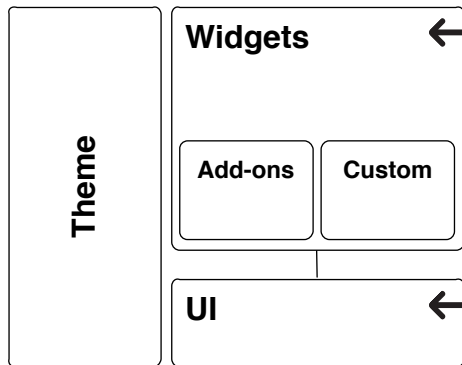


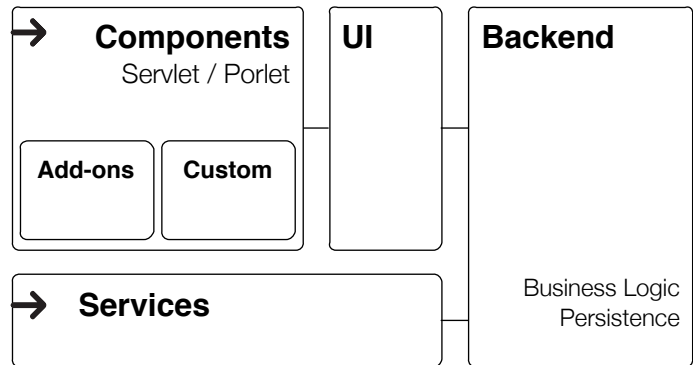
# Architecture

- Vaadin is a component framework
- Vaadin is a server-side framework, meaning, that the code you write is executed as Java on the server

## Browser



## Web Server



- Vaadin widgets: Logic of how an individual component behaves.
- Add-ons: You might extend the core framework with 3rd party widgets.
- Custom: You might have your own custom widgets.
- Theme: The theme defines the look and feel for your application. You may use an existing one or create your own.
- UI: You might have Vaadin independent UI code in your browser. For example, you Vaadin application might be embedded into a web page.
- Components: Server-side APIs of the components.
- Add-ons: The server-side APIs for your 3rd party add-ons.
- Custom: APIs for your custom widgets
- UI: The user interface logic of your application.
- Backend: The business logic of your application.
- Services: Web services that might be used by the Vaadin independent parts of your application.

### NOTE

When using the server-driven programming model, your user interface logic and business logic only exists on the server and is never exposed to the client (browser).

- Vaadin's client-side logic is based on using Google Web Toolkit (GWT)
- All state changes are managed on the server-side, the client side merely reflects the changes in the browser.
- Vaadin is event driven
- The framework handles client-server communication for you

# Components

## Common features in all components

→ You can define a **caption** for all Vaadin components

JAVA

```
TextField textField = new TextField();
textField.setCaption("Username");
```

Username

→ **Icons** can be assigned to components. Icons are in some cases rendered as a part of the component (Button) or in combination with the component (ComboBox)


JAVA

```
Button button = new Button("Lock");
button.setIcon(new ThemeResource("lock.gif"));
```



JAVA

```
ComboBox comboBox =
    new ComboBox("Access rights");
comboBox.setIcon(new ThemeResource("lock.gif"));
```

 Access rights

→ Defining a **Description** for a component will show the string as the component's tooltip

JAVA

```
TextField toolTipField = new TextField();
toolTipField.setDescription("This is a tooltip");

TextField htmlToolTipField = new TextField();
htmlToolTipField.setDescription("This is  
a tooltip <br/> You can also use <b>HTML</b>");
```

This is a tooltip

This is a tooltip  
You can also use **HTML**

→ You can toggle a component's visibility on the server side with **setVisible()**

→ Non-visible components are not sent to the browser, but stay in the server-side state

NOTE

Changing the visibility of a component using CSS will still leave the component in the DOM tree.

