11
Writing Custom Components

In this chapter we will cover:

- Anatomy of PrimeFaces components
- Writing a JavaScript widget
- Writing a Cascading Style Sheets file
- Writing a model class
- Writing component classes
- Writing a renderer class
- Adding AJAX support for p:ajax
- Binding all the parts together
- Running it

Introduction

Despite a comprehensive set of components, framework users sometimes have special requirements for existing components regarding their look and feel and functionality or need new, custom components. Writing of custom components requires a strong understanding and expert knowledge of the full web stack: JSF, JavaScript with jQuery, and CSS. It may not be a problem to acquire these skills. The hurdle is rather in getting first experience of how to combine all these technologies when writing custom components. The purpose of this chapter is exclusively to facilitate the start of component development on top of the PrimeFaces infrastructure.

In this chapter, we will go through the complete process of building reusable PrimeFaces components based on the core functionality. First of all, we will learn a component's structure and constituent parts. After that we will develop a custom `Layout` component as an alternative implementation to the PrimeFaces' one. For the sake of convenience, we will skip some advanced features of the `Layout` component, such as state management and updatable nested layouts. This will allow us to simplify matters and concentrate on the important parts only. At the end of this chapter, we will be equipped with necessary knowledge to be able to create different custom components for the next few JSF/PrimeFaces web applications.

Anatomy of PrimeFaces components

PrimeFaces follows the goal of keeping things clean and being lightweight. Most of the PrimeFaces components have the same structure: HTML markup and a script block.

In this recipe, we will explore a component's anatomy in terms of generated page elements related to a typically component. The `Calendar` component, serves as a learning example. Furthermore, we will give an overview of the basic modules we need to develop. This overview makes a good start for the next recipes.

How to do it...

Place a `p:calendar` tag on a page as shown in the next snippet:

```
<h:form id="form">
...
    <p:calendar id="popupCal" value="#{calendarBean.date}" />
...
</h:form>
```

The rendered HTML markup and the script block related to the `p:calendar` tag will be as follows:

```
<span id="form:popupCal">
    <input id="form:popupCal_input" name="form:popupCal_input"
    type="text"
    class="ui-inputfield ui-widget ui-state-default ui-corner-all" />
</span>

<script id="form:popupCal_s" type="text/javascript">
    $(function(){
        PrimeFaces.cw('Calendar','widget_form_popupCal',
```

```
            {id:'form:popupCal',popup:true,locale:'en_US',dateFormat:
            'm/d/y'});
        });
    </script>
```

This generated output can be viewed in every browser. A right-click on a page normally shows a context menu with **View Page Source** or a similar menu item.

How it works...

The rendered HTML markup contains a container element with the component client ID and one or more child elements. The container element allows us to update the whole component at once by AJAX. The container element for `p:calendar` is a simple `span` element that has a styled input field inside a pop-up calendar without buttons.

The `PrimeFaces.cw()` JavaScript method creates a special JavaScript object, called widget. All the PrimeFaces widgets are namespaced with the `PrimeFaces.widget.*` namespace. The widget for the `Calendar` component is called, for example, `PrimeFaces.widget.Calendar`. Every widget object is instantiated with the `new` operator and assigned to a global variable in the window scope. The name of the variable is passed into the method `PrimeFaces.cw()`. For the previously mentioned calendar example, it was generated automatically as `widget_form_popupCal`. It is also possible to set the widget name explicitly by the tag attribute `widgetVar`.

If the widget variable is auto-generated and not known directly, it can be accessed via a utility function `#{p:widgetVar('componentId')}`. If you have, say, a `p:dialog` tag with the ID `dialog1` and would like to show the dialog, the call `#{p:widgetVar('dialog1')}.show()` does the job.

The widget has various responsibilities such as progressive enhancement of the markup and communication with the server side via AJAX. For example, it adds styles on the fly to the DOM instead of rendering them with the markup (this helps in reducing page size). Another responsibility of the widget is the binding of callbacks for various interaction events on the component. PrimeFaces follows the unobtrusive JavaScript pattern and does not embed DOM events within the event attributes in markup. It happens mostly in the widget's script.

As we can also see, the script block has an ID in order to be referenced by JavaScript. Every script block gets removed from the DOM after widget creation. This removal is necessary to avoid JavaScript collisions on the client side. In the third recipe of this chapter, *Writing a JavaScript widget*, we will see how it works and understand why it is necessary.

Writing a JavaScript widget is only one step or module in the entire development process. The next listing shows a short overview of all the modules that we need to develop custom components.

Module	Description
JavaScript widget	The client-side part of the component that is responsible for progressive enhancement of the markup, AJAX communication with the server-side, binding of event handlers for user interactions, and more.
Cascading style sheets	CSS defines the skeleton of the component, which includes margin, padding, dimensions, positioning (structural style classes), and styling/theming such as text colors, border colors, and background images (skinning style classes). Style classes are described in detail the first recipe, *Understanding structural and skinning CSS*, of *Chapter 2*, *Theming Concept*.
Component class	The server-side part of the component that represents the component in the component tree. The component class also acts as receiver for events that are triggered by the widget and propagated to the server side.
Renderer class	The server-side part of the component that is mostly responsible for markup and script rendering. It writes the component's bits and bytes into the current response. The code snippet for the `p:calendar` component given previously was generated by the calendar's `renderer` class.
Tag handler	The tag handler is a special Java class whose instance is invoked by a view (facelets) handler. Tag handlers run during the view build/restore time when JSF constructs the component tree. It is, say, possible to create and add some subcomponents or JSF listeners dynamically during view build/restore time. Writing of tag handlers is beyond the scope of this chapter/book.
Model classes	Sometimes a component is created by means of model classes. An example is `p:tree`, which is bound to a `TreeNode` instance representing a root node. The root node itself has other `TreeNode` instances as child nodes and so on. Also, the `Layout` component that we will implement in this chapter has a model class—`LayoutOptions`.

Module	Description
XML configuration	All the components' parts have to be bound together. This occurs in the `faces-config.xml` file where the `component` class, `component` family, `renderer` class, and `renderer` type should be registered, as well as in the `*.taglib.xml` file where the component's tag name, tag handler class, and references to component/renderer type are registered. Furthermore, the `*.taglib.xml` object describes the exposed attributes component's (name, type, required flag, and so on) along with the description. There is also an alternative way to `faces-config.xml` with the `@FacesComponent` / `@FacesRenderer` annotations.

There's more...

To avoid UI flicker during progressive enhancements, PrimeFaces tends to hide the visibility of the markup and display it once the script is finished. This is useful because the widget's script is executed when the DOM is fully loaded—the script's logic is surrounded with `$(function(){...})`. Please refer to the jQuery documentation to learn more about the DOM-ready event (`http://api.jquery.com/ready`).

Writing a JavaScript widget

Writing a JavaScript widget is probably the most complicated part during component development. A widget is the heart of every rich UI component and requires accuracy and the best effort when developing.

In this recipe, we will introduce you to the widget structure and implement the base code for the `Layout` widget. Parts requiring AJAX interaction will be skipped. These topics are covered in the recipe *Adding AJAX support for p:ajax* in this chapter.

How to do it...

