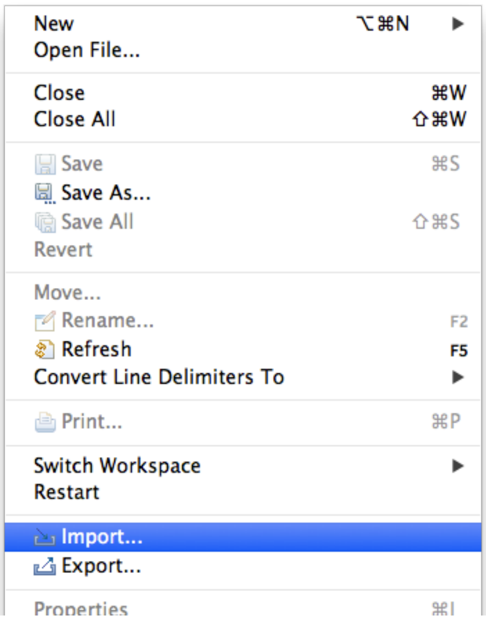
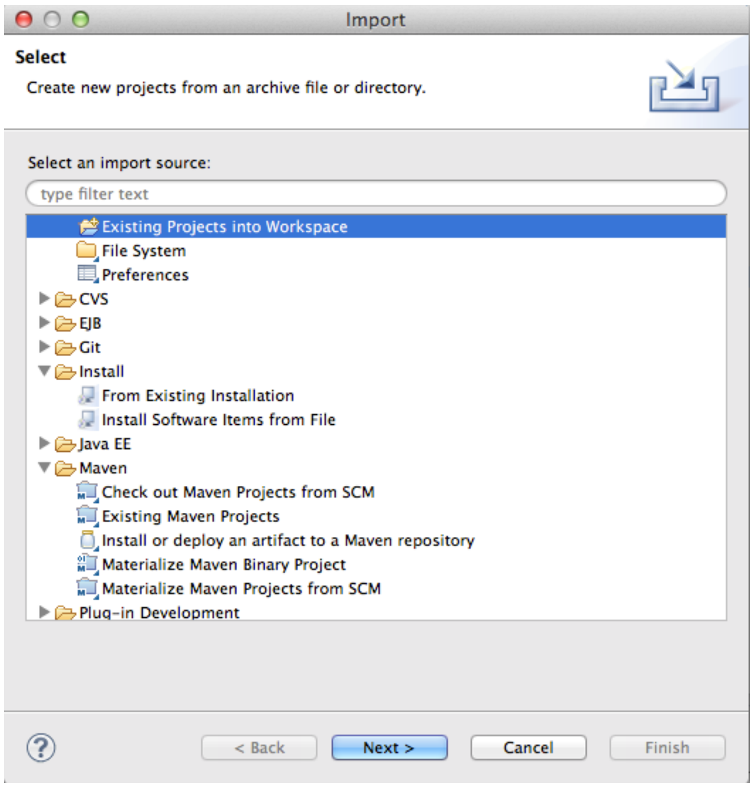
Instructions