JAVA

```
Label invisibleLabel = new Label("This will not be visible");
invisibleLabel.setVisible(false);
```

- You can **disable** any component on the server-side with `setEnabled()`
  - A disabled component will be rendered with less opacity than normal
  - The server does not process events from a disabled component

JAVA

```
TextField disabledTextField =
    new TextField("Disabled field");

disabledTextField.setEnabled(false);
```

Disabled field

- Component can be marked **read-only** with `setReadOnly()`
  - Read-only components do not accept new values, not even when set programmatically
  - If you try to set a value to a read-only component, it will throw an `Property.ReadOnlyException`
  - Binding a read-only property to a field will make the field as well read-only

JAVA

```
TextField readOnlyTextField =
    new TextField("Read-only field");
readOnlyTextField.setValue("field value");
readOnlyTextField.setReadOnly(true);
```

Read-only field  
field value

- All components support **locales**. For example, by defining a locale for a `DateField` this will change the language in which the month and week names are rendered

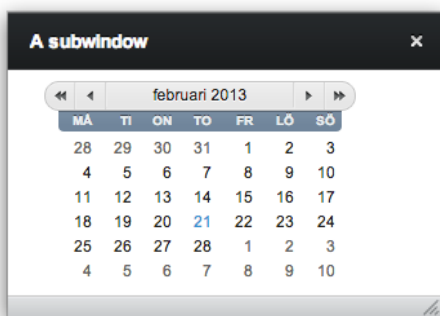
JAVA

```
InlineDateField inlineDateField =
    new InlineDateField();
inlineDateField.setLocale(
    new Locale("sv", "SV"));
```

| februari 2013 |    |    |    |    |    |    |
|---------------|----|----|----|----|----|----|
| MÅ            | TI | ON | TO | FR | LÖ | SÖ |
| 28            | 29 | 30 | 31 | 1  | 2  | 3  |
| 4             | 5  | 6  | 7  | 8  | 9  | 10 |
| 11            | 12 | 13 | 14 | 15 | 16 | 17 |
| 18            | 19 | 20 | 21 | 22 | 23 | 24 |
| 25            | 26 | 27 | 28 | 1  | 2  | 3  |
| 4             | 5  | 6  | 7  | 8  | 9  | 10 |

- You can **define sizes** for components by using `setWidth` and `setHeight`
  - Sizes can be defined relatively (e.g. 100%) or statically (e.g. 200px)
  - Relative sizes are relative to the parent component or a part of it (see layouts)
  - If the size is undefined, then the component components are rendered by their natural size
  - `setSizeFull() = setWidth("100%") + setHeight("100%")`
  - `setSizeUndefined() = setWidth("-1") + setHeight("-1")`

- All components have an API for defining **CSS class names**
  - **setStyleName()**  
clears all previously added style names and adds the given style name
  - **addStyleName()**  
adds a style name to the component
  - **removeStyleName()**  
removes a given style name (if it exists) from the component
  - See Themes for more information about styling
- **getParent()**  
returns the parent component (= component container) for any attached component
  - If the component is not attached to a component container, **getParent()** will return **null**
- **getUI()**  
returns the UI instance to which the component is attached.
  - If the component is not attached to a component container, **getParent()** will return **null**
- **findAncestor(Class<T> parentType)**  
returns the instance of the closest parent component with the given type
  - If a parent component with the given type cannot be found in the parent hierarchy, **null** is returned



```
inlineDateField
.getParent();
```



verticalLayout

```
inlineDateField
.getParent()
.getParent();
```



subWindow



Check out the sampler to see component demos  
and code examples of how to use components

<http://demo.vaadin.com/sampler>

# Component Containers

- Components that can contain one more components within itself
- You add components and remove components from a component container using the API `addComponent()` and `removeComponent()`, respectively

## Layouts

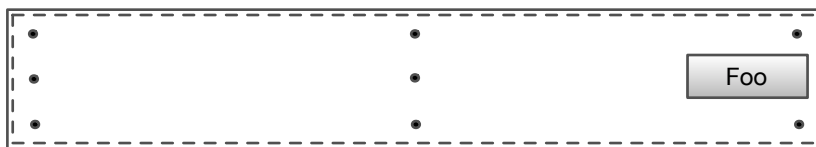
### NOTE

Components that can contain one or more components within itself

- |                         |  |
|-------------------------|--|
| <b>HorizontalLayout</b> | → Rich layouts that support margins, expand ratios and alignment   |
| <b>VerticalLayout</b>   | → Adding a component to a layout will create a "slot" (DIV element) within the layout's DIV in which the component is placed |
| <b>FormLayout</b>       | → Alignment and sizing of components happens relative to and within the slot   |
| <b>GridLayout</b>       |  |