Every widget in PrimeFaces should be namespaced with `PrimeFaces.widget.*` and extended from `PrimeFaces.widget.BaseWidget`. The `PrimeFaces.widget.BaseWidget` class offers the core widget functionality such as removing the script block that belongs to the component. We will write our widget `PrimeFaces.widget.Layout` and extend it from the mentioned `BaseWidget` component using the jQuery `extend()` method.

```
PrimeFaces.widget.Layout = PrimeFaces.widget.BaseWidget.extend({
  /**
   * Initializes the widget from configuration object.
```

```
     */
    init:function (cfg) {
      this._super(cfg);
      this.cfg = cfg;
      this.id = cfg.id;
      this.jqTarget = $(cfg.forTarget);

      var _self = this;

        if (this.jqTarget.is(':visible')) {
          this.createLayout();
        } else {
          var hiddenParent = this.jqTarget.parents
          ('.ui-hidden-container:first');
          var hiddenParentWidget =  hiddenParent.data('widget');

        if (hiddenParentWidget) {
          hiddenParentWidget.addOnshowHandler
            (function () {
            return _self.createLayout();
            });
         }
      }
    },

  createLayout:function () {
    // create layout
    this.layout = this.jqTarget.layout(this.cfg.options);

     // bind "open", "close" and "resize" events
     this.bindEvents(this.jqTarget);
  },

  bindEvents:function(parent) {
          ...
  },

  toggle:function (pane) {
    this.jqTarget.find(".ui-layout-pane").
    each(function() {
      var combinedPosition =
      $(this).data('combinedposition');
```

```
        if (combinedPosition &&
          combinedPosition === pane) {
          $(this).trigger("layoutpanetoggle");
          return false;
          }
        });
      },

        ...
    });
```

As you can see, there is a mandatory `init()` function that expects one JavaScript object—the widget configuration. Such a configuration object is created in JSON notation by the `renderer` class while it is writing the component's markup and JavaScript code into an HTTP response. An example of the configuration object for the the Layout component is shown as follows:

```
{
  "id": "fpl",
  "forTarget": "body",
  "options": {
      "panes": {
          "resizeWhileDragging": true,
          "slidable": false,
          "closable": true,
          "resizable": true,
          "resizeWithWindow": false,
          "spacing": 6
    },
    "north": {
          "closable": false,
          "resizable": false,
          "size": 60
    },
    "south": {
        "closable": false,
        "resizable": false,
        "size": 40
    },
    "west": {
        "minSize": 180,
        "maxSize": 500,
        "size": 210
    },
```

```
        "east": {
            "minSize": 180,
            "maxSize": 650,
            "size": 448,
            "childOptions": {
                "south": {
                    "minSize": 60,
                    "size": "70%"
                },
                "center": {
                    "minHeight": 60
                }
            }
        },
        "center": {
            "minHeight": 60,
            "resizeWhileDragging": false,
            "closable": false,
            "minWidth": 200,
            "resizable": false
        }
      }
    }
```

How it works...

The `init()` function is called during widget initialization. There, we can get the component's ID and the jQuery object for the target HTML element the layout is applied to. This is the body element in the case of the full-page layout and a `div` element around the layout markup in the case of the element layout. We will study these two cases later, in the recipe *Writing a renderer class*.

If the target HTML element (to which the layout is applied) is visible, we can create the layout with the given options directly. This task is done with `this.jqTarget.layout(this.cfg.options)` in the widget's method `createLayout()`. But there is also a special case when the target HTML element is hidden because a layout can be placed within a hidden element (for example, the `TabView` component with the content of hidden tabs). To handle this scenario, we need to take care of a proper layout initialization in hidden containers. PrimeFaces widgets normally register callbacks on widgets of components having hidden elements. In the `init()` method, we found out the first hidden container in the parent hierarchy, got the corresponding widget `hiddenParentWidget`, and registered our callback function with `hiddenParentWidget.addOnshowHandler()`. The callback function again calls the `createLayout()` function when the hidden content gets displayed. This happens, for example, when the user clicks for the first time on a tab in `p:tabView`, which contains the `Layout` component.

The method `bindEvents()` is not relevant here; we will show its implementation in the recipe *Adding AJAX support for p:ajax* in this chapter. The widget provides some other useful client-side methods to toggle, open, close, and resize any specified layout pane. The previous code snippet only shows an example of an implementation for toggling. It is looking for any given pane and fires the event `layoutpanetoggle` on the pane by calling `trigger("layoutpanetoggle")`. All the valid events are specified in the jQuery Layout plugin. Every client-side method expects the pane position as parameter. Nested panes are separated by the underscore character, for example, `toggle("center_north")`.

We will render pane positions, such as `"center_north"`, in the `renderer` class and store these identifiers in the pane's `(div` element), `data-combinedposition` attribute. The access on the client side is then easily possible via `.data('combinedposition')`. This trick allows us to find any pane object by the given pane position.

There's more...

In addition to `init()`, many widgets in PrimeFaces implement the method `refresh()`. This method is called when the component gets updated via AJAX. Partial updates do not recreate widgets; they refresh their internal states. In this way, every PrimeFaces widget is stateful. It depends on the `refresh()` implementation for which parts of the component's state are reinitialized and re-used respectively.

Writing a Cascading Style Sheets file

Cascading Style Sheets (**CSS**) belong to the component's visual part. CSS properties are usually structural style classes. The styling is adopted by PrimeFaces themes so that component developers do not need to worry about colors, images, and other styling settings.

In this recipe, we will write a CSS file for our `Layout` component.

How to do it...

We will write a CSS file, `layout.css`, according to the documentation of the jQuery Layout plugin (`http://layout.jquery-dev.net/documentation.cfm`). The CSS file will be placed in a web project below the `resources` folder.

```
- war
    - resources
        - js
            - chapter11
                - layout.css
```

It contains the following code:

```
.ui-layout-pane {
    padding: 0;
    overflow: hidden;
}

.ui-layout-north,
.ui-layout-south,
.ui-layout-west,
.ui-layout-east,
.ui-layout-center {
    display: none;
}

.ui-layout-resizer-west-dragging,
.ui-layout-resizer-west-open-hover {
    background: url("#{resource['images:chapter11/resizable-
    w.gif']}") repeat-y center;
}

.ui-layout-resizer-west-closed-hover {
    background: url("#{resource['images:chapter11/resizable-
    e.gif']}") repeat-y center;
}

.ui-layout-toggler-west-open {
    background: url("#{resource['images:chapter11/toggle-
    lt.gif']}") no-repeat scroll left center transparent;
}

.ui-layout-toggler-west-closed {
    background: url("#{resource['images:chapter11/toggle-
    rt.gif']}") no-repeat scroll right center transparent;
}

.ui-layout-resizer-east-dragging,
.ui-layout-resizer-east-open-hover {
    background: url("#{resource['images:chapter11/resizable-
    e.gif']}") repeat-y center;
}

.ui-layout-resizer-east-closed-hover {
    background: url("#{resource['images:chapter11/resizable-
    w.gif']}") repeat-y center;
```

```
}

.ui-layout-toggler-east-open {
    background: url("#{resource['images:chapter11/toggle-
    rt.gif']}") no-repeat scroll right center transparent;
}

.ui-layout-toggler-east-closed {
    background: url("#{resource['images:chapter11/toggle-
    lt.gif']}") no-repeat scroll left center transparent;
}

.ui-layout-resizer-north-dragging,
.ui-layout-resizer-north-open-hover {
    background: url("#{resource['images:chapter11/resizable-
    n.gif']}") repeat-x center;
}

.ui-layout-resizer-north-closed-hover {
    background: url("#{resource['images:chapter11/resizable-
    s.gif']}") repeat-x center;
}

.ui-layout-toggler-north-open {
    background: url("#{resource['images:chapter11/toggle-
    up.gif']}") no-repeat scroll top center transparent;
}

.ui-layout-toggler-north-closed {
    background: url("#{resource['images:chapter11/toggle-
    dn.gif']}") no-repeat scroll bottom center transparent;
}

.ui-layout-resizer-south-dragging,
.ui-layout-resizer-south-hover {
    background: url("#{resource['images:chapter11/resizable-
    s.gif']}") repeat-x center;
}

.ui-layout-resizer-south-closed-hover {
    background: url("#{resource['images:chapter11/resizable-
    n.gif']}") repeat-x center;
}
```

```
.ui-layout-toggler-south-open {
    background: url("#{resource['images:chapter11/toggle-
    dn.gif']}") no-repeat scroll bottom center transparent;
}

.ui-layout-toggler-south-closed {
    background: url("#{resource['images:chapter11/toggle-
    up.gif']}") no-repeat scroll top center transparent;
}

.ui-layout-resizer-dragging-limit {
    border: 1px solid #E27D7D;
}

.ui-layout-pane-withsubpanes {
    border: none;
}

.ui-layout-pane-header {
    padding: 4px 1em;
    border-width: 0 0 1px;
}

.ui-layout-pane-content {
    padding: 6px;
    overflow: auto;
}
```

