12
PrimeFaces Extensions in Action

In this chapter we will cover:

- Showing error messages in tooltips
- Rotating and resizing the images
- Managing states with TriStateCheckbox and TriStateManyCheckbox
- Supporting numeric inputs with InputNumber
- Creating a pleasant editor interface for code-like content
- Calling Java methods with parameters on the client side
- Building a dynamic form described by a model
- Timeline – displaying events in chronological order

Introduction

PrimeFaces Extensions (`http://primefaces-extensions.github.com/`) is an open source, community-driven project founded by Thomas Andraschko and Oleg Varaksin, who are power users and well-known members in the PrimeFaces community. The project contains various components that are neither available in PrimeFaces nor in any other JSF UI library, extensions to existing features, converters, validators, useful tools such as a Maven plugin for web resource optimization, and others. PrimeFaces Extensions is the official extensions project developed in a similar manner and with the same underlying APIs as PrimeFaces. The community can expect more from this project in the near future.

In this chapter, we will give an introduction to PrimeFaces Extensions and present some wonderful components that can spark interest in developing rich web applications. The project's homepage contains many links, and to the *Getting Started* section amongst others to a showcase (`https://github.com/primefaces-extensions/primefaces-extensions.github.com/wiki/Getting-Started`). The *Getting Started* section describes how to set up and work with this additional component library. Maven users have a straightforward setup. No extra repositories need to be configured in `pom.xml`. The PrimeFaces Extensions project is available in the Maven Central repository. The following dependency is enough to use the project:

```
<dependencies>
    ...
    <dependency>
        <groupId>org.primefaces.extensions</groupId>
        <artifactId>primefaces-extensions</artifactId>
        <version>0.6.2</version>
    </dependency>
    ...
</dependencies>
```

The latest version can be determined by searching in the Maven Central repository. Users who are planning to use the `CKEditor` or `CodeMirror` component should also add necessary dependencies to the component's resources (CSS, JavaScript files).

```
<dependency>
    <groupId>org.primefaces.extensions</groupId>
    <artifactId>resources-ckeditor</artifactId>
    <version>0.6.2</version>
</dependency>

<dependency>
    <groupId>org.primefaces.extensions</groupId>
    <artifactId>resources-codemirror</artifactId>
    <version>0.6.2</version>
</dependency>
```

After these preparations, the component tags of PrimeFaces Extensions can be used in Facelets with the namespace `xmlns:pe="http://primefaces.org/ui/extensions"`.

Resources in the delivered JAR files can be compressed or uncompressed. PrimeFaces Extensions streams down the compressed resources for the JSF ProjectStage "Production", and it streams down uncompressed resources for the ProjectStage "Development". This allows debugging resources and reports issues more efficiently. JSF ProjectStage can be configured in `web.xml` by the context parameter `javax.faces.PROJECT_STAGE`.

Showing error messages in tooltips

Tooltips are nifty speech bubble tips. They are integral parts of every web application. Implementation of the `Tooltip` component in PrimeFaces Extensions leaves nothing to be desired. It provides useful features such as global, shared, auto-shown tooltips, tooltips with mouse tracking, positioning using corners, effects, and more. Tooltips are highly configurable and support AJAX updates.

In this recipe, we will demonstrate using global tooltips with the jQuery selectors—so-called global limited tooltips. In global mode, the tooltip uses the `title` values of the JSF components. Placing one tooltip is enough instead of using a tooltip for each target component. Global tooltips are normally applied to all components on a page, but this behavior can be controlled in a much smarter way with global limited tooltips. Such tooltips can be applied only to the component specified in the `for` attribute or to HTML elements specified in `forSelector` (jQuery selector). We will develop a practical example where tooltips appear only for components that did not pass validation. This is quite a nice replacement for regular error messages. It is also preferable to show error messages in tooltips on big, crowded forms or in large data tables.

How to do it...

The following XHTML code shows three input fields. The first field is declared as required, the second one has a converter `f:convertDateTime` to accept only dates, and the third field has a validator `f:validateLength` to accept only inputs with length `1`. Furthermore, there is a `submit` button and a global limited `pe:tooltip` component.

```
<h:panelGrid id="details" columns="2"
  columnClasses="formLabel,formEdit">
  <p:outputLabel for="name" value="Name (required)"/>
  <p:inputText id="name" value="#{tooltipsController.name}"
    required="true"
    title="#{component.valid ? '' :
    tooltipsController.getErrorMessage(component.clientId)}"/>

  <p:outputLabel for="date" value="Birth Date (date)"/>
  <p:inputText id="date" value="#{tooltipsController.birthDate}"
    title="#{component.valid ? '' :
    tooltipsController.getErrorMessage(component.clientId)}">
    <f:convertDateTime pattern="dd.MM.yyyy"/>
  </p:inputText>
```

```
    <p:outputLabel for="children" value="Children (number)"/>
    <p:inputText id="children"
    value="#{tooltipsController.children}" title="#{component.valid ? '' :
      tooltipsController.getErrorMessage(component.clientId)}">
      <f:validateLength maximum="1"/>
    </p:inputText>

</h:panelGrid>

<p:commandButton value="Submit" process="details" update="details"
  style="margin-top: 10px;"/>

<pe:tooltip global="true" myPosition="left center"
  atPosition="right center" forSelector=".ui-state-error"/>
```

Values of the `title` attribute on `p:inputText` are bound to the bean's method `getErrorMessage(String clientId)`.

```
@ManagedBean
@ViewScoped
public class TooltipsController implements Serializable {

    private String name;
    private Date birthDate;
    private int children;

    // getter / setter
    ...

    public String getErrorMessage(String clientId) {
        FacesContext fc = FacesContext.getCurrentInstance();
        Iterator<FacesMessage> iter = fc.getMessages(clientId);
        if (iter.hasNext()) {
            return iter.next().getDetail();
        }

        return "";
    }
}
```

The following screenshot shows what an error message in the tooltip looks like:

How it works...

All invalid components gets the `ui-state-error` style class. The idea is to apply a global tooltip (`global="true"`) to the invalid components with `forSelector` set to `ui-state-error`. We have decided to show validation messages on the right side of the fields that have not passed the validation. The tooltip's position is configured by the following two attributes:

- `myPosition`: Defines the corner of the tooltip in relation to the target element
- `atPosition`: Defines the corner of the target element

A validation message appears when the user clicks on an invalid field and disappears on blur. An interesting part in the bean is the `getErrorMessage(String clientId)` method. This method looks for the error message for a component with a given client ID. In the example, only message details are displayed.