### JAVA

```
HorizontalLayout layout = new HorizontalLayout();
Button button = new Button("Foo");
layout.addComponent(button);
layout.setComponentAlignment(button, Alignment.MIDDLE_RIGHT);
```



- HorizontalLayout
- - - Slot
- Alignment positions

- Slot sizes can be changed using expand ratios
- If only one slot has an expand ratio defined, will it take all the extra space available in the layout

### JAVA

```
HorizontalLayout layout = new HorizontalLayout();
layout.setSizeFull();

NativeButton button = new NativeButton("My size is 1/5");
NativeButton button2 = new NativeButton("My size is 4/5");
button.setSizeFull();
button2.setSizeFull();

layout.addComponent(button);
layout.addComponent(button2);
layout.setExpandRatio(button, 1);
layout.setExpandRatio(button2, 4);
```



- HorizontalLayout
- - - Slot

- **CssLayout** - simply puts components inside of a DIV element and leave the rendering to the browser. Allows you to define custom CSS rules for components directly in Java code.
- **AbsoluteLayout** - components are positioned within the layout using coordinates, e.g. `addComponent(label, "100px", "400px")`
- **CustomLayout** - you define the DOM structure of the layout using HTML templates. Components are put into placeholder elements.

#### HTML

WebContent/VAADIN/themes/mytheme/layouts/loginScreen.html

```
<div class="login-screen">
  <div class="logo"></div>
  <div class="fields" location="username"></div>
  <div class="fields" location="password"></div>
</div>
```

#### JAVA

```
CustomLayout layout = new CustomLayout("loginScreen");
TextField usernameField = new TextField("Username");
PasswordField passwordField = new PasswordField("Password");
layout.addComponent(usernameField, "username");
layout.addComponent(passwordField, "password");
```

#### NOTE

Components with a relative size can only be inside of a layout with a defined size. If the parent layout's size is undefined then your application may not render as expected. If your application does not render as expected, add `?debug` to your application URL and click on the Analyze Layouts-button (AL).

## Non-Layout ComponentContainers

### → TabSheet

A multicomponent container that allows switching between the components with "tabs".

| Tab1  | Tab2 | Tab3 | Tab4 |
|---|------|------|------|
| Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud. |      |      |      |

### → Accordion

A multicomponent container similar to TabSheet, except that the "tabs" are arranged vertically.

| Tab1  |
|---|
| Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore. |
| Tab3  |
| Tab4  |

- **HorizontalSplitPanel** and **VerticalSplitPanel** are a two-component containers that divide the available space into two areas to accommodate the two components.

# Single Component Containers

- SingleComponentContainers manage a single component
- Set or replace the component in the container with **setContent()**



## UI

### NOTE

UI is always at the top of your component hierarchy

- By default, each browser tab gets its own UI instance
- If you refresh your browser, a new UI instance will be created
- To preserve the UI instance between browser refreshes, annotate your UI with **@PreserveOnRefresh**

### JAVA

```
@PreserveOnRefresh
public class MyUI extends UI {

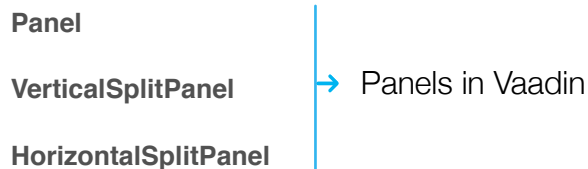
    @Override
    protected void init(VaadinRequest request) {
        VerticalLayout layout = new VerticalLayout();
        layout.setMargin(true);
        setContent(layout);
    }
}
```

# Panel

## NOTE

Only Panels, Windows, TabSheets, Accordions and UIs support scrolling

- Panel and its content can have different sizes
- To enable scrolling, the content should be larger than the panel in the direction that we want to scrolling

