Exercise 1: NumericTextField

This exercise's purpose to demonstrate how to use connectors. The task is to create a TextField that only allows the user to enter Integer values.

Things you'll need to do

1. Create a new Vaadin 7 project, call it NumericTextField
2. Create a new Vaadin widget using the Eclipse plugin (right click on project, New > Other > Vaadin Widget, select **Connector only** template, call your widget "NumericTextField" (For more detailed instructions, see Appendix 1)
3. Enable SuperDevMode (optional, see Appendix 2)
4. Have the server-side component extend `com.vaadin.ui.TextField`
5. Your connector should extend `TextFieldConnector`
6. createWidget should return a `VTextField`
7. Register a `KeyDownHandler` to the widget using `getWidget().addKeyDownHandler(...)`. This listener will be called each time a user presses a key while the focus is on the widget. Check which key the user pressed, if the key code of the pressed key is not one of the number keys, then prevent the widget from processing the event. This is done by calling `event.preventDefault()`

Hint: You need to compare key code (accepted vs entered). The key code for the key for "1" is *not* 1, to get the key code, use

```
      int keyCodeForOne = (int) '1';
```

Bonus 1: User should be able to press delete, backspace and the left/right arrow keys.

Bonus 2: On the server side, override the protected setValue method and validate, that the value is a valid integer.

Disclaimer: In a real world scenario, we shouldn't implement a NumericTextField in this way, because a NumericTextField should handle values of type Integer, while TextField handles values of type String.

Book of Vaadin
16.4. Integrating the two sides with a connector

Exercise 2: CalendarPicker

The goal of this exercise is to practice the usage of RPC and shared state. In this exercise, we will take an existing GWT widget and make it compatible with server-side Vaadin applications.

Things you'll need to do

1. Create a new Vaadin 7 project called CalendarPicker
2. Create a new Vaadin 7 widget using the **full fledged** template, call your widget CalendarPicker (See Appendix 1)
3. Enable SuperDevMode (optional, see Appendix 2)
4. Remove the client RPC interface
5. In the server RPC interface, define one method, setDate(Date date)
6. The state object should have a field for the selected date
7. Set a default value for the date
8. Implement the server RPC in the server-side component. The setDate method should update the selected date in state
9. The connector should create a DatePicker widget
10. The connector should register itself as a ValueChangeHandler<Date> to the widget, in order to listen to the selected date
11. State changes should be passed to the widget
12. **NOTE!** Your component will look a bit funky, as it doesn't have the DatePicker widget's theme. To include a widget specific theme,
    a. create a folder called "public" into the same package where your server-side component class relies.
    b. Create a folder called "calendarpicker" into the public folder.
    c. Copy the styles.css file from the exercise.zip file to the calendarpicker folder
    d. Update your CalendarpickerWidgetset.gwt.xml file to include the style, the file should look like