Global and global limited tooltips are smart enough and survive AJAX updates on components to which tooltips are applied.

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/chapter12/messagesTooltips.jsf`.

Rotating and resizing the images

Image manipulation is not a trivial task on the web. Common tasks such as image area selection, rotation, and resizing are possible with two components from PrimeFaces Extensions: `ImageAreaSelect` and `ImageRotateAndResize`. The underlying widgets use the HTML5 canvas element if available, or Internet Explorer's Matrix Filter, and so are compatible with all major browsers.

In this recipe, we will only discuss the `ImageRotateAndResize` component, which can be used for rotating or resizing images and provides the AJAX events to catch new image values on the server side for further processing. We will develop a simple example to show basic functionality so that users can start to use this component immediately.

How to do it...

We will resize and rotate the logo from the PrimeFaces Extensions project, `pfext-logo.png`. This can be done on the client side with the `pe:imageRotateAndResize` tag and the exposed widget variable. To catch new image values on the sever side, two AJAX events are available: `rotate` and `resize`.

```
<p:growl id="growl" showDetail="true"/>

<h:graphicImage id="myImage"
  value="/resources/images/pfext-logo.png"/>

<pe:imageRotateAndResize id="rotateAndResize" for="myImage"
  widgetVar="rotateAndResizeWidget">
  <p:ajax event="rotate" update="growl"/>
  listener="#{imageController.rotateListener}"
  <p:ajax event="resize" update="growl"/>
  listener="#{imageController.resizeListener}"
</pe:imageRotateAndResize>

<p:commandButton type="button" icon="ui-icon-arrowreturnthick-1-w"
  value="Rotate -90" onclick="rotateAndResizeWidget.rotateLeft(90)"/>
<p:commandButton type="button" icon="ui-icon-arrowreturnthick-1-e"
  value="Rotate +90" onclick="rotateAndResizeWidget.rotateRight(90)"/>
<p:commandButton type="button" icon="ui-icon-zoomin"
  value="Scale +1.05" onclick="rotateAndResizeWidget.scale(1.05)"/>
```

```
<p:commandButton type="button" icon="ui-icon-zoomout"
value="Scale -0.95" onclick="rotateAndResizeWidget.scale(0.95)"/>
```

The AJAX listeners `rotateListener(RotateEvent)` and `resizeListener(ResizeEvent)` are defined in a managed bean `ImageController`.

```
@ManagedBean
@ViewScoped
public class ImageController implements Serializable {

  public void rotateListener(RotateEvent e) {
    FacesMessage msg = new
      FacesMessage(FacesMessage.SEVERITY_INFO,
      "Image rotated", "Degree:" + e.getDegree());

    FacesContext.getCurrentInstance().addMessage(null, msg);
  }

  public void resizeListener(ResizeEvent e) {
    FacesMessage msg = new
      FacesMessage(FacesMessage.SEVERITY_INFO,
      "Image resized", "Width:" + e.getWidth() + ", Height: " +
      e.getHeight());

    FacesContext.getCurrentInstance().addMessage(null, msg);
  }
}
```

The following screenshot shows a growl notification with new image dimensions when the user presses the **Scale +1.05** button:

How it works...

The widget provides the following client-side methods to rotate, resize, and scale any image:

- `rotateLeft(degree)`: Rotates the image in the left direction with the given degree
- `rotateRight(degree)`: Rotates the image in the right direction with the given degree
- `resize(width, height)`: Resizes the image to the given width and height
- `scale(scaleFactor)`: Scales the image with the given factor

`RotateEvent` provides a new rotation degree and `ResizeEvent` provides new height and width of the image, which is referenced via the `for` attribute in `pe:imageRotateAndResize`.

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/chapter12/rotateResize.jsf`.

Managing states with TriStateCheckbox and TriStateManyCheckbox

Some applications use checkboxes that allow a third state in addition to the two provided by a normal checkbox. The third state indicates that it is indeterminate—neither checked nor unchecked. Imagine you have to manage the access rights in a user management system. You can allow a specific right (checkbox is checked), deny it (checkbox is not checked), or you may not set it at all (use system default or the inherited right). There are thus three states that are represented visually by three separate icons. Normally, checkbox implementations allow users to toggle among all states.

In this recipe, we will learn how to use a three-state checkbox implementation, `TriStateCheckbox`, which adds a new state for a common `SelectBooleanCheckbox` component.

How to do it...

Usage of `pe:triStateCheckbox` is similar to `p:selectBooleanCheckbox`. We will implement four three-state checkboxes. The first one demonstrates the basic usage, the second one shows an ajaxified three-state checkbox, and the third and fourth samples show how to use `pe:triStateCheckbox` with custom icons and item label.