Setting up the environment

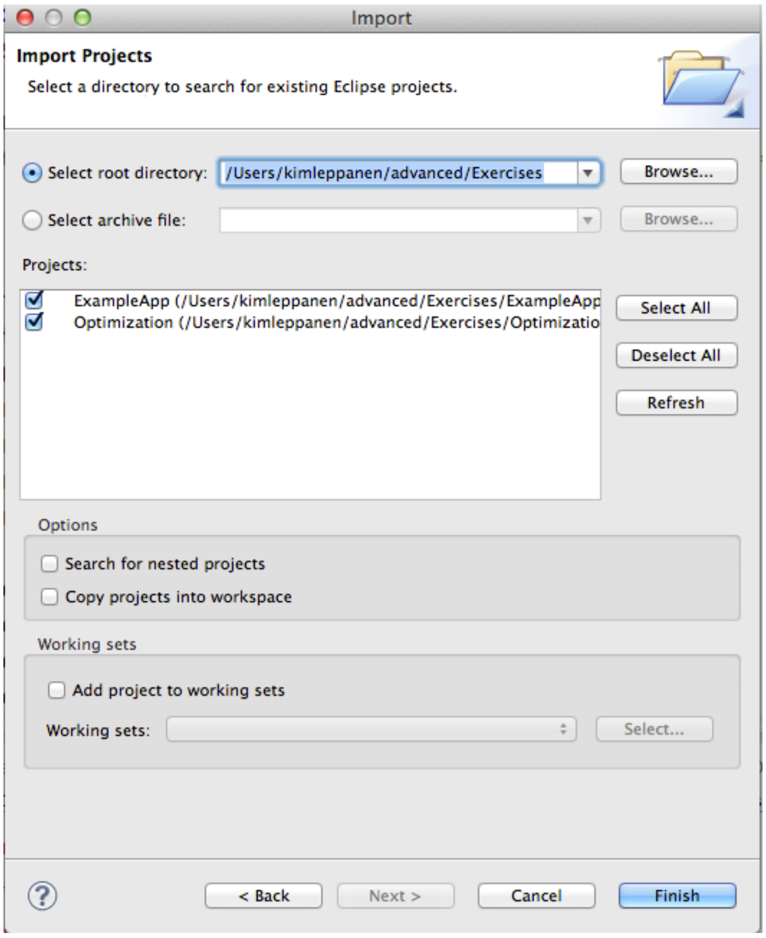
Start by downloading the advanced.zip file and extract it to location of your choice.

Open Eclipse, choose File > Import and choose “Existing project into Workspace”




Import
Select
Create new projects from an archive file or directory.
Select an import source:
type filter text
Existing Projects into Workspace
File System
Preferences
CVS
EJB
Git
Install
From Existing Installation
Install Software Items from File
Java EE
Maven
Check out Maven Projects from SCM
Existing Maven Projects
Install or deploy an artifact to a Maven repository
Materialize Maven Binary Project
Materialize Maven Projects from SCM
Plug-in Development
< Back
Next >
Cancel
Finish

Browse to the folder where you extracted the zip, and choose the Exercises folder. Select the ExampleApp and Optimization projects and click on finish.


Import

Import Projects

Select a directory to search for existing Eclipse projects.

(●) Select root directory: /Users/kimleppanen/advanced/Exercises [Browse...]

( ) Select archive file: [Browse...]

Projects:

- [✓] ExampleApp (/Users/kimleppanen/advanced/Exercises/ExampleApp
- [✓] Optimization (/Users/kimleppanen/advanced/Exercises/Optimizatio

[Select All] [Deselect All] [Refresh]