How it works...

The style classes are specified by the jQuery Layout plugin. Every layout pane gets a class `ui-layout-<pane>`, where `<pane>` is the pane position: `north`, `south`, `west`, `east`, or `center`. We hide all the layout panes on initial page load with the setting `display: none`. As already stated in the recipe *Anatomy of PrimeFaces components* in this chapter, we sometimes need to hide the visibility of the markup to avoid UI flicker during progressive enhancement, especially in Internet Explorer. The JavaScript of the Layout plugin turns on the visibility of all the panes automatically when they have been fully created and are ready to be displayed.

The jQuery layout plugin specifies style classes for the resizer and toggler images as well. These are for instance `ui-layout-resizer-west-dragging` and `ui-layout-toggler-west-open` for any `west` pane. We would like to provide default images so that users can use them by default or overwrite them with custom CSS settings. Default images are placed in the same folder as `layout.css` and referenced in the JSF manner with `background: url("#{resource['images:chapter11/<image-name>']}")`.

The last three style classes are not specified by the jQuery Layout plugin. We will apply them in the `renderer` class to the layout panes without subpanes or specific elements within panes. The specific elements are the pane header and pane content. We intend to define a pane header by using `facet="header"`. All the content in the pane except the header is considered as pane content and gets the class `ui-layout-pane-content`.

See also

The recipe *Writing a renderer class* in this chapter explains the plain markup structure more precisely. The reader will see what layout panes, the pane header, and the pane content look like in HTML.

Writing a model class

As we have seen in the recipe *Writing a JavaScript widget* in this chapter, the widget's `init()` method expects a configuration object with layout options. Options are passed when creating a layout. That means the `layout` tag should have an attribute `options` that is bound to a bean such as `options="#{layoutBean.layoutOptions}"`. The JSON structure of layout options is specified by the jQuery Layout plugin (`http://layout.jquery-dev.net`). There are seven main keys with values representing options. The main keys are: `"panes"` (contains options for all panes), `"north"` (contains options for the north pane), `"center"` (contains options for the center pane), `"south"` (contains options for the south pane), `"west"` (contains options for the west pane), `"east"` (contains options for the east pane), and `"childOptions"` (contains options for the nested sub-layout). The available options are described on the plugin's documentation page. An example of such a structure is shown in the following snippet:

```
{
    "panes": {
        "resizeWhileDragging": true,
        "slidable": false,
        "spacing": 6
    },
    "north": {
        "closable": false,
        "resizable": false,
        "size": 60
    },

    ...

    "east": {
        "minSize": 180,
        "maxSize": 650,
```

```
            "size": 448,
            "childOptions": {
                "south": {
                    "minSize": 60,
                    "size": "70%"
                },
               "center": {
                   "minHeight": 60
               }
            }
        }
    }
```

Attentive readers might probably have recognized that layout options can be deeply nested due to nested layouts. The task is now to write a model class that will help to create layout options in Java. We also need to serialize created options to JSON so that the `renderer` class can take this JSON representation and write it into the widget's script.

In this recipe, we will develop a Java model class `LayoutOptions` and provide functionality for its JSON serialization. `LayoutOptions` should allow creating nested options to satisfy our requirements for building nested layouts.

How to do it...

The structure of the class `LayoutOptions` is driven by the main keys that represent `LayoutOptions` on individual parts. This allows us to build nested layouts like a tree with nested tree nodes.

```
public class LayoutOptions implements Serializable {

  // direct options
  private Map<String, Object> options = new HashMap<String, Object>();

  // options for all panes
  private LayoutOptions panes;

  // options for every specific pane (depends on position)
  private LayoutOptions north;
  private LayoutOptions south;
  private LayoutOptions west;
  private LayoutOptions east;
  private LayoutOptions center;
```

```
// options for child layout
private LayoutOptions child;

public LayoutOptions() {
}

public Map<String, Object> getOptions() {
  return options;
}

public void setOptions(Map<String, Object> options) {
  this.options = options;
}

public void addOption(String key, Object value) {
  options.put(key, value);
}

public void setPanesOptions(LayoutOptions layoutOptions) {
  panes = layoutOptions;
}

public LayoutOptions getPanesOptions() {
  return panes;
}

public void setNorthOptions(LayoutOptions layoutOptions) {
  north = layoutOptions;
}

public LayoutOptions getNorthOptions() {
  return north;
}

// getters / setters for other pane options
...

public void setChildOptions(LayoutOptions layoutOptions) {
  child = layoutOptions;
}
```

```
    public LayoutOptions getChildOptions() {
      return child;
    }

    public String render() {
      return GsonLayoutOptions.getGson().toJson(this);
    }
  }
```

How it works...

The starting point, when building layout options, consists of filling the direct options. Direct options can be added by using `addOption(String key, Object value)`. It is advisable to create direct options for all panes first and then weave them together by setters, for example, using `setNorthOptions(LayoutOptions layoutOptions)`. A fully working example will be demonstrated in the two last recipes of this chapter, Binding all parts together and Running it.

The method `render()` will be invoked later in the `renderer` class. This method avails itself of the singleton instance `GsonLayoutOptions` based on Gson. **Gson** is a Java library that converts Java Objects into their JSON representations (`http://code.google.com/p/google-gson`).

```
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class GsonLayoutOptions {

  private static final GsonLayoutOptions INSTANCE = new
    GsonLayoutOptions();
  private Gson gson;

  private GsonLayoutOptions() {
    GsonBuilder gsonBilder = new GsonBuilder();
    gsonBilder.registerTypeAdapter(LayoutOptions.class, new
      LayoutOptionsSerializer());
    gson = gsonBilder.create();
  }

  public static Gson getGson() {
    return INSTANCE.gson;
  }
}
```

The class `LayoutOptionsSerializer` serves as a custom serializer in order to align the output format of layout options with the specified one. The implementation of this class can be found on the book's homepage in GitHub (`https://github.com/ova2/primefaces-cookbook`).

Writing component classes

The `component` class represents a component in the component `tree`. It defines properties that are reflected in the component's tag as attributes. Attributes participate in the state saving. Furthermore, components are responsible for specifying resource dependencies, receiving AJAX events, and some other tasks.

In this recipe, we will develop two component classes: `Layout` and `LayoutPane`. The `Layout` class will not contain any logic to handle AJAX events. This is the task of the *Adding AJAX support for p:ajax* recipe of this chapter, where we will extend the current implementation. The `LayoutPane` component is intended to be used as a child component of the `Layout` class. It represents a layout pane with one of the five positions—north, south, center, west, or east.