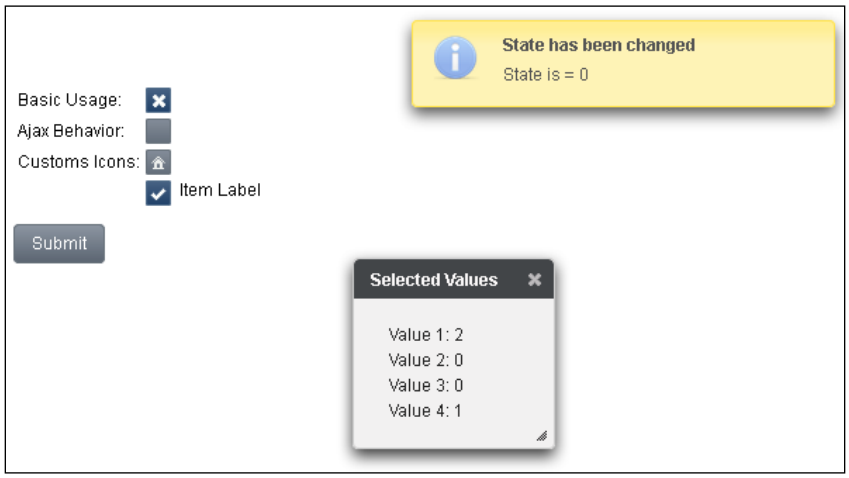
```
<p:growl id="growl" showDetail="true"/>

<h:panelGrid id="triStateGrid" columns="2">
  <h:outputText value="Basic Usage: "/>
  <pe:triStateCheckbox
    value="#{triStateCheckBoxController.value1}"/>

  <h:outputText value="Ajax Behavior: "/>
  <pe:triStateCheckbox id="ajaxTriState"
    value="#{triStateCheckBoxController.value2}">
    <p:ajax update="growl"
    listener="#{triStateCheckBoxController.addMessage}"/>
  </pe:triStateCheckbox>

  <h:outputText value="Customs Icons: "/>
  <pe:triStateCheckbox
    value="#{triStateCheckBoxController.value3}"
    stateOneIcon="ui-icon-home"
    stateTwoIcon="ui-icon-plus"
    stateThreeIcon="ui-icon-minus"/>

  <h:panelGroup/>
  <pe:triStateCheckbox
    value="#{triStateCheckBoxController.value4}"
    itemLabel="Item Label"/>
</h:panelGrid>

<p:commandButton value="Submit"
  process="triStateGrid" update="display"
  oncomplete="dlg.show()" style="margin-top:10px;"/>
```

```
<p:dialog header="Selected Values" widgetVar="dlg">
  <h:panelGrid id="display" columns="1" style="margin: 10px;">
    <h:outputText value="Value 1:
    #{triStateCheckBoxController.value1}"/>
    <h:outputText value="Value 2:
    #{triStateCheckBoxController.value2}"/>
    <h:outputText value="Value 3:
    #{triStateCheckBoxController.value3}"/>
    <h:outputText value="Value 4:
    #{triStateCheckBoxController.value4}"/>
  </h:panelGrid>
</p:dialog>
```

The current selection will be displayed in a dialog when the user clicks on the **Submit** button. The checkbox values are bound to String values in a managed bean.

```
@ManagedBean
@ViewScoped
public class TriStateCheckBoxController implements Serializable {

  private String value1;
  private String value2;
  private String value3;
  private String value4;

  public TriStateCheckBoxController() {
    value1 = "2";
    value4 = "1";
  }

  // getter / setter
    ...

  public void addMessage() {
    FacesMessage msg = new
    FacesMessage(FacesMessage.SEVERITY_INFO,
     "State has been changed", "State is = " + value2);
    FacesContext.getCurrentInstance().addMessage(null, msg);
  }
}
```

The dialog with selected values as well as a growl notification for the second ajaxified `pe:triStateCheckbox` is shown in the following screenshot:



How it works...

The values of `TriStateCheckbox` are mapped to the String values `0`, `1`, and `2`, which represent the three states. The component can be customized with the CSS classes for icons. There are three tag attributes for that:

- `stateOneIcon`: The icon for the first state as a CSS class
- `stateTwoIcon`: The icon for the second state as a CSS class
- `stateThreeIcon`: The icon for the third state as a CSS class

There's more...

There is also a `pe:triStateManyCheckbox` tag, which accepts many `f:selectItem` or `f:selectItems` components. The showcase of the PrimeFaces Extensions contains several examples for simple usage, as well as with converters and validators. Converters allow state mapping departing from the `0`, `1`, and `2` string values.

See also

- Using the two-state checkboxes is the subject of the *Discovering selectBooleanCheckbox and selectManyCheckbox* recipe in *Chapter 3*, *Enhanced Inputs and Selects*.

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/chapter12/triStateCheckboxes.jsf`.

Supporting numeric inputs with InputNumber

Web applications often have requirements to limit inputs to numbers. The `InputNumber` component in PrimeFaces allows you to input currency, percentages, and any type of numeric data in various formats. It has built-in capabilities such as customizable number of digits and decimal digits, adding a symbol and the symbol's position, defining the decimal and thousand separator characters, minimum or maximum values, and a lot of rounding methods.

In this recipe, we will see some basic features of `InputNumber` in action. Advanced features, especially the rounding methods, can be explored in the deployed showcase on the homepage of PrimeFaces Extensions.

How to do it...

The following code snippet includes four examples that demonstrate different attribute combinations of the `InputNumber` component. We will see how the component behaves with a standard configuration, with a currency symbol and a group or decimal separator, with a minimum or maximum value, and with custom decimal places. The attached `p:ajax` tag will show the input value in the corresponding output field when the focus leaves the input field.

```
<h:panelGrid columns="3">
  <h:panelGroup style="font-weight:bold;">
    Input Number</h:panelGroup>
  <h:panelGroup style="font-weight:bold;">
    Double Value</h:panelGroup>
  <h:panelGroup
    style="font-weight:bold;">
    Description
  </h:panelGroup>

  <pe:inputNumber id="input1"
    value="#{inputNumberController.input1}">
    <p:ajax event="blur" update="valueInput1"/>
```

```
    </pe:inputNumber>
    <p:inputText id="valueInput1" disabled="true"
      value="#{inputNumberController.input1}"/>
    <h:outputText value="Without configuration."/>

    <pe:inputNumber id="input2"
    value="#{inputNumberController.input2}"
      symbol=" CHF" symbolPosition="s"
      decimalSeparator="," thousandSeparator=".">
      <p:ajax event="blur" update="valueInput2"/>
    </pe:inputNumber>
    <p:inputText id="valueInput2" disabled="true"
      value="#{inputNumberController.input2}"/>
    <h:outputText value="Currency symbol and decimal / thousand
    separators."/>

    <pe:inputNumber id="input3"
    value="#{inputNumberController.input3}"
      minValue="-1000.999" maxValue="1000">
      <p:ajax event="blur" update="valueInput3"/>
    </pe:inputNumber>
    <p:inputText id="valueInput3" disabled="true"
      value="#{inputNumberController.input3}"/>
    <h:outputText value="Maximun and minimum values (-1000.999 to
    1000.000)."/>

    <pe:inputNumber id="input4" decimalPlaces="6"
      value="#{inputNumberController.input4}">
      <p:ajax event="blur" update="valueInput4"/>
    </pe:inputNumber>
    <p:inputText id="valueInput4" disabled="true"
      value="#{inputNumberController.input4}"/>
    <h:outputText value="Custom decimal places."/>
  </h:panelGrid>
```

The following screenshot shows the four developed examples:

Input Number	Double Value	Description
22,222.00	22222.0	Without configuration.
55.685,00 CHF	55685.0	Currency symbol and decimal / thousand separators.
999.000	999.0	Maximun and minimum values (-1000.999 to 1000.000).
0.123456	0.123456	Custom decimal places.