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Google Inc.//DTD Google Web Toolkit 1.7.0//
EN" "http://google-web-toolkit.googlecode.com/svn/tags/1.7.0/distro-
source/core/src/gwt-module.dtd">
<module>
     <inherits name="com.vaadin.DefaultWidgetSet" />
     <stylesheet src="calendarpicker/styles.css"/>
</module>
```

Don't forget to recompile your widgetset!

Bonus: Now the CalendarPicker extends AbstractComponent, but since the component is used for selecting a value, a more appropriate superclass would be AbstractField. If your component is a Field, then it can be used in, for example, a form.

Your bonus task is to make your CalendarPicker component a Field. A property of a Field is that it manages its own value through the set/getValue.

Things you'll need to do

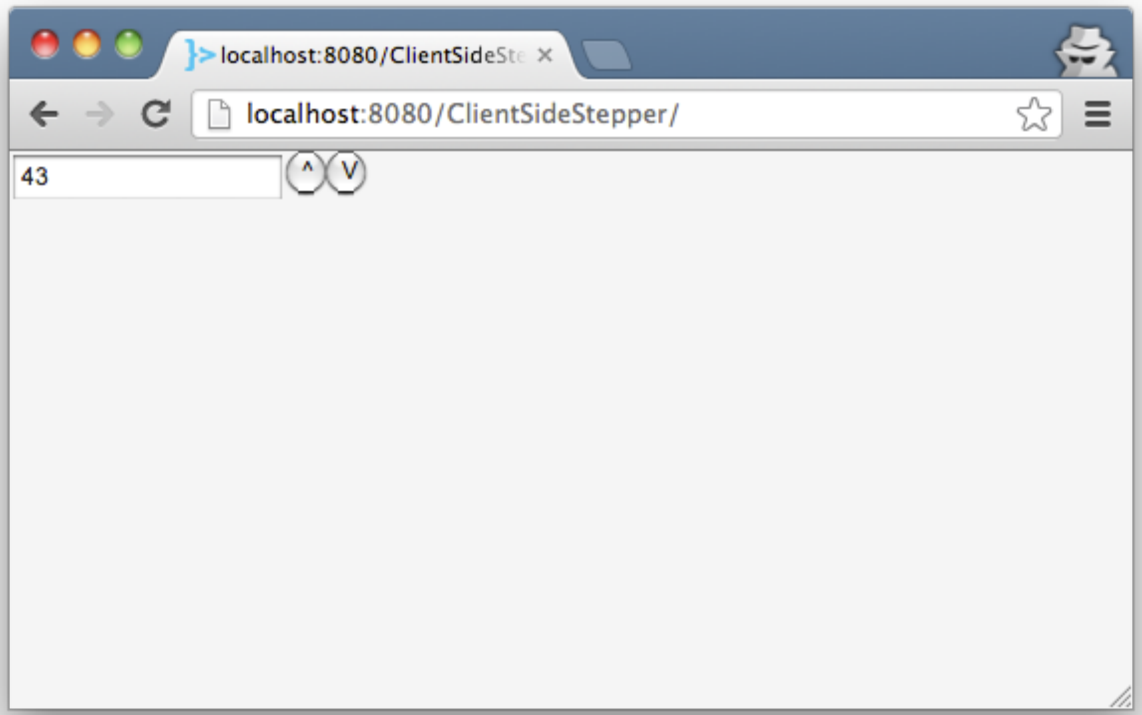
1. CalendarPicker extend `AbstractField<Date>`

2. Change the RPC's setDate method so that instead of setting a value to the state object, it calls the field's setValue with the new date, if and only if 1) the component is not in readOnly state (`isReadOnly()`) and the value has actually changed
3. The connector should extend `AbstractFieldConnector`
4. The state object should extend `AbstractFieldState`

Book of Vaadin
16.5. Shared State

Exercise 3: Stepper (client-side implementation)

The purpose of this exercise is to practice creating new widgets on the client side. The component we are about to create is a numeric textfield, except that it also contains two buttons for stepping the value up and down.


localhost:8080/ClientSideSte
localhost:8080/ClientSideStepper/
43
^
V

The behavior of this component should be exactly the same as in the first widget exercise, except that this also contains the two buttons. When I click on the first button, the text field's value should be increased by one. When clicking on the second button, the value should be decreased by one.

The widget itself will be created as a composition of existing GWT widgets. It it very much like using Vaadin components to build layouts and/or CustomComponents

Things you'll need to do

1. Create new project and new widget called Stepper
2. Server-side widget should extend `AbstractField<Integer>`
3. State should extend `AbstractFieldState` and contain one field: `Integer value`
4. Connector should extend AbstractFieldConnector
5. The StepperWidget should be created as a GWT composition component
    a. Extends HorizontalPanel
    b. Implements ClickHandler and HasValue<Integer>
    c. Has three components in the panel, an IntegerBox and two buttons
    d. Proxies set/getValue methods to the IntegerBox
    e. Add a KeyDownHandler to the IntegerBox, similar to the one in the first widget exercise
    f. Implement ClickHandler so that the value in the IntegerBox is increased/decreased
6. Connector should pass state changes to the widget

```
public class StepperWidget extends HorizontalPanel implements ClickHandler,
            HasValue<Integer> {

      ....


