Getting Started with Vaadin Framework 8

BY **MATTI TAHVONEN** - ORIGINAL BY **MARKO GRÖNROOS**

CONTENTS

ABOUT VAADIN FRAMEWORK

Vaadin Framework is a productive and easy-to-use UI library for developing web applications in Java or your JVM language of choice. Vaadin's strong abstraction of web technologies gives you a component-based approach to build your apps, almost like you're building traditional desktop apps, but through state-of-the-art web technologies. You use an object-oriented approach to compose your UI from smaller UI components and layouts, and hook your logic to the UI with event listeners.

In addition to providing high-level UI components, which saves you from time-consuming HTML, CSS, and JavaScript programming, Vaadin also abstracts away the communication between the server and the browser. This makes both the UI and backend development easier (no need to expose REST services) and increases security.

Vaadin Framework is open source and licensed under the liberal Apache 2 license.

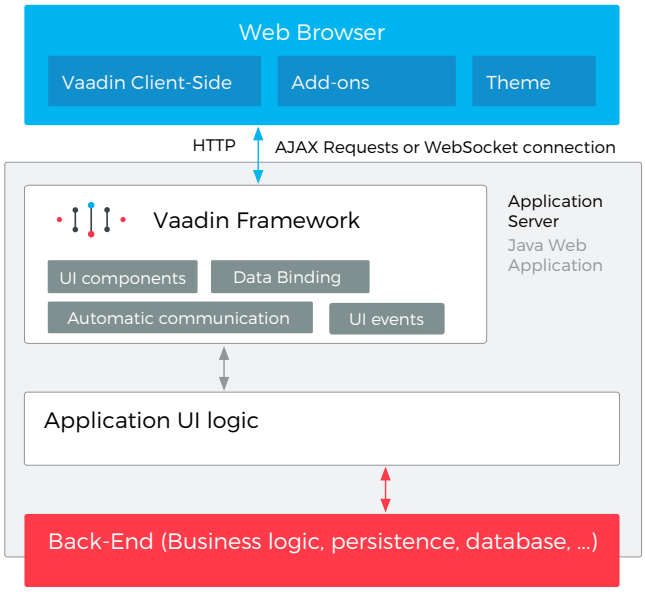


Figure 1 *Vaadin Client-Server Architecture*

For extending the framework, there are numerous add-ons built by Vaadin and the community. The Directory service currently lists nearly 700 add-ons for Vaadin Framework.

BOOTSTRAPPING A PROJECT

You can create a Vaadin application project easily from Maven archetypes. To create a simple app stub, use the following command (or use the same groupId and archetypeId in your IDE):

```
mvn archetype:generate -DarchetypeGroupId=com.vaadin
-DarchetypeArtifactId=vaadin-archetype-application
-DarchetypeVersion=LATEST
```

start.spring.io also provides an option to include Vaadin dependencies to your Spring project. The most commonly used IDEs (IntelliJ, Eclipse, and Netbeans) also have built in support or plugins that can help to create Vaadin applications.

THE UI CLASS: THE ENTRY POINT TO YOUR APPLICATION

The UI class (extends the com.vaadin.UI) is the entry point to your application. The framework creates an instance when the user opens the page in the browser and calls the init method init().

```
@Title("My Vaadin UI")
public class HelloWorld extends com.vaadin.UI {
  @Override
  protected void init(VaadinRequest request) {
    // Create the content root layout for the UI
    VerticalLayout content = new VerticalLayout();
    setContent(content);
    // Display the greeting
    content.addComponent(new Label("Hello World!"));
  }
}
```

BECOMING A VAADIN EXPERT

While Vaadin has the lowest learning curve on the market, the web application ecosystem keeps evolving continuously.

With our subscription based offerings you'll make sure to always benefit from the newest technologies and stay ahead of the learning curve.

Benefit from the easy integration of our charts library or speed up your development with the visual designer and our various other tools.

Vaadin also offers exceptionally good documentation, tutorials and official hands-on trainings all around the world.

Attend our official Vaadin trainings to learn about Vaadin Best Practices, the right application architecture or how to best benefit from the latest features in browsers or our tools.

We offer a broad selection of over 30 courses that will help you stay a web development expert.

