SAMPLE CHAPTER

Robert T. Cooper
Charlie E. Collins

GWT in Practice

- 39 TECHNIQUES
- GET IT DONE
- GET SAVVY

MANNING

GWT in Practice

by Robert Cooper and Charles Collins

Sample Chapter 4

4 Core Application Structure

This chapter covers

- Creating composite view components
- Data binding for the model and view
- The relationship of the controller to the RPC service layer
- Using JPA in the model
- Completing a front-to-back GWT application, client, and server

You must unlearn what you have learned.

—Yoda (*The Empire Strikes Back*)

You have been building web applications for years now. The flaws in the process are obvious, but if you are like us, coping with these flaws has become second nature. You know GWT provides a whole new metaphor for web development, and chapter 2's calculator example demonstrated that GWT development is much more like traditional desktop application development. Now you can hit rewind on your experiences and knowledge and look at web development in a new way. In this second part of the book, we will go beyond the introductory material you saw in the previous chapters, and look at more goal-oriented GWT techniques.

When working with GWT, you are, of course, no longer building navigation and pages in the way you did before. Even more module-centric web frameworks like JSF are still essentially page-based, with the model, view, and controller layers executing within the application server. Now you need to look at your application as having a series of widgets making up the view, a controller that orchestrates actions, and a smart model layer.

In this chapter, we will take a look at a basic use-case for a web application you have likely written many times before: user registration. We will move through the whole structure of this example and look at how these parts relate in GWT applications. We will start by building a model layer that is more intelligent than most of the models you have likely used in other web applications. Next, we will show how to construct the view components and connect them to the model. Finally, we will look at the controller, at how to use the controller layer to interact with services on the server, and at how to use the Java Persistence API (JPA) to store elements in the database.

4.1 Building a model

The model layer of a GWT application has to be a little bit smarter than the model in many traditional web applications. It needs to be responsible for notifying the view layer of changes, as well as for receiving updates to its own structure. In desktop Java development, this is accomplished through the use of `PropertyChangeEvents` and `PropertyChangeListeners` that bind the data from the model to the view components. The view layer "listens" for changes to the model and updates itself accordingly, using the Observer pattern. This can be done in GWT development as well. You saw a small example of this in chapter 2, with the calculator's model, and we will expand on that concept here, using a more formal and representative approach that demonstrates what you will need to do over and over.

Figure 4.1 shows a basic `User` class that we will work with. It is a simple class containing a user name and other common fields, as well as two `Address` objects (one shipping and one billing). To get started, we need to enable these for data binding.

PROBLEM

We need to enable data binding to user interface elements in our model.

SOLUTION

The basis of data binding between the model and the view is triggering events in the model that notify the view of changes, and vice versa. In the calculator example from chapter 2, we built an event system from scratch. We did this in order to be explicit and not involve additional concepts and dependencies in our first example. Nevertheless, this manual method is somewhat inconvenient.

Here we will use the `java.beans.PropertyChangeSupport` class. This isn't a class provided by GWT yet, but it is available as part of the GWTx project (http://code.

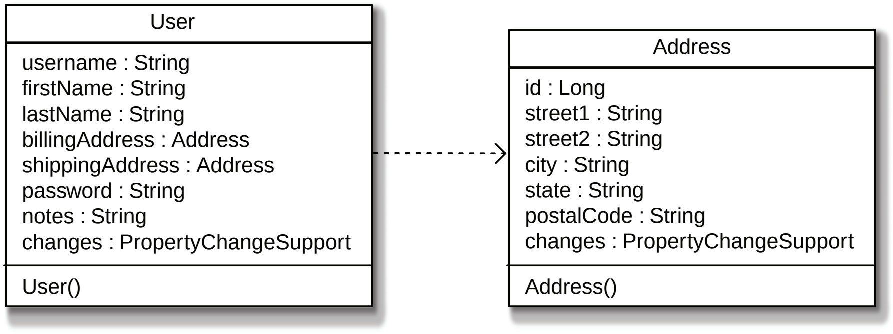


Figure 4.1 The `User` and `Address` classes serving as the model. Notice the `PropertyChangeSupport` attribute, which we will use to notify the view of changes to the model beans.

google. com/p/gwtx/), which adds to the core Java classes provided by GWT. Listing 4.1 shows the `User` object instrumented with the `PropertyChangeSupport` class.

Listing 4.1 Using the `PropertyChangeSupport` class in the `User` object

```
public class User implements IsSerializable {

    private String username;
    private String firstName;
    private String lastName;
    private Address billingAddress = new Address();
    private Address shippingAddress = new Address();
    private String password;
    private String notes;
    private transient PropertyChangeSupport changes =
        new PropertyChangeSupport(this);

    public User() {
        super();
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        String old = this.firstName;
        this.firstName = firstName;
        changes.firePropertyChange(
"firstName", old, firstName);
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        String old = lastName;
        this.lastName = lastName;
```

1 Include IsSerializable marker

2 Construct Property-ChangeSupport

3 Include PropertyChangeSupport to fire changes

```
        changes.firePropertyChange("lastName", old, lastName );
    }

    public Address getBillingAddress() {
        return billingAddress;
    }

    public void setBillingAddress(Address billingAddress) {
        Address old = this.billingAddress;
        this.billingAddress = billingAddress;
        changes.firePropertyChange(
"billingAddress", old, billingAddress);
    }

     // The rest of the getters and setters are removed for brevity.

    public void addPropertyChangeListener(
PropertyChangeListener l) {                         ❹ Add global
        changes.addPropertyChangeListener(l);         change listener
    }

    public void addPropertyChangeListener(
        String propertyName,                        ❺ Add property-
PropertyChangeListener l) {                           specific listener
        changes.addPropertyChangeListener(
propertyName, l);
    }

    public void removePropertyChangeListener(PropertyChangeListener l) {
        changes.removePropertyChangeListener(l);
    }

    public void removePropertyChangeListener(
        String propertyName, PropertyChangeListener l) {
        changes.removePropertyChangeListener(propertyName, l);
    }

}
```

This gives us a model object capable of notifying the view of changes and exposing methods for attaching listeners.

