

WICKET IN ACTION

Martijn Dashorst
Eelco Hillenius



Wicket in Action

by Martijn Dashorst and Eelco Hillenius

Sample Chapter 8

Copyright 2008 Manning Publications

brief contents

PART 1 GETTING STARTED WITH WICKET.....1

- 1 ■ What is Wicket? 3
- 2 ■ The architecture of Wicket 24
- 3 ■ Building a cheesy Wicket application 45

PART 2 INGREDIENTS FOR YOUR WICKET APPLICATIONS.....79

- 4 ■ Understanding models 81
- 5 ■ Working with components:
labels, links, and repeaters 106
- 6 ■ Processing user input using forms 139
- 7 ■ Composing your pages 173

PART 3 GOING BEYOND WICKET BASICS197

- 8 ■ Developing reusable components 199
- 9 ■ Images, CSS, and scripts: working with resources 223
- 10 ■ Rich components and Ajax 238

PART 4	PREPARING FOR THE REAL WORLD.....	265
11	■ Securing your application	267
12	■ Conquer the world with l10n and i18n	282
13	■ Multitiered architectures	299
14	■ Putting your application into production	321

8

Developing reusable components

In this chapter:

- Learn the advantages of creating custom components
- Create complex compound components that include their own navigation

In chapter 7, we looked at group components strategies. You learned that panels are particularly well suited for creating components that can be reused in a variety of contexts, without the need to know anything about their internal structure.

In this chapter, we'll look at creating reusable components. The more generic and context-independent components are, the easier it is to reuse them. We'll first look at examples of generic components. We'll start with a locale component, which is simple; later, we'll add features to illustrate how you can create compound components. After that, we'll discuss how to develop a date-time panel, to illustrate how you can create compound components that participate in form processing.

In the second half of this chapter, we'll examine a domain-specific component: a discount list for the cheese store example. It will illustrate that components can have their own independent navigation. The discount list will also use some of the

components developed earlier in this chapter, and it will be the component we'll build upon in the chapters that follow.

Before we get back into coding, let's look at why you should take the time to create reusable components.

8.1 **Why create custom reusable components?**

Creating custom reusable components takes effort. You need to think about the proper abstractions, encapsulations, and couplings, and you have to design an API, document it, and so on. Why go through the effort? Here are a few good reasons:

- *To battle code duplication.* Code duplication (also known as *copying and pasting*) is one of the larger evils in software engineering. You'll get into situations where you fix a bug in one place but forget about duplicate code elsewhere. Code duplication is a telltale sign of software that isn't well thought out.
- *To save time.* If you solved a problem once and need to address a similar problem somewhere else, being able to reuse a component can be a huge time-saver. Even if the component needs to be tweaked to fit in this new use case, it's typically cheaper to do this than to solve the problem again from scratch. Often, the further your project progresses, the more time you'll save by being able to reuse components you wrote at an earlier stage.
- *To improve quality.* Less code means fewer bugs. And instead of implementing a quick (and often dirty) solution, you take a step back to think about what you really need to solve. That process often results in better code. On top of that, reusing components gets them more exposure (testing hours), so issues are often found more quickly.
- *To divide tasks more easily.* Breaking pages into sets of components enables you to better delegate development tasks across multiple team members.
- *To achieve better abstraction.* One of the main ideas behind modularization in programming is that you can manage complexity by breaking big problems into smaller ones. Custom components can help you tackle issues one at a time. Imagine a component that combines a search panel, a pageable results list, filters, and sort headers. Once you have that, you only have to focus on how you connect the data.

The remainder of this chapter looks at examples of creating custom components. We'll start with a component for switching a user's locale.

8.2 **Creating a component that selects the current locale**

Java's `Locale` object represents the combination of a language and country. Examples of locales are Thai/Thailand, English/USA, and English/UK. Wicket utilizes the user's locale to perform date and number conversions, do message lookups, even determine which file the markup is loaded from. We'll take a closer look at such capabilities in chapter 12, where you'll use them as part of a user properties panel.

Before we start implementing the `Locale` object, let's see what we mean by developing custom reusable components.

8.2.1 What are reusable custom components?

It sounds exciting to learn about authoring custom components, but you've already seen quite a few in previous chapters. For instance, this code fragment is a custom component:

```
add(new Link("add") {
    public void onClick() {
        setResponsePage(new EditContactPage(new Contact()));
    }
});
```

It isn't a reusable custom component, because the only way to put this functionality into another page is to copy it. But making it a reusable component is easy:

```
public class AddLink extends Link {
    private AddLink(String id) {
        super(id);
    }

    public void onClick() {
        setResponsePage(new EditContactPage(new Contact()));
    }
}
```

Because the second code fragment is defined in its own public class, you can put it on any page or any panel by instantiating it and adding it.

Another example of a reusable component is a required text field. Without it, we'd define a text field that enforces input like this:

```
TextField f = new TextField("name");
f.setRequired(true);
```

If we did that for 10 fields, we'd get a lot of code bloat. To avoid that, we can create a custom component that hides the call to `setRequired`. Listing 8.1 shows the code for such a component.

Listing 8.1 RequiredTextField component

```
public class RequiredTextField extends TextField {
    public RequiredTextField(String id) {
        super(id);
        setRequired(true);
    }

    public RequiredTextField(String id, IModel model) {
        super(id, model);
        setRequired(true);
    }
}
```

Using this code, we can declare a required component in one line:

```
RequiredTextField f = new RequiredTextField("name");
```

This component is trivial. But the need to hide implementation details becomes more obvious when we look at the implementation of a date format label (see listing 8.2). This component prints the date of its model object in MEDIUM notation as used in `java.text.DateFormat`.

