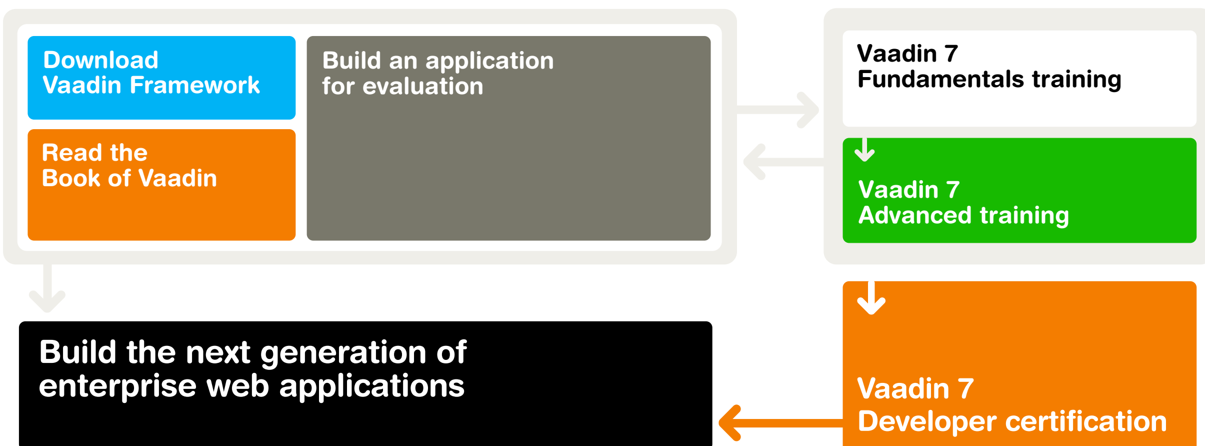
Becoming a Vaadin Expert



How to update yourself

It's not always easy to keep up with the latest Java web development tools and trends. To help you with this, Vaadin offers exceptionally good documentation, tutorials and official hands-on trainings all around the world.

Attend the official Vaadin trainings to learn about Vaadin best practices, the right application architecture and how to best benefit from the latest features. To boost your career take the certification exam and become a Vaadin Certified Developer.

Develop your skills and build amazing Java web applications.

vaadin}>

Brought to you by:

CONTENTS INCLUDE:

- Creating a Server-side UI
- Components
- Themes
- Class Diagram
- Data Binding
- Widget Integration... and more!

Vaadin 7: Modern Web Apps in Java

By: Marko Grönroos

ABOUT VAADIN

Vaadin is a web application development framework that allows you to build web applications much as you would with traditional desktop frameworks, such as AWT or Swing. A UI is built hierarchically from user interface components contained in layout components. User interaction is handled in an event-driven manner.

Vaadin supports both a server-side and a client-side development model. In the server-side model, the application code runs on a server, while the actual user interaction is handled by a client-side engine that runs in the browser. The client-server communications and client-side technologies, such as HTML and JavaScript, are invisible to the developer. The client-side engine runs as JavaScript in the browser, so there is no need to install plug-ins.

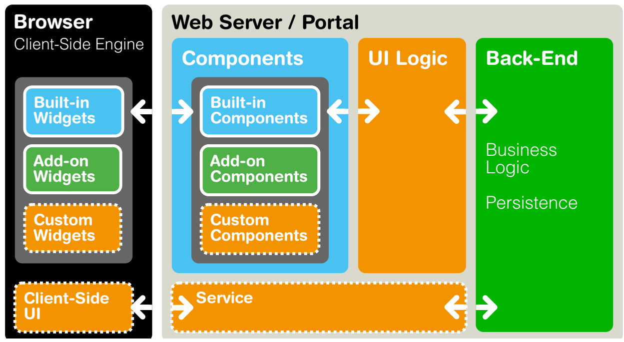


Figure 1: Vaadin Client-Server Architecture

The client-side development model allows building new client-side widgets and user interfaces with the GWT toolkit included in Vaadin. The widgets can be integrated with server-side component counterparts to enable using them in server-side applications. You can also make pure client-side UIs, which can communicate with a back-end service.