DISCUSSION

This may be a lot more code than you are used to seeing in a model bean in a traditional web application, but it is mostly boilerplate. We first need to implement `IsSerializable` or `Serializable` ❶ (the GWT marker interfaces that tell the compiler this class should be made serializable), because we want this object to move back and forth between the client and the server. This time, though, we have a nonserializable property: the `PropertyChangeSupport` instance `changes` is marked transient ❷ and isn't part of the serialized version that will be sent back and forth. It is constructed each time with the object.

Once we have the `PropertyChangeSupport` class, we need to instrument the setter methods to fire change events. The sequence is repetitive, but very important. First you store the current value, then you set the new value, and only after the new value is set do you fire the change event ❸. It is critical that the change events only be fired after the instance value has been set, not at the top of the setter method. The reason for this is what happens inside the `PropertyChangeSupport` class.

`PropertyChangeSupport` (PCS) provides collections and fires change events much like the interfaces used in the previous calculator example. It also checks each call to `firePropertyChange()` to make sure the old and new values are different before firing the event. Figure 4.2 shows the execution sequence for a change event being fired and updating an element with a two-way bind.

If you called `firePropertyChange()` at the top of the setter method, when the last step in this sequence was reached, the current instance value on the model object would still be the same, and the whole event cycle would go into an infinite loop!

Finally, we need to provide methods on the model object for listeners to be added to the model. There are two kinds of listeners supported by the `PropertyChangeSupport` class. Global listeners ❹ will receive an event any time a change is fired. The listener would then look at the value returned by the call to `getPropertyName()` on the `PropertyChangeEvent` instance to decide what to do. The other type of listeners are property specific ❺. These will only be fired if the specified property changes, and they are generally the most useful listeners. However, if you have a complex model or one that requires a lot of translation between the model data and the data needed to properly render the view elements, it can be easier to just listen for any change and reset the entire view component.

Now we have built the model layer of our application, and we have given it the ability to notify listeners of changes, which can be used to bind data in the view layer. The

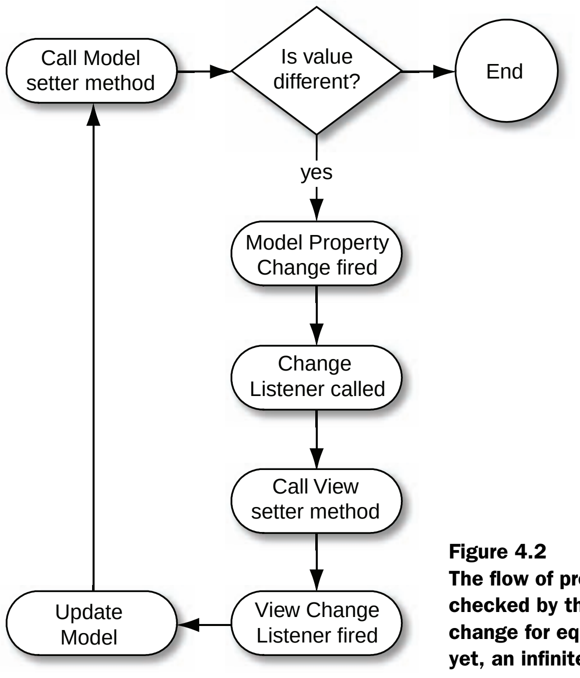


Figure 4.2
The flow of property change events firing will loop if not checked by the model. If the model doesn't check the change for equality, or if the change hasn't happened yet, an infinite loop is formed.

next part of the user registration application we need to build is the view layer itself. The view will both listen to changes coming from the model, and fire its own changes to update the model based on user input.

4.2 Building view components

Unlike page- or template-based technologies, the view layer of a GWT application is composed of Java classes. As a result, the best way to build your user interface is to build a component (or several) for each element of your view. Like building page templates for editing an object type when using a traditional system like Struts Tiles, components can be composed of several different types of view elements.

Most well-designed API libraries derive from a few core classes that encapsulate common functionality. GWT is no different. The `com.google.gwt.user.client.ui` package includes such foundational elements. These items are not typically instantiated directly but are the backbone of the API. The package starts with a root `UIObject` base class, and then fans out into the various component hierarchies that make up the remainder of the library. `UIObject`, as its name implies, is where you begin when building view components.

Figure 4.3 shows a simplified class diagram for the top portion of the API. You can see `UIObject`, `Widget`, and the other descendant elements that are used to create view components.

The API fans out from `UIObject` to three direct descendants: the further subclassed `Widget`, and two individual offshoots, `MenuItem` and `TreeItem`. The subclasses of `Widget` also fall into two distinct subsets: those that are not themselves further subclassed (at the lower left in figure 4.3), and those that are further subclassed (at the lower right in figure 4.3).

These are the core classes used to build GWT applications. When constructing a view layer, there are two general patterns for building UI elements: extending a base GWT widget, or constructing a composite. We will look at both of these methods in turn.

4.2.1 Extending widgets

Declaring your view class as extending a core widget is an easy way to start. You can then perform your setup in the constructor and add methods specific to your implementation, as we did in chapter 2's calculator. We will start with this same pattern here, by constructing a `Panel` that provides an edit view for an `Address` object.

PROBLEM

We need to construct a particular view for a model element, and this view will be used within our module.

SOLUTION

Extending a base GWT widget is certainly the easiest way to get started. We will use this technique to create an `AddressEdit` class. Figure 4.4 shows the address edit portion of a larger component we will complete in this chapter for editing user data, a `UserEdit` widget.