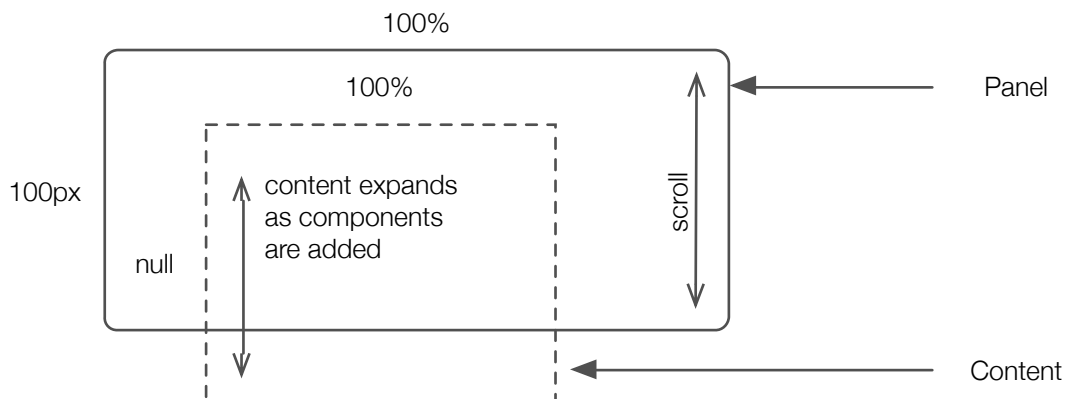


## Example: Panel vertical scroll

- To enable vertical scrolling, ensure that the content layout has an undefined height, or it's height is larger than the size of the panel.
- If content layout's height is undefined, it will grow as components are added, hence the panel will show scrollbars only if required.



A component inside a layout with undefined height must have a static or undefined height (i.e. not relative)



## JAVA

```
VerticalLayout panelLayout = new VerticalLayout();

Label label = new Label("Hello world");
label.setHeight("200px");
panelLayout.addComponent(label);

Panel panel = new Panel();
panel.setContent(panelLayout);
panel.setHeight("100px"); //Force panel to be smaller than label
panel.setWidth("100%");
```

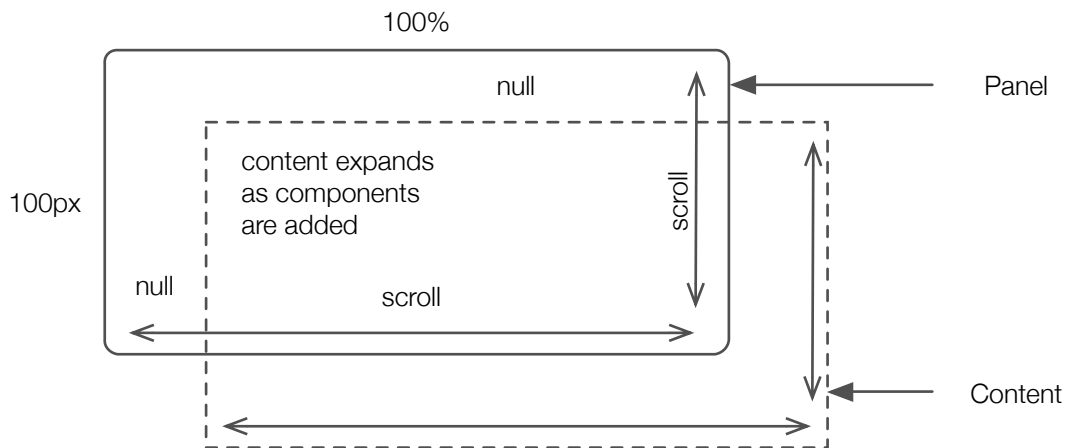


## Example: Panel vertical and horizontal scroll

- To enable both vertical and horizontal scrolling, ensure that the content layout has an undefined size, or it's size is larger than the size of the panel.



A component inside a layout with undefined size must have a static or undefined size (i.e. not relative)



### JAVA

```
VerticalLayout panelLayout = new VerticalLayout();
panelLayout.setSizeUndefined();

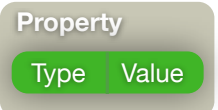
Label label = new Label("Hello world");
label.setHeight("200px");
label.setWidth("200px");
panelLayout.addComponent(label);

Panel panel = new Panel();
panel.setContent(panelLayout);
panel.setHeight("100px"); //Force panel to be smaller than label
panel.setWidth("100px");
```

# Vaadin Data Model

## Property

- Property is an interface for managing values
- Typically implementations also implement **Property.ValueChangeNotifier**, which means that the property can send **ValueChangeEvent**s in case of value changes.
- Values are maintained through **getValue()** and **setValue()**
- A property is typed: e.g. **Property<String>** which means that **getValue()** will only return values of the specified type. This is also true for **setValue(value)**.