CREATING A SERVER-SIDE UI

A server-side Vaadin application consists of one or more UI classes that extend the **com.vaadin.UI** class and implement the **init()** method.

```
@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!"));
    }
}
```

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

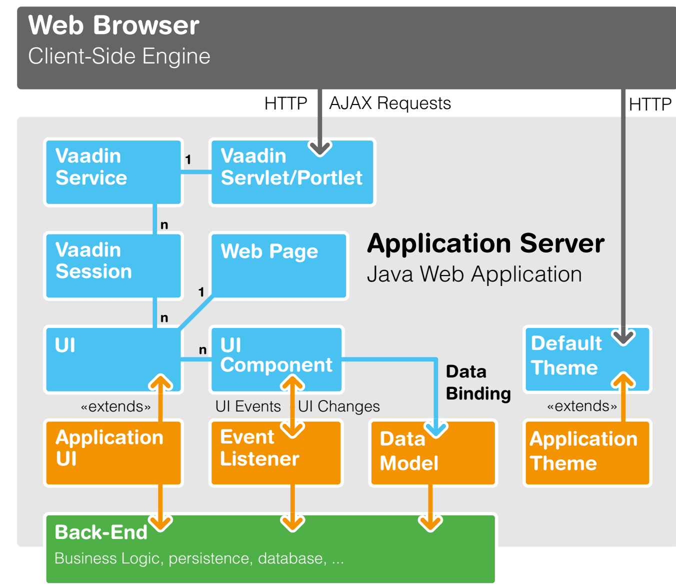
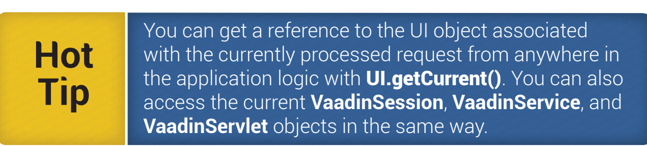


Figure 2: Architecture for Vaadin Applications

You can create a Vaadin application project easily with the Vaadin Plugin for Eclipse, with NetBeans, or with Maven.



Hot 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.

Event Listeners

In the event-driven model, user interaction with user interface components triggers 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(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        Notification.show("Thank You!");
    }
});
layout.addComponent(button);
```

Value changes in a field component can be handled correspondingly with a **ValueChangeListener**. By setting the ***immediate*** property of a component to **true**, user interaction events can be fired immediately when the focus changes. Otherwise, they are delayed until the first immediate interaction, such as a button click.

In addition to the event-driven model, UI changes can be made from the server-side with server push.

Deployment

Vaadin applications are deployed to a Java application server as web applications. A UI runs as a Java Servlet, which needs to be declared in a **web.xml** deployment descriptor, or with the **@WebServlet** and **@VaadinServletConfiguration** annotations in a Servlet 3.0 capable server as follows:

