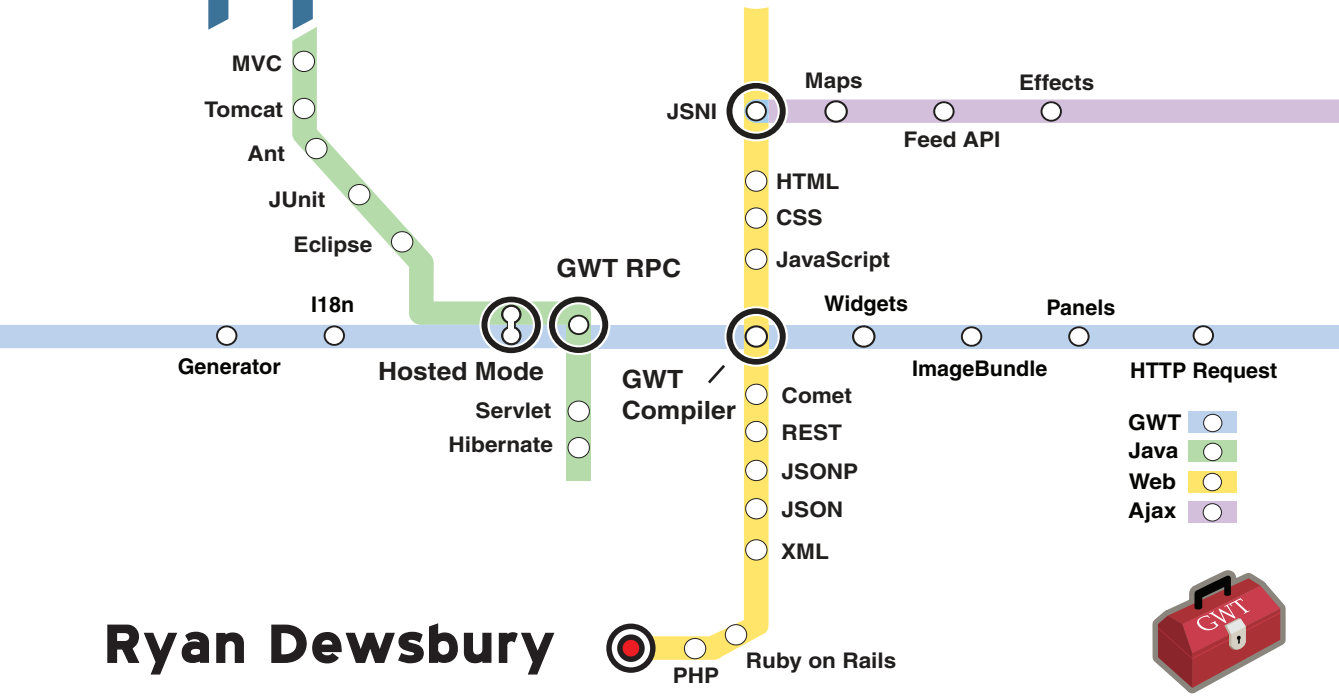
PRENTICE HALL

Google™ Web Toolkit Applications



Ryan Dewsbury

Contents

detail about how to build a GWT-RPC servlet, but you can see how the DAOs map to the servlet and to the database using Hibernate.

Writing the RPC Service

Using GWT-RPC, our model objects are automatically serialized when they are used as parameters to a RPC call. The only restrictions are that they must implement the `Serializable` interface, which they do, and they must have a zero argument constructor, which they also have. On the server we can reuse these classes, and we can even map them directly to the database using Hibernate, an object-relational mapping (ORM) tool for Java.

The first step in implementing the RPC service is to declare the service interface and have it extend GWT's `RemoveService` interface:

```
public interface RPCObjectFactoryService extends RemoteService{
   /**
    * @gwt.typeArgs <com.gwtapps.databaseeditor.client.model.BaseObject>
    */
   List getAll( String type );
   /**
    * @gwt.typeArgs <com.gwtapps.databaseeditor.client.model.BaseObject>
    */
   List getAllFrom( String type, String Id, String member );
   BaseObject getById( String type, String Id );
   void save( BaseObject object );
   void delete( String type, String id );
   void addTo(String type, String Id, String member, BaseObject objectToAdd);
}
```

This interface has the same six methods that we need to implement from the DAOs, but adds a first parameter to each method to indicate the type of object that should be used. This will either be `Story` or `User` for this application. Notice the `gwt.typeArgs` annotation that has been added. This tells the GWT compiler that the objects in the returned list must be of the type `BaseObject`. Both the `Story` and `User` classes extend `BaseObject`, so they can be transported in this list. This annotation is required to reduce the amount of code generated by the RPC code generator. If we didn't specify this, it would have to generate serialization code for every object that could be in the list.