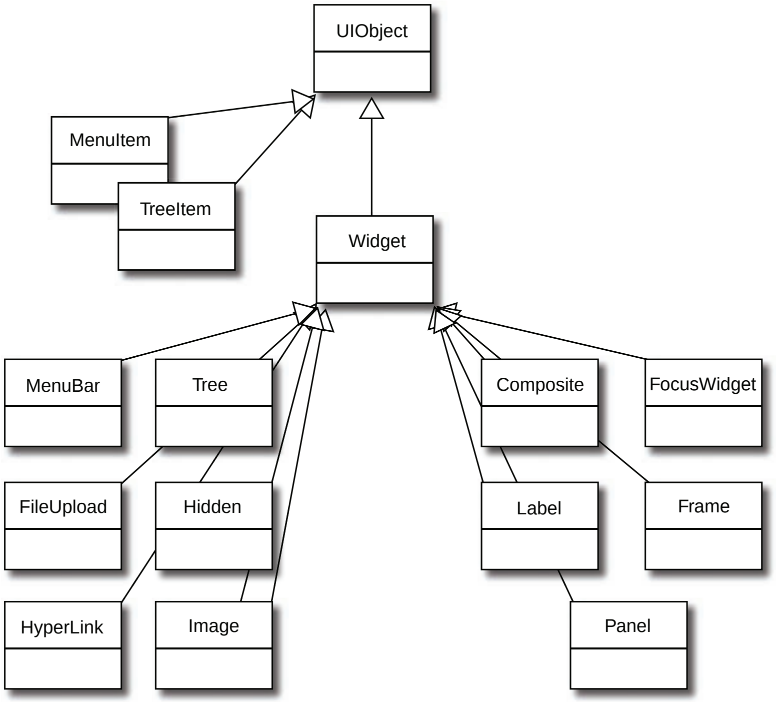
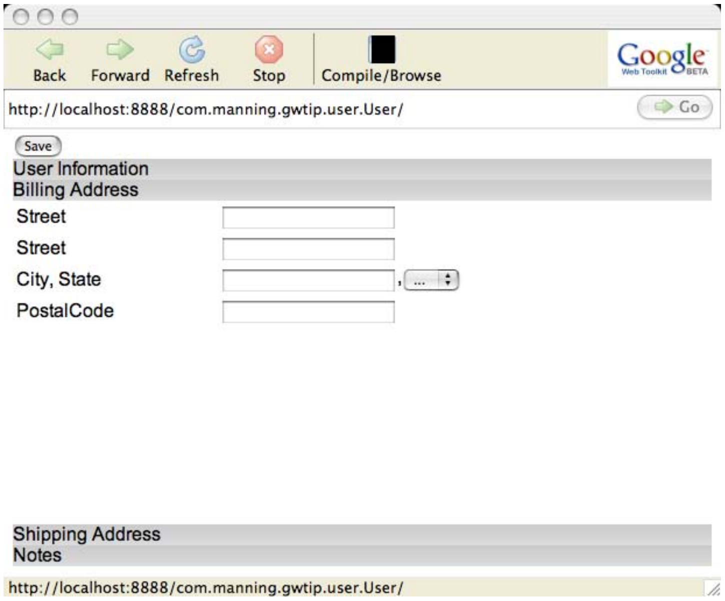


Figure 4.3 The top-level hierarchy of classes in the `client.ui` API. Notice that almost everything is a `Widget`, except for `MenuItem` and `TreeItem`, which are for limited use inside other widgets.



http://localhost:8888/com.manning.gwtip.user.User/

User Information
Billing Address

Street	
Street	
City, State	, ...
PostalCode	

Shipping Address
Notes

http://localhost:8888/com.manning.gwtip.user.User/

Figure 4.4 The `AddressEdit` widget showing editing of the `billingAddress` property nested in a `StackPanel`. The `StackPanel` shows the other elements of the larger `UserEdit` component.

Here the `AddressEdit` class simply extends the `FlexTable` widget, allowing us to lay out a fairly standard address entry form. Listing 4.2 shows how we build this form in the constructor of `AddressEdit`.

Listing 4.2 AddressEdit.java

```
public class AddressEdit extends FlexTable {

    private Address address;                                  ❶ Include model reference
    private TextBox street1 = new TextBox();
    private TextBox street2 = new TextBox();                  ❷ Construct component widgets for Address
    private TextBox city = new TextBox();
    private ListBox state = new ListBox();
    private TextBox postalCode = new TextBox();

    public AddressEdit(final Address address) {
        super();

        state.addItem("...");
        state.addItem("AL");
        state.addItem("AK");
        // rest omitted.

        this.setStyleName("user-AddressEdit");
        this.address = address;
        this.setWidget(0,0, new Label("Street"));
        this.setWidget(0,1, street1);                         ❸ Lay out grid
        this.setWidget(1,0, new Label("Street"));
        this.setWidget(1,1, street2);
        this.setWidget(2,0, new Label("City, State"));
        HorizontalPanel hp = new HorizontalPanel();
        hp.add(city);
        hp.add(new Label(","));                               ❹ Create subpanel where needed
        hp.add(state);
        this.setWidget(2,1, hp);

        this.setWidget(3,0, new Label("PostalCode"));
        this.setWidget(3,1, postalCode);
    }
}
```

`AddressEdit` extends a simple widget, `FlexTable`, and lays out the elements we need to edit an `Address` object.

DISCUSSION

Extending `FlexTable` ❶ makes laying out the widgets in the constructor method very easy. We simply create the elements we need ❷ and add them. `FlexTable` works much like a standard HTML table element, dynamically reconfiguring itself as elements are added to grid positions ❸.

`FlexTable` and `Grid` are both subclasses of `HTMLTable` and, as such, use the same methods for inserting elements into a table structure at a specified row and column. The difference is that `FlexTable` adds methods that enable you to insert new cells dynamically, while `Grid` does not. `Grid` does allow for the entire table to be resized, rows or columns, but does not allow on-the-fly expansion like `FlexTable` does. This is because Grid, by definition, must maintain its rectangular nature, and FlexTable is, how shall we put this, flexible.

Inside our widget we are using a HorizontalPanel to format city and state in the traditional sequence. Along with VerticalPanel, it is the most basic of container Widgets. This panel simply lays out the widgets added to it horizontally in the order they are added; in this case, a TextBox, a Label, and a ListBox are added ❹.