CORE

- ✓ Vaadin Framework
- ✓ Access to 600+ add-ons
- ✓ Discussion forum
- ✓ Extensive documentation

PRO

- ✓ All CORE features
- ✓ Designer
- ✓ TestBench
- ✓ Charts
- ✓ Spreadsheet
- ✓ Board

PRIME

- ✓ All CORE + PRO features
- ✓ Expert Chat
- ✓ Development on Demand
- ✓ Warranty

AVAILABLE FOR PRO AND PRIME

TRAINING SUBSCRIPTION

Unlimited access to expert lead trainings and certifications online and live.

vaadin.com/pricing

vaadin

Normally, you need to:

- Extend the UI class.
- Build an initial UI from components
- Define event listeners to implement the UI logic.

Optionally, you can also:

- Set a custom theme for the UI.
- Bind components to data.
- Bind components to resources.

Tip: You can get a reference to the UI object associated with the currently processed request from anywhere in the application logic with UI.getCurrent(). You can also access the current VaadinSession, VaadinService, and VaadinServlet objects in the same way.

DEPLOYMENT

To expose UI classes for end users, you'll need a VaadinServlet deployed to a servlet container. The VaadinServlet handles server requests and manages user sessions and UIs. If you use Vaadin Spring or Vaadin CDI (available in the Vaadin Directory), the servlet is configured automatically, and you only need to expose your UI class using `@SpringUI` or `@CDIUI("")` annotation. In a basic servlet container project, like the one that is generated by vaadin-archetype-application, the servlet can be declared as such:

```
@WebServlet(urlPatterns = "/*", name = "MyUIServlet",
asyncSupported = true)
@VaadinServletConfiguration(ui = MyUI.class,
productionMode = false)
public static class MyUIServlet extends VaadinServlet {
}
```

Most of the time you don't need to work at the servlet level at all, but you can add customization to the servlet if you want. In Spring and CDI projects, you can just expose a customized version of SpringVaadinServlet or VaadinCDIServlet and the automatic default version is then ignored.

Vaadin UIs can also be deployed as portlets in a portal. Refer to vaadin.com/docs for Portlet deployment.

EVENTS LISTENERS

In the event-driven model, user interactions with user interface components trigger server-side events, which you can handle with event listeners.

In the following example, we handle click events for a button with an anonymous class:

```
Button button = new Button("Click Me");
button.addClickListener(event-> {
  Notification.show("Thank You!");
});
layout.addComponent(button);
```

Different components provide different listeners. For example, value changes in a field component can be handled with a ValueChangeListener, and TabSheet allows you to listen to tab changes using SelectedTabChangeListener. Many field components allow to track focus using FocusListener.

In addition to the event-driven model, UI changes can be made from other places than UI initiated threads. Changes are reflected automatically if you have enabled server push (using vaadin-push module and @Push annotation to UI class) or use polling configured to the UI class. The push mode primarily uses WebSockets for communication and falls back to long polling if it's not supported.

```
public void notifyUI(YourAppUI activeUI, String msg) {
  // ensure proper synchronization using
  // the UI.access method when called
  // from a non-UI initiated event
  activeUI.access(() -> {
    activeUI.showMessageInUI(msg);
    // method in your UI class
  });
}
```

COMPONENTS

Vaadin components include field, layout, and other components. The component classes, related helper classes, and their inheritance hierarchy is illustrated in Figure 2.

COMPONENT PROPERTIES

Common component properties are defined in the Component interface and the AbstractComponent base class for all components.

PROPERTY	DESCRIPTION
Caption	A label usually shown above, left of, or inside a component, depending on the component and the containing layout.
Description	A longer description that is usually displayed as a tooltip when the mouse hovers over the component.
Enabled	If false, the component is shown as grayed out and the user cannot interact with it. (Default: true.)
Icon	An icon for the component, usually shown left of the caption, specified as a resource reference.
Locale	The current country and/or language for the component. Meaning and use is application-specific for most components. (Default: UI locale.)
Visible	Whether the component is visible or not. (Default: true.)