```
@WebServlet(value = "/*", asyncSupported = true)
@VaadinServletConfiguration(
    productionMode = false,
    ui = HelloWorld.class)
public class myServlet extends VaadinServlet {
}
```

The **VaadinServlet** handles server requests and manages user sessions and UIs. All that is normally hidden, but you may need to do some tasks in the custom servlet class. The Eclipse plugin creates the servlet class as an inner class of the UI class. In a Servlet 2.4 capable server, you need to use a **web.xml**.

Vaadin UIs can also be deployed as portlets in a portal.

COMPONENTS

Vaadin components include field, layout, and other components. The component classes and their inheritance hierarchy is illustrated in Figure 4.

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 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.
immediate	If *true*, value changes are communicated immediately to the server-side, usually when the selection changes or the field loses input focus. (Default: *false*)
locale	The current country and/or language for the component. Meaning and use is application-specific for most components. (Default: UI locale)
readOnly	If *true*, the user cannot change the value. (Default: *false*)
visible	Whether the component is actually visible or not. (Default: *true*)

Field Properties

Field properties are defined in the Field interface and the AbstractField base class for fields.

Property	Description
required	Boolean value stating whether a value for the field is required. (Default: *false*)
requiredError	Error message to be displayed if the field is required but empty. Setting the error message is highly recommended for providing the user with information about a failure.

Sizing

The size of components can be set in fixed or relative units in 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 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 fit the content.

Figure 4 shows the default sizing of components.

Notice that a layout with an undefined size must not contain a component with a relative (percentual) size.

Validation

Field values can be validated with **validate()** or **isValid()**. You add validators to a field with **addValidator()**. Fields in a **FieldGroup** can all be validated at once.

Built-in validators are defined in the **com.vaadin.data.validator** package and include:

Validator	Description
CompositeValidator	Combines other validators
DateRangeValidator	The date/time is between a specified range
Double(Range)Validator	The input is a double value (and within a range)
EmailValidator	The input is a valid email address
Integer(Range)Validator	The input is an integer (and within a range)
NullValidator	The input value either is null or is not
RegexpValidator	The input string matches a regular expression
StringLengthValidator	The length of the input string is within a range

You can also implement a custom **Validator** by defining its **validate()** method.

Fields in a **FieldGroup** bound to a **BeanItem** can be validated with the Bean Validation API (JSR-303), if an implementation is included in the class path.

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.

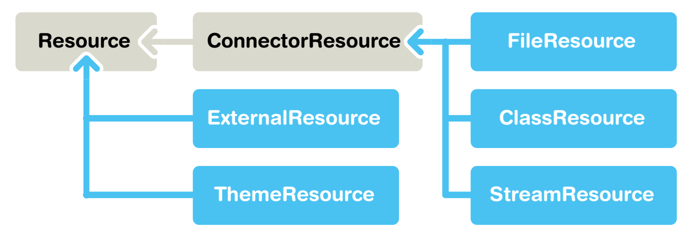


Figure 3: Resource Classes and Interfaces

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

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 set 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. Their default sizes are marked in Figure 4, and can be changed with the sizing methods described earlier. 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!

Margins

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

Setting **setSpacing(true)** enables spacing between the layout slots. The spacing can be adjusted with CSS as the width or height of 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 a layout cell 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);
```