How to do it...

The `Layout` class specifies resource dependencies by JSF annotations `@ResourceDependencies` for multiple resources and `@ResourceDependency` for single resource. The component should extend `javax.faces.component.UIComponentBase` and implement the interfaces `org.primefaces.component.api.Widget` and `javax.faces.component.behavior.ClientBehaviorHolder`. The component should set a renderer type in the constructor and provide a component family in the return value of `getFamily()`. Other constants are only interesting for the renderer and will be explained in the next recipe. Attributes that participate in the state saving are listed in `enum PropertyKeys`. The provided setter/getter will also be explained later.

```
@ResourceDependencies({
  @ResourceDependency(library = "primefaces",
                      name = "jquery/jquery.js"),
  @ResourceDependency(library = "primefaces",
                      name = "primefaces.js"),
  @ResourceDependency(library = "css",
                      name = "chapter11/layout.css"),
  @ResourceDependency(library = "js",
                      name = "chapter11/jquery.layout.js"),
  @ResourceDependency(library = "js",
                      name = "chapter11/layout.js")
})
public class Layout extends UIComponentBase implements Widget,
  ClientBehaviorHolder {
```

```
    public static final String COMPONENT_FAMILY =
      "org.primefaces.cookbook.component";
    private static final String DEFAULT_RENDERER =
      "org.primefaces.cookbook.component.LayoutRenderer";

    public static final String POSITION_SEPARATOR = "_";
    public static final String STYLE_CLASS_PANE =
   "ui-widget-content ui-corner-all";
    public static final String STYLE_CLASS_PANE_WITH_SUBPANES =
      "ui-corner-all ui-layout-pane-withsubpanes";
    public static final String STYLE_CLASS_PANE_HEADER =
      "ui-widget-header ui-corner-top ui-layout-pane-header";
    public static final String STYLE_CLASS_PANE_CONTENT =
      "ui-layout-pane-content";

    protected enum PropertyKeys {
      widgetVar,
      fullPage,
      options,
      style,
      styleClass
    }

    public Layout() {
      setRendererType(DEFAULT_RENDERER);
    }

    @Override
    public String getFamily() {
      return COMPONENT_FAMILY;
    }

    public String getWidgetVar() {
      return (String) getStateHelper().eval(PropertyKeys.widgetVar,
      null);
    }

    public void setWidgetVar(String widgetVar) {
      getStateHelper().put(PropertyKeys.widgetVar, widgetVar);
    }

    public boolean isFullPage() {
      return (Boolean) getStateHelper().eval(PropertyKeys.fullPage,
      true);
    }
```

```
    public void setFullPage(boolean fullPage) {
      getStateHelper().put(PropertyKeys.fullPage, fullPage);
    }

    public Object getOptions() {
      return getStateHelper().eval(PropertyKeys.options, null);
    }

    public void setOptions(Object options) {
      getStateHelper().put(PropertyKeys.options, options);
    }

    public String getStyle() {
      return (String) getStateHelper().eval(PropertyKeys.style,
      null);
    }

    public void setStyle(String style) {
      getStateHelper().put(PropertyKeys.style, style);
    }

    public String getStyleClass() {
      return (String) getStateHelper().eval(PropertyKeys.styleClass,
      null);
    }

    public void setStyleClass(String styleClass) {
      getStateHelper().put(PropertyKeys.styleClass, styleClass);
    }

    public String resolveWidgetVar() {
      FacesContext context = FacesContext.getCurrentInstance();
      String userWidgetVar = (String)
      getAttributes().get(PropertyKeys.widgetVar.toString());

      if (userWidgetVar != null) {
        return userWidgetVar;
      }

      return "widget_" + getClientId(context).replaceAll("-|" +
        UINamingContainer.getSeparatorChar(context), "_");
    }
  }
```

The component class `LayoutPane` does not specify any resources and does not implement the interfaces `Widget` and `ClientBehaviorHolder` because no widget exists for this component. It is used as a child component under `Layout`.

```
public class LayoutPane extends UIComponentBase {

  public static final String COMPONENT_FAMILY =
    "org.primefaces.cookbook.component";
  private static final String DEFAULT_RENDERER =
    "org.primefaces.cookbook.component.LayoutPaneRenderer";

  protected enum PropertyKeys {
    position,
    combinedPosition
  }

  public LayoutPane() {
    setRendererType(DEFAULT_RENDERER);
  }

  @Override
  public String getFamily() {
    return COMPONENT_FAMILY;
  }

  // position "north" | "south" | "west" | "east" | "center"
  public String getPosition() {
    return (String)
    getStateHelper().eval(PropertyKeys.position, "center");
  }

  public void setPosition(String position) {
    getStateHelper().put(PropertyKeys.position, position);
  }

  public String getCombinedPosition() {
    return (String) getStateHelper().eval(PropertyKeys.
    combinedPosition, "center");
  }

  public void setCombinedPosition(String combinedPosition) {
    getStateHelper().put(PropertyKeys.combinedPosition,
    combinedPosition);
  }
}
```

How it works...

`@ResourceDependencies` is a container annotation to specify multiple `@ResourceDependency` annotations. The latter makes sure that the declared resources, JavaScript and CSS files, are loaded along with the component. There are three attributes to be considered in `@ResourceDependency`, discussed as follows:

- `name`: This is the filename of the resource.
- `library`: This is the path name where the resource can be found; it corresponds to the library name in JSF.
- `target`: This is the target element where the resource is expected to be referenced. This attribute is optional. When not provide, the value `head` is assumed. That means the resource will be included in the `head` section. Another option for JavaScript files is `body`. The referenced resource will then be added to the `body` tag at the page end.

Resources with names `jquery/jquery.js` and `primefaces.js` from the library `primefaces` always need to be included due to the required core functionality.
The resource with name `jquery.layout.js` comes from the jQuery Layout plugin.
The remaining resources, `layout.css` and `layout.js`, are written by the component developer—they were described in the previous recipes, *Writing a JavaScript widget* and *Writing a Cascading Style Sheets file*, of this chapter.

JSF manages multiple resource declarations properly, so a declared resource will not be added to the page if it was already declared by another component on this page.

Extending from the `javax.faces.component.UIComponentBase` class is useful if we do not want to worry about base implementations of some methods such as `getClientId()` and `getChildren()`. Furthermore, a component needs to implement interfaces `org.primefaces.component.api.Widget` if it provides a JavaScript widget and `javax.faces.component.behavior.ClientBehaviorHolder` if it can be ajaxified and should work with attached `p:ajax`. The widget interface declares the method `resolveWidgetVar()`, whose implementation is always the same and can be taken from the `Layout` class when developing custom components.

The last subject to be discussed here is state management with the aim to preserve the component state between requests. That occurs in setter/getter via an instance of `javax.faces.component.StateHelper`. The call `getStateHelper().put(Serializable key, Object value)` stores the specified key/value pair. The call `getStateHelper().eval(Serializable key, Object defaultValue)` finds a value associated with the specified key. The latter call allows you to evaluate value expressions and to return the value from the backing bean.

Writing a renderer class

The `renderer` class generates the component's markup and writes an associated JavaScript block into the current response. Before starting to think about renderer class, the recommended practice consists of designing a plain HTML structure. This step may not be necessary if a component has a simple markup or does not have a renderer at all. But it is recommended to start with a proper sketch in case of a more complex markup.