Listing 8.2 The DateFmtLabel component

```
public class DateFmtLabel extends Label {
    public DateFmtLabel(String id) {
        super(id);
    }

    @Override
    public final IConverter getConverter(Class type) {
        return new StyleDateConverter("M-", true);
    }
}
```

If we have this component, we can do this:

```
add(new DateFmtLabel("until"));
```

Assuming a date value is provided—say, through a compound property model—it's formatted as follows: Sep 26, 2007. Internally, a format converter that can handle dates is configured, and the implementation details are hidden from those who wish to reuse this component. With preconfigured components like these, you can easily enforce consistency in your projects.

In the next section, we'll develop a custom component that displays the current locale and lets users change to another one.

8.2.2 Implementing the locale-selector component

In action, the locale-select component looks like the drop-down menu shown in the partial screen shot in figure 8.1.



Figure 8.1 The locale-select component in action

The current locale is English. If we select Thai from the drop-down menu, the display changes as shown in figure 8.2.

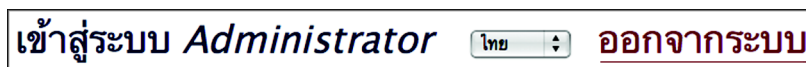


Figure 8.2 The locale changed to Thai

Listing 8.3 shows how the component is implemented.

Listing 8.3 Implementation of the locale select component

```
public class LocaleDropDown extends DropDownChoice {
    private class LocaleRenderer extends ChoiceRenderer {
        @Override
        public String getDisplayValue(Object locale) {
            return ((Locale) locale).getDisplayNames(getLocale());
        }
    }

    public LocaleDropDown(String id, List<Locale> supportedLocales) {
        super(id, supportedLocales);
        setChoiceRenderer(new LocaleRenderer());
        setModel(new IModel() {
            public Object getObject() {
                return getSession().getLocale();
            }

            public void setObject(Object object) {
                getSession().setLocale((Locale) object);
            }

            public void detach() {
            }
        });
    }

    @Override
    protected boolean wantOnSelectionChangedNotifications() {
        return true;
    }
}
```

Render choices

Display in current locale's language

Use custom-defined model

Use session's locale directly

ChoiceRenderers are used by components such as the drop-down menu to determine what should be rendered as visible values for users and what as internal identifiers. Letting the `wantOnSelectionChangeNotifications` method return `true` results in a postback every time a user changes the selection in the drop-down menu.

The nice thing about this code is that there isn't much to it. By extending the drop-down menu component, we let that component do the heavy lifting, and we can focus on the specific functionality we need.

Again, this is an example of how you can fairly easily build custom components by hard-wiring a particular component configuration. Instead of creating a custom class, we could have instantiated a drop-down menu and set the choice renderer and model on it directly. If we needed this functionality only once, that would be a fine choice. But if we might need the functionality multiple times, a single line of code now suffices:

```
add(new LocaleDropDown("localeSelect", Arrays
    .asList(new Locale[] { Locale.ENGLISH,
        Locale.SIMPLIFIED_CHINESE, new Locale("th") })));
```

It's nice that the component lets you switch from English to Thai; but if your Thai language skills are lacking, you suddenly won't understand what was on the page. Suppose, as an exercise, we provide a link that resets the session's locale to the value it had when the component was constructed. We want the link's display to be transparent to the user; the component should be a single entity that can be constructed as you just saw.

Figure 8.3 shows the locale-select component with a Reset link.

How do you create a component that consists of two components? The next section explains.



Figure 8.3 The drop-down menu with a Reset link

8.2.3 *Creating a compound component*

As you learned in chapter 7, panels are a good choice to create compound components. Panels can easily be reused in separate contexts without requiring users to know about their internal structure. That comes in handy here, because we're about to create a combination of components: the drop-down menu and a Reset link. We don't want users to have to include the markup for both components in their pages; we make it possible for them to use, say, a `` tag as a placeholder.

The code in listing 8.4 is the first step in developing the compound component. We wrap the locale drop-down menu we developed in the previous section in a panel.

Listing 8.4 Locale drop-down menu nested in a panel

```
public class LocaleDropDownPanel extends Panel {
    private static class LocaleDropDown extends DropDownChoice {
        private class LocaleRenderer extends ChoiceRenderer {
            @Override
            public String getDisplayValue(Object locale) {
                return ((Locale) locale).getDisplayNames(getLocale());
            }
        }

        LocaleDropDown(String id, List<Locale> supportedLocales) {
            super(id, supportedLocales);
            setChoiceRenderer(new LocaleRenderer());
            setModel(new IModel() {
                public Object getObject() {
                    return getSession().getLocale();
                }

                public void setObject(Object object) {
                    getSession().setLocale((Locale) object);
                }

                public void detach() {
                }
            });
        }
    }
}
```

```

@Override
protected boolean wantOnSelectionChangedNotifications() {
    return true;
}

public LocaleDropDownPanel(String id, List<Locale> supportedLocales) {
    super(id);
    add(new LocaleDropDown("localeSelect", supportedLocales));
}
}

```

And here's the code for `LocaleDropDownPanel.html`:

```

<wicket:panel>
  <select wicket:id="localeSelect">
    <option value="nl">Dutch</option>
    <option value="en">English</option>
  </select>
</wicket:panel>

```

Pretty straightforward, isn't it?