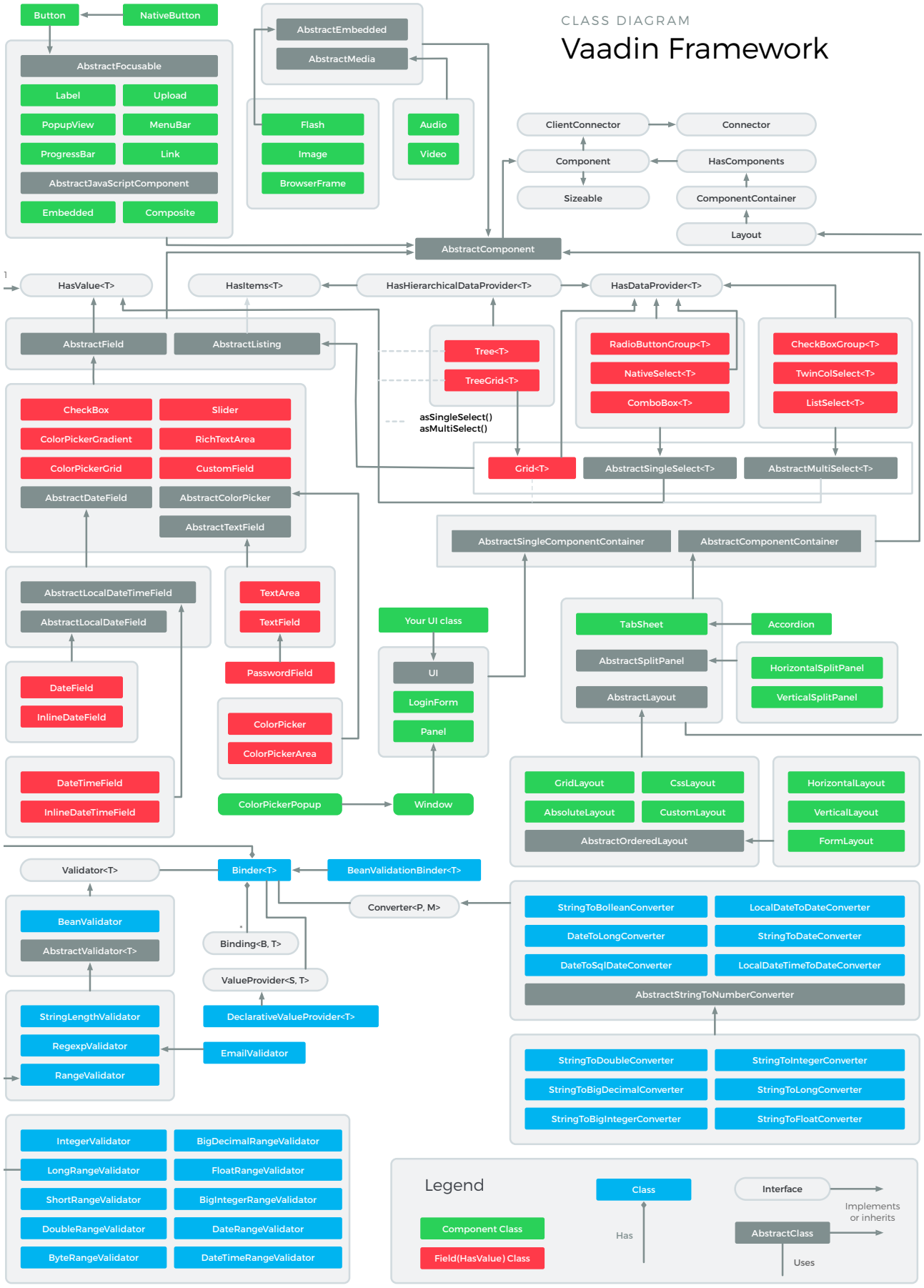


Figure 2 *Class diagram*

FIELD PROPERTIES

Field implementations often extend from AbstractField<T> and contain the following commonly used properties.

PROPERTY	DESCRIPTION
value	The actual value of the field. The type depends on the field and possible configuration. Value changes can be monitored using ValueChangeListener, but often values are bound automatically using com.vaadin.data.Binder.
requiredIndicatorVisible	A Boolean controlling whether the component should be marked as required in the UI (most often using red * character).
readOnly	If true, the user cannot change the value. (Default: false.)
tabIndex	The tab index property is used to specify the order in which the fields are focused when the user presses the Tab key.