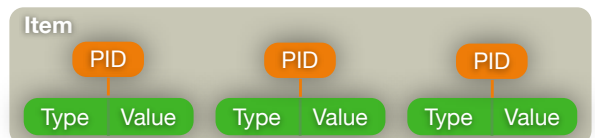


## Item

### NOTE

An item is a collection of **Properties**

- Each property has a *propertyId*, which is a unique identifier for a **Property**



### JAVA

```
Person person = new Person();

BeanItem<Person> personItem = new BeanItem<Person>(person);
personItem.addNestedProperty("address.street");

Property<String> firstNameProperty = personItem.getItemProperty("firstName");
Property<String> lastNameProperty = personItem.getItemProperty("lastName");
Property<String> addressProperty = personItem.getItemProperty("address.street");

System.out.println(firstNameProperty.getValue() + " "
    + lastNameProperty.getValue() + " ["
    + addressProperty.getValue()+ "]");
```

### JAVA

```
//Example classes used in this code example above
public class Person{
    private String firstName = "Anna";
    private String lastName = "Jonsson";
    private Address address = new Address();
    //Getters and setters
}

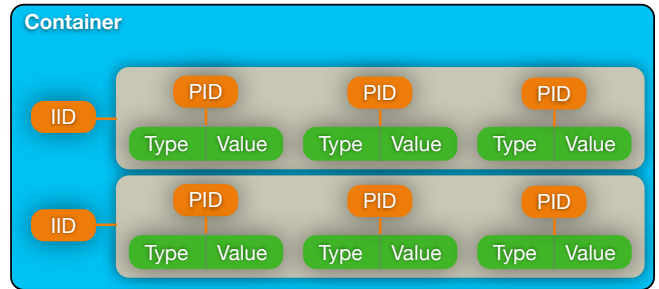
public class Address{
    private String street = "Example street 1";
    //Getters and setters
}
```

# Container

## NOTE

A container is a collection of **Items**

- Each **Item** has a *itemId*, which is a unique identifier for an **Item**



- Typically (**but not always**) implement some of the sub interfaces:
- **Container.Filterable**  
Container supports filtering - filters can be added to the container
- **Container.Sortable**  
Container supports sorting - sort the container based on a property
- **Container.Indexed**  
Container is indexed - items in the container are in a given order

## IndexedContainer

- An in-memory container for which you define properties and manually populate it with items.
- Values for an item's properties are manually set

## JAVA

```
IndexedContainer indexedContainer = new IndexedContainer();
indexedContainer.addContainerProperty("First name", String.class, "");
indexedContainer.addContainerProperty("Last name", String.class, "");

List<Person> persons = getPersons();
for(Person person : persons){
    Item item = indexedContainer.addItem(person);
    item.getItemProperty("First name").setValue(person.getFirstName());
    item.getItemProperty("Last name").setValue(person.getLastName());
}

table.setContainerDataSource(indexedContainer);
```

## HierarchicalContainer

- An extension of **IndexedContainer** that supports parent/child relations.
- Typically used to populate components like the Tree or TreeTable

## BeanItemContainer

- Automatically creates properties based on a Bean type.
- Can be automatically populated by giving a collection of beans of the defined type.
- Container automatically creates items and sets the values for the properties

JAVA

```
Collection<Person> persons = getPersons();

BeanItemContainer<Person> beanItemContainer
    = new BeanItemContainer<Person>(Person.class);
beanItemContainer.addAll(persons);

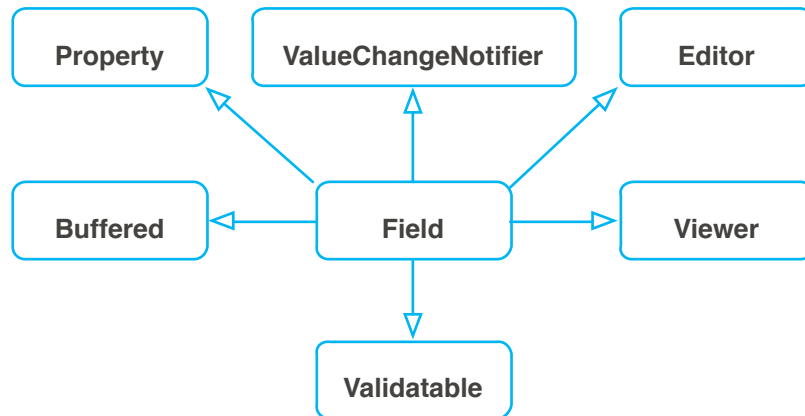
table.setContainerDataSource(beanItemContainer);
```

## SQLContainer

- A container that connects directly to an SQL database.