The option elements in the markup will be discarded. They're here so you can preview the markup in an arbitrary editor—or even your browser—and have an idea what the panel will look like. If we don't care about the preview, we can do this:

```

<wicket:panel>
  <select wicket:id="localeSelect" />
</wicket:panel>

```

The instantiation works much the same as before:

```

add(new LocaleDropDownPanel("localeSelect",
    Arrays.asList(new Locale[] { Locale.ENGLISH,
        Locale.GERMAN, Locale.SIMPLIFIED_CHINESE })));

```

But the markup used as a placeholder for the component is now something like this

```

<span wicket:id="localeSelect" />

```

rather than this:

```

<select wicket:id="localeSelect">
  <option value="nl">Dutch</option>
  <option value="en">English</option>
</select>

```

If we tried the latter, the resulting markup would be as follows:

```

<select>
  <select name="localeSelect:localeSelect"
    onchange="window.location.href='?wicket:interface=
5:localeSelect:localeSelect::IOnChangeListener::&localeSelect:locale
Select=' + this.options[this.selectedIndex].value;">
    <option selected="selected" value="0">English</option>
    <option value="1">German</option>
    <option value="2">Chinese (China)</option>
  </select>
</select>

```

A `select` tag nested within another `select` tag isn't valid HTML, so the output is wrong. The HTML looks this way because panels replace what is between the tags they're attached to, not the tags themselves.

If we care about previewability, we can use tags. Here we've added dummy markup for a `select` next to where our panel (which in turn contains a `select`) will be replaced:

```
<span wicket:id="localeSelect" />
<wicket:remove>
  <select>
    <option value="nl">Dutch</option>
    <option value="en">English</option>
  </select>
</wicket:remove>
```

These `<wicket:remove>` tags instruct Wicket to skip everything between them, so you can insert any markup you want for the purpose of previewability.

In case you think this is a half-baked solution, we can do something smart to let users use a `<select>` tag as a placeholder for our panel. Using our panel (which essentially is a specialized `select`) feels like using a normal `select`. At runtime, we can convert the tag to something harmless (like a `` tag) by putting this in our panel:

```
@Override
protected void onComponentTag(ComponentTag tag) {
    super.onComponentTag(tag);
    tag.setName("span");
}
```

The `name` property of `ComponentTag` is mutable and determines what the actual HTML tag is when it's rendered. If we render the component with this code in place, the output is as follows:

```
<span>
  <select name="localeSelect:localeSelect"
onchange="window.location.href='?wicket:interface=
5:localeSelect:localeSelect::IOnChangeListener::&
localeSelect:localeSelect=' +
this.options[this.selectedIndex].value;">
    <option selected="selected" value="0">English</option>
    <option value="1">German</option>
    <option value="2">Chinese (China)</option>
  </select>
</span>
```

This is the case regardless of what tag is used in the markup: it's always set to ``.

Most components shipped with Wicket don't alter tags like we just did. You have fewer surprises that way, which increases the chance that you'll write robust programs. But changing the tag can be a convenient trick to facilitate better previewability in your projects.

The locale-select component currently has the same functionality it had before, but now it's wrapped in a panel. In the next section, we'll add the Reset link.

8.2.4 Adding a Reset link

The Reset link implements the functionality to change the locale back to what the user's locale was when the component was instantiated. The first step is to save the locale before it is changed. In this example, we do that lazily through the model (see listing 8.5).

Listing 8.5 setObject implementation that saves the current locale

```
public void setObject(Object object) {
    Session session = getSession();
    Locale keep = (Locale) session.getMetaData(SAVED);
    if (keep == null) {
        session.setMetaData(SAVED, getLocale());
    }
    session.setLocale((Locale) object);
}
```

We store the locale as session metadata. Metadata exists for components, request cycles, sessions, and applications; you can use it to store arbitrary objects such as configuration data, authorization data, or just about anything you wish. In this example, it makes sense to use this facility so we don't have to force users of our component to provide a custom session that stores the initial locale as a property.

The metadata key is defined like this:

```
static MetadataKey SAVED = new MetadataKey(Locale.class) { };
```

Now, we can add to the panel a link that uses this metadata to reset the locale (see listing 8.6).

Listing 8.6 Implementation of the Reset link

```
add(new Link("reset") {
    @Override
    public void onClick() {
        Session session = getSession();
        Locale keep = (Locale) session.getMetaData(SAVED);
        if (keep != null) {
            session.setLocale(keep);
            session.setMetaData(SAVED, null);
        }
    }
});
```

The link gets the saved locale from the session, if it exists, and if so, sets the locale to that value and nulls the metadata entry.

Here's the panel template:

```
<wicket:panel>
  <select wicket:id="localeSelect" />
  <a href="#" wicket:id="reset">[reset]</a>
</wicket:panel>
```

Let's look at what we've achieved so far. We created a component that lets users switch their locale. To use this component, you don't have to know anything about how it's implemented; nor does it have to know anything about what else is on the page it's placed on. The component can handle input, such as selection changes or a click of the Reset link, independent of what is on the page. The component is truly self-contained. You'll see it again in chapter 12 on localization.

Remember the `DateFmtLabel` component from the beginning of this chapter? In the next section, we'll develop an input-receiving and time-enabled counterpart, which will show how you can develop composite components that participate in form processing.

8.3 *Developing a compound component: DateTimeField*

Our goal in this section is to create a component, `DateTimeField`, that provides the user with separate input fields for the date, hours, and minutes. The component should hide from users the internal implementation details; users should provide a model that works as a date and be done with it.

When it's finished, you'll be able to use the component as shown in listing 8.7.