SIZING

The size of components can be set in fixed or relative units by either dimension (width or height), or be undefined to shrink to fit the content.

METHOD	DESCRIPTION
setWidth() **setHeight()**	Set the component size in either fixed units (px,pt, pc, cm, mm, in, em, or rem) or as a relative percentage (%) of the available area provided by the containing layout. The null value or -1 means undefined size (see below), causing the component to shrink to fit the content.
setSizeFull()	Sets both dimensions to 100% relative size.
setSizeUndefined()	Sets both dimensions as undefined, causing the component to shrink to its minimum size.

Notice that a layout with an undefined size must not contain a component with a relative (percentual) size. For more tips on how to use relative sizes, refer to the Layout section.

RESOURCES

Icons, embedded images, hyperlinks, and downloadable files are referenced as resources.

```
Button button = new Button("Button with an icon");
button.setIcon(new ThemeResource("img/myimage.png"));
```

External and theme resources are usually static resources. Connector resources are served by the Vaadin servlet.

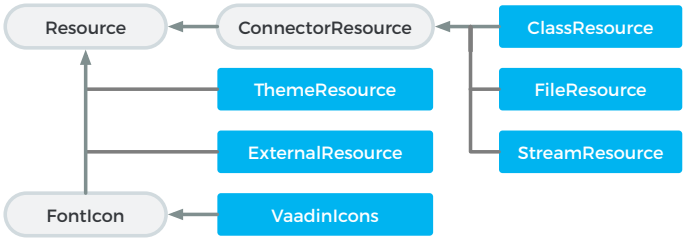


Figure 3 *Resource Classes and Interfaces*

CLASS NAME	DESCRIPTION
ExternalResource	Any URL.
ThemeResource	A static resource served by the application server from the current theme. The path is relative to the theme folder.
FileResource	Loaded from the file system.
ClassResource	Loaded from the class path.
StreamResource	Provided dynamically by the application.
FontIcon	Font icon sets, such as VaadinIcons, contain a selection of special resources that can be used as icons.

LAYOUT COMPONENTS

The layout of a UI is built hierarchically from layout components, or more generally component containers, with the actual interaction components as the leaf nodes of the component tree.

You start by creating a root layout and setting it as the UI content with setContent(). Then you add the other components to that with addComponent(). Single-component containers, most notably Panel and Window, only hold a single content component, just as UI, which you must set with setContent().

The sizing of layout components is crucial. You can either use explicit size or define it relatively from the area provided by the

parent component. Notice that if all the components in a layout have relative size in a particular direction, the layout may not have undefined size in that direction!

Tip: Besides building your layouts in Java, you can also layout your view in declarative HTML format or visually by using Vaadin Designer inside Eclipse or IntelliJ.

MARGINS

Certain layout components support setting margins programmatically. Setting setMargin(true) enables all margins for a layout, and with a MarginInfo parameter you can enable each margin individually. The margin sizes can be adjusted with the padding property (as top, bottom, left, and right padding) in a CSS rule with a corresponding v-top-margin, v-bottom-margin, v-left-margin, or v-right-margin selector. For example, if you have added a custom mymargins style to the layout:

```
.mymargins.v-margin-left {padding-left: 10px;}
.mymargins.v-margin-right {padding-right: 20px;}
.mymargins.v-margin-top {padding-top: 30px;}
.mymargins.v-margin-bottom {padding-bottom: 40px;}
```

SPACING

Certain layout components also support the setSpacing(true) method that controls spacing between the layout slots. Spacing can be adjusted with CSS as the width or height of the elements with the v-spacing style. For example, for a vertical layout:

```
.v-vertical > .v-spacing {height: 50px;}
```

For a GridLayout, you need to set the spacing as left/top padding for a .v-gridlayout-spacing-on element:

```
.v-gridlayout-spacing-on {
  padding-left: 100px;
  padding-top: 50px;
}
```

ALIGNMENT

When an area provided by the layout component is larger than a contained component, the component can be aligned within the cell with the setComponentAlignment() method as in the example below:

```
VerticalLayout layout = new VerticalLayout();
Button button = new Button("My Button");
layout.addComponent(button);
layout.setComponentAlignment(button,
  Alignment.MIDDLE_CENTER);
```