In this recipe, we will sketch a plain HTML markup for the upcoming component and develop two renderer classes, `LayoutRenderer` and `LayoutPaneRenderer`. Both `Layout` and `LayoutPane` have associated renderers. Code lines for supporting client (AJAX) behaviors will be added in the next recipe.

How to do it...

The markup structure is predetermined by the jQuery Layout plugin. We only need to add specific CSS classes to the style pane header and pane content. These classes were already mentioned in the recipe *Writing a Cascading Style Sheets file* in this chapter. The attribute `data-combinedposition` should be added to the pane's `div` element as well. The value of this attribute is used in the widget as described in the recipe *Writing a JavaScript widget* in this chapter. A typical markup output is shown below:

```
<div class="ui-layout-center ui-widget-content ui-corner-all
     ui-layout-pane-content"
     data-combinedposition="center">
     Center content
</div>

<div class="ui-layout-west ui-widget-content ui-corner-all"
     data-combinedposition="west">
    <div class="ui-widget-header ui-corner-top ui-layout-pane-header">
        West header
    </div>
    <div class="ui-layout-content ui-layout-pane-content"
        style="border:none">
        West content
    </div>
</div>

<div class="ui-layout-east ui-corner-all
     ui-layout-pane-withsubpanes"
     data-combinedposition="east">
    <div class="ui-layout-center ui-widget-content ui-corner-
    all"
```

```
            data-combinedposition="east_center">
            <div class="ui-widget-header ui-corner-top
                ui-layout-pane-header">
                East-Center header
            </div>
            <div class="ui-layout-content ui-layout-pane-content"
                style="border:none">
                East-Center content
            </div>
        </div>
        <div class="ui-layout-south ui-widget-content
             ui-corner-all ui-layout-pane-content"
             data-combinedposition="east_south">
            East-South content
        </div>
    </div>

    <div class="ui-layout-south ui-widget-content
         ui-corner-all ui-layout-pane-content"
         data-combinedposition="south">
        South content
    </div>
```

The `LayoutRenderer` class should wrap the rendered markup in a `div` element if the layout is an element layout. The element layout is not applied to the entire page and needs a surrounding `div` element to be able to be partially updatable. The `layout` class overrides the default implementation for some methods (they are annotated with `@Override`) and implements a new method `encodeScript()`, which is responsible for the rendering of the script block.

```
public class LayoutRenderer extends CoreRenderer {

  @Override
  public void encodeBegin(FacesContext fc, UIComponent
  component) throws IOException {
    ResponseWriter writer = fc.getResponseWriter();
    Layout layout = (Layout) component;

    encodeScript(fc, layout);

    if (!layout.isFullPage()) {
      writer.startElement("div", layout);
      writer.writeAttribute("id", layout.getClientId(fc),
      "id");
```

```
        if (layout.getStyle() != null) {
          writer.writeAttribute("style", layout.getStyle(),
          "style");
        }

        if (layout.getStyleClass() != null) {
          writer.writeAttribute("class", layout.getStyleClass(),
          "styleClass");
        }
      }
    }

    @Override
    public void encodeEnd(FacesContext fc, UIComponent component) throws
      IOException {
      ResponseWriter writer = fc.getResponseWriter();
      Layout layout = (Layout) component;

      if (!layout.isFullPage()) {
        writer.endElement("div");
      }
    }

    @Override
    public boolean getRendersChildren() {
      return false;
    }

    protected void encodeScript(FacesContext fc, Layout layout) throws
      IOException {
      ResponseWriter writer = fc.getResponseWriter();
      String clientId = layout.getClientId();

      startScript(writer, clientId);
      writer.write("$(function() {");
      writer.write("PrimeFaces.cw('Layout', '" +
      layout.resolveWidgetVar() + "',{");
      writer.write("id:'" + clientId + "'");

      if (layout.isFullPage()) {
        writer.write(",forTarget:'body'");
      } else {
```

```
        writer.write(",forTarget:'" + ComponentUtils.
        escapeJQueryId(clientId) + "'");
      }

      LayoutOptions layoutOptions = (LayoutOptions) layout.getOptions();
      if (layoutOptions != null) {
        writer.write(",options:" + layoutOptions.render());
      } else {
        writer.write(",options:{}");
      }

      writer.write("},true);});");
      endScript(writer);
    }
  }
```

The `LayoutPaneRenderer` class is a little more complex than `LayoutRenderer`. But, unlike `LayoutRenderer`, it does not need to render any script block. Remember that `LayoutPane` does not have an associated widget.

```
public class LayoutPaneRenderer extends CoreRenderer {

  @Override
  public void encodeBegin(FacesContext fc, UIComponent
  component) throws IOException {
    ResponseWriter writer = fc.getResponseWriter();
    LayoutPane layoutPane = (LayoutPane) component;

    String position = layoutPane.getPosition();
    String combinedPosition = position;

    UIComponent parent = layoutPane.getParent();
    while (parent instanceof LayoutPane) {
      combinedPosition = ((LayoutPane) parent).getPosition() +
        Layout.POSITION_SEPARATOR + combinedPosition;
      parent = parent.getParent();
    }

    // save combined position
    layoutPane.setCombinedPosition(combinedPosition);
```

```
boolean hasSubPanes = false;
for (UIComponent subChild : layoutPane.getChildren()) {
  // check first level
  if (hasSubPanes) {
    break;
  }

  if (subChild instanceof LayoutPane) {
    if (!subChild.isRendered()) {
      continue;
    }

    hasSubPanes = true;
  } else {
    for (UIComponent subSubChild : subChild.getChildren()) {
      // check second level
      if (subSubChild instanceof LayoutPane) {
        if (!subSubChild.isRendered()) {
          continue;
        }

        hasSubPanes = true;

        break;
      }
    }
  }
}

UIComponent header = layoutPane.getFacet("header");

writer.startElement("div", null);
writer.writeAttribute("id", layoutPane.getClientId(fc),
"id");
if (hasSubPanes) {
  writer.writeAttribute("class", "ui-layout-" + position +
  " " + Layout.STYLE_CLASS_PANE_WITH_SUBPANES, null);
} else {
  if (header != null) {
    writer.writeAttribute("class", "ui-layout-" + position
    + " " + Layout.STYLE_CLASS_PANE, null);
  } else {
```

```
        writer.writeAttribute("class",
              "ui-layout-" + position + " " +
              Layout.STYLE_CLASS_PANE + " "+
              Layout.STYLE_CLASS_PANE_CONTENT, null);
      }
    }

    writer.writeAttribute("data-combinedposition",
    combinedPosition, null);

    // encode header
    if (header != null) {
      writer.startElement("div", null);
      writer.writeAttribute("class",
      Layout.STYLE_CLASS_PANE_HEADER, null);

      header.encodeAll(fc);

      writer.endElement("div");
    }

    // encode content
    if (header != null) {
      writer.startElement("div", null);
      writer.writeAttribute("class", "ui-layout-content " +
        Layout.STYLE_CLASS_PANE_CONTENT, null);
      writer.writeAttribute("style", "border:none", null);
      renderChildren(fc, layoutPane);

      writer.endElement("div");
    } else {
      renderChildren(fc, layoutPane);
    }
  }

  @Override
  public void encodeEnd(FacesContext fc, UIComponent
  component) throws IOException {
    ResponseWriter writer = fc.getResponseWriter();

    writer.endElement("div");
  }
```

```
    @Override
    public boolean getRendersChildren() {
      return true;
    }

    @Override
    public void encodeChildren(FacesContext fc, UIComponent
    component) throws IOException {
      // nothing to do
    }
  }
```

How it works...