# Fields

- Fields are components for entering/selecting data
- Field interface extends the following interfaces:



- **Property** - Get/set value with `getValue()` and `setValue()`
- **ValueChangeNotifier** - If value changes, a **ValueChangeEvent** will be sent to all listeners
- **Property.Editor/Viewer** - Properties can be bound to the field and the field will use the property's value as its own value. If the field value changes, the property's value will be automatically changed.
- **Buffered** - Value propagation from the field can be controlled with `commit()` and `discard()` on the server side
- **Validatable** - Validators can be added to the field

## JAVA

```
BeanItem<Person> personItem = new BeanItem<Person>(new Person());
final Property<String> firstNameProperty = personItem.getItemProperty("firstName");

final TextField firstNameField = new TextField("First name");
firstNameField.setPropertyDataSource(firstNameProperty);
firstNameField.setBuffered(true);

Button commitButton = new Button("Commit");
commitButton.addClickListener(new Button.ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        System.out.println("Prop:" + firstNameProperty.getValue()
            + " -- Field: " + firstNameField.getValue());
        firstNameField.commit();
        System.out.println("Prop:" + firstNameProperty.getValue()
            + " -- Field: " + firstNameField.getValue());
    }
});
```

# Validators

## NOTE

Validators allow you to validate the values of fields against a set of rules defined in a validator

- Many default validator implementations for common validations in the core
- Most validators accept null values
- If you do not want empty values, then you can mark the field as `setRequired(true)`, although it is not a validator, it behaves in a similar way

## Example: DoubleRangeValidator

- Validates that the given value is of type Double
- Validates that the value is between two given limit values

# Converters

## NOTE

A converter transforms the format of the value between the data source and the presentation. Locale is taken into account when converting.

## Example: TextField

- TextField only handles String values, if you bind a Property with type Double it must be converted between the presentation and model type.

## JAVA

```
ObjectProperty<Double> doubleProperty = new ObjectProperty<Double>(new Double(0));

TextField doubleField = new TextField("Double field");
doubleField.setImmediate(true);

doubleField.setConverter(new StringToDoubleConverter());

doubleField.addValidator(
    new DoubleRangeValidator("Value {0} is not between -2 and 2", -2d, 2d));

doubleField.setPropertyDataSource(doubleProperty);
```

Double field

0



Double field

45

Value 45.0 is not between -2 and 2

# Field Groups

## NOTE

Field groups are used to create web forms

- You bind an **Item** to a layout that contain **Fields** using a **FieldGroup**
- The recommended way to use field groups is to create the form layout separately and annotate the fields with `@PropertyId("<propertyId>")`
- The `commit()` and `discard()` API works the same way as for buffered fields.
- Commits are done transactionally, so either all values are committed or none are.

## JAVA

```
public class PersonForm extends HorizontalLayout{
    @PropertyId("firstName")
    private TextField personFirstName = new TextField("First name");
    @PropertyId("lastName")
    private TextField personLastName = new TextField("Last name");
    @PropertyId("address.street")
    private TextField street = new TextField("Home address");

    public PersonForm() {
        setSpacing(true);
        addComponent(personFirstName);
        addComponent(personLastName);
        addComponent(street);
    }
}
```

## JAVA

```
BeanItem<Person> personItem = new BeanItem<Person>(new Person());
personItem.addNestedProperty("address.street");

PersonForm personFormLayout = new PersonForm();

final FieldGroup binder = new FieldGroup();
binder.setItemDataSource(personItem);
binder.bindMemberFields(personFormLayout);

Button commit = new Button("Commit", new Button.ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        try { binder.commit(); }
        catch (CommitException e) { //TODO handle exception
        }
    }
});
```

|                                       |                                      |   |
|---------------------------------------|--------------------------------------|---|
| First name                            | Last name                            | Home address                                  |
| <input type="text" value="Anna"/>     | <input type="text" value="Jonsson"/> | <input type="text" value="Example street 1"/> |
| <input type="button" value="Commit"/> |                                      |   |

