Exercise 1: Connecting Components

The goal of this exercise is to demonstrate how you can interconnect Vaadin components in order to automatically keep your data up-to-date in your views. A `Component` that implements the `HasValue` interface in Vaadin contains a typed value. If you change this value, a registered listener can catch the `ValueChangeEvent` and react appropriately to the event (e.g. transfer the new value to the data model or another component). The `HasValue` interface offers the `ValueChangeListener` interface for catching the change event.

In this exercise, you should create a `Slider` component whose minimum value is 0 and maximum value is 100. Next you need to create a `Label component` and bind the slider to the label so, that when the slider's value changes, its numeric representation is shown in the label.

Bonus task: If you are quick with this exercise, try adding a `ProgressBar` component to the layout. `ProgressBar` is a visual representation of a progress. It takes as an argument a float value between the value 0 and 1. In other words, if we want the `ProgressBar` to be halfway, the its value needs to be 0.5.

Note: The `ProgressBar` component accepts a float value between 0 and 1 and the `Slider` returns a double value between the set min- and max-threshold. The Slider should be bound to the `ProgressBar`, so that if you choose the value 75 in the `Slider`, then the `ProgressBar` would be at the state 75%. There is one problem yet to solve and that is that Slider handles type Double while `ProgressBar` expects a Float, so in practice you need to convert the double into a Float which should be within the desired range (75d => 0.75f).

Helpful links to Vaadin documentation

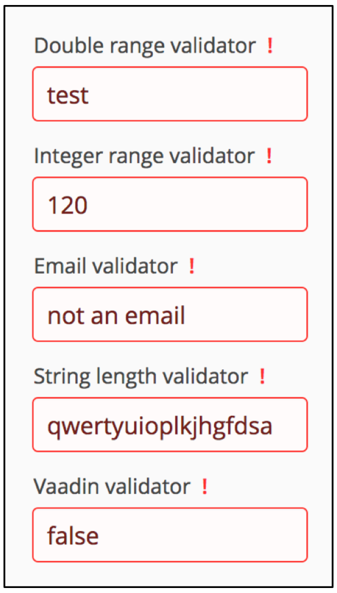
- Reacting to Value Changes
- Slider
- Label
- ProgressBar

Exercise 2: Validation

An essential part of manipulating user input is validating the input values. In this exercise we practice applying validators on fields.

You will start with a view that has five `TextField` components in it. Each `TextField` should have its own validator connected to a binder which is bound to the field. The validators you need to use are: `DoubleRangeValidator`, `IntegerRangeValidator`, `EmailValidator`, `StringLengthValidator` and a custom validator.

- `DoubleRangeValidator` and `IntegerRangeValidator` should accept values between 1 and 100'
- EmailValidator should only accept valid email addresses
- String length validator should accept strings with a maximum length of ten characters
- The custom validator should implement the Validator Interface and check if the user entered "Vaadin"



Double range validator !
test

Integer range validator !
120

Email validator !
not an email

String length validator !
qwertyuioplkjhgfdsa

Vaadin validator !
false

Note that the value of a `TextField` is always a String. If you enter the value "54", it is still a String and will not work with the `Integer/DoubleRangeValidator`. For the validator to pass, you'll first have to apply a converter to the binder that is bound to the `TextField`. The appropriate converters are `StringToInteger-` and `StringToDoubleConverter`.

Hints:

- To get started, you need to implement a simple java bean to use with the `Binder` instance. Your bean should have five properties, one for each `TextField`.
- Bind the fields using `binder.forField(textField)`
- If you encounter null value exceptions with the binder, take a look at the method `Binder.BindingBuilder.withNullRepresentation(String)`

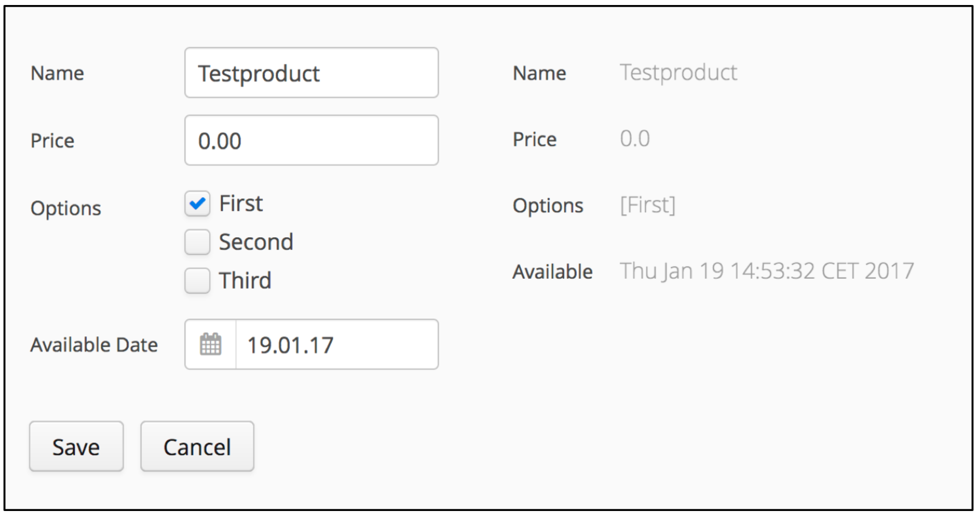
Helpful links to Vaadin documentation

- TextField
- Binding Data to Forms
- Validating and Converting User Input

Exercise 3: Binding Data to Forms

In this exercise, we want to create a form for editing a `Product` bean and display the current values of its properties in a read-only view on the right hand side. Your task is to create the form for editing the product properties and use the `Binder` helper class to bind the product object to the fields of the form.

A selection component called `CheckBoxGroup` should be used for options; other required components are `TextField` and `DateField`. When you click on save, the values from the Binder should be committed into the product bean and the read-only view updated. If you click on cancel, any changes in the form should be reverted (essentially reading back the values from the product bean).



Name	Testproduct	Name	Testproduct
Price	0.00	Price	0.0
Options	✓ First Second Third	Options	[First]
		Available	Thu Jan 19 14:53:32 CET 2017
Available Date	19.01.17		

Steps for this exercise:
1. Create a new class for editing the product; it should extend a layout and contain four fields
2. Create a Binder instance and bind it with the input fields on the editor form
3. Create a Save button that writes the binder values to the product bean and updates the read-only view
4. Create a Cancel button which makes the binder re-read the product bean
5. Implement the `refreshReadOnlyView` method

Hints:
- If you need to use a converter for a field, you need to do it like this: `binder.forField(field).withConverter(…).bind(…)`
- The rest of the fields can be annotated with @PropertyId("propName") and bound with `binder.bindInstanceFields(editorForm)`
- To convert a `Date` to `LocalDate` and vice versa you can use these snippets:
  - `Date.from(value.atStartOfDay().atZone(ZoneId.systemDefault()).toInstant())`
  - `Instant.ofEpochMilli(value.getTime()).atZone(ZoneId.systemDefault()).toLocalDate()`

Bonus task: If you are quick with this exercise, try implementing a Converter that allows you to enter currencies to the Price field. In the application below, I can enter euro values with either the postfix "€" or "EUR" and it will be correctly interpreted as a double value.

Helpful links to Vaadin documentation

- Binding Beans to Forms
- Validating and Converting User Input