      @Override
      public Integer getValue() {
            return integerBox.getValue();
      }

      @Override
      public void setValue(Integer newValue) {
            integerBox.setValue(newValue);
      }

      @Override
      public void setValue(Integer value, boolean fireEvents) {
            integerBox.setValue(value, fireEvents);
      }

      @Override
      public HandlerRegistration addValueChangeHandler(
                  ValueChangeHandler<Integer> handler) {
            return integerBox.addValueChangeHandler(handler);
      }
}
```

Bonus: Implement functionality to enable stepping the integer values using mouse wheel.

Exercise 4: Stepper (server-side implementation)

Not everything needs to be done on the client-side. The goal of this exercise is to create the exact same component as in the previous example, except that this time it will be created completely on the server-side with the help of the NumericTextField created in the first exercise.

Things you'll need to do
1. Create a new project, DO NOT create a new widget
2. Package the NumericTextField from the first exercise as an add-on and import it to this project. This is done simply by copying the resulting jar-file to WebContent/WEB-INF/lib
3. Compile your custom widgetset at this point, as you have a new add-on in your project
4. Create a new server-side class, called Stepper
5. Stepper should extends `CustomField<Integer>`
6. Implement the layout and functionality of the stepper component just like you would do in a normal Vaadin application

The benefit of this approach is that we are not introducing any new client-side code. With introducing new client-side code lies a risk of introducing weird, client-side, browser-specific bugs. Creating the component as a server-side composition allows us to use client-side code that has already been tested and verified. This approach comes with a price: we cannot implement the mouse scroll feature of the previous exercise's bonus task without making modifications to the client side :(

Book of Vaadin
5.25. Composite Fields with CustomField

Exercise 5: Component extensions - CssInject

The purpose of this exercise is to practice using the Extension API. The goal is to enable the injection of CSS rules for any widget component directly from server side code. This happens by creating an extension, which transfers CSS rules (property name & value) from server side with shared state to the client side connector. In the connector the rules are simply set into the extended component's widget's element.

Things you'll need to do

1. Create new project called CSSInject
2. Create a new Vaadin widget. You only need a Connector and SharedState for this exercise
3. Server-side component should extend `AbstractExtension` and have API for adding and removing style rules
4. State should extend `AbstractComponentState` and contain one field for storing the CSS properties: `Map<String, String>`
5. Connector should extend `AbstractExtensionConnector`
   - 5.1. you can to get the extended component's element by calling `ComponentConnector.getWidget().getElement()`
   - 5.2. you can access the element's style properties from `Element.getStyle()`
6. Apply the given styles to the element
7. Don't forget to remove from the element CSS attributes that have been removed from the state!

Book of Vaadin
16.7. Component and UI Extensions

Exercise 6: Application level extension - Refresh

The idea of this exercise is to continue try out creating another extension - the Refresher. The refresher has one single purpose - it is an application level that will **poll** the server on a given interval.

For this extension you'll need a SharedState and a ServerRPC. The state should contain two properties: interval for which the polling is done (defined in milliseconds) and is the extension enabled or disabled (hint: maybe the superclass of your state contains something that you need, such as "enabled" attribute ;)).

Your RPC should contain one method, refresh() which will be called every time the timer loops. Your server should react to this RPC call by sending out an event notifying listeners, that a poll has occurred.

Things you'll need to do
1. Create a new widget
2. Your connector should extend `AbstractExtensionConnector`.
3. Your ServerRPC should have a method refresh()
4. Create a GWT timer that will do refresh calls to the server after the given intervals