How it works...

The `pe:inputNumber` tag transforms a normal input field into a field that only accepts numeric inputs. Any symbol can be appended or prepended to the numeric value by the `symbol` attribute. In the second example, we applied the Swiss currency by setting `symbol` to `CHF`. The symbol's position is specified by the `symbolPosition` attribute. A symbol is prepended by default. We set `symbolPosition` to `s`, which means suffix, so that the symbol was appended. Other useful settings are `decimalSeparator` and `thousandSeparator`. They define the decimal and thousand separator characters respectively. In the example, we used `decimalSeparator=","` and `thousandSeparator="."`. The minimal allowed input value can be set by the `minValue` attribute and the maximal allowed one by the `maxValue` attribute. In the third example, we set `minValue` to `-1000.999` and `maxValue` to `1000`. Decimal places are configured by the `decimalPlaces` attribute. The default number of decimal places is `2`. In the fourth example, we increased this value to `6` by setting `decimalPlaces` to `6`.

There's more...

The `pe:inputNumber` tag also works with validators and converters. The usage of validators and converters is the same as for other JSF standard and PrimeFaces input components. The following example demonstrates how we can force inputs such as `$ 60.00`. The `<f:validateDoubleRange minimum="50"/>` setting ensures that the user cannot input values less than 50. An example with a converter is shown for the sake of completeness as well.

```
<pe:inputNumber  value="#{inputNumberController.input6}" symbol="$ "
  validatorMessage="The value must be 50.00 or greater">
  <f:validateDoubleRange minimum="50"/>
</pe:inputNumber>

<pe:inputNumber  value="#{inputNumberController.distance}"
  symbol="meters " converter="inputNumberConverter"/>
```

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/chapter12/inputNumber.jsf`.

Creating a pleasant editor interface for code-like content

`CodeMirror` is a component that can be used to create a relatively pleasant editor interface for code-like content—computer programs, HTML markup, and similar. It supports different modes. If a mode has been defined for a language we are editing, the code will be colored, and the editor will optionally help us with indentation. A full list of all supported modes with descriptions can be found on the homepage of the underlying native widget (`http://codemirror.net`).

In this recipe, we will see samples to learn how to use this component and also learn how the code completion works.

How to do it...

The `CodeMirror` component is created by the `pe:codeMirror` tag. We will use it with the `value`, `mode`, `theme`, `lineNumbers`, `completeMethod`, and `extraKeys` attributes. Our intention is to implement server-side as well as client-side code completion for a JavaScript code.

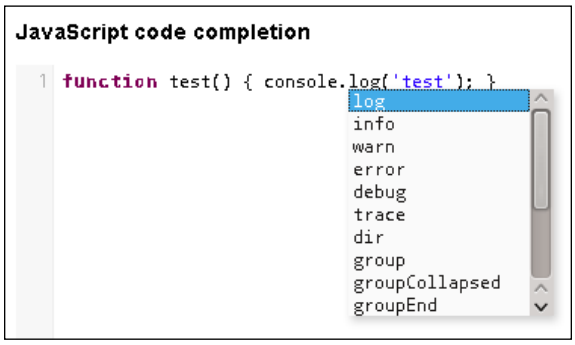
```
<h3 style="margin-top:0">Server-side code completion</h3>

<pe:codeMirror id="codeMirror1"
    mode="#{codeMirrorController.mode}"
    widgetVar="myCodeMirror"
    theme="eclipse" value="#{codeMirrorController.content}"
    lineNumbers="true"
    completeMethod="#{codeMirrorController.complete}"
    extraKeys="{'Ctrl-Space':
      function(cm){myCodeMirror.complete();}}"/>

<h3 style="margin-top:20px">JavaScript code completion</h3>

<pe:codeMirror id="codeMirror2" mode="#{codeMirrorController.mode}"
    theme="eclipse" value="#{codeMirrorController.content}"
    lineNumbers="true"
    extraKeys="{'Ctrl-Space':
    function(cm){CodeMirror.simpleHint
      (cm, CodeMirror.javascriptHint);}}"/>
```

```
<p:commandButton actionListener="#{codeMirrorController.changeMode}"
    update="codeMirror1 codeMirror2"
    value="Change mode with AJAX" style="margin-top:10px;"/>
```

The `CodeMirrorController` bean has the `javascript` mode by default and simple JavaScript content bound to the `value` attribute of the `pe:codeMirror` tag. Also, a server-side method, `complete(String token, String context)`, is available to be used via the `completeMethod` attribute.

```
@ManagedBean
@ViewScoped
public class CodeMirrorController implements Serializable {

    private String content;
    private String mode = "javascript";

    public CodeMirrorController() {
        content = "function test() { console.log('test'); }";
    }

    public void changeMode() {
        if (mode.equals("css")) {
            mode = "javascript";
        } else {
            mode = "css";
        }
    }

    public List<String> complete(String token, String context) {
        ArrayList<String> suggestions = new ArrayList<String>();

        suggestions.add("context: " + context);
        suggestions.add("token: " + token);

        return suggestions;
    }

    // getter / setter for content and mode
    ...
}
```

The following screenshot demonstrates the mentioned code completion for JavaScript code:



How it works...

The current content of the editor can be acquired from the `value` attribute when `pe:codeMirror` is processed. The mode may be set to a string value, which simply names the mode or is a MIME type associated with the mode. Alternatively, it may be an object containing configuration options for the mode, with a name property that names the mode, for example, `{name: "javascript", json: true}`. Information about the configuration parameters for each mode can be found on the homepage of the underlying native widget (`http://codemirror.net`). In the developed example, the user can switch between the `javascript` and `css` modes. So, it is possible to switch the predefined mode at runtime and start to type a CSS code.

The theme to style the editor with is provided by the `theme` attribute. We have used the `eclipse` theme. Beside `eclipse`, the following themes are also available:

- `ambiance`
- `blackboard`
- `cobalt`
- `elegant`
- `erlang-dark`
- `lesser-dark`
- `monokai`
- `neat`
- `night`
- `rubyblue`
- `vibrant-ink`
- `xq-dark`

The `lineNumbers` attribute defines whether or not to show line numbers to the left of the editor. Line numbers are not shown by default. There is also the `gutter` attribute, which can be used to force a "gutter" (empty space on the left-hand side of the editor) to be shown even when no line numbers are active. This is useful for setting markers.