# Resources

- Vaadin contains components for showing different kinds of resources in your user interface

|              |                                     |
|--------------|-------------------------------------|
| Flash        |                                     |
| Image        |                                     |
| Embedded     | → Accepts a Resource object as data |
| BrowserFrame |                                     |
| Video        |                                     |
| Audio        |                                     |

- **Embedded** can be used for resource types not covered by the built-in types
- Component icons are defined using resources
- When using the **Link** component, the target is given as a resource

## Resource objects

- **FileResource** - reads the content of a File object
- **ClassResource** - reads a file that is in the *classpath*
- **ThemeResource** - reads a file from the application's Theme folder
- **StreamResource** - used for serving dynamic content through a stream
- **ExternalResource** - points to a resource that is not on on the server, for example, a web site URL



# Themes

NOTE

Themes define the look and feel of your application

- Consists of cascading style sheets (CSS) and images (or other resources)
- Can also contain custom layouts for your application

## Sass (Syntactically Awesome Stylesheets)

- Vaadin uses Sass for defining its themes
- **SCSS (Sassy CSS)** - Sass has two dialects (SASS and SCSS), Vaadin uses SCSS
- Sass makes your CSS more maintainable as it supports for instance variables and inheritance

SCSS code is compiled to normal CSS for the browser

→ In development mode - automatically on the fly

→ In production mode - you need to have a compiled theme in your web application package

- **Variables** - use variables to define often used values

SCSS

```
$blue: #3bbfce;
$margin: 16px;

.border {
  padding: $margin;
  margin: $margin;
  border-color: $blue;
}
```

CSS

```
.border {
  padding: 16px;
  margin: 16px;
  border-color: #3bbfce;
}
```

- **Nesting** - Nest CSS selectors to reduce typing

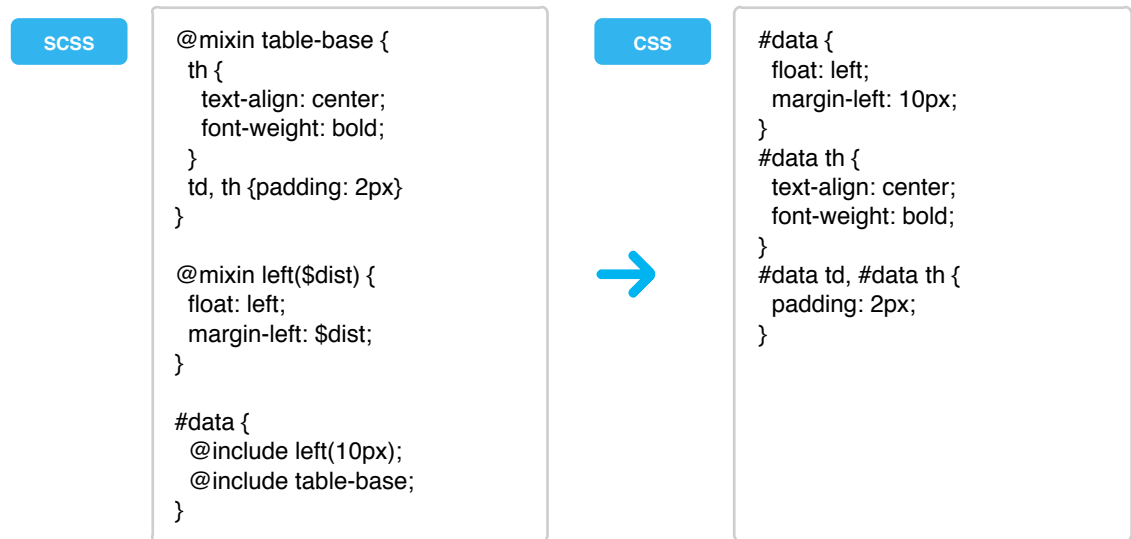
SCSS

```
li {
  font: {
    family: serif;
    weight: bold;
    size: 1.2em;
  }
}
```

