained, the example invoker above can be registered on the binding like this:

```java
Service ejbService = ....;
ejbService.setInvoker(new EJBInvoker(ejbHome));
```

If you are using an EJB3 container you can use the following invoker, which is just a simplified version of the above:

```java
public class EJB3Invoker extends AbstractInvoker {
    private Object ejb;

    public EJB3Invoker(String jndiName) throws NamingException {
        ejb = new InitialContext().lookup(jndiName);
    }

    public Object getServiceObject(final Exchange context) {
        return ejb;
    }
}
```

**Executors**

In addition to providing your own Invokers, you can also supply Executors for your service. Executors are a way to control scheduling for your service. To supply your own executor for a service just do:

```java
Service service = ....; // look up the service from CXF, or create it
service.setExecutor(new MyExecutor());
```