Next, the interface's asynchronous version needs to be implemented for the client application since each method call must be asynchronous:

```
public interface RPCObjectFactoryServiceAsync{
   void getAll( String type, AsyncCallback callback );
   void getAllFrom( String type, String Id, String member,
      AsyncCallback callback );
   void getById( String type, String Id, AsyncCallback callback );
   void save( BaseObject object, AsyncCallback callback );
   void delete( String type, String id, AsyncCallback callback );
   void addTo(String type, String Id, String member, BaseObject objectToAdd,
      AsyncCallback callback );
}
```

This is almost the same as the previous interface except any return value is set to `void` and an extra `AsyncCallback` parameter is added to each method. The `AsyncCallback` implementation receives the return value, if any.

To implement the service on the server we need to implement the `RPCObjectFactoryService` interface and GWT's `RemoteServiceServlet`:

```
public class RPCObjectFactoryServiceImpl
   extends RemoteServiceServlet
   implements RPCObjectFactoryService {

   public void addTo( String type, String Id, String member, BaseObject
      objectToAdd) {
   }

   public List getAll(String type) {
      List result = null;
      return result;
   }

   public List getAllFrom(String type, String Id, String member) {
      List result = null;
      return result;
   }

   public BaseObject getById(String type, String Id) {
      BaseObject result = null;
      return result;
   }

   public void save(BaseObject object) {
   }
```

```
    public void delete(String type, String id) {
    }
}
```

This code leaves the implementation of these methods empty until the Hibernate mappings are built and the servlet can load and save objects from a database.

To run the servlet in GWT's hosted browser you need to add the following line to the module XML file:

```
<servlet path="/objectFactory"
class="com.gwtapps.databaseeditor.server.RPCObjectFactoryServiceImpl"/>
```

Now that we have an RPC service set up, we need to connect it to the DAO implementation so it can be used by the application's view. Fortunately, the `ObjectDAO` interface is a close match to the service interface, and the model objects can be automatically used with the service, so this work is fairly straightforward. The following implements the `ObjectDAO` interface for RPC:

```
protected class RPCObjectDAO implements ObjectDAO {
    private final String type;
    public RPCObjectDAO( String type ){
        this.type = type;
    }

    public void getAll(CollectionListener handler) {
        service.getAll(type, new CollectionCallback( handler ) );
    }

    public void getAllFrom( BaseObject object, String member,
        CollectionListener handler) {
        service.getAllFrom(type, object.getId(), member,
            new CollectionCallback( handler ) );
    }

    public void getById(String id, ObjectListener handler) {
        service.getById(type, id, new ObjectCallback( handler ) );
    }

    public void save(BaseObject object) {
        service.save( object , new RefreshCallback() );
    }

    public void delete(BaseObject object) {
        service.delete( type, object.getId(), new RefreshCallback() );
    }
```

```
   public void addTo( BaseObject object, String member, BaseObject
      objectToAdd) {
      service.addTo( type, object.getId(), member, objectToAdd,
         new RefreshCallback() );
   }
}
```

The DAO takes a string as a parameter, which for this application is either `Story` or `User`, and uses the parameter in each call to the service. The service is a member variable on the outer class, which is the `RPCObjectFactory`:

```
public class RPCObjectFactory implements ObjectFactory{

   protected class CollectionCallback implements AsyncCallback{
      private CollectionListener handler;
      public CollectionCallback(CollectionListener handler) {
         this.handler = handler;
      }
      public void onFailure(Throwable exception)
         { GWT.log( "error", exception );}
      public void onSuccess(Object result) {
         handler.onCollection((List)result);
      }
   }

   protected class ObjectCallback implements AsyncCallback{
      private ObjectListener handler;
      public ObjectCallback(ObjectListener handler) {
         this.handler = handler;
      }
      public void onFailure(Throwable exception)
         { GWT.log( "error", exception ); }
      public void onSuccess(Object result) {
         handler.onObject((BaseObject)result);
      }
   }

   protected class RefreshCallback implements AsyncCallback{
      public void onFailure(Throwable exception)
         { GWT.log( "error", exception ); }
      public void onSuccess(Object result) {
         listener.onRefresh();
      }
   }

   private RPCObjectDAO storyDAO = new RPCObjectDAO("Story");
   private RPCObjectDAO userDAO = new RPCObjectDAO("User");
```

```
   private RPCObjectFactoryServiceAsync service;
   public RPCObjectFactory(String baseUrl) {
      service = (RPCObjectFactoryServiceAsync)
         GWT.create( RPCObjectFactoryService.class );
      ServiceDefTarget endpoint = (ServiceDefTarget) service;
      endpoint.setServiceEntryPoint( baseUrl );
   }

   public ObjectDAO getStoryDAO() {
      return storyDAO;
   }

   public ObjectDAO getUserDAO() {
      return userDAO;
   }

   public void setListener(ObjectFactoryListener listener) {
      this.listener = listener;
   }
}
```

To handle the callbacks from the RPC calls, the `RPCObjectFactory` class implements three callback inner classes that extend GWT's `AsyncCallback` interface. The `CollectionCallback` class is used for RPC calls that expect a list of objects as a return value. It relays the list to the `CollectionListener` interface implemented in the application's view. The `ObjectCallback` class is used for RPC calls that expect a single object return value, and it relays the returned object to an `ObjectListener` interface implemented in the view. The third callback, `Refresh`, is used when the currently viewed item in the interface will need to be refreshed. In this application the `save` and `delete` DAO methods use this callback.