Extending a base GWT widget is problematic for widgets you want to expose as part of an API. Since the AddressEdit widget extends FlexTable, it is possible for external code to call the setWidget() method outside of the constructor and alter the composition of the widget. We don't want people doing this, since AddressEdit is really only intended to be used inside the scope of editing a user. To get around this problem, GWT provides the Composite widget.

4.2.2 *Extending composite*

In this chapter's AddressEdit example, and in chapter 2's calculator, we created a widget by combining provided panels and input components in a custom manner. We wanted to reuse all of the Widget features provided by the toolkit, so we extended a Panel, such as VerticalPanel. While this approach does work and is simple (which is why we have used it so far), it is not the preferred way to create custom GWT components. Instead of naively subclassing panel elements, it is much better to create a GWT Composite class.

Composite is a class that exposes only the getElement() method, as required by UIObject, and provides several protected methods for use by widgets that subclass the component. These methods are getWidget(), setWidget(), and initWidget(). Composite objects contain only a single base widget, which for obvious reasons will usually be a container of some kind. These widgets are constructed inside the local constructor of a Composite child, and then are added to the composite with initWidget(). The call to initWidget() must always be made before a Composite can be added to a container, and should therefore be the last call in your constructor.

In this section, to demonstrate the Composite concept, we will create the UserEdit class as a composite element.

PROBLEM

We need to create a widget, but we do not wish to expose the public methods of a basic GWT widget.

SOLUTION

Start by extending Composite, and expose only the methods you want, or none at all. Listing 4.3 shows the UserEdit class and the construction of a Composite widget.

Listing 4.3 Building a Composite UserEdit class

```
public class UserEdit extends Composite {
    private User user;                                   ◁─┐
    private StackPanel stack = new StackPanel();           ❶  Create model layer
    private TextBox username = new TextBox();                 object for widget
    private PasswordTextBox password = new PasswordTextBox();
    private PasswordTextBox passwordConfirm = new PasswordTextBox();
    private TextBox firstName = new TextBox();
    private TextBox lastName = new TextBox();
    private TextArea notes = new TextArea();
    private AddressEdit billingAddress;
    private AddressEdit shippingAddress;

   public UserEdit(final UserCreateListener controller, final User user) {
        super();
        this.user = user;
        stack.setHeight("350px");
        VerticalPanel basePanel = new VerticalPanel();
        Button save = new Button("Save");
        basePanel.add(save);                                  Create FlexTable
        basePanel.add(stack);                                 to lay out basic
        FlexTable ft = new FlexTable();                       elements
        ft.setWidget(0,0, new Label("Username"));
        ft.setWidget(0,1, username);
        ft.setWidget(1,0, new Label("Password"));
        ft.setWidget(1,1, password);
        ft.setWidget(2,0, new Label("Confirm"));
        ft.setWidget(2,1, passwordConfirm );

        ft.setWidget(3,0, new Label("First Name"));
        ft.setWidget(3,1, firstName);
        ft.setWidget(4,0, new Label("Last Name"));
        ft.setWidget(4,1, lastName);

        stack.add(ft, "User Information" );
        billingAddress = new AddressEdit(
            user.getBillingAddress());                              Create
        stack.add(billingAddress, "Billing Address");               AddressEdit
        shippingAddress = new AddressEdit(                          classes for
           user.getShippingAddress());                              addresses
        stack.add(shippingAddress, "Shipping Address");
        notes.setWidth("100%");
        notes.setHeight("250px");
        stack.add(notes, "Notes");

      this.initWidget(basePanel);
    }
}
```

In this example, we are using our `UserEdit` class as a component in a larger edit view of the `User` model object. While the object graph contains other structured objects (the `Address` classes), the `UserEdit` class brings these together with the editors for the other portions of our model that we created individually. It also provides direct edit widgets for simple values directly on the `User` class. At first blush, this looks very similar to the process we used in constructing the `AddressEdit` class, but it is actually a bit different.

DISCUSSION

Listing 4.3 creates our model layer object ❶ and a bunch of widgets relating to the fields on the model. The outer container is simply a `VerticalPanel`, and then we use a `StackPanel` to separate out different aspects of the `User` object. A `StackPanel` is a special kind of container that provides an expanding and collapsing stack of widgets with a label that toggles between them (much like the sidebar in Microsoft Outlook).

The use of the `StackPanel` is in keeping with one of the new principles you should note if you are coming from traditional web development: Build interface components, not navigation. In a traditional web application, each element of the stack might be a separate page, and it would be necessary to use a `Session` construct on the server to store the intermediate states of the `User` object. Here we can simply build the entire `User` object's construction process into one component that lets the user move through them. This means less resource use on the server, because we are spared a request-response cycle when we move between the different sections; we no longer have to maintain state information for each user accessing the application.

Once we have constructed the `UserEdit` object, it has no exposed methods other than `getElement()`, and it is public rather than package-scoped like `AddressEdit`. These aren't completed classes, however. We still need to enable them to interact with the model layer. This means handling user input via events and making changes to the model to update the data.

4.2.3 Binding to the model with events

We discussed in section 4.1 why we need events on the model layer, and how to provide that functionality. Now, in the view layer, we need to build the binding logic into our widgets. GWT includes a number of basic event types.

In fact, many of the GWT widgets provide much of the event notification you will ever need. In our `UserEdit` example thus far, we made use of `Button`, which extends `FocusWidget`, which in turn implements `SourcesClickEvents` and `SourcesFocus-Events` to raise events for our `ClickListener` implementation. Likewise, we used `TextBox,` which itself implements `SourcesKeyboardEvents`, `SourcesChangeEvents`, and `SourcesClickEvents`. In the GWT API, event types are specified by these `Sources` interfaces, which tell developers what events a widget supports. We can use these, along with the `PropertyChangeEvents` from our model layer to provide a two-way binding with the view.

Problem

We need to bind the data from a widget or a view component to a property on a model object.

Solution

We will revisit the `UserEdit` composite class to demonstrate data binding. Listing 4.4 shows the changes we will make to the constructor, and the new methods we will add to support this concept.

