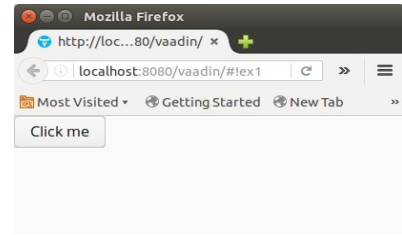
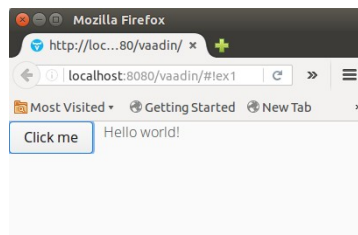
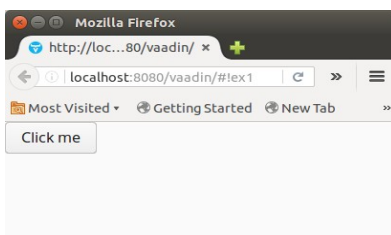


Basic Vaadin Exercises

For each of the exercises create a separate view class. Example, for exercise 1, you can create a class called “Ex1View”, the package is not relevant. The view class should implement *View* and extend a container component like for example *HorizontalLayout*. Of course, you are free to create a composite of different layouts (like adding other layouts to the top level layout). You will navigate between the different exercises by adding them to the navigator in the main UI class. E.g. `navigator.addView(“ex1”,Ex1View.class);` Navigation can then be done via de browser, eg. `http://localhost:8080/vaadin/#!ex1`

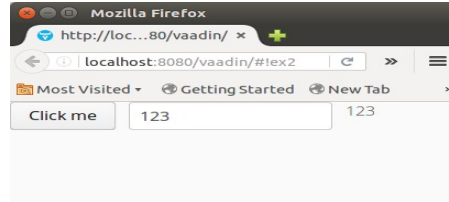
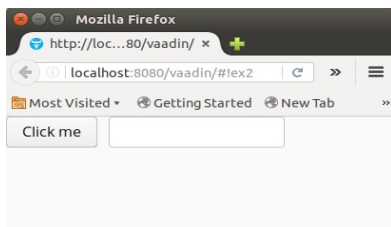
(1)

Add a single button on the view. If you click the button, the text “Hello world!” should appear on the right of the button (choose what you think would be an appropriate component for this). If you click it again, the text should disappear. Important: the text should appear on the right of the button as illustrated here:



(2)

Create a view with a single button and a text field. If you click the button, the text you've entered in the text field should appear on the right side of the button:



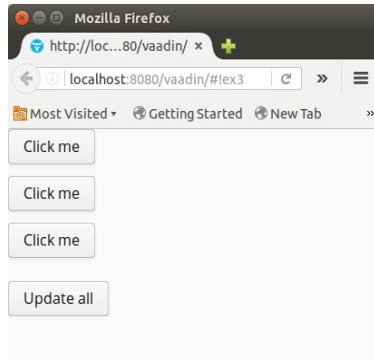
(a) Add a converter to the text field which will try to convert the entry to an Integer. An error message should appear next to the text field if the value is not an integer. Note; in case of Vaadin the message is rendered as an exclamation mark (!) which shows the message as tooltip when you hover over the image. In case of a conversion error, the contents of the text field must not be shown in the area on the right of the text field.



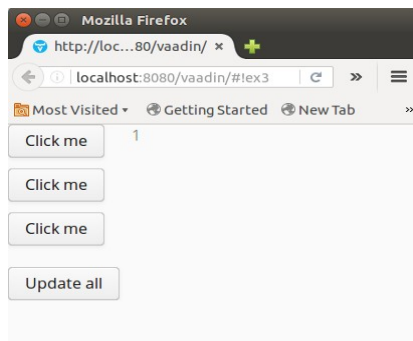
(b) At this time the conversion error only appears when clicking the button. Try to change your code so that the conversion error appears from the moment you jump out of the text field.

(3)

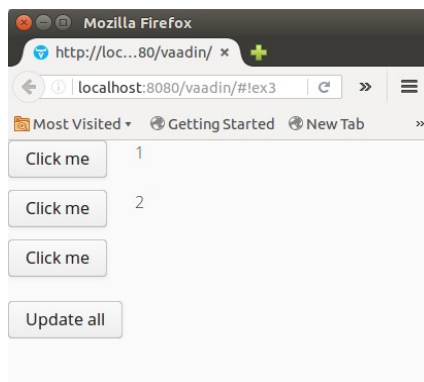
Here we are going to test the Ajax behavior. We'll be putting some buttons on this view, and each button triggers a partial update. Do you need to do something special for this? Create a single counter (private int counter =0;) create a method to increment the counter (let's say “*increment()*”) and one to display the counter (let's say: “*getCounter()*”). Next, create 3 text fields and 3 buttons. Also create one button with text “update everything” (we'll explain this in a minute). Button 1,2 and 3 call the “*increment()*” and each output field shows the current value of the counter by accessing the counter property.