EXPANDING COMPONENTS TO AVAILABLE SPACE

The ordered layouts and GridLayout support expand ratios, to allow some components to take the remaining space left over from other components. The ratio is a float value and components have 0.0f default expand ratio. The expand ratio must be set after the component is added to the layout.

```
VerticalLayout layout = new VerticalLayout();
layout.setSizeFull();
layout.addComponent(new Label("Title"));
// Doesn't expand
TextArea area = new TextArea("Editor");
area.setSizeFull();
layout.addComponent(area);
layout.setExpandRatio(area, 1.0f);
```

VerticalLayout and HorizontalLayout also contain a helper method called addComponentsAndExpand(Component...) that adds a component and adjusts expand ratios and component sizes that suit most common use cases. The above can be flattened to:

```
VerticalLayout layout = new VerticalLayout();
layout.addComponent(new Label("Title"));
// Doesn't expand
TextArea area = new TextArea("Editor");
layout.addComponentsAndExpand(area);
```

The Grid component also supports expand ratios for columns.

CUSTOM LAYOUT

The CustomLayout component allows the use of an HTML template that contains location tags for components, such as . The components are inserted in the location elements with the addComponent() method as shown below:

```
CustomLayout layout = new CustomLayout(
  getClass().getResourceAsStream("mylayout.html"));
layout.addComponent(new Button("Hello"), "hello");
```

The layout content is given as an InputStream or it can also be a named resource file in your theme directory.

ADD-ON COMPONENTS

Your application might require less frequently needed features or UI components, which are not included in the core distribution. Hundreds of Vaadin add-on components are available in the Vaadin Directory, both free and commercial. To use a component, just copy the dependency information for Maven or download the jar files from Directory into your project.

Many of the add-ons contain client side extensions. These add-ons require you have a widgetset that combines both built in and custom client side extensions into one compact client-side engine. If you created your add-on with vaadin-archetype-application, your application is ready for add-on usage out of the box, but remember to execute "mvn clean install" (or the same via your IDE) to ensure the client side engine is rebuilt.

If your project doesn't have vaadin-maven-plugin (e.g. a start.spring.io project or simplified example app), add the following build plugin to your project's pom.xml file:

```
<plugin>
  <groupId>com.vaadin</groupId>
  <artifactId>vaadin-maven-plugin</artifactId>
  <version>${version.vaadin}</version>
  <executions>
    <execution>
      <goals>
        <!-- Needed for theme: -->
        <goal>update-widgetset</goal>
        <goal>compile</goal>
        <!-- Needed for theme: -->
        <goal>update-theme</goal>
        <goal>compile-theme</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

THEMES

Vaadin allows customization of the appearance of the user interface with themes. Themes can include Sass or CSS style sheets, custom layout HTML templates, and graphics.

BASIC THEME STRUCTURE

Custom themes are placed under the /VAADIN/themes/ folder of the web application (src/main/webapp in Maven projects). This location is fixed and the VAADIN folder specifies that these are static resources specific to your Vaadin application. The structure is illustrated in Figure 4.

Each theme has its own folder with the name of the theme. A theme folder must contain a styles.scss (for Sass) or a styles.css (for plain CSS) style sheet. Custom layouts must be placed in the layout's sub-folder, but other contents may be named freely.

Custom themes need to inherit a base theme. The suggested base theme is Valo, which is easily customizable using Sass variables. Such a theme is created for you automatically if you bootstrap your project is built on top of the vaadin-archetype-application archetype.

SASS THEMES

Sass (Syntactically Awesome StyleSheets) is a stylesheet language based on CSS3, with some additional features such as variables, nesting, mixins, and selector inheritance. Sass themes need to be compiled to CSS. Vaadin includes a Sass compiler that compiles stylesheets during the build process. You can also use on-the-fly compilation during development, by disabling the compilation from the build script.

To enable multiple themes on the same page, all the style rules in a theme should be prefixed with a selector that matches the name of the theme. It is defined with a nested rule in Sass. Sass themes are usually organized in two files: a styles.scss and a theme-specific file such as mytheme.scss.

With this organization, the styles.scss would be as follows:

