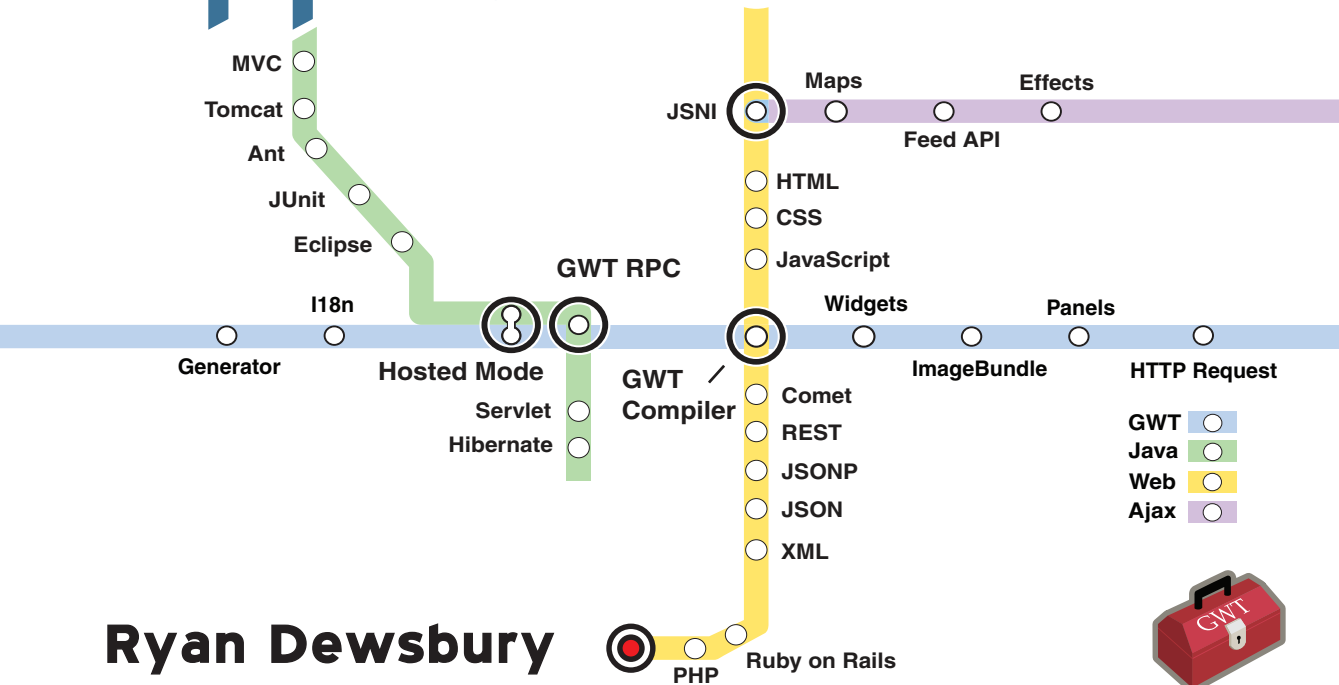


Google™ Web Toolkit

Applications



Ryan Dewsbury

Contents

Preface xvii

About the Author xxiii

PART I *Understanding the Google Web Toolkit* 1

Chapter 1 *First Steps with the Google Web Toolkit* 3

The Emergence of Ajax	3
Rethinking Web Applications	5
Adobe Flash and Flex	6
Microsoft Silverlight	7
Java FX	8
Software Engineering for Ajax	8
Building Rich Interfaces with Widgets and Panels	9
Getting Better Performance with Asynchronous Communication	10
Providing Interoperation Through Web Standards and Web Services	10
Speeding Development Using Java Tools	11
Evaluating Your Background	11
Web Designers	12
Web Site Developers	12
Ajax Developers	13
Web Application Developers	13
Desktop Application Developers	13
The Importance of Application Development Skills	14

A Quick Tutorial	16
Starting a GWT Project	17
Customizing a Generated Application	20
Creating a Dynamic Interface	22
Using Asynchronous HTTP	27
Overview of Toolkit Packages	30
Overview of GWT Applications	34
Common Application Patterns	34
Sample Applications	35
Summary	36