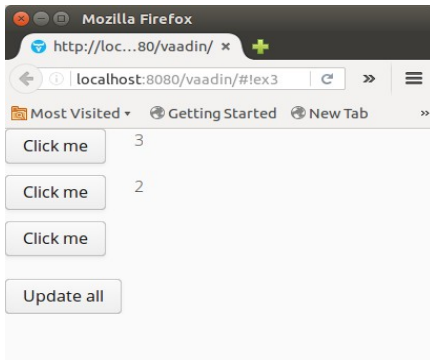
After pushing on button 1:



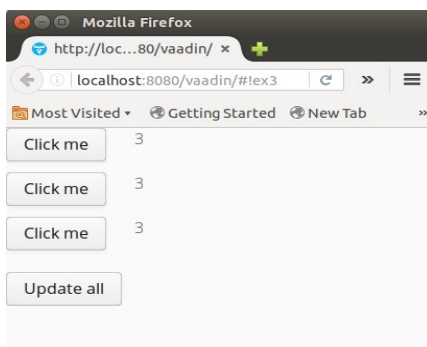
After pushing on button 2:



After pushing on button 1 again:

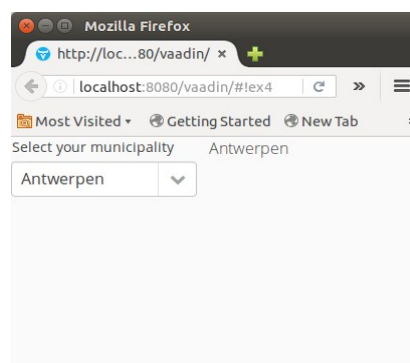
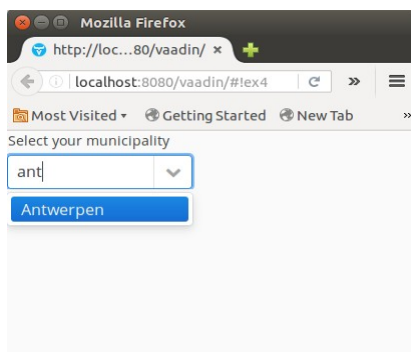


Finally, the “update everything” button will update all three output fields to the current value of the counter:



(4)

We're building an autocomplete. Use the vaadin combobox to achieve this. An example of this component can be found in the Vaadin sampler (<http://demo.vaadin.com/sampler/#ui/data-input/multiple-value/combo-box>). You can get a list of Flemish municipalities here: http://nl.wikipedia.org/wiki/Lijst_van_gemeenten_in_het_Vlaams_Gewest (use your magical text skills to get them in a comma separated list so you can add them in your source). The goal is that the municipalities are shown from the moment you start typing into the component. Once you selected a municipality, a label must appear on the right showing the selection;



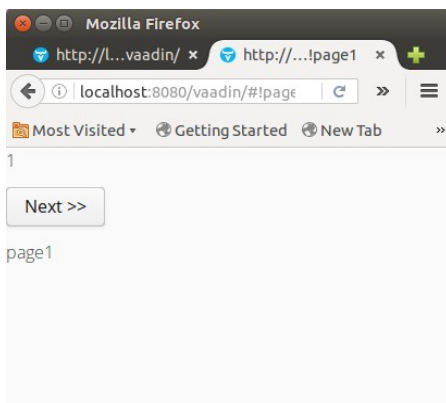
(b) Try to configure the combobox so that only starts to make suggestions from the moment you enter 3 or more characters. For example, entering 'a' or 'an' would yield no suggestions. From the moment you type a third character like 'ant' the combobox will start making suggestions with all municipalities starting with 'ant'.