```
@import "mytheme.scss";
@import "addons.scss";

// This file prefixes all rules with the theme
// name to avoid causing conflicts with other themes.
// The actual styles should be defined in mytheme.scss

.mytheme {
  @include addons;
  @include mytheme;\
}
```

The mytheme.scss, which contains the actual theme rules, would define the theme as a Sass mix-in as follows:

```
@import "../valo/valo.scss";

@mixin mytheme {
  @include valo;

  // Insert your own theme rules here
  .mycomponent { color: red; }
}
```

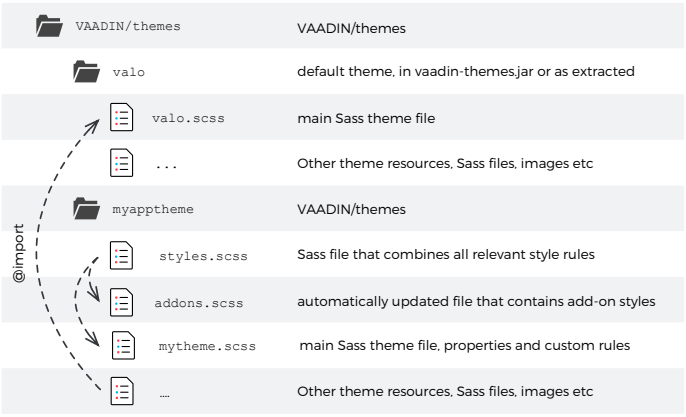


Figure 4 *Theme Contents*

APPLYING A THEME

Every component has a default CSS class based on the component type, and you can add custom CSS classes for UI components with addStyleName(), as shown below.

You set the theme for a UI with the @Theme annotation.

```
@Theme("mytheme")
public class MyUI extends UI {

    @Override
    protected void init(VaadinRequest vaadinRequest) {
       //...
       Label label = new Label("This is my themed
component");
       label.addStyleName("mycomponent");
       layout.addComponent(label);
    }
```

DATA BINDING

Vaadin allows binding your Java domain classes directly to components. This way, you avoid writing a lot of boilerplate code.

LISTING DATA IN GRID

Grid, the component that shows your data in a tabular format, often plays a central part in a business application. The easiest way to use Grid is to pass the data as a list, stream or an array. In the following code example, Person is your domain object (Java Bean):

```
Grid<Person> grid = new Grid<>(Person.class);
grid.setItems(listOfPersons);
// define columns and the order as bean properties
// (default: show all)
grid.setColumns("firstName", "lastName", "email");
```

Alternatively, you can define columns programmatically. In this case, you don't need to pass the domain type as a constructor parameter:

```
Grid<Person> grid = new Grid<>();
grid.setItems(listOfPersons);
grid.addColumn(Person::getFirstName)
  .setCaption("First name");
// ...
```

The addColumn method returns a Column object that can be used to further configure the column and the data representation. In the above example, we only configure the caption of the column. With bean based listing your can get a reference to the Column with getColumn("propertyName").

LAZY LOADING DATA IN GRID

The setItems method stores the given data in your server memory for rapid access. If your data grid contains a lot of data and you wish to save some server memory, you can also do a lazy binding with your backend using setDataProvider method:

```
grid.setDataProvider(
  (sortOrders, offset, limit) ->
    service.findAll(offset, limit).stream(),
  () -> service.count()
);
```

The example above uses the easiest solution, in which you only provide two lambda expressions: the first to provide the given range of items and the second one that provides the total number of items available. Alternatively, you can provide a custom implementation of the DataProvider interface.

SELECTION COMPONENTS

Selection components (ComboBox, RadioButtonGroup, ListSelect...) let your users pick one or several selections from a set of given items. Grid is also a selection component. You can use it as a single or multi-selection component using asSingleSelect() or asMultiSelect() method.

To set the available items, you use the same setItems or setDataProvider methods, as with Grid. Most select components by default render the selections using item's toString method. Selecting a String from a list of strings could be accomplished as follows:

```
ComboBox<String> comboBox = new ComboBox<>();
comboBox.setItems("Foo","Bar","Car");
comboBox.addValueChangeListener(e ->
  Notification.show(e.getValue()));
```

If toString presentation doesn't fit to your use case, you can in most select components customize the caption as follows:

```
ComboBox<Person> cb = new ComboBox<>();
cb.setItemCaptionGenerator(p -> p.getFirstName() +
  " " + p.getLastName());
```

A similar concept is available for icons.

BINDER FOR HANDY FORMS

Binder is a helper class that allows you to bind a single Java object's properties to be displayed in multiple Vaadin components. The most typical use case is a form. Binder also supports two-way binding, so that values input by the end user automatically get written back to your domain model.

Binder will also help you to do value conversion and validation for the data. The following binds firstName(String) and age (Integer) to two text fields.

```
Binder<Person> b = new Binder<>();
b.forField(firstNameField)
  // additional configuration
  .asRequired("First name must be defined")
  .bind(Person::getFirstName, Person::setFirstName);
b.forField(ageField)
  .withConverter(new StringToIntegerConverter(
    "Must be valid integer!"))
  .withValidator(integer -> integer > 0,
    "Age must be positive")
  .bind(p -> p.getAge(), (p, i) -> p.setAge(i));
b.setBean(person);
```

Alternatively, you can make an automatic binding to your UI classes member fields. The binding is made based on naming convention, but it can be overridden with @PropertyId annotation.

```
private TextField age = new TextField();
private TextField firstName = new TextField();

private void bindFields(Person person) {
  Binder<Person> b = new Binder<>(Person.class);
  // additional configuration supported
  // also in name based binding
  b.forMemberField(age).withConverter(
    new StringToIntegerConverter("Must be integer"));
  // this binds all found fields
  b.bindInstanceFields(this);
  b.setBean(person);
}
```

Converters and Validators are often written by developers, but there is also a selection of built-in helper classes available. See Figure 2 to see the built-in helpers.

WIDGET INTEGRATION

The easiest way to create new components is composition with the CustomComponent or by extending from some layout component. If that is not enough, you can create an entirely new component by creating a client-side widget (in GWT or JavaScript), a server-side counterpart, and binding the two together with a connector, using a shared state and RPC calls.

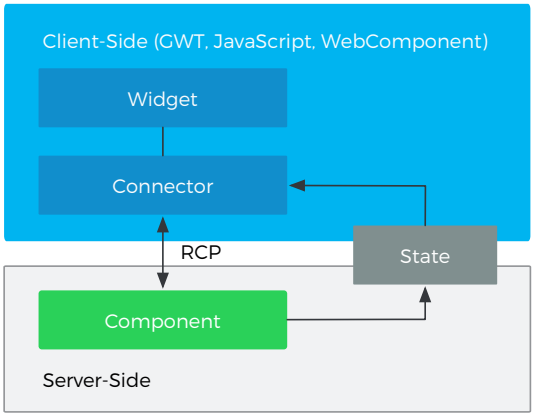


Figure 5: *Integrating a Client-Side widget with a Server-Side API*

Shared state is used for communicating component state from the server-side component to the client-side connector, which should apply them to the widget. The shared state object is serialized by the framework.

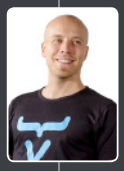
You can make RPC calls both from the client-side to the server-side, typically to communicate user interaction events, and vice versa. To do so, you need to implement an RPC interface.

CREATING A WIDGET PROJECT

You can create widgets directly to your application project, but a much better habit is to create a separate add-on project for your custom widgets and then add dependencies to it from your application projects. Then you can build better tests that are prepared for reuse in your other Vaadin projects or even sharing your add-ons with others via the Directory service.

For widgets, there is a specially tailored project template: vaadin-archetype-widget. Create a project using that archetype and follow its instructions in its README file. The archetype also creates stubs for required classes.

ABOUT THE AUTHOR



MATTI TAHVONEN has a long history in Vaadin R&D: developing the core framework from the dark ages of pure JS client side to the GWT era and creating number of official and unofficial Vaadin add-ons. His current responsibility is to keep you up to date with latest and greatest Vaadin related technologies. When not hacking with Java related technologies, he is most likely in the local bushes, coaching orienteers or driving MTB, or sailing in the Baltic Sea. You can follow him on Twitter – @MattiTahvonen

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com