Listing 8.7 Example of how `DateTimeField` can be used

```
public class DateTimeFieldPage extends WebPage {
    private Date date = new Date();

    public DateTimeFieldPage() {
        Form form = new Form("form") {
            @Override
            protected void onSubmit() {
                info("new date value: " + date);
            }
        };
        add(form);
        PropertyModel model = new PropertyModel(this, "date");
        form.add(new DateTimeField("dateTime", model));
        add(new FeedbackPanel("feedback"));
    }
}
```

Here's the markup:

```
<form wicket:id="form">
    <span wicket:id="dateTime">[date time field here]</span>
    <input type="submit" value="set" />
</form>
<div wicket:id="feedback">[feedback here]</div>
```

When rendered in a browser, it looks like figure 8.4.

As you can see, this component is a composite. Let's examine how to implement it.



Figure 8.4
The `DateTimeField` component as rendered in a browser

8.3.1 Composite input components

Things can get tricky when you want to create compound components that act like form components. You can nest form components in panels, and their individual models will be updated without any problem; but the model of the panel isn't automatically updated. That often isn't an issue: the locale drop-down menu we developed works fine embedded in a normal panel, and the panel doesn't need to have a model of its own. But consider a date-time field that works on a model (which produces a date) and which internally breaks dates into separate date (day of month) and time (hours and minutes) fields. You could let each of these nested components update its part of the model, but then you wouldn't have a single action for updating the model object of the outer component. Also, because validation is only executed for form components, you would have to pass validators to nested components—and that would bloat your component's API and expose implementation details.

The solution is to use a special kind of component that is both a panel and a form component: `FormComponentPanel`. Like normal panels, form component panels are associated with markup files; but unlike panels, they participate in form processing. We'll base the date-time field on this special component. In the next section, we'll start by embedding the form components that do the real job of receiving input for us.

8.3.2 Embedding form components

The first part of writing the date-time field is straightforward. We already know that we need to nest three text-field components: one for the date, one for the hours, and one for the minutes. These components should work with their own models, and the date-time field should use these model values to update its own model as an atomic operation. In other words, the component should update its model only when all the inputs of the nested components are valid and can be combined to form a date that passes the component's validation.

We'll look next at how a date-time field component can be implemented. The code is extensive, so it's broken up over several sections. Listing 8.8 shows the first part.

Listing 8.8 *DateTimeField* embedding the form components

```
public class DateTimeField extends FormComponentPanel {  
    private Date date;  
    private Integer hours;  
    private Integer minutes;  
    private final DateTextField dateField;  
    private final TextField hoursField;  
    private final TextField minutesField;  
  
    public DateTimeField(String id) {  
        this(id, null);  
    }  
  
    public DateTimeField(String id, IModel model) {  
        super(id, model);  
    }  
}
```

```

setType(Date.class);
PropertyModel dateFieldModel = new PropertyModel(this, "date");
add(dateField = new DateTextField("date", dateFieldModel));
dateField.add(new DatePicker());
hoursField = new TextField("hours", new PropertyModel(this,
    "hours"), Integer.class);
add(hoursField);
hoursField.add(NumberValidator.range(0, 24));
hoursField.setLabel(new Model("hours"));
minutesField = new TextField("minutes", new PropertyModel(
    this, "minutes"), Integer.class);
add(minutesField);
minutesField.add(NumberValidator.range(0, 59));
minutesField.setLabel(new Model("minutes"));
}

```

Each field works on its own model object (date, hours, and minutes). Note that we don't have to add getters and setters for the private members date, hours, and minutes, because property models can work on them directly. We can decide to regard such fields as implementation details and not expose them via getters and setters.

The component exposes two constructors. The one without a model argument is useful when you want to use the component with compound property models.

The hours and minutes text fields both have validators attached to ensure valid input, and they have labels set for error reporting. You've seen how this works in earlier chapters.

A last interesting bit from this fragment is the use of a factory method that produces the date text field:

```
add(dateField = new DateTextField("date", dateFieldModel));
```

By default—in this component—this factory method is implemented like this:

```

protected DateTextField newDateTextField(String id,
    PropertyModel dateFieldModel) {
    return DateTextField.forShortStyle(id, dateFieldModel);
}

```

By delegating the construction of the date text field to a factory method, we enable users to provide their own versions or configurations of the text field. They could, for instance, specify a date pattern by overriding the factory method like this:

```

dateTimeField = new DateTimeField("dateTime", model) {
    @Override
    protected DateTextField newDateTextField(String id,
        PropertyModel dateFieldModel) {
        return DateTextField.forDatePattern(id, dateFieldModel,
            "dd-MM-yyyy");
    }
};

```

There are no surprises in the first part of the date-time field. Next, we'll look at how to synchronize the models of the nested components with the model of the top component.

This wasn't relevant for the locale-selection component earlier, because it works with its own model and isn't meant to interface with a model provided by users. This component, however, is meant to be used as follows:

```
form.add(new DateTimeField("dateTime", model));
```

Users will expect the date-time field to use the provided model object. If the model produces a date like 12 January 2008, 11:00 AM, they will expect the date and time fields to display values accordingly; and if end users change these fields and submit them as part of a form, the users will expect the date to be changed properly.

We need to synchronize the models that are used by the embedded components in a separate step so the change is atomic: either all nested fields validate and the date is updated properly, or the nested fields don't validate, in which case the date isn't updated. The next section shows how to do this.