To do this, you might be tempted to use the *FilteringMode* to configure the combobox how matches are filtered based on the entered data. For example, you can configure a “starts with” filter

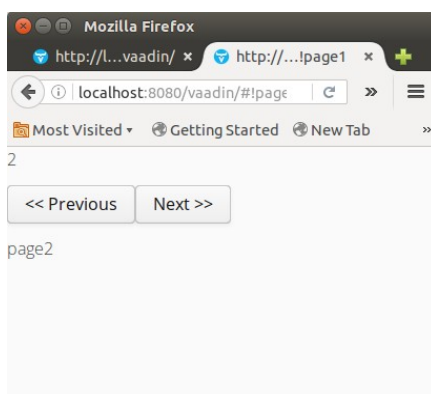
mode. This means that from the moment the suggestions in the combobox start with the text you have entered, the suggestions are displayed. Unfortunately there is no out of the box callback mechanism which you can use to supply your own implementation, so there is no easy way to tell the combobox it should wait with suggestions until the input string is 3 or more in length. However, there is a workaround. By extending the *ComboBox* component, you can override the *changeVariables* method. This method will give you access to the filterstring the user is entering by extracting it from the parameter of the method like this: *String filterString = (String) variables.get("filter");* you can store the filterstring as an instance variable of the extended *ComboBox* you are creating. Finally, you can override the method *List<?> getFilteredOptions()* to only start returning suggestions if the *filterString* is at least 3 or more characters. Try to see if you can make this work

(5)

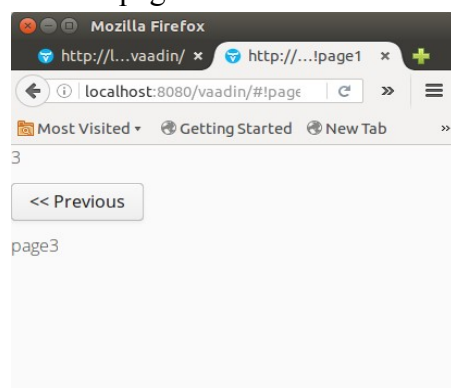
You're going to create a “wizard” consisting of several pages that the user has to walk through. The contents of the wizard is of no importance (this is an exercise on navigation). The first page shows a text displaying “0” and a static text somewhere on the bottom to identify the page “page1”. Each time you navigate forward (or backward) the counter will increase when you navigate to the next (or previous) page. You will navigate with a button on the page (not with the browser back/forward or by changing the URL yourself). You will create three pages. The first page looks like this:



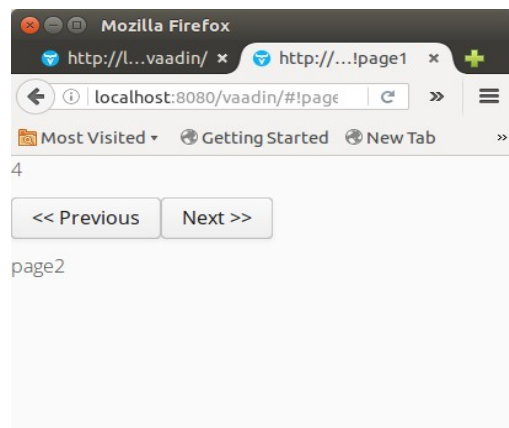
The next page will look like:



The last page will look like:

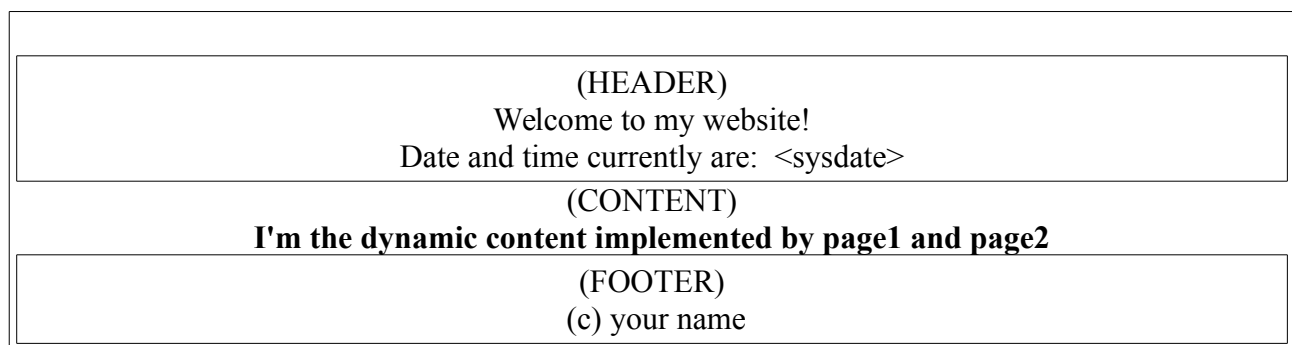


Remember, the counter is dynamic. If I would go back on the third page, the counter would be incremented again on the second page (hint: use the VaadinSession):