Expand Ratios

The ordered layouts and **GridLayout** support expand ratios that 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);
```

Also **Table** supports expand ratios for columns.

Custom Layout

The **CustomLayout** component allows the use of a HTML template that contains location tags for components, such as **<div location="hello"/>**. The components are inserted in the location elements with the **addComponent()** method as shown below:

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

The layout name in the constructor refers to a corresponding **.html** file in the **layouts** subfolder in the theme folder, in the above example **layouts/mylayout.html**. See Figure 5 for the location of the layout template file.

ADD-ON COMPONENTS

Hundreds of Vaadin add-on components are available from the Vaadin Directory, both free and commercial. You can download them as an installation package or retrieve with Maven, Ivy, or a compatible dependency manager. Please follow the instructions given in Vaadin Directory at http://vaadin.com/directory.

Most add-ons include widgets, which need to be compiled to a project widget set. In an Eclipse project created with the Vaadin Plugin for Eclipse, select the project and click the **Compile Vaadin widgets** button in the tool bar.

THEMES

Vaadin allows customizing 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 (under **WebContent** in Eclipse projects). This location is fixed and the **VAADIN** folder specifies that these are static resources specific to Vaadin. The structure is illustrated in Figure 5.

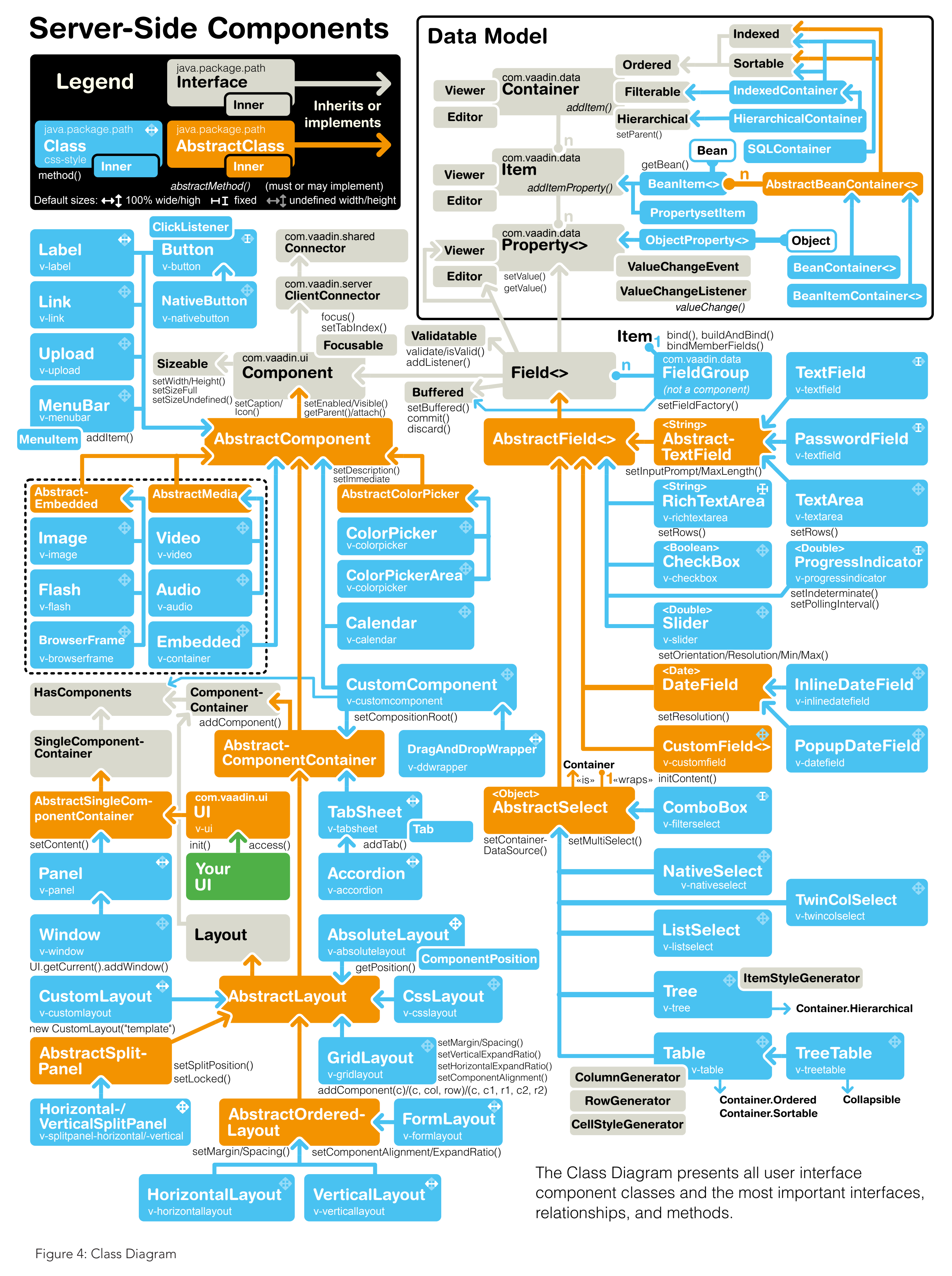
Each theme has its own folder with the name of the theme. A theme folder must contain a **styles.scss** (for Sass) or a **styles.css** (for plain CSS) style sheet. Custom layouts must be placed in the **layouts** sub-folder, but other contents may be named freely.

Custom themes need to inherit a base theme. The built-in themes in Vaadin 7 are **reindeer**, **runo**, and **chameleon**, as well as a **base** theme on which the other built-in themes are based.

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 on-the-fly during development, and can also be used for building production packages.

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**.



The Class Diagram presents all user interface component classes and the most important interfaces, relationships, and methods.

Figure 4: Class Diagram

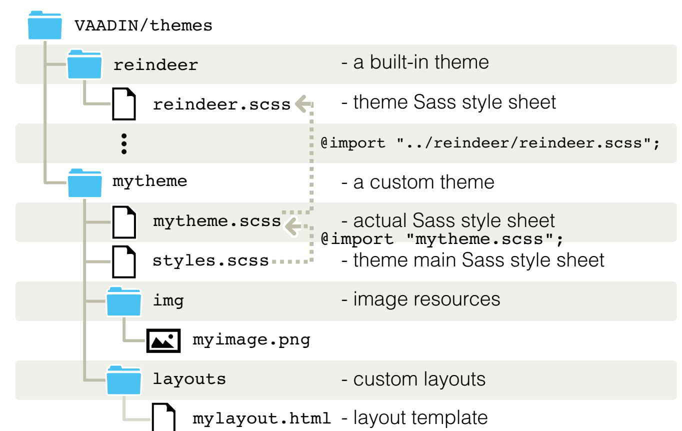


Figure 5: Theme Contents

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