Listing 4.4 UserEdit.java, modified to include `PropertyChangeSupport`

```
public class UserEdit extends Composite{
     // Previously shown attributes omitted
    private PropertyChangeListener[] listeners =
        new PropertyChangeListener[5];
```

Create Array to hold PropertyChangeListeners

```
public UserEdit(final UserCreateListener controller, final User user) {
     super();                                               ❶ Create
                                                              PropertyChangeListener
 // Previously shown widget building omitted.                 for model

 listeners[0] = new PropertyChangeListenerProxy(
    "street1",
    new PropertyChangeListener() {                  Add PropertyChangeListener
        public void propertyChange(                 to model object
    PropertyChangeEvent propertyChangeEvent) {
        street1.setText(                                          Repeat ❷
        (String) propertyChangeEvent.getNewValue());            for each
            }                                                   property
         });
 address.addPropertyChangeListener(listeners[0]);
 street1.addChangeListener(                              ❸ Create change
     new ChangeListener() {                                listener for view
         public void onChange(Widget sender) {
             address.setStreet1(street1.getText());        Update
         }                                                 model
     });

     // Repeating pattern for each of the elements

     save.addClickListener( new ClickListener() {          ❹ Check
         public void onClick(Widget sender) {                passwordConfirm
             if(!password.getText().equals(                  before updating
                     passwordConfirm.getText())) {
                 Window.alert("Passwords do not match!");
                 return;
             }
             controller.createUser(user);         ❺ Call controller
         }
     });

     this.initWidget(basePanel);

 }                                                ❻ Clean up model
                                                    listener
 public void cleanup(){
     for (int i=0; i < listeners.length; i++) {
         user.removePropertyChangeListener(listeners[i]);
     }
   billingAddress.cleanup();                      Clean up child
     shippingAddress.cleanup();                 ❼ view elements
 }
```

Now we have the basics of data binding and housekeeping in the `UserEdit` class.

DISCUSSION

Providing two-way data binding, unfortunately, requires a good deal of repetitive code ❷. In Swing it is possible to simplify a lot of this boilerplate code with reflection-based utility classes, but since the Reflection API isn't available in GWT code, we must repeat this code for each of our properties. First, we create a `PropertyChangeListener` ❶ that watches the model and will update the view if the model changes. We wrap it in a `PropertyChangeListenerProxy` that will filter the events to just those we want to watch. While not critical here, it is a very good practice to provide this binding in your widgets. This ensures that if another part of the application updates the model, the view will reflect it immediately and you will not have a confused state between different components that are looking at the same objects.

NOTE While the PropertyChangeSupport class will let you add Property-ChangeListeners specifying a property name, it will wrap these in the PropertyChangeListenerProxy class internally. When it does this, you lose the ability to call removePropertyChangeListener() without specifying a property name. Since we just want to loop over all of these listeners in our cleanup() method, we wrap them as we construct them so the cleanup will run as expected.

Next, we create a ChangeListener and attach it to the widget responsible for the property ❸. With each change to the widget, the model will be updated. In this case, we are using TextBoxes, so we call the getText() method to determine their value. If you have done Ajax/DHTML programming before, you know that the common pattern for the <input type="text"> element is that the onChange closure only fires when the value has changed and the element loses focus. Sometimes this is important to keep in mind, but since we know that the element will lose focus as the user clicks the Save button, we don't have to worry about it here. If you need that kind of detail about changes, you could use a KeyboardListener on the TextBoxes, which will fire on each keystroke while the box is focused.

For some widgets, you might have to provide a bit of logical conversion code to populate the model. The following is a small section from the AddressEdit class, where we update the state property on the Address object:

```
listeners[4] = new PropertyChangeListener() {
    public void propertyChange(
        PropertyChangeEvent propertyChangeEvent) {
        for(int i=0; i < state.getItemCount(); i++) {
            if(state.getItemText(i).equals(
                propertyChangeEvent.getNewValue())) {
                state.setSelectedIndex(i);
                break;
            }
        }
    }

};
address.addPropertyChangeListener("state", listeners[4]);
state.addChangeListener(new ChangeListener() {
    public void onChange(Widget sender) {
        String value = state.getItemText(state.getSelectedIndex());
        if(!"...".equals(value)) {
            address.setState(value);
        }
    }
});
```

This looks much like the repeating patterns with the `TextBoxes`, but in both listeners we must determine the value in relation to the `SelectedIndex` property of the state `ListBox`.

When the user clicks the Save button, we need to make the call back to the controller layer to store the user data ❺. You will notice that we are doing one small bit of data validation here: we are checking that the `password` and `passwordConfirm` values are the same ❹. The `passwordConfirm` isn't actually part of our model; it is simply a UI nicety. Where you do data validation can be an interesting discussion on its own. In some situations, you might know the valid values and simply put the checks in the setters of the model and catch exceptions in the `ChangeListeners` of the view. This can provide a lot of instant feedback to users while they are filling out forms. For larger things like either-or relationships, or things that require server checks, providing validation in the controller is the best option. Of course, since GWT is Java-based, you can use the same validation logic on the server and the client, saving on the effort you might have expended in more traditional Ajax development.

The final important thing to notice here is the `cleanup()` ❻ method. This simply cycles through the `PropertyChangeListeners` we added to the model class and removes them. This is important because once the application is done with the `UserEdit` widget, it needs a means to clean up the widget. If we didn't remove these listeners from the longer-lived `User` object, the `UserEdit` reference could never be garbage-collected, and would continue to participate in the event execution, needlessly taking up processor cycles. Of course, since the `AddressEdit` widget is doing this as well, we also need to clean up those listeners ❼.

Why do we clean up the `PropertyChangeListeners` and not the `ChangeListeners` and `ClickListeners` we used on the child widgets? Those change listeners will fall out of scope and be garbage-collected at the same time as our `UserEdit` class. Since they are private members, and the `UserEdit Composite` masks any other operations into itself, classes outside of the scope of `UserEdit` can't maintain references to them.

Now that we have the model and the view, and we have established the relationship between them, we need to set up the controller and send the user registration information back to the server.