8.3.3 Synchronizing the models of the embedded components

To keep the models of the nested components and the top component synchronized, we need to override two methods: `onBeforeRender`, which prepares for rendering, and `convertInput`, which handles receiving input. `onBeforeRender` is defined at the level of the `Component` base class. We'll use it as a hook into the component lifecycle so that we can synchronize the internal models right before the nested components are rendered (see listing 8.9).

Listing 8.9 *DateTimeField* preparing for rendering

```
@Override
protected void onBeforeRender() {
    date = (Date) getModelObject();
    if (date != null) {
        Calendar calendar = Calendar.getInstance(getLocale());
        calendar.setTime(date);
        hours = calendar.get(Calendar.HOUR_OF_DAY);
        minutes = calendar.get(Calendar.MINUTE);
    }
    dateField.setRequired(isRequired());
    super.onBeforeRender();
}
```

**Synchronize
member
variables**

**Synchronize
required flag**

**Call super
(required)**

This code reads the current value of the model object—which should be a date—and extracts the days, hours, and minutes values from it so they can be used by the nested text fields. It's important to realize that the date-time field doesn't "own" its model or model value. The model is a reference to some data passed in, so it may have been changed from the outside between requests. For instance, in the `DateTimeFieldPage` example (listing 8.7), we could include a link in the page to set the date-time to "now":

```
add(new Link("now") {
    @Override
    public void onClick() {
```

```

        date = new Date();
    }
});

```

In this case, the date used by the model of the date-time field would be changed without our direct knowledge. So, it's a good idea to determine the current model value right before rendering, assuming it might have been changed since the last time we checked (and saved) it, which explains the call to `getModelObject()`.

Notice two other things in the method implementation: we have to remember to call the super-implementation of the `onBeforeRender` method (although in this case it doesn't matter whether that is done toward the start or end of the method), and we set the required bit of the date text field according to whether the component is required. In this case, hours and minutes are always optional.

The second method, `convertInput`, handles the receiving of user input. Listing 8.10 defines this method.

Listing 8.10 `DateTimeField` receiving input

```

@Override
protected void convertInput() {
    Date date = (Date) dateField.getConvertedInput();
    if (date != null) {
        Calendar calendar = Calendar.getInstance(getLocale());
        calendar.setTime(date);
        Integer hours = (Integer) hoursField.getConvertedInput();
        Integer minutes = (Integer) minutesField.getConvertedInput();
        if (hours != null) {
            calendar.set(Calendar.HOUR_OF_DAY, hours % 24);
            calendar.set(Calendar.MINUTE,
                (minutes != null) ? minutes : 0);
        }
        setConvertedInput(calendar.getTime());
    } else {
        setConvertedInput(null);
    }
}

```

The `convertInput` method is called during the first phase of component validation (before any validators are executed). Implementations should parse user input and either set the converted input using `setConvertedInput` or report that the input couldn't be interpreted directly. A form component panel typically doesn't receive user input directly. But because its nested components do, and because it wants to update its own model value accordingly, we override this method.

Form processing functions like validating and updating models are done using depth-first (*postorder*) traversals of the component tree. In effect, this means the children of compound components are processed before the top component. That is exactly what we need here, because we want to construct the date from the already-processed nested components. The tricky thing is that when Wicket calls `convertInput`, form processing hasn't finished performing validation, and the models of the

nested components aren't yet updated. We can't use the date, hours, and minutes member variables to construct the date. Instead, we can manually call `getConvertedInput` on the nested components. We can safely do that because `convertInput` is called only when a form component is marked "valid" (meaning it passed all validation), and the method to determine that (`isValid`) returns `true` only when all children are valid. We can implement `convertInput` assured that the input of the nested components is valid.

After doing a bit of date calculation, we set the converted date. Note that because we can assume all the validators of the nested components executed successfully, we know that the hours and minutes values we get from the nested components are valid: we added validators to them to enforce that.

One last method will make the component well-rounded:

```
@Override
public String getInput() {
    return dateField.getInput() + ", " + hoursField.getInput() + ":"
        + minutesField.getInput();
}
```

This method is used by the default implementation of `convertInput`. It's also useful at various locations for error reporting (for example, for validators with messages that use the `${input}` variable).

We'll use this component in the next and last sections of this chapter, where we'll develop another custom component: the discount list.

8.4 Developing a discount list component

The locale-select component and date-time field are both examples of generic components; they can function in a large variety of contexts. In this section, we'll develop a component that is specific for a certain domain. We may be able to reuse it across our domain—the cheese store—but even if we use it only once, developing it as a separate component still makes sense. Doing so allows us to focus on problems one at a time; and once we have the component, we can place it on any page or panel. That also makes refactoring a lot easier.

The component we're about to develop lists discounts and has an administration function for editing those discounts. Switching between the normal list and editing is handled by the component.

The domain model can be described as follows. A discount consists of a reference to a cheese, a description, a discount (which is a percentage), and a date from/until when the offer is valid. Figure 8.5 shows a UML diagram.

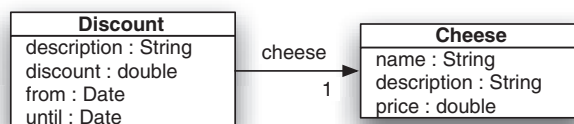


Figure 8.5
The discount list component's mini-domain model

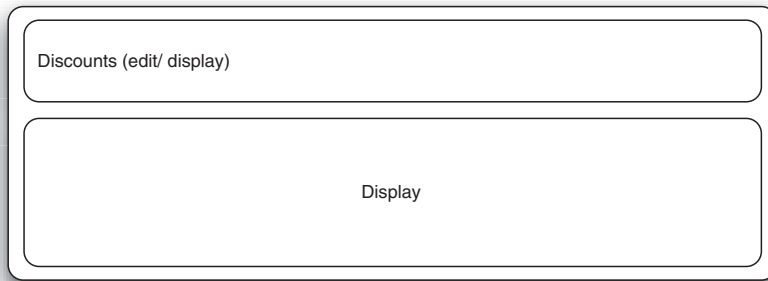


Figure 8.6 The layout of the discount list component

Schematically, the layout of the discount-list component can be drawn as shown in figure 8.6.