Chapter 2 User Interface Library Overview 37

Static Widgets	38
Label	38
HTML	41
Image	42
Hyperlink	44
Form Widgets	47
Button	47
ToggleButton and PushButton	49
Checkbox	53
RadioButton	54
ListBox	54
SuggestBox	56
TextBox	58
PasswordTextBox	59
TextArea	60
RichTextArea	61
FileUpload	62
Hidden	62
Complex Widgets	63
Tree	63
MenuBar	67
Simple Layout Panels	70
FlowPanel	70
HorizontalPanel and VerticalPanel	71
HorizontalSplitPanel and VerticalSplitPanel	72
FlexTable and Grid	73
DeckPanel	77
DockPanel	78
HTMLPanel	79

Complex Layout Panels	80
StackPanel	80
TabPanel	81
Simple Container Panels	84
Composite	84
SimplePanel	85
ScrollPane	85
FocusPanel	86
Complex Container Panels	87
FormPanel	87
DisclosurePanel	89
PopupPanel	91
DialogBox	93
User Interface Framework Glue	95
Event Interfaces	95
Feature Interfaces	101
Summary	103

Chapter 3 Server Integration Techniques 105

Basic Server Integration Techniques	105
Asynchronous HTTP Requests	106
Working with Plain Text and HTML	109
Integrating with Traditional Server-Side Script Technologies	111
Using Data Format Libraries	117
Reading and Writing XML	117
Reading and Writing JSON	122
Third-Party Server Integration	125
The Same Origin Policy	125
Using JavaScript APIs	126
Using JSONP	127
Using an HTTP Proxy	128
Advanced Server Integration Techniques	128
Stateful Servers	128
Integrating with Stateless Servers	130
Using GWT-RPC	131
Summary	137

Chapter 4 Software Engineering for Ajax 139

Setting Up the Development Environment	139
Installing the Java Development Kit	140

Installing the Google Web Toolkit	140
Installing Eclipse	144
Adding Projects to Eclipse	145
Writing Java Code in Eclipse	149
Creating Classes in Eclipse	149
Using the Eclipse Java Editor	153
Debugging in Eclipse	158
Organizing Your Application Structure	164
Testing Applications	168
Using JUnit	170
Benchmarking	176
Building and Sharing Modules	180
Using Modules	180
Creating a Reusable Module	181
Sharing a Compiled Application (Mashups)	183
Deploying Applications	183
Deploying to a Web Server	184
Deploying a Servlet to a Servlet Container	184
Automating Deployment with Ant	187
Summary	190

Chapter 5 Using the Toolkit Effectively 191

Using Asynchronous Programming	191
Handling the Back Button	197
Creating Elegant Interfaces with CSS	200
Connecting GWT Widgets to CSS	200
Specifying Colors	203
Specifying Units	203
Using Font Properties	205
Using Text Properties	205
Using Margin, Border, and Padding Properties	207
CSS Examples	208
Using the Cursor Property	210
Extending the Toolkit	212
Building New Widgets	212
Using the JavaScript Native Interface	215
Providing Cross-Browser Support	218
Using Other JavaScript Libraries	221
Internationalizing Applications	226
Declaring Constants	227
Declaring Messages	229

Localization Tools	230
Managing Locales	232
Using the Dictionary	234
Generating Code	235
Using Generated Code	235
Writing a Code Generator	237
Writing the generate Method	239
Improving Performance	242
Handling Long Processing	242
Using ImageBundle	244
Caching on Apache	246
Caching on a Servlet Container	247
Compression on Apache	249
Compression on a Servlet Container	249
Summary	251

PART II ***Rich Web Applications by Example*** **253**

Chapter 6 **Gadget Desktop Application** **255**

Using the Container Application Pattern	256
Designing the Model	257
Using the Abstract Factory Pattern	258
Making the GadgetClass Class	259
Making the Gadget Class	261
Making Easy User Preferences	262
Building a Columned Container Interface	264
Defining the View	266
Using a VerticalPanel for the Main Layout	267
Using HorizontalPanel for the Menu	267
Using TabPanel for Page Selection	269
Using HorizontalPanel and FlowPanel for Columns	271
Putting Gadgets in the View	274
The Hello World Gadget	274
Making a Gadget Container	276
Using ImageBundle, ToggleButton, and PushButton	280
Creating Drag-and-Drop Gadgets	283
Docking Widgets Between Panels	284
Dragging Widgets Using Mouse Events	285
Dropping a Widget	288