```
private class Poller extends Timer {
      @Override
      public void run() {
            // TODO - RPC call
      }
}
```

5. Your onStateChange method should cancel the poller (`poller.cancel();`) and restart it with the given interval.

   ```
   poller.scheduleRepeating(getState().interval);
   ```

6. Your server-side extension API should have methods for getting and setting the interval value and the extensions enabled-mode.
7. Your server-side extension API should have methods for registering poll event listeners

   ```
   public static interface PollListener {
      public void poll();
   }
   ```

8. Your server-side extension should send events when the client notifies the server of a refresh.

Bonus: Your current connector schedules repetitions for an indefinite amount of times. Your task is to make this limit variable by calling setLimit(Int) from a Refresher object. This limit should be stored in the RefresherState and considered by your connector in onStateChanged().

Book of Vaadin - 16.7. Component and UI Extensions

Exercise 7: JavaScript widget exercise

The idea of this exercise is to learn how to utilise existing JavaScript widgets to build Vaadin components. There are several ways how a JavaScript widget could be integrated with Vaadin. In this exercise we do the integration by using the JavaScript component API provided by the Vaadin framework.

Your task is to create a timer component that has three public methods: start, stop and reset. In addition, when the timer reaches 10 seconds, a notification "Time is up" should be shown. The code for the client-side JavaScript widget (the timer) is provided for you.

Things you will need to do:
Initial setup:

1. Create a new Vaadin project
2. Create a new class, `TimerComponent` that extends `AbstractJavaScriptComponent`
3. Copy `timer.js` file from the /Exercises folder in the training material to the same package with your component. This is the widget being integrated.
4. Create `connector.js` file for your component's connector. The connector is implemented using JavaScript and should be placed in the same folder with your component and the `timer.js`. In that file define a stub connector. (See the Book of Vaadin for details)
5. The connector should initialise the client side widget in constructor.
6. Annotate your server-side component to load both JavaScript files. (`@JavaScript({"connector.js","timer.js"})`)

Shared state:

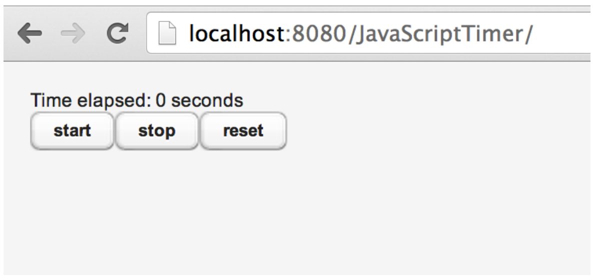
7. Create a shared state for the component. The state should extend from JavaScriptComponentState and have only one field: `boolean started`