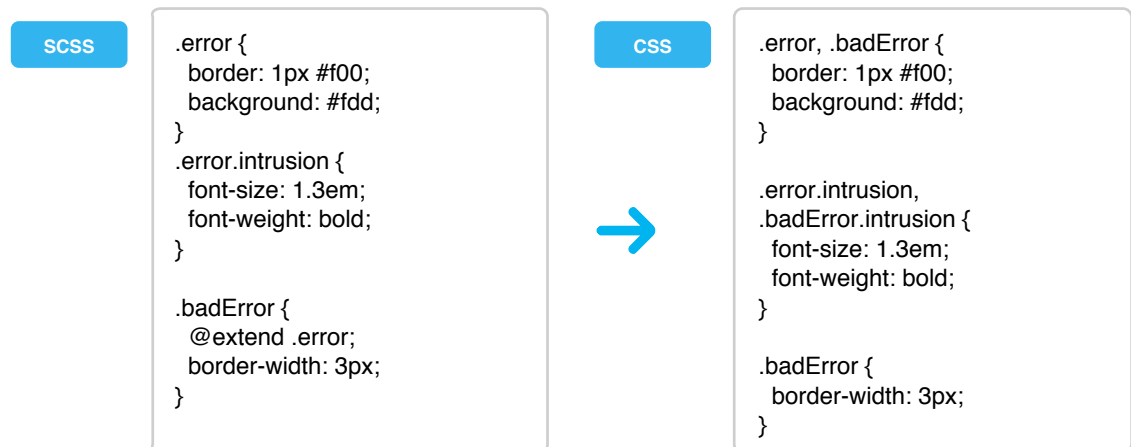
CSS

```
li {
  font-family: serif;
  font-weight: bold;
  font-size: 1.2em;
}
```

→ **Mixins** - Use Mixins to apply blocks of CSS rules to your selectors



→ **Selector inheritance** - extend selectors to provide customised behaviour



→ Themes are located under: WebContent/VAADIN/themes/<theme name>/

→ Apply a theme on a UI by annotating the class with @Theme("<theme name>")

## Using a theme

→ The Vaadin naming convention for component style names is:

### NOTE

.v-<component>-<attribute>

#### Examples

.v-button-caption  
.v-datefield-button  
.v-accordion-item-caption

.v-datefield-textfield  
.v-accordion-item

→ Define your own CSS class names to components with `addStyleName()`

JAVA

```
Label myLabel = new Label("Hello world");
myLabel.addStyleName("foo");
```

→ Target label from CSS with `.<style>` or `.v-label-<style>`

HTML

```
<div class="v-label v-widget foo v-label-foo">Hello world</div>
```

NOTE

`addStyleName()` - add a style name to a component  
`setStyleName()` - clears previous style names and adds the given style name

→ Use browser tools (like Firebug) to see how your CSS class names are applied to elements

NOTE

**Remember cross-browser testing!**



Wrap commonly used styles into their own components. For example, if you are using the style "warning-label" for labels that show a warning to the user:

Instead of calling `myLabel.addStyleName("warning-label")` in multiple places, use a class for the warning label:

JAVA

```
public class WarningLabel extends Label {

    public WarningLabel() {
        addStyleName("warning-label");
    }
}
```

# Add-ons

- You can extend the core framework by using 3rd party add-ons
- Add-ons are typically distributed through the Vaadin directory
- The directory is located at <https://vaadin.com/directory>
- All add-ons in the directory are also available through Maven

## Adding an add-on to your project

1. Search for the add-on you want and download the JAR file
2. Place the JAR file in your classpath
  - In **Eclipse**, place the downloaded add-on into WebContent/WEB-INF/lib/
3. If your add-on has new client-side code (typical for components, but not always), then you need to recompile your widgetset. In Eclipse, you can click on the recompile widgetset button, if you have installed the Vaadin plugin for Eclipse.



4. You can now use the add-on as if it were a part of the core framework!

### NOTE

**When using add-ons:** You need to recompile your widgetset every time the client-code has some modifications. Typically this is when you introduce new add-ons to your project or when you change the version of the Vaadin framework.

If you do not use any add-ons, then you do not ever need to recompile the widgetset.

- If you are using Maven, then click on "Maven POM" in the Vaadin directory instead of downloading the JAR file, this will give you information about how to include the add-on to your project.

### XML

```
<dependency>
  <groupId>com.vaadin.addon</groupId>
  <artifactId>vaadin-calendar</artifactId>
  <version>2.0.0</version>
</dependency>

<repository>
  <id>vaadin-addons</id>
  <url>http://maven.vaadin.com/vaadin-addons</url>
</repository>
```