The component's top section has a static title and a link that displays either Edit or Display, depending on the component's state. The rest of the section (Display) displays either a read-only list with discounts or a form with a list of input fields to directly edit those discounts.

When the component is in list mode, it displays the discounts, and the link says Edit. If the link is clicked, the display changes to a form in which the list can be directly edited and which has buttons for adding and removing rows. In edit mode, the link in the title section says Display; when clicked, it changes the display back to the normal list.

In the next section, we'll look at the top-level component that contains the header and list sections.

8.4.1 The container

The container, `DiscountsPanel`, nests the header and list components and needs to track whether it's in edit mode (if not, it should display the read-only list). Listing 8.11 shows the code for the container component.

Listing 8.11 The container component

```
public class DiscountsPanel extends Panel {
    private boolean inEditMode = false;
    public DiscountsPanel(String id) {
        super(id);
        add(new DiscountsList("content"));
        final Link modeLink = new Link("modeLink") {
            @Override
            public void onClick() {
                inEditMode = !inEditMode;
                setContentPanel();
            }
        };
        modeLink.addActionListener(this);
    }
}
```

← **Current mode**

Switch mode

```

    }
};
add(modeLink);
modeLink.add(new Label("linkLabel", new AbstractReadOnlyModel() {
    @Override
    public Object getObject() {
        return inEditMode ? "[display]" : "[edit]";
    }
}));
}

void setContentPanel() {
    if (inEditMode) {
        addOrReplace(new DiscountsEditList("content"));
    } else {
        addOrReplace(new DiscountsList("content"));
    }
}
}

```

Mode-dependent link label

Add or replace child

As you can see, the component initially nests the `DiscountList` component (`inEditMode` starts out being `false`). Whenever the mode link is clicked, the mode is switched and the content component is replaced accordingly.

Reflect on what we've achieved. Using component replacement, we created a component that can perform its own self-contained navigation. We created a portlet-like miniapplication that can function in any page without further configuration. This is quite a contrast to many of Wicket's competitors, which force you to do everything with page navigation.

As great as component replacement is in many cases, you need to keep a couple of things in the back of your mind.

COMPONENT-REPLACEMENT GOTCHAS

Consider that as soon as you start applying component replacement, you lose bookmarkability. In order to provide Back button support, Wicket records versions of pages that make structural changes; that way, if the Back button is clicked, Wicket can roll back to a previous structure.

Also keep in mind that the identifiers of the replacement and replaced components must be the same. Unless your component actively changes the tag it's linked to (as in section 8.1.4), you have to be careful that the replacement is compatible with the structure of the component it replaces. For instance, you can't replace a text field with a list view. But if you use panels and fragments as we do in this example, you'll never run into this problem.

The last thing we need to finish the top-level part of the component is the markup. As you can see, it's straightforward:

```

<wicket:panel>
  <div>
    <div>
      Special discounts &nbsp;  
    
```

```

        <a href="#" wicket:id="modeLink">
            <span wicket:id="linkLabel">[label]</span>
        </a>
    </div>
    <span wicket:id="content">[panel content here]</span>
</div>
</wicket:panel>

```

Mode-switch link

Content section

The first part of the component is finished. Using the component is as simple as this:

```
add(new DiscountsPanel("discounts"));
```

In the next two sections, we'll develop the discount list panels, which will be placed one at a time in the content section of the component we just built. We'll start with the default panel: the read-only discounts list.

8.4.2 The read-only discounts list

In view mode, the read-only list is the component that is displayed as the content. Figure 8.7 shows the discount list in view mode. It displays discounts the way end users would see them, except that it has an Edit link embedded.

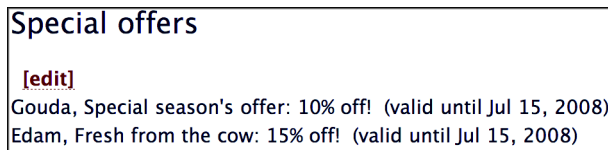


Figure 8.7
The discount list component
in view mode

For the implementation, we'll embed a refreshing-view component (a list that recalculates its children on every render) in a panel, so we can easily use the list elsewhere without having to worry about what the internal structure looks like. Listing 8.12 shows the implementation.

Listing 8.12 Implementation of the read-only list component

```

public class DiscountsList extends Panel {
    public DiscountsList(String id) {
        super(id);
        add(new RefreshingView("discounts") {
            @Override
            protected Iterator getItemModels() {
                return new ModelIteratorAdapter(MyDataBase.getInstance()
                    .listDiscounts().iterator()) {
                    @Override
                    protected IModel model(Object object) {
                        return new CompoundPropertyModel((Discount) object);
                    }
                };
            }
        });
    }

    @Override
    protected void populateItem(Item item) {

```

Wrap discounts list

CompoundPropertyModel

```

        item.add(new Label("cheese.name"));
        item.add(new PercentLabel("discount"));
        item.add(new Label("description"));
        item.add(new DateFmtLabel("until"));
    }
}
});
}
}

```

The `getItemModels` method needs to return an iterator that produces `IModel` objects. The `ModelIteratorAdapter` wraps the iterator of the discounts list, and we wrap each object that is produced by the iterator in a compound property model. Because every list item will have a compound property model set, we can add components without explicitly providing their models; the child components will use their identifiers as property expressions on those models.