In the constructor, you can see the client-side object for the service being created using GWT's deferred binding. The `ServiceDefTarget` interface is used to connect the client-side service object to the service servlet.

With the servlet set up and the `RPCObjectFactory` connecting the DAO layer with the service, we can run the application on RPC by adding the `RPCObjectFactory` to the application's entry point like this:

```
public class DatabaseEditor implements EntryPoint{
   public void onModuleLoad() {
      //create view
      DatabaseEditorView view = new DatabaseEditorView();
      RootPanel.get("databaseEditorView").add( view );
```

```
      //create objectFactory
      RPCObjectFactory objectFactory =
         new RPCObjectFactory( "/objectFactory" );

      //give the view the object factory
      view.setObjectFactory( objectFactory );
   }
}
```

At this point, however, we haven't connected the servlet with the database. That is handled in the next section using Hibernate.

Using Hibernate to Store the Model

Hibernate, an object-relational mapping tool for Java applications, lets you map object-oriented Java classes and relationships to a relational database. This application uses Hibernate to map the fields from the `Story` and `User` objects to the database tables described earlier.

To get started with Hibernate, download the Hibernate package from www.hibernate.org and put the Hibernate JAR files on your classpath. Once you have Hibernate installed and on your classpath, you can begin to use it in your GWT services. Note that you can't use Hibernate in your client application since the client code is run in a browser and does not have access to a database.

To use Hibernate in the Database Editor RPC servlet, we first need to create the Hibernate configuration file called `hibernate.cfg.xml` and store it in the root of our package. In this file you configure Hibernate to connect to your database. The following is an example of the Hibernate configuration file:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>
   <property name="connection.url">jdbc:mysql://localhost/
socialnews?autoReconnect=true</property>
   <property name="connection.username">root</property>
```

```
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="connection.password"></property>
    <property name="transaction.factory_class">
    org.hibernate.transaction.JDBCTransactionFactory</property>


         <!-- JDBC connection pool (use the built-in) -->
         <property name="connection.pool_size">1</property>

         <!-- Enable Hibernate's automatic session context management -->
         <property name="current_session_context_class">thread</property>

         <!-- Disable the second-level cache  -->
         <property name="cache.provider_class">
         org.hibernate.cache.NoCacheProvider</property>

         <!-- Echo all executed SQL to stdout -->
         <property name="show_sql">true</property>

         <mapping
         resource="com/gwtapps/databaseeditor/client/model/User.hbm.xml"/>
         <mapping
         resource="com/gwtapps/databaseeditor/client/model/Story.hbm.xml"/>

     </session-factory>

</hibernate-configuration>
```

You should refer to the Hibernate documentation for information about the configuration options in this file. A brief overview of the file shows that a MySQL database called "socialnews" is chosen and the root user is used to connect. Two other important lines in this file for this application are the mapping elements at the end. Each one points to a mapping XML file that defines how one class should be mapped in the database. There is one mapping file for the `User` class called `User.hbm.xml` and one mapping file for the `Story` class called `Story.hbm.xml`.

The `User.hbm.xml` file is as follows:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping  PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

```
<hibernate-mapping>
   <class name="com.gwtapps.databaseeditor.client.model.User" table="users">
      <id name="id" column="id" type="long">
         <generator class="native"/>
      </id>
      <property name="name"/>
      <property name="email"/>
      <property name="password"/>
   </class>
</hibernate-mapping>
```

The mapping sits inside a `hibernate-mapping` element. The first child element is a `class` element that indicates the full class name of the class that is being mapped. Inside the `class` element is first the ID mapping, which is set to the type `long` and has a generator set to automatically generate a new ID when new `User` objects are saved. The remaining three elements in the `class` element are property elements that indicate the other fields that should be mapped. This mapping essentially tells Hibernate to map `User` objects to a table called `users`, as we've described earlier in this chapter.

The `Story.hbm.xml` file is implemented like this:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping  PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name="com.gwtapps.databaseeditor.client.model.Story" table="stories">
      <id name="id" column="id" type="long">
         <generator class="native"/>
      </id>
      <property name="title"/>
      <property name="url"/>
      <property name="description"/>
      <set name="digs" table="user_dug" cascade="save-update">
            <key column="story_id"/>
   <many-to-many
class="com.gwtapps.databaseeditor.client.model.User"
column="user_id"/>
      </set>
   </class>
</hibernate-mapping>
```