4.3 The controller and service

You may have noticed that we passed a `UserCreateListener` instance into the `UserEdit` constructor. It is important in the design of your application that your custom widgets externalize any business logic. If you want to promote reuse of your view code, it shouldn't needlessly be tied to a particular set of business logic. In this example, though, our controller logic is pretty simple.

In this section, we will build the controller and the service servlet that will store our user in the database, pointing out places where you can extend the design with other functionality.

4.3.1 Creating a simple controller

The overall job of the controller is to provide access to business logic and provide a control system for the state of the view. Think, for a moment, about the controller level of an `Action` in a Struts application. Suppose it is triggered based on a user event, a form submission. It then validates the data and passes it into some kind of business logic (though many Struts applications, unfortunately, put the business logic right in the `Action` bean) and directs the view to update to a new state—redirecting to some other page. You should think of the controller in a GWT application as filling this role, but in a very different manner.

We will now take a look at a simple controller in the form of a `UserCreateListener`.

Problem

We need to create a controller to manage the action events and state for our view class. This will trigger the use-case actions of our application.

Solution

We will start by creating a simple implementation of the `UserCreateListener` interface, as presented in listing 4.5.

Listing 4.5 `UserCreateListenerImpl`—the controller for the `UserEdit` widget

```
package com.manning.gwtip.user.client;

import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.rpc.ServiceDefTarget;
public class UserCreateListenerImpl implements UserCreateListener {
    private UserServiceAsync service =
        (UserServiceAsync) GWT.create(UserService.class);        ❶ Create a service

    public UserCreateListenerImpl() {
        super();
        ServiceDefTarget endpoint = (ServiceDefTarget) service;
        endpoint.setServiceEntryPoint
             (GWT.getModuleBaseURL()+"UserService");         ❷ Bind the server address
    }

    public void createUser(User user){
        if ("root"
              .equals(
                 user.getUsername())) {                  Validate the data
            Window.alert("You can't be root!");
          return;
        }
        service.createUser(user, new AsyncCallback() {
            public void onSuccess(Object result) {
                Window.alert("User created.");
                // here we would change the view to a new state.
            }

            public void onFailure(Throwable caught) {
                Window.alert(caught.getMessage());            Alert user
                }
            });
        }
    }
```

Now we have a controller class for our `UserEdit` widget. This controller will make calls back to the remote service, completing the front end of the application.

DISCUSSION

This is a simple and somewhat trivial example, but it does demonstrate the logical flow you should see in your application. First, we get the service ❶ and bind it ❷, as you saw in chapter 3. Next, we implement the required method, `createUser()`. The method starts with a simple bit of data validation, and this could certainly be more advanced.

A good case would be to create a `UserValidator` object that could perform any basic validation we need. This simple example just shows where this would happen. Once the validation is done, we make the call to the remote service and handle the results. If this were part of a larger application, the `onSuccess()` method might call back out to another class to remove the `UserEdit` panel from the screen and present the user with another panel, like the forward on a Struts action controller.

Another validation case would be to present the user with an error notification if something "borked" on the call to the server. This might indicate an error, or data that failed validation on the server. For example, duplicate usernames can't easily be checked on the client. We have to check this at the point where we insert the user into the database.

All of which brings us to accessing the database in the service. For this, we will use the Java Persistence API.

4.3.2 JPA-enabling the model

One of the most common questions in GWT development is, "How do I get to the database?" You saw the basics of the Tomcat Lite configuration in chapter 3, but most people want to use something fancier than raw JDBC with their database. While JDBC works well, it is more cumbersome to work with than object-oriented persistence APIs. Today, that usually means some JPA provider like Hibernate, Kodo, OpenJPA, or TopLink Essentials.

There are two general patterns for using JPA in a GWT environment. The first is to JPA-enable a model shared between the client and the server. The second is to create a set of DTOs that are suitable for use on the client, and convert them in the service to something suitable for use on the server. Figure 4.5 shows the difference between these approaches in systems.

There are trade-offs to be made with either of these patterns. If you JPA-enable a shared model, your model classes are then limited to what the GWT JRE emulation classes can support, and to the general restrictions for being GWT-translatable (no argument constructor, no Java 5 language constructs currently, and so on). Using the DTO approach and converting between many transfer objects adds complexity and

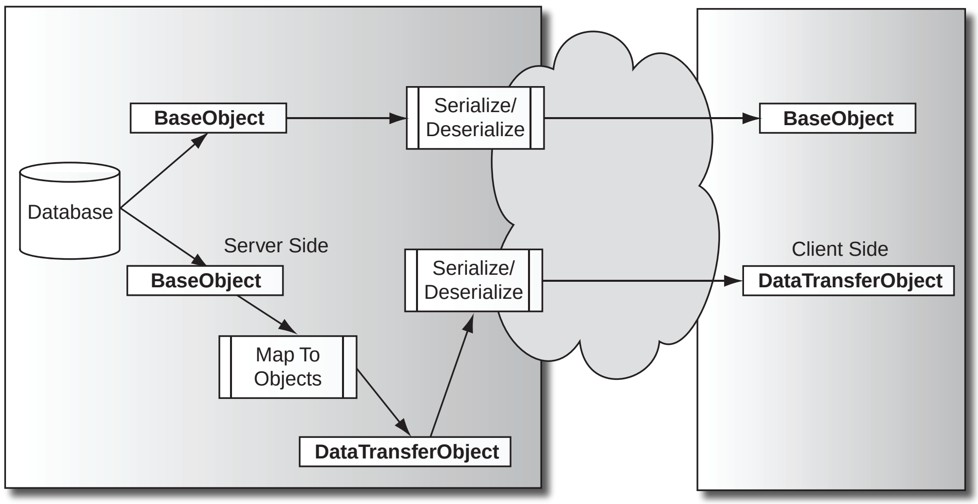


Figure 4.5 Flow from the server to the client with and without `DataTransferObjects`. Note that an additional mapping step is needed if DTOs are used.

potentially a lot of lines of code to your application, but it also provides you with finer-grained control over the actual model your GWT client uses.