`CodeMirror` provides server-side support for code completion (activated by `Ctrl-Space`) via `completeMethod`. The activation is done by the `extraKeys` attribute, which specifies extra key bindings for the editor. When it is given, it should be an object with property names such as `Ctrl-A`, `Home`, or `Ctrl-Alt-Left`. `completeMethod` expects two strings, `context` and `token`, and returns a list of suggestions.

`context` is the current context such as `console` (see the example in the *How it works...* section of this recipe)—when we hit `Ctrl-Space` after `console.` (dot after `console`).

`token` is the current token such as `console`—when we hit `Ctrl-Space` somewhere in the console. If, for example, we hit `Ctrl-Space` after `o` in `log`, `context` is `console` and `token` is `log`.

The client-side code completion works in a similar way to server-side code completion, but without a binding to the server-side method. In the example, the `extraKeys` object tells us that we want to have simple suggestions (hints) for JavaScript.

There's more...

`CodeMirror` also provides CSS styles for some modes. The following CSS files are available for the modes `spec`, `tiddlywiki`, and `tiki` respectively:

- `codemirror/mode/spec.css`
- `codemirror/mode/tiddlywiki.css`
- `codemirror/mode/tiki.css`

If you would like to include them, use `h:outputStylesheet`; for example, as follows:

```
<h:outputStylesheet library="primefaces-extensions" name="codemirror/
mode/tiki.css"/>.
```

Included styles will be applied to all `CodeMirror` instances on a page.

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/chapter12/codeMirror.jsf`.

Calling Java methods with parameters on the client side

Some Java or web developers are probably familiar with the **Direct Web Remoting** (**DWR**) framework (`http://directwebremoting.org/dwr`). DWR is a Java library that enables Java on the server and JavaScript in a browser to interact with and call each other. It has several modules for integration, such as Spring, Struts, and Guice. There is also a JSF integration, but it's not good enough to be used in JSF 2. It requires an expensive XML configuration or a configuration via annotations. JSF 2 standard lacks the ability to call arbitrary methods on JSF managed beans as well. Some attempts were made to push up this intention and integrate this technique in JSF 2, but nothing has happened until now. PrimeFaces Extensions has a component `RemoteCommand` that enhances the component of the same name in PrimeFaces and brings more power and flexibility. It covers two important use cases. The first one is assigning JavaScript parameters to bean properties in a convenient way. If needed, validators and converters can be applied as well. The second one is executing remote Java methods with any parameters from the client side. This is a DWR-like approach with zero configuration.

In this recipe, we will consider only the second case and discuss how to call any method in a managed bean from JavaScript.

How to do it...

Our task can be done with the main tag `pe:remoteCommand` and its child tags `pe:methodParam` and `pe:methodSignature`. We will develop an example with three parameters and attached converters. Two command buttons should execute the `pe:remoteCommand` tag and show sent data in a growl component.

```
<p:growl id="growl" showDetail="true"/>

<pe:remoteCommand id="applyDataCommand" name="applyData"
    process="@this" update="growl"
    actionListener="#{remoteCommandController.printMethodParams}">
    <pe:methodSignature parameters="java.lang.String,
       java.util.Date,
       org.primefaces.cookbook.model.chapter12.Circle"/>
    <pe:methodParam name="subject"/>
    <pe:methodParam name="date">
       <f:convertDateTime type="both" dateStyle="short"
       locale="en"/> </pe:methodParam>
```

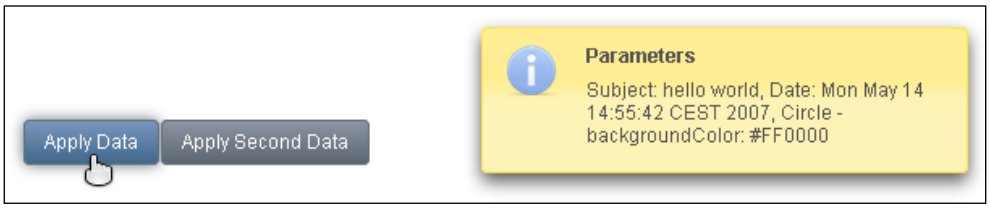
```
        <pe:methodParam name="circle">
            <pe:convertJson/>
        </pe:methodParam>
    </pe:remoteCommand>

    <script type="text/javascript">
        circle = {
            radius:50,
            backgroundColor:"#FF0000",
            borderColor:"#DDDDDD",
            scaleFactor:1.2
        };
        circle2 = {
            radius:32,
            backgroundColor:"#FF0320",
            borderColor:"#DDFFFD",
            scaleFactor:1.6
        };
    </script>

    <p:commandButton value="Apply Data" type="button"
        onclick="applyData('hello world', '5/14/07 12:55:42 PM',
    JSON.stringify(circle))"/>
    <p:commandButton value="Apply Second Data" type="button"
        onclick="applyData('hello user', '7/11/01 11:55:42 PM',
        JSON.stringify(circle2))"/>
```

The managed bean `RemoteCommandController` contains a method with three parameters to be called from the client side.

```
    @ManagedBean
    @RequestScoped
    public class RemoteCommandController {

        private String subject;
        private Date date;
        private Circle circle;

        public void printMethodParams(String subject, Date date,
        Circle circle) {
            FacesMessage msg = new
            FacesMessage(FacesMessage.SEVERITY_INFO, "Parameters",
            "Subject: " + subject + ", Date: " + date + ", Circle -
            backgroundColor: " +
```

```
            circle.getBackgroundColor());

            FacesContext.getCurrentInstance().addMessage(null, msg);
        }

        // getter / setter
        ...
    }
```

The following screenshot shows what the growl notification looks like when the user presses the first button. We can see that the client-side data has been correct converted, and transferred to the server.



How it works...

The name of the `pe:remoteCommand` component (which in our case is set to `applyData`) should match the name of the JavaScript function. In the example, the JavaScript function `applyData` is called via `onclick` in `p:commandButton`.