It is a good practice to extend your own renderers from the PrimeFaces core renderer `org.primefaces.renderkit.CoreRenderer`. The core renderer offers handy utility methods such as `startScript(ResponseWriter writer, String clientId)` and `endScript(ResponseWriter writer)` to render start/end lines for a JavaScript block, and much more. The script block should be executed when the DOM is fully loaded. Therefore, the script's logic is surrounded with `$(function(){...})` and follows the structure discussed in the first recipe of this chapter, *Anatomy of PrimeFaces components*. The component's client ID and the target element, the layout is applied to, are passed into the widget's configuration object. Layout options are acquired by the call `layoutOptions.render()`, and passed into this object as well. The whole markup is generated in the overridden methods `encodeBegin()` and `encodeEnd()`. All the methods that were overridden are as follows:

- `void encodeBegin(FacesContext context, UIComponent component)`: It renders the beginning of this component to the output stream.
- `void encodeEnd(FacesContext context, UIComponent component)`: It renders the ending of this component to the output stream.
- `boolean getRendersChildren()`: It returns a flag indicating whether this renderer is responsible for rendering the children of the component that is to be rendered.
- `void encodeChildren(FacesContext context, UIComponent component)`: It renders the child components of this component. This method will only be called if the `getRendersChildren()` method returns the value `true`.

Adding AJAX support for p:ajax

If we need to execute a JavaScript callback or a listener method on a backing bean for certain events, the component should support `p:ajax` backed by the `org.primefaces.component.behavior.ajax.AjaxBehavior` class. To support `AjaxBehavior` more work in widget, component and renderer classes is required.

In this recipe, we will learn how to add the AJAX support and make our `Layout` component ajaxified. We are interested in three events on the layout panes—close, open, and resize. The close event fires after a layout pane gets closed; the open event fires after a layout pane gets opened; and the resize event fires when the pane is resizing.

How to do it...

The next listings show additional code snippets that are added to handle the attached `p:ajax` tag for the three previously mentioned events. The first class to be extended is `Layout`. It will get a collection of supported events, overridden implementations for `processDecodes(FacesContext fc)`, `processValidators(FacesContext fc)`, `processUpdates(FacesContext fc)`, `queueEvent(FacesEvent event)`, and some utility methods.

```
public class Layout extends UIComponentBase implements Widget,
ClientBehaviorHolder {

  private static final Logger LOG =
  Logger.getLogger(Layout.class.getName());

  ...

  private static final Collection<String> EVENT_NAMES =
    Collections.unmodifiableCollection(Arrays.asList("open",
    "close", "resize"));

  @Override
  public Collection<String> getEventNames() {
    return EVENT_NAMES;
  }

  @Override
  public void processDecodes(FacesContext fc) {
    if (isSelfRequest(fc)) {
      this.decode(fc);
```

```
    } else {
      super.processDecodes(fc);
    }
  }

  @Override
  public void processValidators(FacesContext fc) {
    if (!isSelfRequest(fc)) {
      super.processValidators(fc);
    }
  }

  @Override
  public void processUpdates(FacesContext fc) {
    if (!isSelfRequest(fc)) {
      super.processUpdates(fc);
    }
  }

  @Override
  public void queueEvent(FacesEvent event) {
    FacesContext context = FacesContext.getCurrentInstance();
    Map<String, String> params =
    context.getExternalContext().getRequestParameterMap();
    String eventName =
    params.get(Constants.PARTIAL_BEHAVIOR_EVENT_PARAM);
    String clientId = this.getClientId(context);

    if (isSelfRequest(context)) {
      AjaxBehaviorEvent behaviorEvent =
         (AjaxBehaviorEvent) event;
      LayoutPane pane = getLayoutPane(this,
         params.get(clientId + "_pane"));
      if (pane == null) {
        LOG.warning("LayoutPane by request parameter '" +
          params.get(clientId + "_pane") + "' was not found");

        return;
      }

      if ("open".equals(eventName)) {
        OpenEvent openEvent = new OpenEvent(pane,
        behaviorEvent.getBehavior());
```

```
        openEvent.setPhaseId(behaviorEvent.getPhaseId());
        super.queueEvent(openEvent);

        return;
      } else if ("close".equals(eventName)) {
        CloseEvent closeEvent = new CloseEvent(pane,
        behaviorEvent.getBehavior());
        closeEvent.setPhaseId(behaviorEvent.getPhaseId());
        super.queueEvent(closeEvent);

        return;
      } else if ("resize".equals(eventName)) {
        double width = Double.valueOf(params.get(clientId +
        "_width"));
        double height = Double.valueOf(params.get(clientId
        + "_height"));

        ResizeEvent resizeEvent = new ResizeEvent(pane,
        behaviorEvent.getBehavior(), width, height);
        event.setPhaseId(behaviorEvent.getPhaseId());
        super.queueEvent(resizeEvent);

        return;
      }
    }

    super.queueEvent(event);
  }

  public LayoutPane getLayoutPane(UIComponent component,
    String combinedPosition) {
    for (UIComponent child : component.getChildren()) {
      if (child instanceof LayoutPane) {
        if (((LayoutPane)
        child).getCombinedPosition().equals(combinedPosition)) {
          return (LayoutPane) child;
        } else {
          LayoutPane pane = getLayoutPane(child,
          combinedPosition);
          if (pane != null) {
            return pane;
          }
        }
      }
    }
```

```
        return null;
      }

      private boolean isSelfRequest(FacesContext context) {
        return this.getClientId(context).equals(

      context.getExternalContext().getRequestParameterMap().get
        (Constants.PARTIAL_SOURCE_PARAM));
      }
    }
```

The method `queueEvent(FacesEvent event)` instantiates three event classes, depending on the user interactions with layout panes. Every event class should extend `javax.faces.event.AjaxBehaviorEvent`. As an example, we would like to show what the `ResizeEvent` class looks:

```
  public class ResizeEvent extends AjaxBehaviorEvent {

    private double width;
    private double height;

    public ResizeEvent(UIComponent component,
      Behavior behavior, double width, double height) {
      super(component, behavior);
      this.width = width;
      this.height = height;
    }

    public final double getWidth() {
      return width;
    }

    public final double getHeight() {
      return height;
    }

    @Override
    public boolean isAppropriateListener(FacesListener
      facesListener) {
      return (facesListener instanceof AjaxBehaviorListener);
    }
```

```
    @Override
    public void processListener(FacesListener facesListener) {
      if (facesListener instanceof AjaxBehaviorListener) {
        ((AjaxBehaviorListener) facesListener).
        processAjaxBehavior(this);
      }
    }
  }
```

The `LayoutRenderer` class will get additional code too. This consists of the overridden method `decode(FacesContext fc, UIComponent component)` and the added call `encodeClientBehaviors(fc, layout)`.

```
  public class LayoutRenderer extends CoreRenderer {

    @Override
    public void decode(FacesContext fc, UIComponent
      component) {
      decodeBehaviors(fc, component);
    }

    ...

    protected void encodeScript(FacesContext fc, Layout
      layout) throws IOException {
      ...

      encodeClientBehaviors(fc, layout);

      writer.write("},true);});");
      endScript(writer);
    }
  }
```