Due to the restrictions in the direct JPA entity approach, and due to other advantages that a DTO layer can provide, it is common to use the DTO approach to communicate between GWT and a server-side model. We will take a look at this pattern, using transfer objects and all of the aspects it entails, in detail in chapter 9, where we will consider a larger Spring-managed application. In this chapter, we will look at JPA-enabling our GWT beans, which is the easiest method for simple stand-alone applications.

Problem

We want to enable our model beans for use with JPA providers to persist them to a database.

Solution

If you have been using JPA in the past, and you recall that GWT client components are bound to a Java 1.4 syntactical structure, you are likely thinking to yourself, "you can't add annotations to those beans!" Good eye—you would be thinking correctly. However, there is another way to describe JPA beans that doesn't normally get much attention but is designed for just such a scenario: using an orm.xml metadata mapping file. You, of course, also need a persistence.xml file to declare the persistence unit. Listing 4.6 shows the persistence unit definition.

Listing 4.6 Persistence.xml for the user

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
             http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
             version="1.0">

    <persistence-unit name="user-service"
        transaction-type="RESOURCE_LOCAL">
        <provider>
            org.hibernate.ejb.HibernatePersistence       <-- Specify using Hibernate
        </provider>
        <class>com.manning.gwtip.user.client.User</class>
        <class>com.manning.gwtip.user.client.Address</class>
        <properties>
            <property name="hibernate.dialect"
           value="org.hibernate.dialect.MySQLDialect"/>  <-- Specify using MySQL
            <property name="hibernate.connection.driver_class"
             value="com.mysql.jdbc.Driver"/>
            <property name="hibernate.connection.username"
                value="userdb"/>
            <property name="hibernate.connection.password"
                value="userdb"/>
            <property name="hibernate.connection.url"
             value="jdbc:mysql://localhost/userdb"/>
            <property name="hibernate.hbm2ddl.auto"
            value="create-drop"/>        <-- Drop and create the DB each time
        </properties>
    </persistence-unit>
</persistence>
```

If you have used JPA before, this will look pretty standard. We aren't using a `DataSource` here, just making direct connections to the database. We are also using Hibernate. Even though we have experience using both Hibernate and TopLink Essentials as JPA providers, we chose Hibernate for this example because although Hibernate requires more dependencies, it is actually easier to demonstrate in the GWT shell. TopLink works in the shell also, but it requires additional steps beyond dependencies, such as an endorsed mechanism override of the embedded Tomcat version of Xerces, and the inclusion of the TopLink agent (we will use TopLink in several other examples later in the book).

Next, we need an orm.xml file to specify the metadata we would normally specify in annotations. Listing 4.7 shows the mapping file for our user objects.

Listing 4.7 The orm.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
                 http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
                 version="1.0">

    <package>com.manning.gwtip.user.client</package>
    <entity class="User"                                   (1) Save steps with
       metadata-complete="true" access="PROPERTY">      <--    metadata-complete
        <table name="USER"/>
        <named-query name="User.findUserByUsernameAndPassword">
            <query>select u from User u
                where u.username = :username
                       and u.password = :password</query>
        </named-query>
        <attributes>
            <id name="username" />
            <one-to-one name="shippingAddress" >       Cascade to
                <cascade>                                Address objects
                    <cascade-all />
                </cascade>
            </one-to-one>
            <one-to-one name="billingAddress" >
                <cascade>
                    <cascade-all />
                </cascade>
            </one-to-one>
        </attributes>
    </entity>
    <entity class="Address" metadata-complete="true" access="PROPERTY">
        <table name="ADDRESS"/>
        <attributes>
            <id name="id">
                <generated-value strategy="IDENTITY"/>     Autoincrement
            </id>                                            ID field
        </attributes>
    </entity>
</entity-mappings>
```

This looks a lot like the annotations you might provide in the Java files themselves. Indeed, the orm.xml file maps pretty much one to one with the annotations. The important thing to pay attention to is the `metadata-complete` attribute on the `<entity>` element ❶. This tells the entity manager to use its default behavior for any properties on the object that aren't explicitly laid out in the file.

Discussion

For the `id` property on the `Address` object, we are using an `IDENTITY` strategy that will use MySQL's autoincrementing field type. This is another area where Hibernate and TopLink differ in use. TopLink doesn't support the `IDENTITY` scheme with its MySQL4 dialect. You must use a virtual sequence. In this case, the address `<entity>` element would look like this:

```
<entity class="Address" metadata-complete="true" access="PROPERTY">
     <table name="ADDRESS"/>
     <sequence-generator
         name="addressId" sequence-name="ADDRESS_ID_SEQUENCE" />
     <attributes>
         <id name="id">
             <generated-value strategy="SEQUENCE" generator="addressId"/>
         </id>
     </attributes>
</entity>
```

MySQL doesn't support sequences as a database structure, but TopLink will create a table to maintain the sequence value. Hibernate balks at this configuration, because it knows that MySQL doesn't support sequences. In short, don't expect these configuration files to be write-once-run-anywhere. Everything from the JPA provider used, right down to the database used, is coupled in your application. Hopefully as the EJB 3/JPA implementations mature, these issues will go away.

The orm.xml file is not well documented. We have found the best documentation to be simply looking at the schema file itself (http://java.sun.com/xml/ns/persistence/orm_1_0.xsd) and using a validating XML editor. Another option is to use the OpenJPA reverse-engineering tool (http://incubator.apache.org/openjpa/), which has the ability to create an orm.xml file for you from an existing database schema.

Now we have our JPA mappings, enabling us to store the objects from the model we created in the first step to the database. The last step is to create the service component that will bridge the gap between the client-side controller we created in section 4.3.1 and the database.

4.3.3 Creating a JPA-enabled service

We mentioned in chapter 3 that it is generally a best practice to create a separate local service and have your `RemoteServiceServlet` proxy calls into it. While we will be looking at that pattern in detail in chapter 9, we will simply create our service code in the servlet here. Since we are JPA-enabling our model, the fact that our service is dependent on the GWT libraries doesn't have any negative ramifications.

PROBLEM

We need to create a JPA-enabled service servlet for our application to take model objects sent from the controller layer of the client and store them in a database.

