

BOOK OF VAADIN

VAADIN FRAMEWORK 8

Volume 1

Vaadin Team

2017
Vaadin Ltd

BOOK OF VAADIN

Vaadin Team

Vaadin Ltd

Revision 1

Published: 2017-01-10

Vaadin Framework 8.0

This book can be downloaded for free at

<https://vaadin.com/book>

Published by

Vaadin Ltd

Ruukinkatu 2-4

20540 Turku

Finland

Abstract

Vaadin is a web application development framework that enables developers to build high-quality user interfaces with Java. It provides a set of ready-to-use user interface components and allows creating your own components. The focus is on ease-of-use, re-usability, extensibility, and meeting the requirements of large enterprise applications.

Copyright © 2000-2017 Vaadin Ltd

All rights reserved. This work is licensed under the Creative Commons CC-BY-ND License Version 2.0.

Printed by Bookwell Oy in Juva, Finland, 2017.

Preface	v
1. Who is This Book For?	v
2. Organization of This Book	vi
3. Supplementary Material	ix
4. Support	x
5. Acknowledgements	x
6. About Vaadin Ltd	xi
Chapter 1. Introduction	1
1.1. Overview	1
1.2. Example Application Walkthrough	3
1.3. Support for IDEs	5
1.4. Goals and Philosophy	6
1.5. Background	7
Chapter 2. Installing the Development Toolchain	11
2.1. Overview	11
2.2. A Reference Toolchain	13
2.3. Installing Java SDK	15
2.4. Installing a Web Server	16
2.5. Installing the Eclipse IDE and Plugin	17
2.6. Installing the NetBeans IDE and Plugin	25
2.7. Installing and Configuring IntelliJ IDEA	28
Chapter 3. Creating a Vaadin Application	31
3.1. Overview	31
3.2. Vaadin Libraries	32
3.3. Overview of Maven Archetypes	34
3.4. Creating and Running a Project in Eclipse	35
3.5. Creating a Project with the NetBeans IDE	43
3.6. Creating a Project with IntelliJ IDEA	48
3.7. Creating a Project with Maven	50
3.8. Vaadin Installation Package	53
3.9. Using Vaadin with Scala	53
Chapter 4. Architecture	55
4.1. Overview	55
4.2. Technological Background	60
4.3. Client-Side Engine	64
4.4. Events and Listeners	66
Chapter 5. Writing a Server-Side Web Application	69
5.1. Overview	69
5.2. Building the UI	74

5.3. Designing UIs Declaratively	79
5.4. Handling Events with Listeners	86
5.5. Images and Other Resources	87
5.6. Handling Errors	92
5.7. Notifications	95
5.8. Application Lifecycle	99
5.9. Deploying an Application	107
Chapter 6. User Interface Components	117
6.1. Overview	118
6.2. Interfaces and Abstractions	121
6.3. Common Component Features	123
6.4. Field Components	137
6.5. Selection Components	141
6.6. Component Extensions	145
6.7. Label	146
6.8. Link	149
6.9. TextField	152
6.10. TextArea	156
6.11. PasswordField	158
6.12. RichTextArea	159
6.13. Date Input with DatePicker	161
6.14. Button	167
6.15. CheckBox	169
6.16. ComboBox	170
6.17. ListSelect	173
6.18. NativeSelect	174
6.19. CheckBoxGroup and RadioButtonGroup	175
6.20. TwinColSelect	177
6.21. Grid	179
6.22. MenuBar	196
6.23. Upload	199
6.24. ProgressBar	202
6.25. Slider	203
6.26. PopupView	205
6.27. Composition with CustomComponent	207
6.28. Composite Fields with CustomField	209
6.29. Embedded Resources	210
Chapter 7. Managing Layout	215
7.1. Overview	216
7.2. UI, Window, and Panel Content	218
7.3. VerticalLayout and HorizontalLayout	219

7.4. GridLayout	227
7.5. FormLayout	231
7.6. Panel	233
7.7. Sub-Windows	236
7.8. HorizontalSplitPanel and VerticalSplitPanel	240
7.9. TabSheet	243
7.10. Accordion	247
7.11. AbsoluteLayout	249
7.12. CssLayout	252
7.13. Layout Formatting	256
7.14. Custom Layouts	264
Chapter 8. Vaadin Designer	267
8.1. Overview	267
8.2. Installing in Eclipse	269
8.3. Installing in IntelliJ IDEA	270
8.4. Licensing	270
8.5. Getting Started	271
8.6. Designing	274
8.7. Theming and Styling	283
8.8. Wiring It Up	285
8.9. Templates	289
8.10. Limitations	292

Preface

This book provides an overview of the Vaadin Framework and covers the most important topics that you might encounter while developing applications with it. The book is a compilation of the most important documentation available in Vaadin Docs at vaadin.com/docs. A more detailed documentation of the individual classes, interfaces, and methods is given in the Vaadin API Reference.