Options

- [ ] Search for nested projects
- [ ] Copy projects into workspace

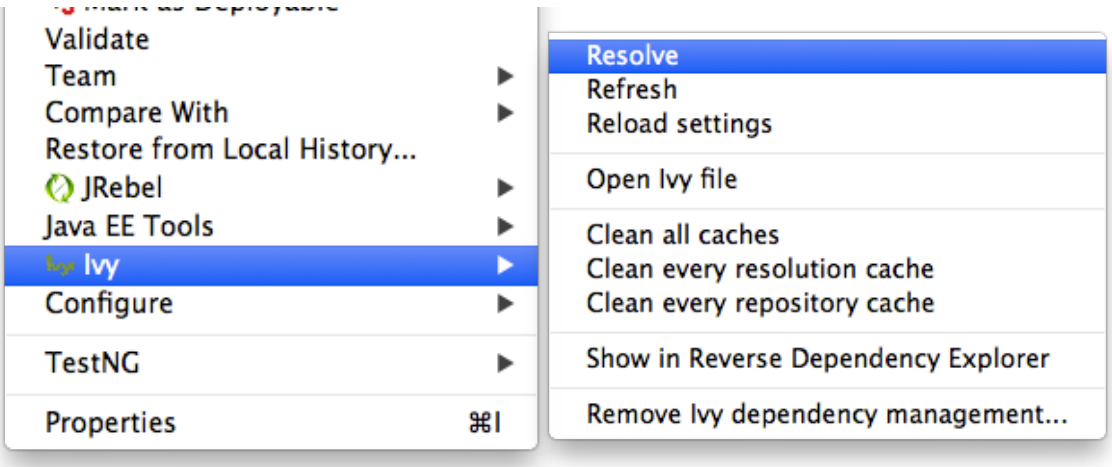
Working sets

- [ ] Add project to working sets

Working sets: [Select...]

[< Back] [Next >] [Cancel] [Finish]

Right-click on the imported projects and choose Ivy > Resolve.


Validate
Team
Compare With
Restore from Local History...
JRebel
Java EE Tools
Ivy
Configure
TestNG
Properties ⌘I
Resolve
Refresh
Reload settings
Open Ivy file
Clean all caches
Clean every resolution cache
Clean every repository cache
Show in Reverse Dependency Explorer
Remove Ivy dependency management...

You should now be all set up. Try deploying the project to Tomcat and browse to http://localhost:8080/ExampleApp/

Background

Our client wants to create a web application for monitoring their expenses on a department basis. In the first sprint the application will contain three views: A Dashboard that is a simple placeholder view, an Auditing view which contains a list of audit log messages and a Department view that is used to maintain all of the employees in a department.

This initial version of the application will only show the expenses and employees of the "Services" department. The customer has provided a stub implementation of their domain model and services.

The customer is not satisfied with the current structure of the application code. As a distinguished Vaadin expert, you have been hired to help with the project. Your task will be to implement the navigation for the project and restructure the application to be more maintainable. The stub project contains the initial implementation of the application excluding navigation.

Exercise 1: Navigation

The first exercise is to implement navigation between views. Create the base application so that it uses the Navigator API to handle the views and navigation between them.

Things you'll need to do

1. There are three views in the application, make the views Navigator compatible by implementing the *com.vaadin.navigator.View* -interface
2. In the UI class, create a navigator instance and register views to the navigator
3. Implement the click listeners in the UI class so that clicking on a button will transfer the user to the corresponding view

Bonus task: The URL to access the department view is http://localhost:8080/ExampleApp/#!department

It is possible give parameters to views which can then be handled in the view's enter method. Your task is to check, if an person ID is given as a view parameter, then the person with that ID should automatically be selected from the table for editing.

Book of Vaadin
Navigator: 383

Exercise 2: Presenters

The customer has requested you to refactor their application to implement the Model-View-Presenter pattern. You'll start this task by refactoring application logic into presenters and creating a layer of abstraction (using interfaces) between the presenter and the view implementation.

Improve code compartmentalizing and testability by separating UI code and -logic from each other. Views should still listen to events from the UI components but the presenter will decide what should be done when an event happens. Presenters are also the entry point to other subsystems like services.

Things you'll need to do:

1. Make views implement an interface that the presenters will use
2. Create a presenter for the Department view.
3. The view will create its own presenter upon construction and give a reference to itself in the presenters constructor.
4. The contents of the view should be refreshed upon navigation
5. Move the business logic from the view implementations to the presenters.
6. DepartmentPresenter should handle at least:
   - Fetching employees from service and updating the view with them upon navigation
   - Handle the saving of a *Person*-entity through the service
   - Handle the cancel event
   - Decide what to do when a *Person* is selected

When using the Vaadin Navigator API, one can easily separate the navigation event from the view creation event. If the view is given as a class-reference to the navigator, the view will be recreated upon each navigation. However, if an instance of the view is given to the navigator the state of the view is preserved. We should try to make our views so that either mode can be used. This is why we should only create the components of the view in the constructor and fill/refresh the data of the view on the *enter*-event.