As an attentive reader could probably guess, `pe:methodParam` specifies a name of the parameter and `pe:methodSignature` specifies a comma- or space-separated list with fully qualified class names. Class names should match the passed parameters in the same order as they were defined. Converters are used as well. They can also be defined as usual in JSF by the `converter` attribute in `pe:methodParam`. The attached date converter converts a date string to a date object (`java.util.Date`), and a JSON converter (from the PrimeFaces Extensions project) converts a JavaScript object (`circle`) to a plain Java class (`org.primefaces.cookbook.model.chapter12.Circle.java`).

The bean method `printMethodParams` will be called when the user pushes either of the two command buttons. The JavaScript parameters, defined via `pe:methodParam`, get assigned to the corresponding Java parameters. The method generates a message that is shown as a growl pop up.

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/chapter12/remoteCommand.jsf`.

Building a dynamic form described by a model

A HTML form on a web page allows the user to enter data. Users fill in the forms using checkboxes, radio buttons, text fields, select lists, and so on. Normally, we can build a form in quite a straightforward manner by using the `h:panelGrid` or `p:panelGrid` tags, if the count of rows or columns, positions of elements, and so on are known. That is true for static forms. But it is not possible to use `h:panelGrid` or `p:panelGrid` if a form is described dynamically at runtime; for instance, if the entire definition of the form is placed in a database or an XML file. The `DynaForm` component makes it possible to build a dynamic form with labels, inputs, selects, and any other elements by using a model in Java. There are no limitations.

In this recipe, we will learn the process for building dynamic forms and show some basic features. Advanced use cases can be found in the showcase on the homepage of PrimeFaces Extensions.

How to do it...

Let us write a dynamic form in XHTML containing one or many `p:inputText`, `p:calendar`, `p:selectOneMenu`, `p:inputTextarea`, and `p:rating` components. The form should have `submit` and `reset` buttons. Furthermore, valid input values should be shown in a dialog when the form is submitted.

```
<h:panelGroup id="dynaFormGroup">
  <p:messages id="messages" showSummary="true"/>

  <pe:dynaForm id="dynaForm"
    value="#{dynaFormController.model}" var="data">
    <pe:dynaFormControl type="input" for="txt">
      <p:inputText id="txt" value="#{data.value}"
      required="#{data.required}"/>
    </pe:dynaFormControl>
    <pe:dynaFormControl type="calendar" for="cal"
      styleClass="calendar">
      <p:calendar id="cal" value="#{data.value}"
        required="#{data.required}" showOn="button"/>
    </pe:dynaFormControl>
```

```
      <pe:dynaFormControl type="select" for="sel"
        styleClass="select">
        <p:selectOneMenu id="sel" value="#{data.value}"
          required="#{data.required}">
          <f:selectItems value="#{dynaFormController.languages}"/>
        </p:selectOneMenu>
      </pe:dynaFormControl>
      <pe:dynaFormControl type="textarea" for="tarea">
        <p:inputTextarea id="tarea" value="#{data.value}"
        required="#{data.required}" autoResize="false"/>
      </pe:dynaFormControl>
      <pe:dynaFormControl type="rating" for="rat">
        <p:rating id="rat" value="#{data.value}"
        required="#{data.required}"/>
      </pe:dynaFormControl>

      <f:facet name="buttonBar">
        <p:commandButton value="Submit"
          action="#{dynaFormController.submitForm}"
          process="dynaForm"
          update=":mainForm:dynaFormGroup :mainForm:inputValues"
          oncomplete="handleComplete(xhr, status, args)"/>
        <p:commandButton type="reset" value="Reset"
          style="margin-left: 5px;"/>
      </f:facet>
    </pe:dynaForm>
  </h:panelGroup>

  <p:dialog header="Input values" widgetVar="inputValuesWidget">
    <p:dataList id="inputValues"
      value="#{dynaFormController.bookProperties}"
      var="bookProperty" style="margin:10px;">
      <h:outputText value="#{bookProperty.name}:
      #{bookProperty.formattedValue}"
        style="margin-right: 10px;"/>
    </p:dataList>
  </p:dialog>

  <h:outputScript id="dynaFormScript" target="body">
    /* <![CDATA[ */
    function handleComplete(xhr, status, args) {
      if(args && args.isValid) {
```

```
        inputValuesWidget.show();
      } else {
        inputValuesWidget.hide();
      }
    }
    /* ]]> */
  </h:outputScript>
```