Now we can also inspect the skipped method in the recipe *Writing a JavaScript widget* of this chapter. The implementation binds three callbacks to some special events in order to execute client behavior functions.

```
  bindEvents: function(parent) {
    var _self = this;

    // bind events
    parent.find(".ui-layout-pane")
```

```
    .bind("layoutpaneonopen",function () {
      var behavior = _self.cfg.behaviors ?
      _self.cfg.behaviors['open'] : null;
      if (behavior) {
        var combinedPosition = $(this).data('combinedposition');
         var ext = {
          params:[
            {name:_self.id + '_pane', value:combinedPosition}
          ]
        };

        behavior.call(_self, combinedPosition, ext);
      }
    }).bind("layoutpaneonclose",function () {
      var behavior = _self.cfg.behaviors ?
      _self.cfg.behaviors['close'] : null;
      if (behavior) {
        var combinedPosition = $(this).data('combinedposition');
        var ext = {
        params:[
          {name:_self.id + '_pane', value:combinedPosition}
        ]
      };

      behavior.call(_self, combinedPosition, ext);
    }
  }).bind("layoutpaneonresize", function () {
    var layoutPane = $(this).data("layoutPane");

    if (!layoutPane.state.isClosed && !layoutPane.state.isHidden) {
      var behavior = _self.cfg.behaviors ? _self.cfg.behaviors['resize']
      : null;
      if (behavior) {
        var combinedPosition = $(this).data('combinedposition');
          var ext = {
            params:[
              {name:_self.id + '_pane', value:combinedPosition},
              {name:_self.id + '_width', value:layoutPane.state.
              innerWidth},
```

```
                {name:_self.id + '_height',
                value:layoutPane.state.innerHeight}
            ]
              };

              behavior.call(_self, combinedPosition, ext);
            }
          }
        });
    }
```

How it works...

The mandatory method `getEventNames()` in the `Layout` class comes from the `ClientBehaviorHolder` and returns all the supported event names. The methods `processValidators(FacesContext fc)` and `processUpdates(FacesContext fc)` were overridden because the child components are not interested in the validation and update phases when a request is initiated by the component itself; for example, when the user has closed, opened, or resized a pane. The method `processDecodes(FacesContext fc)` was overridden because we need to decode the AJAX events when they are sent by the attached `p:ajax`. Decoding happens in the renderer class—`decode(FacesContext fc, UIComponent component)` calls `decodeBehaviors(fc, component)` from the `org.primefaces.renderkit.CoreRenderer`. During decoding, events may be enqueued for later processing by event listeners that have registered an interest. The enqueuing happens again in the `Layout` class in the method `queueEvent(FacesEvent event)`. This method is responsible for the creation of event instances that are put into the event queue. AJAX listeners later get these event instances as parameters when invoked.

What is the connecting link between Java code and the JavaScript widget? The magic happens in the call `encodeClientBehaviors(fc, layout)` (see `LayoutRenderer` in the preceding code snippet). This method prepares the callback functions for client behaviors and puts them into the widget's configuration object. The callback functions are then accessible in the widgets via the configuration object `cfg`, as `cfg.behaviors['event name']`, for example, `cfg.behaviors['close']`. We can call any callback function with all the parameters we need, using `cfg.behaviors['event name'].call(...)`. On the server side, sent parameters are extracted from the the request map.

It is recommended that you check out the PrimeFaces project and explore the code for ajaxified components to understand how AJAX behaviors work.

Binding all the parts together

Any component is used via its tag in Facelets. The `Layout` component can be used, say, as `pc:layout`; and the `LayoutPane` component as `pc:layoutPane`, where `pc` is any namespace of your choice (`pc` here is an abbreviation for "PrimeFaces cookbook"). But how should JSF find the corresponding component and renderer classes for a component tag? This occurs by the means of a proper configuration. A configuration binds the component's parts together.

In this recipe, we will learn about configuration details and prepare for all the steps in the last recipe of this chapter, *Running it*.

How to do it...

First of all, we will have to register the component class, component family, renderer class, and the renderer type in `faces-config.xml`. We have to do this for both our components.

```
<component>
  <component-type>org.primefaces.cookbook.component.Layout</component-
type>
  <component-class>org.primefaces.cookbook.component.Layout</
component-class>
</component>
<component>
  <component-type>org.primefaces.cookbook.component.LayoutPane</
component-type>
  <component-class>org.primefaces.cookbook.component.LayoutPane</
component-class>
</component>

<render-kit>
  <renderer>
    <component-family>org.primefaces.cookbook.component</component-
family>
    <renderer-type>org.primefaces.cookbook.component.LayoutRenderer</
renderer-type>
    <renderer-class>org.primefaces.cookbook.component.LayoutRenderer</
renderer-class>
</renderer>
<renderer>
  <component-family>org.primefaces.cookbook.component</component-
family>
  <renderer-type>org.primefaces.cookbook.component.
LayoutPaneRenderer</renderer-type>
  <renderer-class>org.primefaces.cookbook.component.
LayoutPaneRenderer</renderer-class>
```

```
  </renderer>
</render-kit>
```

The next code listing shows a Facelet tag library, `cookbook.taglib.xml`, which is created under the `WEB-INF` folder. The configuration demonstrates how the `layout` tag, with its attributes, can be registered and described. A configuration for the `layoutPane` tag is similar and hence is not shown here.

```
<?xml version="1.0"?>
  <facelet-taglib version="2.0"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facelettaglibrary_2_0.xsd">

    <namespace>http://primefaces.org/ui/cookbook</namespace>

    <tag>
        <description><![CDATA[Page Layout
        component.]]></description>
        <tag-name>layout</tag-name>
        <component>
            <component-type>org.primefaces.cookbook.component.Layout</
component-type>
            <renderer-type>org.primefaces.cookbook.component.
LayoutRenderer</renderer-type>
        </component>
        <attribute>
            <description>
                <![CDATA[Unique identifier of the component in
                a namingContainer.]]>
            </description>
            <name>id</name>
            <required>false</required>
            <type>java.lang.String</type>
        </attribute>
        ...
        <attribute>
            <description>
                <![CDATA[Style class of the main Layout
                container element. Default value is null.]]>
            </description>
            <name>styleClass</name>
            <required>false</required>
            <type>java.lang.String</type>
        </attribute>
```

```
        </tag>

        ...

    </facelet-taglib>
```

The tag library should be disclosed in `web.xml` by its path.

```
<context-param>
  <param-name>javax.faces.FACELETS_LIBRARIES</param-name>
  <param-value>/WEB-INF/cookbook.taglib.xml</param-value>
</context-param>
```

How it works...

The place for XML configuration files is either the `WEB-INF` (WAR projects) or `META-INF` (JAR projects) folder. The component family and renderer types for `faces-config.xml` were defined in the component classes `Layout` and `LayoutPane` (see the recipe *Writing component classes* of this chapter). The same types should be put in the XML tags `component-family` and `renderer-type`. Two other tags, `component-class` and `renderer-class`, point to the fully qualified class names.

The Facelet tag library, `cookbook.taglib.xml`, is the second important place where the component tag name, the tag handler class, references to the component, and the renderer types are registered. The component tag's name is linked over the component and renderer types to the data in `faces-config.xml`. Furthermore, this file describes the component's exposed attributes (name, type, required flag, and so on) along with the description. Component tags can be used in Facelets (XHTML files) under the specified namespace `http://primefaces.org/ui/cookbook`.

There is also an alternative way for registration in `faces-config.xml`. The component and renderer classes can be annotated with `@FacesComponent` and `@FacesRenderer` respectively (package `javax.faces.component`). In this case, no entries in `faces-config.xml` are needed.

A `contex-param` entry in `web.xml` is only required if we develop components in a WAR project. Putting components in a JAR file makes this entry redundant.