Below is illustrated how the enter pattern can be used in the Auditing view.

```
public class AuditingPresenter {
	private AuditingView view;

	public void setView(AuditingView auditingView) {
		view = auditingView;
	}

	public void enter() {
		fetchInitialData();
	}

	private void fetchInitialData() {
		for (String message : AuditLogService.getAuditLogMessages()) {
			view.addAuditLog(message);
		}
	}

}

public class AuditingViewImpl extends VerticalLayout implements View,
		AuditingView {

	private AuditingPresenter presenter;
```

```
      private CssLayout messageLayout;

      public AuditingViewImpl() {
            ...

            presenter = new AuditingPresenter();
            presenter.setView(this);
      }

      @Override
      public void enter(ViewChangeEvent event) {
            messageLayout.removeAllComponents();
            presenter.enter();
      }

      @Override
      public void addAuditLog(final String message) {
            messageLayout.addComponent(new Label(message));
      }

}
```

Below is the View interface of the example solution. It should give you an idea of what needs to be done.

```
public interface DepartmentView {

      public void setEmployees(List<Person> employees);

      public void showEmployeeInForm(Person employee);

      public Person commitChanges();

      public void discardChanges();

      public void selectEmployee(Person employee);

      public void setDepartment(Department department);

}
```

Exercise 3: Push

Sometimes we have operations in our backend that takes a while to perform. If we perform there processes in the same thread as in which the HTTP request is processed, we will block any user interface interactions until the process is done. In most cases, we want the user to be able to continue using the user interface even though we are doing “heavy stuff” in the backend. Hence, the heavy process are typically performed in a separated thread.

If we modify the user interface from a thread that is outside the HTTP request, we won't see the changes in the browser until the user does something that triggers an HTTP request (so that we can send the changes of the UI in the response of that request). For the changes to be seen in the user interface immediately, we can use AJAX push, which will allows us to "push" the changes to the browser, even if we don't have an active HTTP request going on.

In this exercise, we will practice enabling push in a Vaadin application. The situation we are trying to simulate is that PersonService's getEmployees method takes multiple seconds before it returns its data. We want to push the user information to the UI when the process is done. On a high level, we need to enable push support in the Vaadin application, make the heavy process to be accessed in a separate thread and then handle possible concurrency issues that may occur if we modify the UI from a thread outside the HTTP request.

Things you'll need to do enable push support:

1. Add `<async-supported>true</async-supported>` to your web.xml under `<servlet>`
2. Adding push dependencies to ivy.xml
```
    <dependency org="com.vaadin" name="vaadin-push"
rev="&vaadin.version;" conf="default->default" />
```
3. Enabling push for your application by adding `@Push` annotation for your UI class

Things you'll need to do in your application:

1. We want to have a progress bar in the user interface showing that we are loading data. This component already exists within the DepartmentInfo class. To make it visible, we need to call setLoadingState on the class. It takes as an argument a float value between 0 and 1, indicating the loading progress. Given the value 0, the progress bar will become visible. Given the value 1, it will be hidden.

   Add the new method, `public void setDataLoadingState(float percentageProgress)`, to your DepartmentView. The implementation of this method should update the loading state in the DepartmentInfo class.

2. In your DepartmentPresenter class, create an ExecutorService that will provide you a safe way to get threads for executing long running processes in the backend.
```
    private final ExecutorService pool =
Executors.newFixedThreadPool(10);
```
3. Create an inner class that implements the `Runnable` interface, this runnable is responsible for fetching the persons from the backend and giving the details to the view once the data has been loaded. Here is a pseudo implementation of the runnable you need
```
    class UIUpdateRunnable implements Runnable {
            ProgressingFuture<List<Person>> employeesAsync;

            @Override
            public void run() {
                    // call backend's getEmployeeAsync() to fetch data. Method will return
                    // a future object
                    // while the future is not ready to be accessed (isDone returns false),
                    // wait for it to complete

                    // once the future is done, get the employee list from the future (call
                    // employeesAsync.get()) and set the employee list to the view

                    // update the data loading state to 1, so that the progress bar will be
                    // hidden
                }
        }
```