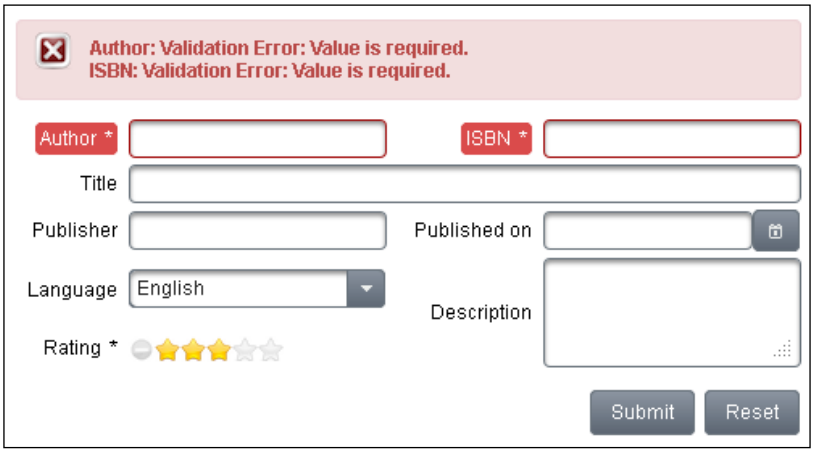
The `value` attribute of the `pe:dynaForm` component points to a model class, `DynaFormModel`. The model is created completely in Java—row by row and control by control.

```
@ManagedBean
@ViewScoped
public class DynaFormController implements Serializable {

  private static List<SelectItem> LANGUAGES =
    new ArrayList<SelectItem>();

  private DynaFormModel model;

  public DynaFormController() {
    model = new DynaFormModel();

    // add rows, labels and editable controls
    // set relationship between label and editable controls

    // 1. row
    DynaFormRow row = model.createRegularRow();

    DynaFormLabel label11 = row.addLabel("Author", 1, 1);
    DynaFormControl control12 = row.addControl(
      new BookProperty("Author", true), "input", 1, 1);
    label11.setForControl(control12);

    DynaFormLabel label13 = row.addLabel("ISBN", 1, 1);
    DynaFormControl control14 = row.addControl(
      new BookProperty("ISBN", true), "input", 1, 1);
    label13.setForControl(control14);

    // 2. row
    row = model.createRegularRow();

    DynaFormLabel label21 = row.addLabel("Title", 1, 1);
    DynaFormControl control22 = row.addControl(
```

```
      new BookProperty("Title", false), "input", 3, 1);
    label21.setForControl(control22);

    // 3. row
    row = model.createRegularRow();

    DynaFormLabel label31 = row.addLabel("Publisher", 1, 1);
    DynaFormControl control32 = row.addControl(
      new BookProperty("Publisher", false), "input", 1, 1);
    label31.setForControl(control32);

    DynaFormLabel label33 = row.addLabel("Published on", 1, 1);
    DynaFormControl control34 = row.addControl(
      new BookProperty("Published on", false), "calendar", 1, 1);
    label33.setForControl(control34);

    // 4. row
    row = model.createRegularRow();

    DynaFormLabel label41 = row.addLabel("Language", 1, 1);
    DynaFormControl control42 = row.addControl(
      new BookProperty("Language", false), "select", 1, 1);
    label41.setForControl(control42);

    DynaFormLabel label43 = row.addLabel("Description", 1, 2);
    DynaFormControl control44 = row.addControl(
      new BookProperty("Description", false), "textarea", 1, 2);
    label43.setForControl(control44);

    // 5. row
    row = model.createRegularRow();

    DynaFormLabel label51 = row.addLabel("Rating", 1, 1);
    DynaFormControl control52 = row.addControl(
      new BookProperty("Rating", 3, true), "rating", 1, 1);
    label51.setForControl(control52);
  }

  public DynaFormModel getModel() {
    return model;
  }
```

```
    public List<BookProperty> getBookProperties() {
      if (model == null) {
        return null;
      }

      List<BookProperty> bookProperties = new
      ArrayList<BookProperty>();
      for (DynaFormControl dynaFormControl : model.getControls()) {
        bookProperties.add((BookProperty) dynaFormControl.getData());
      }

      return bookProperties;
    }

    public String submitForm() {
      FacesMessage.Severity sev =
      FacesContext.getCurrentInstance().getMaximumSeverity();
      boolean hasErrors = ((sev != null) &&
        (FacesMessage.SEVERITY_ERROR.compareTo(sev) >= 0));

      RequestContext requestContext =
      RequestContext.getCurrentInstance();
      requestContext.addCallbackParam("isValid", !hasErrors);

      return null;
    }

    public List<SelectItem> getLanguages() {
      if (LANGUAGES.isEmpty()) {
        LANGUAGES.add(new SelectItem("en", "English"));
        LANGUAGES.add(new SelectItem("de", "German"));
        LANGUAGES.add(new SelectItem("ru", "Russian"));
        LANGUAGES.add(new SelectItem("tr", "Turkish"));
      }

      return LANGUAGES;
    }
  }
```

What does the form look like from the UI point of view? The following screenshot shows the screen obtained after a failed form validation:



The following screenshot demonstrates valid input fields that are shown in the `p:dialog` component after a successful form submission:

How it works...

The main steps to build a dynamic form are as follows:

1. Create a model instance:

```
DynaFormModel model = new DynaFormModel();
```

2. Add a row to a regular grid:

```
DynaFormRow row = model.createRegularRow();
```

3. Add a label:

```
DynaFormLabel label = row.addLabel(value, colspan, rowspan);
```

4. Add an editable control:

```
DynaFormControl control = row.addControl(data, type, colspan,
rowspan);
```

5. Set the relationship between the label and control (optional):

```
label.setForControl(control);
```

Repeat steps 2 to 5 as many times as needed.

The main tag is `pe:dynaForm`. The child tag `pe:dynaFormControl` matches controls created in Java by the `type` attribute. This is usually a *one to many* relation. As we can also see, one of the important features is built-in support for labels. `DynaForm` renders labels automatically—no need to include `p:outputLabel` or `h:outputLabel`.

`h:outputText` can also be used as a normal tag inside `pe:dynaFormControl` with a proper value for `type`.

There's more...

Besides the `buttonBar` facet, `DynaForm` also supports the following facets to include content pieces before or after the main container in regular or extended areas:

- `headerRegular`
- `footerRegular`
- `headerExtended`
- `footerExtended`

Additional highlights:

- Expandable extended view area (grid)
- Open or close state saving
- The `autoSubmit` feature
- Widget's client-side API

It is also possible to build dynamic forms with labels above fields and various elements such as PrimeFaces' `p:separator` component.

Since a dynamic form is created in Java, the XHTML code with `pe:dynaForm` can stay the same—changing the Java model leads to a different appearance of the dynamic form. This is probably the most interesting feature of this component.

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/chapter12/dynaForm.jsf`.

Timeline – displaying events in chronological order

The `Timeline` component visualizes chronological data. Data is organized as events that have either a single date or start and end dates (range). The component allows free movement and zooming in the visible timeline area by dragging and scrolling the mouse wheel. The time scale on the axis is adjusted automatically and supports scales ranging from milliseconds to years. It is highly customizable, and the fastest timeline implementation so far. Dealing with hundreds of events is hence not a problem.

In this recipe, we will demonstrate the first few steps when working with the `Timeline` component. The features this component has are almost countless, so we will learn only the basic usage. For this purpose, we will build a timeline that will show a release history of Primefaces Extensions.

How to do it...

We will create the timeline structure in Java and bind it to the `pe:timeline` tag via the `value` attribute. Besides `value`, the most important attributes of the `pe:timeline` tag are `axisPosition`, `eventStyle`, and `showNavigation`. In the developed example, the first two attributes can be manipulated by a `p:selectOneButton` component.