```
@import "mytheme.scss";

/* Enclose theme in a nested style with the theme name. */
.mytheme {
  @include mytheme; /* Use the mixin defined in mytheme.scss */
}
```

The **mytheme.scss**, which contains the actual theme rules, would define the theme as a Sass mixin as follows:

```
/* Import a base theme.*/
@import "../reindeer/reindeer.scss";

@mixin mytheme {
  /* Include all the styles from the base theme */
  @include reindeer;

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

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.

Applying a Theme

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

```
@Theme("mytheme")
@Title("My Vaadin UI")
public class MyUI extends com.vaadin.UI {
    @Override
    protected void init(VaadinRequest request) {
        …
        // Display the greeting
        Label label = new Label("This is My Component!");
        label.addStyleName("mycomponent");
        content.addComponent(label);
    }
```

DATA BINDING

Vaadin allows binding components directly to data. The data model, illustrated in Figure 4, is based on interfaces on three levels of containment: properties, items, and containers.

Properties

The **Property** interface provides access to a value of a specific class with the **setValue()** and **getValue()** methods.

All field components provide access to their value through the **Property** interface, and the ability to listen for value changes with a **Property.ValueChangeListener**. The field components hold their value in an internal data source by default, but you can bind them to any data source with **setPropertyDataSource()**. Conversion between the field type and the property type is handled with a **Converter**.

For selection components, the property value points to the item identifier of the current selection, or a collection of item identifiers in the ***multiSelect*** mode.

The **ObjectProperty** is a wrapper that allows binding any object to a component as a property.

Items

An item is an ordered collection of properties. The **Item** interface also associates a *property ID* with each property. Common uses of items include form data, **Table** rows, and selection items.

The **BeanItem** is a special adapter that allows accessing any Java bean (or POJO with proper setters and getters) through the **Item** interface.

Forms can be built by binding fields to an item using the **FieldGroup** utility class.

Containers

A *container* is a collection of items. It allows accessing the items by an *item ID* associated with each item.

Common uses of containers include selection components, as defined in the **AbstractSelect** class, especially the **Table** and **Tree** components. (The current selection is indicated by the *property* of the field, which points to the item identifier of the selected item.)

Vaadin includes the following built-in container implementations:

Container Class	Description
IndexedContainer	Container with integer index keys
BeanItemContainer	Bean container that uses bean as item ID
BeanContainer	Bean container with explicit item ID type
HierarchicalContainer	Tree-like container, used especially by the **Tree** and **TreeTable** components
FilesystemContainer	Direct access to the file system
SQLContainer	Binds to an SQL database
JPAContainer	Binds to a JPA implementation

Also, all components that can be bound to containers are containers themselves.