4. In the presenter's enter method, you need to
    4.1. clear the table from old data, you can do this my giving the view an empty list of employees
    4.2. trigger the UIUpdateRunnable by calling `pool.submit(new UIUpdateRunnable());`
    4.3. call the view's setDataLoadingState(0) in order to make the progress bar visible
5. Modify your view to handle UI changes from background threads in a thread safe way, this is done by using the `access()` method. Your backend threads make modifications to the view through three methods, selectEmployee(), setDataLoadingState() and setEmployees(). We need to secure those three methods. Below is the example implementation for setDataLoadingState.

```
@Override
public void setDataLoadingState(final float state) {
        UI ui = getUI();
        if (!isAttached() || ui.getSession().hasLock()) {
                departmentInfo.setLoadingState(state);
        } else {
                ui.access(new Runnable() {
                        @Override
                        public void run() {
                                departmentInfo.setLoadingState(state);
                        }
                });
        }
}
```

Bonus task: Make the progress bar show the actual loading progress. The ProgressingFuture will give you the loading state of the backend, now try to figure out what you need to do to get that visible in the UI.

Bonus task 2: Change the auditing view so, that whenever a new auditing message is added, the message will be automatically pushed to the browser. Note that the auditing service already has a notification mechanism which notifies listeners, when new messages are added.

Book of Vaadin
Server Push: 410

Exercise 4: Separate models from entities

In this exercise we will separate the UI from the back-end by using proxies and DTOs. This will provide resilience for the UI from potential back-end or entity model changes. Proxies will mediate data between the entities and the UI.

Things you'll need to do:
1. Create an EmployeeProxy which will function as the model for a Person entity
2. Create DepartmentDTO that will function as the view's model. It should contain two fields, a String field for the department's name and a list of employees.
3. In the presenter: Fill in the Employee proxies with data from the PersonService
4. In the presenter: Fill the DepartmentDTO with data from the AuthenticationService User object
5. In the presenter: Fill the DepartmentDTO with the employee proxies
6. Refactor the UI components to handle proxies instead of entities

Tip: Vaadin data binding API uses reflection to search for the getters and setters for fields. Hence if a field is named "firstName", Vaadin will look for a getter "getFirstName()" and a setter "setFirstName()". This correlates against the @PropertyId(<fieldName>) annotations in for example EmployeeEditor.

Bonus task: If you've implemented the first bonus task of exercise 3, you will notice that you have a small problem: the presenter is still giving information directly to the view - the loading status. This data should also be in the view, but then your problem is, how do you update the view without rerendering the department name and employees table?

You should set a model to the view once and just update the content of the model. The model is responsible for sending an event whenever its data changes - the view listens to these changes and updates the view accordingly. The easiest way to implement this observer-pattern, is to use PropertyChangeSupport and PropertyChangeListener. Your model should have an instance of PropertyChangeSupport and your view should implement PropertyChangeListener and register itself as a listener to the model. Here's a hint how it all works. This is code that goes into your model:

```
private final PropertyChangeSupport propertyChangeSupport = new
PropertyChangeSupport(this);
...
public void addPropertyChangeListener(PropertyChangeListener arg0) {
            propertyChangeSupport.addPropertyChangeListener(arg0);
      }
...

public void setEmployees(List<EmployeeProxy> employees) {
            List<EmployeeProxy> old = this.employees;
            this.employees = employees;
            propertyChangeSupport.firePropertyChange("employees", old,
employees);
      }
```

Optimization

Exercise 5: Optimization of layout rendering time

The key to this exercise is that not everything as it seems and to simplify, simplify, simplify.Your task is to optimize the rendering time of the given application. To see the rendering time of your application, add the parameter “?debug” to the URL and look at the bottom of the debug window for the phrase “Processing time was <x>ms for <y> characters of JSON”. X marks the spot, that is the rendering time.