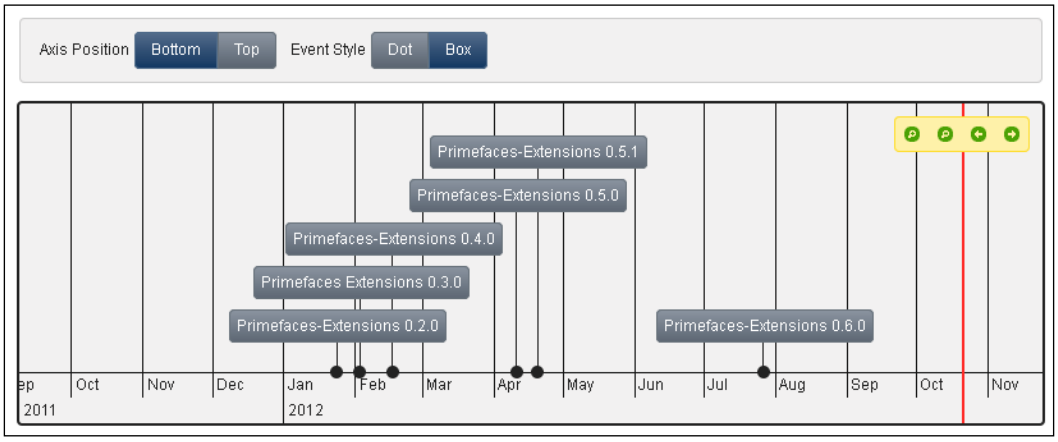
```
<p:fieldset>
  <h:panelGrid columns="4">
    <h:outputLabel value="Axis Position"/>
    <p:selectOneButton id="axisPosition"
      value="#{timelineController.axisPosition}">
      <f:selectItem itemLabel="Bottom" itemValue="bottom"/>
       <f:selectItem itemLabel="Top" itemValue="top"/>
      <p:ajax event="change" process="@this" update="timeline"/>
    </p:selectOneButton>

    <h:outputLabel value="Event Style"/>
    <p:selectOneButton id="eventStyle"
      value="#{timelineController.eventStyle}">
      <f:selectItem itemLabel="Dot" itemValue="dot"/>
      <f:selectItem itemLabel="Box" itemValue="box"/>
      <p:ajax event="change" process="@this" update="timeline"/>
    </p:selectOneButton>
  </h:panelGrid>
</p:fieldset>

<p:spacer width="100%" height="10"/>

<pe:timeline id="timeline" value="#{timelineController.timelines}"
    axisPosition="#{timelineController.axisPosition}"
    eventStyle="#{timelineController.eventStyle}"
    showNavigation="true"
    height="250px" width="800px">
</pe:timeline>
```

The `value` attribute points to a list of instances implementing the `org.primefaces.extensions.model.timeline.Timeline` interface. That means we can create as many timelines as we want. They will all be displayed one upon the other. In the example, we will only create a single timeline.

```
@ManagedBean
@ViewScoped
public class TimelineController implements Serializable {
```

```
  private List<Timeline> timelines;

  private String eventStyle = "dot";
  private String axisPosition = "bottom";

  @PostConstruct
  public void initialize() {
    timelines = new ArrayList<Timeline>();

    Calendar cal = Calendar.getInstance();
    Timeline timeline = new DefaultTimeLine("prh", "Primefaces
    Release History");
    cal.set(2011, Calendar.MARCH, 10);
    timeline.addEvent(new DefaultTimelineEvent
    ("Primefaces-Extensions 0.1", cal.getTime()));
    cal.set(2012, Calendar.JANUARY, 23);
    timeline.addEvent(new DefaultTimelineEvent
    ("Primefaces-Extensions 0.2.0", cal.getTime()));
    cal.set(2012, Calendar.FEBRUARY, 2);
    timeline.addEvent(new DefaultTimelineEvent("Primefaces
    Extensions 0.3.0", cal.getTime()));
    cal.set(2012, Calendar.FEBRUARY, 16);
    timeline.addEvent(new DefaultTimelineEvent
    ("Primefaces-Extensions 0.4.0", cal.getTime()));
    cal.set(2012, Calendar.APRIL, 10);
    timeline.addEvent(new DefaultTimelineEvent
    ("Primefaces-Extensions 0.5.0", cal.getTime()));
    cal.set(2012, Calendar.APRIL, 19);
    timeline.addEvent(new DefaultTimelineEvent
    ("Primefaces-Extensions 0.5.1", cal.getTime()));
    cal.set(2012, Calendar.JULY, 26);
    timeline.addEvent(new DefaultTimelineEvent
    ("Primefaces-Extensions 0.6.0", cal.getTime()));
    timelines.add(timeline);
  }

  public List<Timeline> getTimelines() {
    return timelines;
  }

  // getter / setter
  ...
}
```

The following screenshot shows the result for the preceding code snippet. The navigation bar is accessible at the top-right corner of the timeline. It allows scrolling and zooming along the timeline. An alternative way of scrolling is to drag the timeline using the mouse. The mouse's scroll wheel can be used for zooming in and zooming out too.



How it works...

Timelines can be created in Java by using the `DefaultTimeLine` class, which expects a list of events created by the `DefaultTimelineEvent` class. Each event consists of the title, start or end date, and style class to be applied to the event box. Events have to be added by the `addEvent` method.

The tag attribute `axisPosition` specifies the position of the date axis. Possible values are `top` and `bottom`. The default value is `bottom`. The tag attribute `eventStyle` specifies the display style for events. Possible values are `box` and `dot`. The default value is `box`. The preceding screenshot shows the `box` style. The Boolean flag `showNavigation` controls the visibility of the navigation controls at the top-right corner. Navigation controls are not displayed by default.

There's more...

Every event can be exposed via the `var` attribute. This gives an excellent opportunity to customize the event's content. The content to be shown for every event can be freely defined within the `pe:timeline` tag. Furthermore, there is a client behavior event, `eventSelect`, which is fired after a click on any event. In this way we can show details of the currently selected event, for example, in `p:dialog`. The following code snippet sketches this idea:

```
<pe:timeline id="timeline" value="#{customEventsController.timelines}"
  var="event" height="250px" width="100%">
  <p:ajax event="eventSelect"
    listener="#{customEventsController.onEventSelect}"
    update="viewDialog" oncomplete="viewDlg.show()"/>
  <h:panelGroup layout="block">
    <h:outputText value="#{event.startDate}">
      <f:convertDateTime pattern="dd MMM yyyy"/>
    </h:outputText>
    <br/>
    <h:outputText value="#{event.title}"/>
  </h:panelGroup>
</pe:timeline>

<p:dialog id="viewDialog" widgetVar="viewDlg" header="Event Details">
  <!-- show event details -->
  ...
</p:dialog>
```

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/chapter12/timeline.jsf`.