See also

The last recipe of this chapter, *Running it*, will show how to use the component tags `layout` and `layoutPane` with the specified namespace `http://primefaces.org/ui/cookbook`.

Running it

After we developed a new component, we would like to use it in real projects. This recipe gives an example of how to use a full-page layout in practice. The example covers an XHTML part as well as a managed bean implementation. We will design a full-page layout with two nested layout panes in the east position.

How to do it...

Any layout is by default a full-page layout according to the default value of the `fullPage` attribute. Every full-page layout should be placed as a direct child of the `h:body` or `h:form` tag that is inside the `h:body`. Let us design the following structure.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:pc="http://primefaces.org/ui/cookbook">
<f:view contentType="text/html" locale="en">
  <h:head title="PrimeFaces Cookbook - Layout Component">
  ...
    </h:head>
    <h:body>
        <pc:layout id="fpl" widgetVar="fullLayoutWidget"
                   options="#{showcaseLayout.layoutOptions}">
            <pc:layoutPane position="north">
                North content
            </pc:layoutPane>

            <h:form id="mainForm" prependId="false">
                <pc:layoutPane position="center">
                    Center content
                </pc:layoutPane>

                <pc:layoutPane position="west">
                    <f:facet name="header">
                        West header
                    </f:facet>
                    West content
                </pc:layoutPane>
```

```
                <pc:layoutPane position="east">
                    <!-- nested layout -->
                    <pc:layoutPane position="center">
                        <f:facet name="header">
                            East-Center header
                        </f:facet>
                        East-Center content
                    </pc:layoutPane>
                    <pc:layoutPane position="south">
                        East-South content
                    </pc:layoutPane>
                </pc:layoutPane>

                <pc:layoutPane position="south">
                    South content
                </pc:layoutPane>
            </h:form>

        </pc:layout>
    </h:body>
</f:view>
</html>
```

Layout options will be created in a managed bean, `ShowcaseLayout`.

```
@ApplicationScoped
@ManagedBean(eager = true)
public class ShowcaseLayout {

  private LayoutOptions layoutOptions;

    @PostConstruct
    protected void initialize() {
        layoutOptions = new LayoutOptions();

    // for all panes
    LayoutOptions panes = new LayoutOptions();
    panes.addOption("resizable", true);
    panes.addOption("closable", true);
    panes.addOption("slidable", false);
    panes.addOption("spacing", 6);
    panes.addOption("resizeWithWindow", false);
    panes.addOption("resizeWhileDragging", true);
    layoutOptions.setPanesOptions(panes);
```

```
// north pane
LayoutOptions north = new LayoutOptions();
north.addOption("resizable", false);
north.addOption("closable", false);
north.addOption("size", 60);
layoutOptions.setNorthOptions(north);

// south pane
LayoutOptions south = new LayoutOptions();
south.addOption("resizable", false);
south.addOption("closable", false);
south.addOption("size", 40);
layoutOptions.setSouthOptions(south);

// center pane
LayoutOptions center = new LayoutOptions();
center.addOption("resizable", false);
center.addOption("closable", false);
center.addOption("resizeWhileDragging", false);
center.addOption("minWidth", 200);
center.addOption("minHeight", 60);
layoutOptions.setCenterOptions(center);

// west pane
LayoutOptions west = new LayoutOptions();
west.addOption("size", 210);
west.addOption("minSize", 180);
west.addOption("maxSize", 500);
layoutOptions.setWestOptions(west);

// east pane
LayoutOptions east = new LayoutOptions();
east.addOption("size", 448);
east.addOption("minSize", 180);
east.addOption("maxSize", 650);
layoutOptions.setEastOptions(east);

// nested east layout
LayoutOptions childEastOptions = new LayoutOptions();
east.setChildOptions(childEastOptions);
```

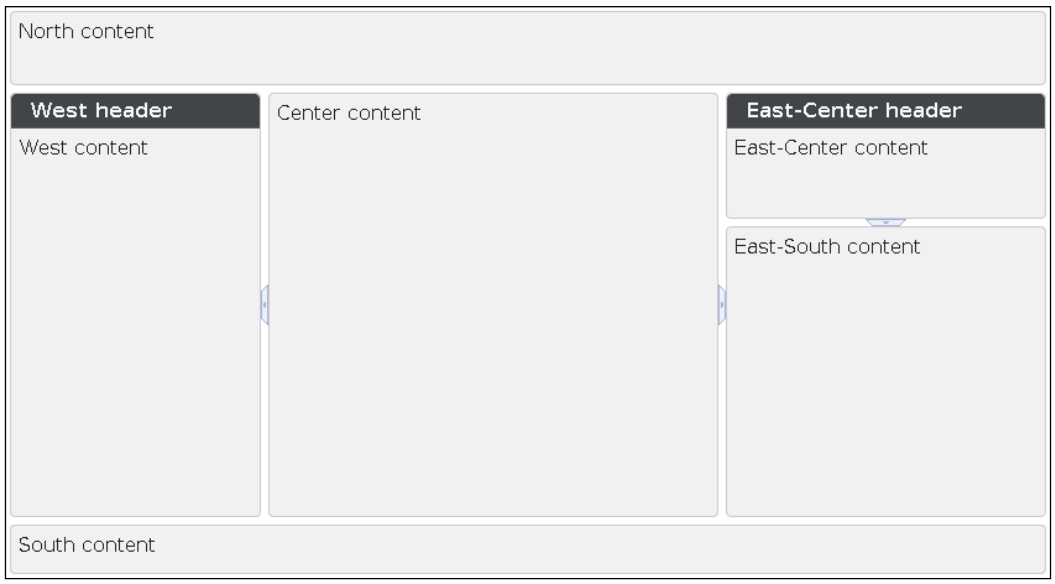
```
        // east-center pane
        LayoutOptions eastCenter = new LayoutOptions();
        eastCenter.addOption("minHeight", 60);
        childEastOptions.setCenterOptions(eastCenter);

        // south-center pane
        LayoutOptions southCenter = new LayoutOptions();
        southCenter.addOption("size", "70%");
        southCenter.addOption("minSize", 60);
        childEastOptions.setSouthOptions(southCenter);
      }

      public LayoutOptions getLayoutOptions() {
        return layoutOptions;
      }

      public void setLayoutOptions(LayoutOptions layoutOptions) {
        this.layoutOptions = layoutOptions;
      }
    }
```

We are done. The end result looks impressive:



How it works...

There is only one `pc:layout` tag on the page. Adding nested `pc:layoutPane` tags is enough to create nested layouts. That means one `pc:layoutPane` component can be put as a direct child of another `pc:layoutPane` component. This process can be continued as long as needed. There is only one requirement—every layout (nested or not) always expects a pane with `position` set to `center`. The XHTML part is thus clarified.

The algorithm for building layout options in Java is straightforward as well. There are setters for every pane position. There is also a method for setting options describing nested layouts. We set empty options for the nested east layout with `east.setChildOptions(childEastOptions)`. All the layout options were created during application startup in an application-scoped bean. That approach helps speed up layout creation.

PrimeFaces Cookbook Showcase application

This recipe is available in the demo web application on GitHub (`https://github.com/ova2/primefaces-cookbook`). Clone the project if you have not done it yet, explore the project structure, and execute the built-in Jetty Maven plugin to see this recipe in action. Follow the instructions in the `README` file if you do not know how to run Jetty.

When the server is running, the showcase for the recipe is available under `http://localhost:8080/primefaces-cookbook/views/chapter11/customComponent.jsf`.