Typical cause for slowly rendering layouts is the excessive use of components, deep layout hierarchies and slow layouts. With Vaadin 7, the layout performance between different layouts has decreased, but there are still differences. Play around and try to see what affects the rendering time.

Creating widgets

Exercise 6: 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"
3. Have the server-side component extend `com.vaadin.ui.TextField`
4. Your connector should extend `TextFieldConnector`
5. createWidget should return a `VTextField`
6. 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
Integrating the two sides with a connector: 465

Exercise 7: 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

3. In the server-side component, override getState to return a CalendarPickerState
4. Remove the client RPC interface
5. In the server RPC interface, define one method, setDate(long 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
<?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.

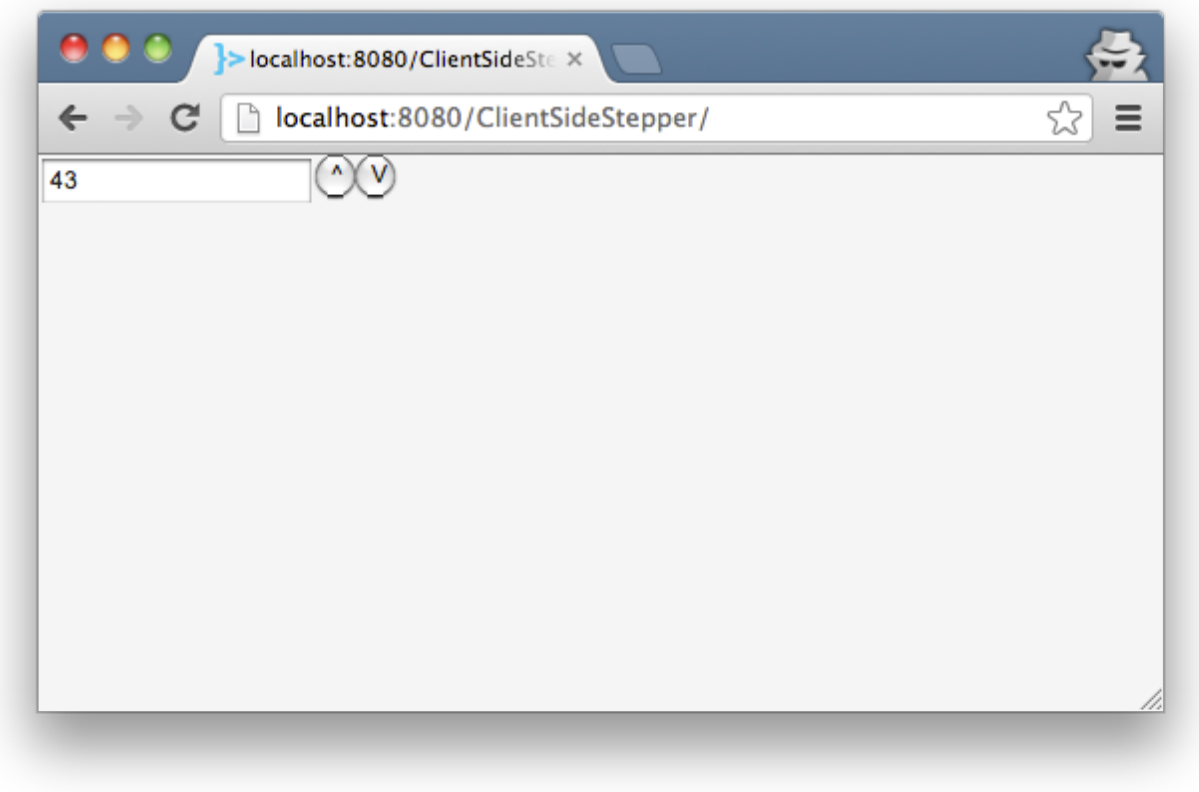
Your bonus task is to make your CalendarPicker component a Field. A property of a Field is that it manages its own value through the set/getValue.

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
Shared State: 466

Exercise 8: 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 9: 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
CustomField: 221