This file has a layout similar to the `User.hbm.xml` file, in which the class and table name are defined along with the autogenerated ID and three

fields. In addition, there is a set defined that maps the digs `List` on the `Story` class to the `user_dug` table. It sets the key to the `story_id` column in the `user_dug` table and defines a many-to-many relationship to `User` objects for the `user_id` column in the `user_dug` table. This mapping allows us to add and delete `User` objects from the digs `List` on a `Story` object and have the changes automatically reflected in the database when the Hibernate transaction is committed.

With the mappings set up, Hibernate can be used inside the `RPCObject FactoryServiceImpl` to implement the service's methods. To use Hibernate in this class we need to get a Hibernate `Session` object. The common way to get a `Session` object is to set up a Hibernate `Session Factory` in a `HibernateUtils` class:

```
public class HibernateUtil {
   private static SessionFactory sessionFactory;
   static {
      try {
         sessionFactory=new Configuration()
         .configure()
         .buildSessionFactory();
      } catch (Throwable ex) {
         throw new ExceptionInInitializerError(ex);
      }
   }
   public static SessionFactory getSessionFactory() {
      // Alternatively, you could look up in JNDI here
      return sessionFactory;
   }
   public static void shutdown() {
      // Close caches and connection pools
      getSessionFactory().close();
   }
}
```

In this helper class a global `SessionFactory` object is statically initialized for use by all clients that connect to the servlet. Each client that connects to the servlet gets `sessionFactory` and calls the `getCurrentSession()` method to retrieve a `Session` object. The `Session` returned from this call will return the same `Session` each time it is called on the current thread (this was set up as an option in the Hibernate configuration file).

Using the `HibernateUtils` class in the `RPCObjectFactoryServiceImpl` class, we are able to implement the service methods to interact with the

database. For example, the following is the implementation of the `getAll` method:

```
public List getAll(String type) {
   List result = null;
   Session session = HibernateUtil.getSessionFactory().getCurrentSession();
   session.beginTransaction();
   result = session.createQuery("from "+type).list();
   session.getTransaction().commit();
   return result;
}
```

This code illustrates the steps involved with using a Hibernate session to interact with the database. First, the current session is retrieved from the `SessionFactory`, and then the `beginTransaction` method is called to indicate a unit of work. After a transaction is started, various method calls can be made on the session to read from and write to the database. In this example a query is created to get a list of all objects of a certain type. Once all of the work is done with the database, the `commit` method is called to save any changes that may have occurred.

The rest of the methods in the `RPCObjectFactoryServiceImpl` follow this pattern. The following code implements each of the methods, which illustrates how to use Hibernate to perform all of the DAO's methods needed for the Database Editor application:

```
public class RPCObjectFactoryServiceImpl
   extends RemoteServiceServlet
   implements RPCObjectFactoryService {

   public void addTo(
String type, String Id, String member, BaseObject objectToAdd) {
      if( type.equals("Story") && member.equals("digs") ){
         Session session =
HibernateUtil.getSessionFactory().getCurrentSession();
         session.beginTransaction();
         Story story = (Story)session.get(Story.class, Id);
         story.getDigs().add( objectToAdd );
         session.getTransaction().commit();
      }
   }

   public List getAll(String type) {
      List result = null;
      Session session =
HibernateUtil.getSessionFactory().getCurrentSession();
```

```
      session.beginTransaction();
      result = session.createQuery("from "+type).list();
      session.getTransaction().commit();
      return result;
   }

   public List getAllFrom(String type, String Id, String member) {
      List result = null;
      if( type.equalsl("Story") && member.equals("digs") ){
         Session session =
HibernateUtil.getSessionFactory().getCurrentSession();
         session.beginTransaction();
         Story story = (Story)session.get(Story.class, Id);
         result = story.getDigs();
         session.getTransaction().commit();
      }
      return result;
   }

   public BaseObject getById(String type, String Id) {
      BaseObject result = null;
      Session session =
HibernateUtil.getSessionFactory().getCurrentSession();
      session.beginTransaction();
      result = (BaseObject)session.get(Story.class, Id);
      session.getTransaction().commit();
      return result;
   }

   public void save(BaseObject object) {
      Session session =
HibernateUtil.getSessionFactory().getCurrentSession();
      session.beginTransaction();
      session.save(object);
      session.getTransaction().commit();
   }

   public void delete(String type, String id) {
      Session session =
HibernateUtil.getSessionFactory().getCurrentSession();
      session.beginTransaction();
      session.delete(session.get(Story.class, id));
      session.getTransaction().commit();
   }

}
```