SOLUTION

Listing 4.8 shows the `RemoteServiceServlet` that will take the model objects and persist them to the database.

Listing 4.8 `UserServiceServlet` with JPA calls

```
public class UserServiceServlet extends RemoteServiceServlet
    implements UserService {
    private EntityManagerFactory factory;     <-- ❶ Cannot use @PersistenceUnit annotation

    public UserServiceServlet() {
        super();
        try{
             factory =
               Persistence.createEntityManagerFactory(     <-- ❷ Create the EntityManagerFactory
            "user-service");
        } catch(Exception e){
            e.printStackTrace();
        }

    }
    public void createUser(User user) throws UserServiceException {
        if("root".equals(user.getUsername())) {
           throw new UserServiceException(
              "You can't be root!");        <──┐ Remember, never
        }                                     ❸ trust the client
       try{
            EntityManager mgr = factory.createEntityManager();
            mgr.getTransaction().begin();
            mgr.persist(user);
            mgr.getTransaction().commit();     │ Catch persistence
        } catch(RollbackException e) {       <─┘ exceptions
            throw new UserServiceException(
           "That username is taken. Try another!");
        } catch(PersistenceException p) {
            throw new UserServiceException(
            "An unexpected error occurred: "+p.toString());
        }
    }
}
```

This likely looks familiar to anyone who has done JPA work, but there are some important things to cover concerning how we are interacting with JPA here.

DISCUSSION

This is a pretty simple class, but there are some important things to note about it. First, we aren't using the `@PersistenceUnit` annotation to get our `EntityManagerFactory` ❶. This would be the "regular" way to get `EntityManagerFactory` in Java EE 5. The Tomcat in GWTShell—or Tomcat in general, for that matter—doesn't support Java EE 5 dependency injection. You can use Spring for this in regular Tomcat, but since all the servlets in the GWT shell are created by the shell proxy service, we can't easily do this in hosted mode. So, we simply create the factory on our own in the constructor ❷.

The next thing of note is that we revalidate the user data ❸. Remember, you should always check the data on both sides. Checking it on the client side improves the user experience; checking it on the server improves application robustness.

Finally, we check for the `RollBackException`, which is thrown if the username is already in the database. This is a broken part of Hibernate. If we were using TopLink, we would catch the "correct" exception, `javax.persistence.EntityExistsException`. These types of differences are another example of the current challenges of using the present generation of JPA providers.

Now we have all the necessary components of a basic web application. We have a model layer that notifies listeners of changes, a view layer that binds to the model and provides updates, and a controller layer that captures our use cases and passes requests to the service. Last, we have a service that takes our model layer and persists it to a database. While these are the same components that a Struts or JSF developer might be used to building, they manifest differently in code and bring with them a new set of design issues for the web developer to consider.

4.4 Summary

In this chapter, we introduced the standard patterns that make up a GWT application. While the typical MVC approach certainly still applies, the way we go about using it is very different from the server-side MVC you might have used in traditional web frameworks in the past. First, the model object on the client is much more intelligent than a simple value object. Data binding is done on an eventing basis, and calls to the server are more service-oriented, representing a use case for your application, not necessarily a "screen."

Some of these things you already know how to do, yet the ways you expect to do them might not work at first in the GWT shell environment. Here, we explored one way to integrate JPA with a GWT application. In chapter 9, we will look at another pattern that integrates Spring, JPA, and DTOs. Chapter 9's approach can be used to make your application integrate more cleanly with existing JEE systems. The direct JPA- and GWT-enabled model pattern we used here is better for simple, standalone applications.

Until now we have focused on building GWT applications using the standard RPC approach for communicating with servers. In addition to the RPC and servlet method, you can also communicate with other server backends, including those that are not servlet based. In the next chapter, we will cover additional means of talking to servers, including basic HTTP and more advanced XML, SOAP, REST, Flash, and Comet. Along the way, we will also deal with the key related concepts of client/server object serialization with Java and JavaScript, and of security.

JAVA / WEB

Free ebook
SEE INSERT

GWT IN PRACTICE

Robert T. Cooper and Charlie E. Collins

With the Google Web Toolkit (GWT), you can build rich internet applications in Java using a Swing-like framework that provides dozens of pre-built components. GWT automatically converts your Java into JavaScript, so you can focus on application design and functionality without concern for browser quirks. This powerful tool opens up many new possibilities for web development, like using Java test and build tools, integrating with frameworks like the Java Persistence API, and easily reusing code throughout the layers of your application.

GWT in Practice is an example-driven, code-rich book written for web developers who have knowledge of the basics of GWT. You'll find scores of cookbook-style solutions for common and uncommon needs like drag-and-drop support for UI elements, data binding, processing streaming data, handling application state, automated builds, and continuous integration. And you will appreciate the attention it pays to the problem of integrating GWT with your existing applications and services.

What's Inside

- Create and customize widgets
- Learn the ins and outs of RPC
- Package, build, and test with Maven, Ant, and JUnit
- Work with the Java Persistence API
- Use GWT from Eclipse, NetBeans, and IDEA

Robert Cooper and **Charlie Collins** are Atlanta-based Java EE developers. Both contribute to many open source projects, including the gwt-maven plugins that support Maven-based builds for Google Web Toolkit.

For owners of this book, more information, code samples, and a free ebook are available from www.manning.com/GWTinPractice

"A true hands-on manual for GWT"

—Edmon Begoli
Oak Ridge National Laboratory

"Cooper and Collins live and breathe GWT—their code is spotless."

—Andrew Grothe
Triware Technologies Inc.

"Expertly explains the genius of this technology—a real gift."

—Peter Pavlovich, Kronos Inc.

"The more complex your project, the more critical is this information."

—Ara Abrahamian, NSI ltd

"The perfect guide to GWT in the real world."

—Devon Hillard
DigitalSanctuary.com

ISBN-13: 978-1933988290
ISBN-10: 1933988290
54499
9 781933 988290

MANNING

$44.99/Can $44.99 [INCLUDING eBOOK]
$27.50 PDF ebook at Manning.com