Buffering

All field components implement the **Buffered** interface that allows buffering user input before it is written to the data source.

Method	Description
commit()	Validates the field or field group and, if successful, writes the buffered data to the data source
discard()	Discards the buffered data and re-reads the data from the data source
setBuffered()	Enables buffering when *true*. Default is *false*.

The **FieldGroup** class has the same methods to manage buffering for all bound fields.

WIDGET INTEGRATION

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

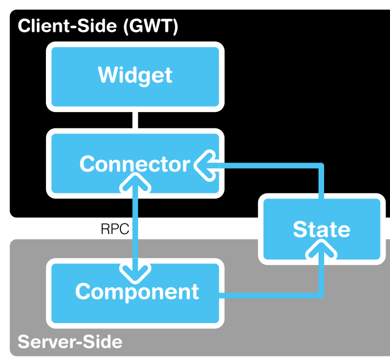


Figure 6: Integrating a Client-Side widget with a Server-Side API

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.

Defining a Widget Set

A ***widget set*** is a collection of widgets that, together with inherited widget sets and the communication framework, forms the Client-Side Engine of Vaadin when compiled with the GWT Compiler into JavaScript.

A widget set is defined in a **.gwt.xml** GWT Module Descriptor. You need to specify at least one inherited base widget set, typically the **DefaultWidgetSet** or other widget sets.

```
<module>
    <inherits name="com.vaadin.DefaultWidgetSet" />
</module>
```

The client-side source files must normally be located in a **client** sub-package in the same package as the descriptor.

You can associate a stylesheet with a widget set with the **<stylesheet>** element in the **.gwt.xml** descriptor:

```
<stylesheet src="mycomponent/styles.css"/>
```

Widget Project Structure

Figure 7 illustrates the source code structure of a widget project, as created with the Vaadin Plugin for Eclipse.

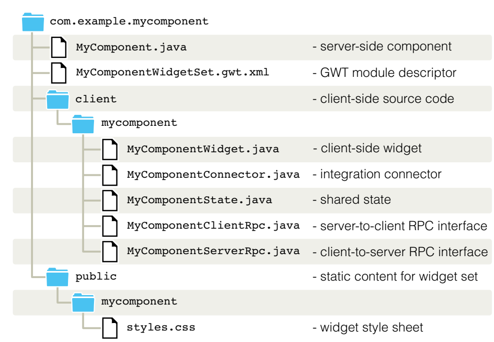


Figure 7: Widget Project Source Structure

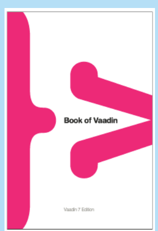
ABOUT THE AUTHOR

Marko Grönroos is a professional writer and software developer working at Vaadin Ltd in Turku, Finland. He has been involved in web application development since 1994 and has worked on several application development frameworks in C, C++, and Java. He has been active in many open source software projects and holds an M.Sc. degree in Computer Science from the University of Turku.

Website: http://iki.fi/magi

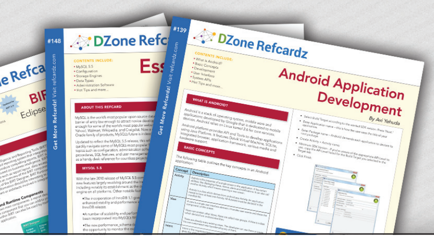
Blog: http://markogronroos.blogspot.com/

RECOMMENDED BOOK



Book of Vaadin is a comprehensive documentation of Vaadin. It shows how to get started, gives a good overview of the features, and tutors you through advanced aspects of the framework.

READ NOW
http://vaadin.com/book



Browse our collection of over 150 Free Cheat Sheets

Free PDF

Upcoming Refcardz

C++
CSS3
OpenLayers
Regex

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. **"DZone is a developer's dream",** says PC Magazine.

Copyright © 2013 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

$7.95

Version 1.0