Cleaning Up User Interfaces with CSS	290
Using CSS with GWT	292
Adding Styles to the Tabs	293
Adding Styles to the Menu	296
Adding Styles to the <code>GadgetContainerView</code>	297
Creating a <code>RoundedPanel</code>	298
Adding Persistency	300
Implementing the <code>CookieStorage</code> Class	300
Handling Browser Differences	302
Loading and Saving Cookies	303
Using Google Gears for Storage	305
Using Other JavaScript Libraries	309
Using Google Maps API	309
Using Google Ajax API for Feeds	311
Building a Weather Gadget	313
Building an RSS News Gadget	316
Summary	318

Chapter 7 Multi-Search Application 319

Using the Aggregator Application Pattern	319
Multi-Search Design	321
The Model	322
The View	324
The <code>MultiSearchView</code> Class	325
The <code>SearchResultsView</code> Class	328
Building a <code>Link</code> Widget	329
The Controller	331
Importing Structured Data Formats	334
Bypassing the Same Origin Policy	334
Loading JSON Feeds with JSONP	336
Integrating with Yahoo! Search	340
Integrating with Google Base	344
Integrating with Flickr Search	348
Integrating with Amazon Search	354
Summary	358

Chapter 8 Blog Editor Application 361

Using the Workspace Application Pattern	361
Building a Web Service Client	363
Blog Editor Design	365
The Model	365

Building a Multiple Document View	367
The <code>BlogEditorView</code> Class	370
The <code>BlogView</code> Class	372
The <code>BlogEntryView</code> Class	376
The <code>EditEntryView</code> Class	378
Adding Rich Text Editing	381
Using <code>RichTextArea</code>	382
Using <code>ImageBundle</code> , Internationalization, and the <code>RichTextToolbar</code>	383
The <code>LoadingPanel</code> Widget	386
The <code>TitleCommandBar</code> Widget	388
Designing the Application Controller	390
Building an HTTP Proxy Servlet	392
A New Cross-Domain <code>RequestBuilder</code>	394
Writing a Proxy Servlet	397
Integrating with the Blogger API	401
Using Atom Publishing Protocol and <code>GData</code>	402
Defining the <code>BloggerService</code> Class	402
Signing In to a Google Account	403
Getting the XML List of Blogs for the Account	410
Getting the XML List of Entries for Each Blog	413
Sending XML to Create and Save an Entry	416
Sending a Delete Request for an Entry	419
Summary	421

Chapter 9 Instant Messenger Application 423

Using the Collaborator Application Pattern	423
Instant Messenger Design	425
The Model	426
Building a Complementary Interface	429
The <code>MessengerView</code> Class	432
The <code>SignInView</code> Class	436
The <code>ContactListView</code> Class	439
The <code>ChatWindowView</code> Class	442
The Controller Overview	447
Using GWT-RPC	450
The <code>RemoteService</code> Interface	452
The <code>RemoteServiceServlet</code> Class	453
Using an Asynchronous Interface	454
Connecting to the Server	456
Adding RPC Events	458

Polling Protocols	458
Event-Based Protocols	460
Implementing Events	462
The Instant Messenger Server	465
Using Server-Side Advanced IO	470
Using Comet on Tomcat	472
Using Continuations on Jetty	476
Summary	478

Chapter 10 Database Editor Application 479

Using the Manager Application Pattern	480
Designing the Model	482
Using Asynchronous Data Access Objects	485
Building a Two-Paned Editor Interface	488
Using the <code>Tree</code> and <code>SplitPanel</code> Widgets	489
Extending and Dynamically Loading Tree Items	490
Creating Workspace Views	494
Using Dialogs for Editing and Creating Objects	502
Server Integration Overview	507
Using Actions	507
Using REST	508
Using RPC	509
Writing a Generic GWT Code Generator	510
Writing the Code Generator	512
Automatically Serializing to XML	521
Automatically Serializing to JSON	521
Integrating with Action-Based PHP Scripts	522
Using PHP to Build the Action API	523
Writing the Action Data Access Layer in the Client	526
Integrating with a RESTful Ruby on Rails Application	530
Using Ruby on Rails to Build a REST API	531
Writing the REST Data Access Layer in the Client	538
Integrating with a GWT-RPC Servlet	542
Writing the RPC Service	543
Using Hibernate to Store the Model	548
Summary	554

Index 555

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 `RemoteService` 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.databaseditor.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 `RPCObjectFactoryServiceImpl` 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.equals("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();
    }
}
```