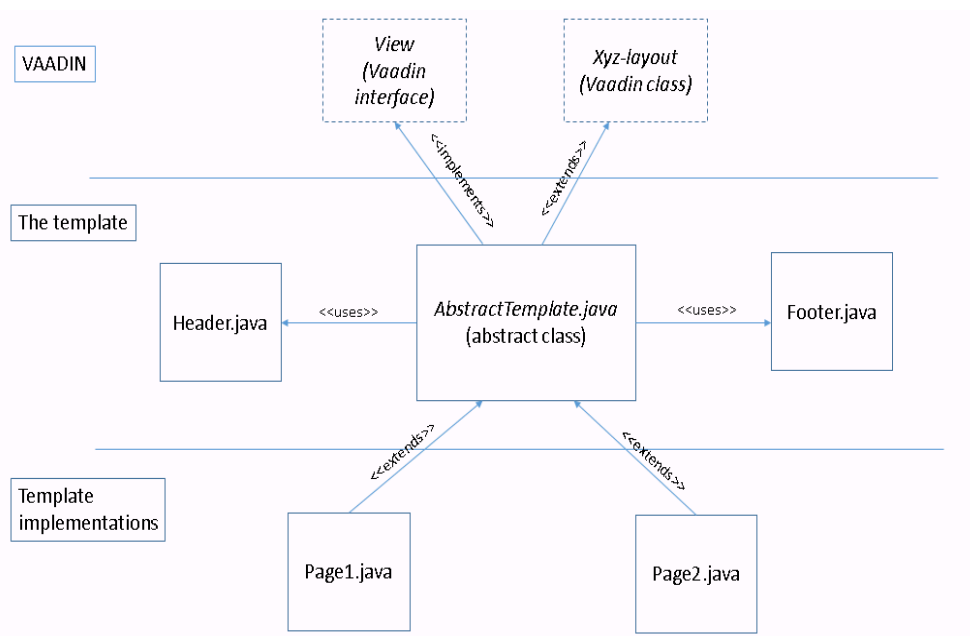


(7)


This is an exercise on creating a template. It has nothing to do with Vaadin itself, but pure Java inheritance (in other words, Vaadin does not need any special templating extension, the Java language already provides OO techniques to do this). The goal is to create a template with header, content and footer. Header and footer are included in the template and are fixed; they do not change. The content on the other hand is dynamic and will change depending on the view.



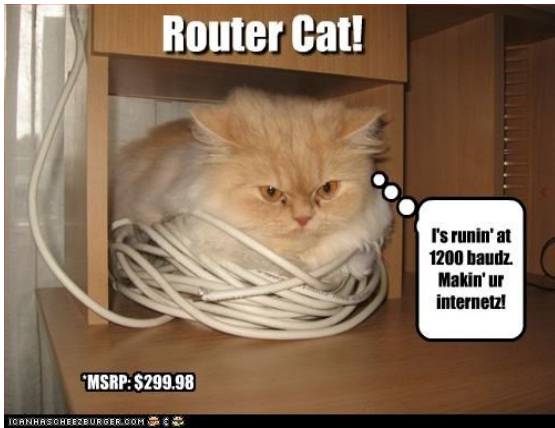
We will let the template be implemented by 2 separate pages. You will have following classes: *Abstracttemplate*, *header*, *footer*, *page1* and *page2*. Header and footer need to be included in the template. Template is an abstract class, so you need an explicit subclass of it to use it as a view “page1” and “page2” are a implementation of the template each providing their contents. In a (sort of) class model this would look like



Going to <http://localhost:8080/Vaadin/#!page1> should show this page:

(HEADER) Welcome to my website! Date and time currently are: <sysdate>
(CONTENT) I'm the dynamic content of "page1": 
(FOOTER) (c) your name

Going to <http://localhost:8080/Vaadin/#!page2> should show this page:

(HEADER) Welcome to my website! Date and time currently are: <sysdate>
(CONTENT) I'm the dynamic content of "page2": 
(FOOTER) (c) your name

Remember, only the dynamic content changed (in this case the image), the header and footer are part of the implemented template. So the classes `page1` and `page2` should **not** include any part of the header or footer. They should just the code for the body content (in this case a simple image that is centered),

Btw; for loading images the easiest way is to place them in a custom theme. A custom theme 'myapp' was already created for you (see `webapp/VAADIN`). This theme extends from the default valo theme. So basically it doesn't change anything to the theme. We are using it to store our images you have seen above. Doing so you will now be able to load the images using a **ThemeResource** . Since Vaadin knows that we are using the 'myapp' theme, it will automatically go look into the theme directory for loading images. Note: the theme loaded until now (in the main UI class) is the “Valo” theme. For this exercise to work (read: for the `themeresource` to find the images) you will need to use our “custom valo” theme with name “myapp”.