8. Override `getState` in the `TimerComponent` to return your shared state
9. Add methods for starting and stopping the timer to your server-side component. Use the shared state to communicate the current status to the client
10. In your JavaScript connector add code that reacts to the state changes and starts and stops the timer accordingly

Call JavaScript from Server:

11. Add method for resetting the timer in your server-side component. Use `callFunction` to invoke the reset function on client side connector (see AbstractJavaScriptComponent api for details)
12. Add corresponding method to client-side connector that will call reset method of your JavaScript widget.

Call Server from JavaScript:

13. Register JavaScriptFunction in your server side component that shows the notification "Time is up". This method will be called by your connector.
14. The connector should invoke the function registered on the server side component when the time (10 seconds) is up.
15. Build UI that contains the timer component and three buttons: Start, Stop and Reset that can be used to interact with the timer

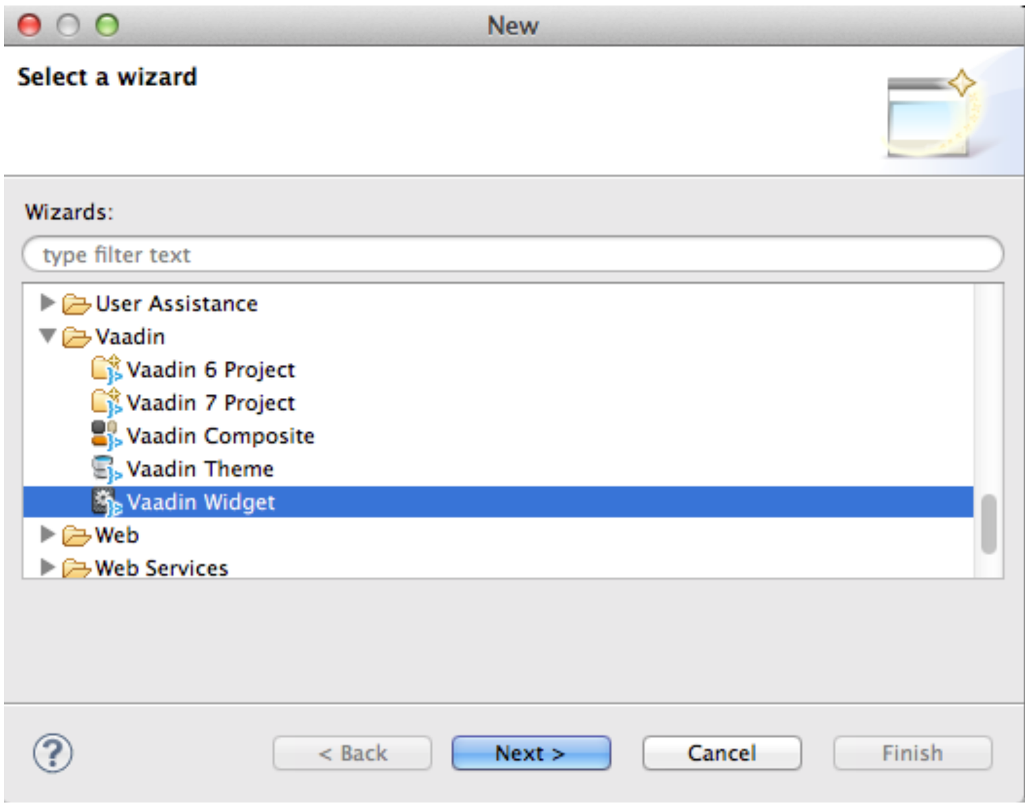


Book of Vaadin
16.12. Integrating JavaScript Components and Extensions

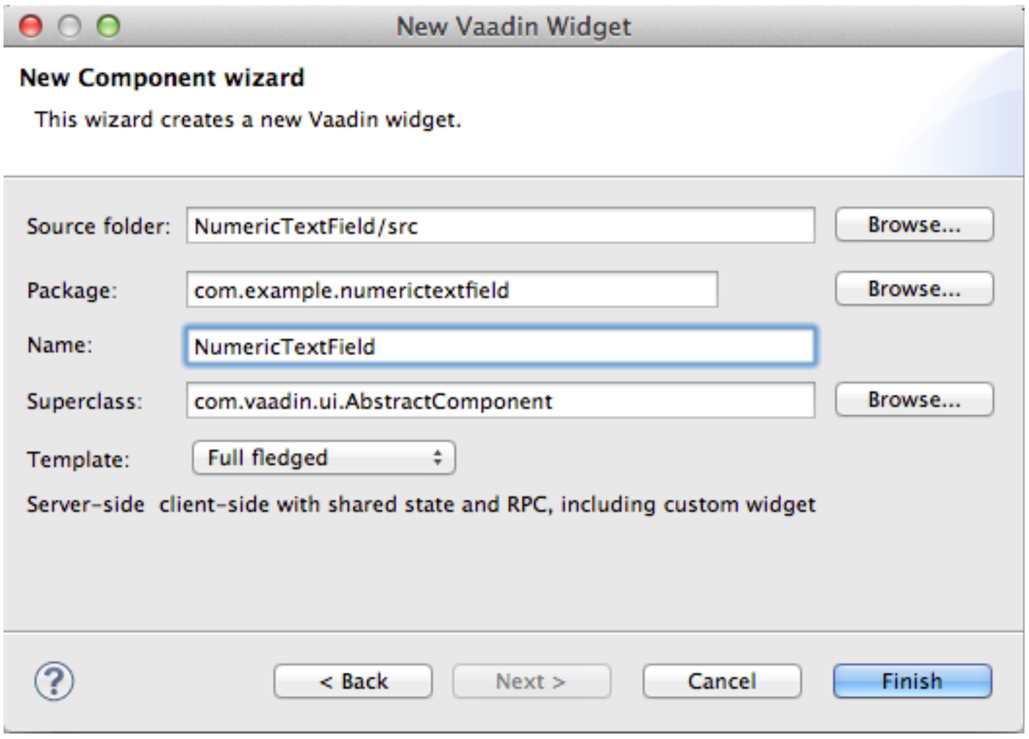
Bonus: Now there's an arbitrary time limit of 10 seconds, after which a notification is shown to the user. Your bonus task is to make it possible to set the time limit freely on the server-side component. You'll need to add appropriate field to your state class and modify the way your connector updates client-side based on the state sent from the server.

Appendix 1: Creating a widget template

1. Create a new Vaadin project or use an existing one
2. Choose File > New > Other > Vaadin > Vaadin Widget
3. Click Next



4. Give name for the created widget
5. Choose “Full fledged” or “Connector only” template
6. Click Finish


New Vaadin Widget

New Component wizard

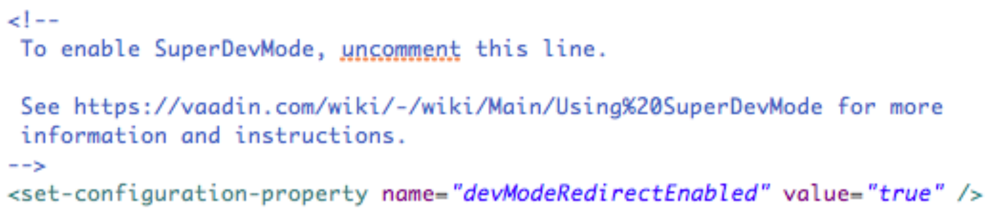
This wizard creates a new Vaadin widget.

Source folder:	NumericTextField/src
Package:	com.example.numerictextfield
Name:	NumericTextField
Superclass:	com.vaadin.ui.AbstractComponent
Template:	Full fledged

Server-side client-side with shared state and RPC, including custom widget

Appendix 2: Enabling SuperDevMode

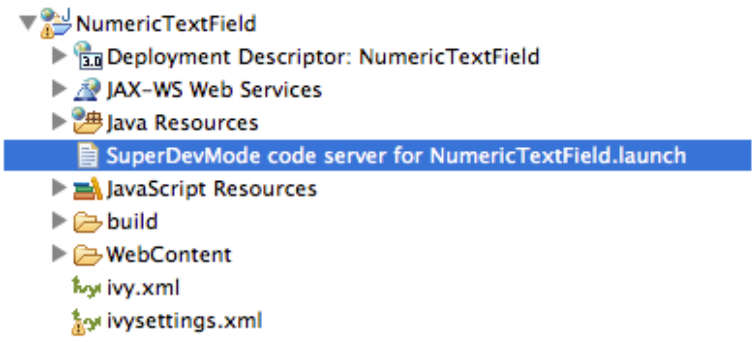
1. Make sure you have already created the widget template (see Appendix 1)
2. Edit your widget set definition file (<widgetname>.gwt.xml)
3. Uncomment devModeRedirectEnabled configuration property