Listing 8.13 shows the markup for the read-only list.

Listing 8.13 Markup for the read-only list component

```
<wicket:panel>
  <li wicket:id="discounts">
    <strong><span wicket:id="cheese.name">name</span></strong>,
    <span wicket:id="description">description</span>:
    <span wicket:id="discount">discount</span> off!&nbsp;
    (valid until <span wicket:id="until">until</span>)
  </li>
</wicket:panel>
```

Note that we use another custom component: a label that formats its model value as a percentage. As an exercise, think about how you would implement that, and compare it to the component available in the code that comes with this book.

Now that we've implemented the read-only list for view mode, we're ready to look at the edit list for the discount-list component's edit mode.

8.4.3 The edit-discounts list

The edit list provides a form for bulk editing discounts; it includes a button for creating a new discount and links for removing discounts. When we're done, it will look like figure 8.8.

Let's start with the simple part and create a panel with a form, a button for a new discount, and a save button that persists the bulk changes. You can see the implementation in listing 8.14.





Special offers									
[display]									
name	%	description	from			until			remove
Gouda	10	Special season's offer	6/15/08		13 : 53	7/15/08		13 : 53	[remove]
Edam	15	Fresh from the cow	6/15/08		13 : 53	7/15/08		13 : 53	[remove]

Figure 8.8 A screenshot of the edit-discounts list

Listing 8.14 Form portion of the list-editing component

```

public final class DiscountsEditList extends Panel {

    private List<Discount> discounts;

    public DiscountsEditList(String id) {

        super(id);
        Form form = new Form("form");
        add(form);
        form.add(new Button("newButton") {
            @Override
            public void onSubmit() {
                DiscountsEditList.this.replaceWith(
                    new NewDiscountForm(DiscountsEditList.this.getId()));
            }
        });
        form.add(new Button("saveButton") {
            @Override
            public void onSubmit() {
                MyDataBase.getInstance().update(discounts);
                info("discounts updated");
            }
        });
        form.add(new FeedbackPanel("feedback"));
    }
}

```

Replace self

Save list contents

...

The Wicket part of this code should hold no secrets by now. To make the example somewhat realistic, we're keeping a reference to a list of discounts retrieved from the database, which after updating is saved back to the database.

It's more interesting to look at the use of `replaceWith`. This method, which is defined on the component base class, is shorthand for doing `getParent().replace(..)`, where `replace` is a method defined on `MarkupContainer`. Either form is fine.

The first part of the repeater is implemented as shown in listing 8.15.

Listing 8.15 Repeater's iterator

```

RefreshingView discountsView = new RefreshingView("discounts") {

    @Override
    protected Iterator getItemModels() {
        if (discounts == null) {
            discounts = DataBase.getInstance().listDiscounts();
        }
        return new ModelIteratorAdapter(discounts.iterator()) {
            @Override
            protected IModel model(Object object) {
                return EqualsDecorator
                    .decorate(new CompoundPropertyModel((Discount) object));
            }
        };
    }
}

```


This is almost the same as the way we defined `getItemModels` in the read-only list. If it were exactly the same, we probably would have made a common base class for it. But here we assign the discounts list we get from the database to the `discounts` member. Because the database returns a snapshot of its current contents when servicing `listDiscounts` calls, we in effect keep a reference to a working copy of the database contents. The `Save` button's `onSubmit` method synchronizes the working copy with the database contents by calling the database's `update` method.

Also, because we're working in a form, we don't want the repeater to discard its child components every time rendering completes (the default behavior when refreshing a view). Instead, it should refresh only when the model objects are changed. We can configure this by setting the `item-reuse` strategy, as follows:

```
discountsView.setItemReuseStrategy(
    ReuseIfModelsEqualStrategy.getInstance());
```

That, together with wrapping the model with `EqualsDecorator`—which returns a model proxy that implements `equals` and `hashCode` using the model object—makes the repeater refresh only when the underlying model changes. Listing 8.16 shows the implementation of `EqualsDecorator` (an elaborate explanation of it is outside the scope of this book).

Listing 8.16 Model proxy that implements `equals` and `hashCode`

```
public final class EqualsDecorator {
    private EqualsDecorator() { }

    public static IModel decorate(final IModel model) {
        return (IModel) Proxy.newProxyInstance(model.getClass()
            .getClassLoader(), model.getClass().getInterfaces(),
            new Decorator(model));
    }

    private static class Decorator implements
        InvocationHandler, Serializable {
        private final IModel model;

        Decorator(IModel model) { this.model = model; }

        public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
            String methodName = method.getName();
            if (methodName.equals("equals")) {
                if (args[0] instanceof IModel) {
                    return Objects.equal(model.getObject(), ((IModel) args[0])
                        .getObject());
                }
            } else if (methodName.equals("hashCode")) {
                Object val = model.getObject();
                return Objects.hashCode(val);
            } else if (methodName.equals("writeReplace")) {
```

```

        return new SerializableReplacement(model);
    }
    return method.invoke(model, args);
}
}

private static class SerializableReplacement implements
    Serializable {
    private final IModel model;

    SerializableReplacement(IModel model) { this.model = model; }

    private Object readResolve() throws ObjectStreamException {
        return decorate(model);
    }
}
}

```

The last code fragment of the list-editing component is shown in listing 8.17: the `populateItem` implementation.