```
<!--
 To enable SuperDevMode, uncomment this line.

 See https://vaadin.com/wiki/-/wiki/Main/Using%20SuperDevMode for more
 information and instructions.
-->
<set-configuration-property name="devModeRedirectEnabled" value="true" />
```

4. Compile the widget set using the compile widget set button in Eclipse
5. Open project properties
6. Choose section Vaadin
7. Click “Create SuperDevMode launch”. This will create a .launch file to your project root
8. Click OK
9. Start your server (Tomcat)
10. Right click on the launch file and select ***Run As..*** (not Debug as…) to launch the SuperDevMode. This will perform an initial compilation of your widget set / module
11. To start using the SuperDevMode, visit your application URL and add ?superdevmode at the end. Alternatively you can launch the super dev mode from the debug window (?debug)
12. After these steps you will see a notification that the widget set is being recompiled, which should only take a few seconds. The page will reload and you are running in SuperDevMode. You can now make changes to your widget code and afterwards press refresh in the browser and immediately start using the new code



```
▼ NumericTextField
  ▶ Deployment Descriptor: NumericTextField
  ▶ JAX-WS Web Services
  ▶ Java Resources
    SuperDevMode code server for NumericTextField.launch
  ▶ JavaScript Resources
  ▶ build
  ▶ WebContent
    ivy.xml
    ivysettings.xml
```

NOTE: To be able to debug Java code in Chrome, open the Chrome Inspector (right click > Inspect Element), click the settings icon in the lower corner of the window and check "Scripts > Enable JS source maps". Refresh the page with the inspector open and you will be able to access java code and set breakpoints in the sources tab.