Listing 8.17 Repeater's `populateItem` implementation

```

@Override
protected void populateItem(Item item) {
    item.add(new Label("cheese.name"));
    item.add(new PercentageField("discount"));
    item.add(new RequiredTextField("description"));
    item.add(new DateTimeField("from"));
    item.add(new DateTimeField("until"));

    final Discount discount = (Discount) item.getModelObject();
    final Link removeLink = new Link("remove") {
        @Override
        public void onClick() {
            MyDataBase.getInstance().remove(discount);
        }
    };
    item.add(removeLink);
    removeLink.add(new SimpleAttributeModifier("onclick",
        "if(!confirm('remove discount for "
            + discount.getCheese().getName()
            + " ?')) return false;"));
}

```

Another custom component

What, another custom component? That's what happens once you get the hang of it: custom components everywhere!

The percentage field is implemented in listing 8.18.

Listing 8.18 Implementation of the percentage field

```

public class PercentageField extends TextField {
    public PercentageField(String id) {
        super(id, double.class);
    }
}

```

Type is double

```

public PercentageField(String id, IModel model) {
    super(id, model, double.class);
}

@Override
public final IConverter getConverter(Class type) {
    return new IConverter() {
        public Object convertToObject(String value, Locale locale) {
            try {
                return getNumberFormat(locale).parseObject(value);
            } catch (ParseException e) {
                throw new ConversionException(e);
            }
        }

        public String convertToString(Object value, Locale locale) {
            return getNumberFormat(locale).format((Double) value);
        }

        private NumberFormat getNumberFormat(Locale locale) {
            DecimalFormat fmt = new DecimalFormat("##");
            fmt.setMultiplier(100);
            return fmt;
        }
    };
}

```

Fixed converter

Conversion exception

If we had used a regular text field, we would have seen 0.20 or something similar for a discount of 20%. That isn't exactly user-friendly. The percentage field component translates 0.20 to 20 and back again, so the user doesn't have to calculate back and forth. It uses a converter to perform that calculation, and the converter in turn uses a decimal formatter.

Converters are responsible for converting model values to user-facing output and user input back to model values. The percentage field component sets up the converter to be used for itself by overriding the `getConverter` method and making the method final to prevent misuse. We'll take another look at converters in chapter 13.

We'll leave the component's markup and the implementation of the new discount form to your imagination (or you can look it up in the source code that comes with this book). It's time to wrap up the chapter.

8.5 Summary

In this chapter, we looked at how to create custom reusable components for Wicket, and why you would want to do so. The first few examples packaged component configuration into new classes. That can be an effective strategy to hide complexity, to enforce consistency throughout your project(s), and to reduce code duplication.

The locale-select component and date-time field component are examples of generic components that can be used in many different contexts. The locale-select component with a Reset link is an example of a composite component that acts as a

single self-contained unit for its users. Users don't have to know that the component combines a drop-down menu and a link: a single line of Java code and a single line of markup are enough to use the component.

The date-time field extends that concept and is a composite component that participates in form processing (it's updated on form submits) and that automatically updates its model depending on the input of its nested form components.

The last example in this chapter created a domain-specific cheese store discount list. It reused some of the components we developed earlier, and it showed how by using component replacement, components can implement their own independent means of navigation, even for editing data.

In the next chapter, we'll discuss Wicket resources, which you can use to include things like images, JavaScript, and CSS references in your custom components.

WICKET IN ACTION

Martijn Dashorst and Eelco Hillenius

FOREWORD BY Jonathan Locke



Wicket bridges the mismatch between the web's stateless protocol and Java's OO model. The component-based Wicket framework shields you from the HTTP under a web app so you can concentrate on business problems instead of the plumbing code. In Wicket, you use logic-free HTML templates for layout and standard Java for an application's behavior. The result? Coding a web app with Wicket feels more like regular Java programming.

Wicket in Action is a comprehensive guide for Java developers building Wicket-based web applications. It introduces Wicket's structure and components, and moves quickly into examples of Wicket at work. Written by core committers, this book shows you the "how-to" and the "why" of Wicket. You'll learn to use and customize Wicket components, to interact with Spring and Hibernate, and to implement rich Ajax-driven features.

What's Inside

- All of Wicket's basic concepts and components
- Security, localization, and internationalization
- Creating custom reusable components
- Wicket's Ajax and JavaScript capabilities
- Working with databases and resources
- Prepare your application for production

About the Authors

Martijn Dashorst has been actively involved in Wicket since it was open-sourced, and speaks regularly at conferences, including JavaOne and JavaPolis. **Eelco Hillenius** has been part of Wicket's core team almost from the start. He works for Teachscape where he is helping to build the next e-learning platform.

For online access to the authors, code samples, and (for owners of this book) a free ebook, go to www.manning.com/WicketinAction

"This is the complete and authoritative guide to Wicket, written and reviewed by the core members of the Apache Wicket team. If there's anything you want to know about Wicket, you are sure to find it in this book."

—From the Foreword by Jonathan Locke
Founder and Architect of Apache Wicket

"With this book, Wicket will become the greatest territory the Dutch have settled since Manhattan."

—Nathan Hamblen
Senior Software Developer
Teachscape, Inc.

"Loved the sample application—it tied everything together."

—Phil Hanna
Senior Software Developer
SAS Institute

"The essential guide for learning and using Wicket."

—Erik van Oosten
Lead Programmer and
Project Manager, JTeam



MANNING

\$44.99 / Can \$44.99 [INCLUDING EBOOK]