This edition covers Vaadin Framework 8 released in early 2017. For the most current documentation, please visit the Vaadin Docs available at vaadin.com/docs.

Writing this manual is an ongoing work and it is rarely completely up-to-date with the quick-evolving product. You can also find PDF and EPUB versions of the book at the website. You may find the other versions more easily searchable than the printed book. The web edition also has some additional technical content, such as some example code and additional sections that you may need when actually doing development. The purpose of the slightly abridged print edition is more to be an introductory textbook to Vaadin, and still fit in your pocket.

1. Who is This Book For?

This book is intended for software developers who use or are considering to use Vaadin to develop web applications.

The book assumes that you have some experience with programming in Java. If not, it is at least as easy to begin learning Java with Vaadin as with any other UI framework. No knowledge of AJAX is needed as it is well hidden from the developer.

You may have used some desktop-oriented user interface frameworks for Java, such as AWT, Swing, or SWT, or a library such as Qt for C++. Such knowledge is useful for understanding the scope of Vaadin, the event-driven programming model.

and other common concepts of UI frameworks, but not necessary.

If you do not have a web graphics designer at hand, knowing the basics of HTML and CSS can help so that you can develop presentation themes for your application. A brief introduction to CSS is provided. Knowledge of Google Web Toolkit (GWT) may be useful if you develop or integrate new client-side components.

2. Organization of This Book

The Book of Vaadin gives an introduction to what Vaadin is and how you use it to develop web applications.

Volume 1

Chapter 1. *Introduction*

The chapter gives an introduction to the application architecture supported by Vaadin, the core design ideas behind the framework, and some historical background.

Chapter 2. *Installing the Development Toolchain*

This chapter gives instructions for installing a toolchain for developing Vaadin applications. A toolchain typically includes an IDE, a Vaadin plugin for the IDE, and a development server. We cover the Eclipse IDE, NetBeans IDE, and IntelliJ IDEA. After this, you should have all the basic tools set up.

Chapter 3. *Creating a Vaadin Application*

This chapter gives practical instructions for creating a Vaadin application project in an IDE or otherwise, and for running it in an integrated development server.

Chapter 4. *Architecture*

This chapter gives an introduction to the architecture of Vaadin and its major technologies, including AJAX, Google Web Toolkit, and event-driven programming.

Chapter 5. *Writing a Server-Side Web Application*

This chapter gives all the practical knowledge required for creating applications with Vaadin, such as window management, application lifecycle, deployment in a

servlet container, and handling events, errors, and resources.

Chapter 6. User Interface Components

This chapter gives the basic usage documentation for all the (non-layout) user interface components in Vaadin and their most significant features. The component sections include examples for using each component, as well as for styling with CSS/Sass.

Chapter 7. Managing Layout

This chapter describes the layout components, which are used for managing the layout of the user interface, just like in any desktop application frameworks.

Chapter 8. Vaadin Designer

This chapter gives instructions for using Vaadin Designer, a visual tool for the Eclipse IDE for creating composite designs, such as for UIs, views, or other composites.

Volume 2:

Chapter 9. Themes

This chapter gives an introduction to Cascading Style Sheets (CSS) and Sass and explains how you can use them to build custom visual themes for your application.

Chapter 10. Binding Components to Data

This chapter gives an overview of the built-in data model of Vaadin, consisting of properties, items, and containers.

Chapter 11. Advanced Web Application Topics

This chapter provides many special topics that are commonly needed in applications, such as opening new browser windows, embedding applications in regular web pages, low-level management of resources, shortcut keys, debugging, etc.

Chapter 12. Portal Integration

This chapter describes the development of Vaadin applications as portlets which you can deploy to any portal supporting Java Portlet API 2.0 (JSR-286). The chapter

also describes the special support for Liferay and the Control Panel, IPC, and WSRP add-ons.

Chapter 13, Client-Side Vaadin Development

This chapter gives an introduction to creating and developing client-side applications and widgets, including installation, compilation, and debugging.

Chapter 14, Client-Side Applications

This chapter describes how to develop client-side applications and how to integrate them with a back-end service.

Chapter 15, Client-Side Widgets

This chapter describes the built-in widgets (client-side components) available for client-side development. The built-in widgets include Google Web Toolkit widgets as well as Vaadin widgets.

Chapter 16, Integrating with the Server-Side

This chapter describes how to integrate client-side widgets with their server-side counterparts for the purpose of creating new server-side components. The chapter also covers integrating JavaScript components.

Chapter 17, Using Vaadin Add-ons

This chapter gives instructions for downloading and installing add-on components from the Vaadin Directory.

Chapter 18, Vaadin Charts

This chapter documents the use of the Vaadin Charts add-on component for interactive charting with many diagram types. The add-on includes the Chart and Timeline components.

Chapter 19, Vaadin Spreadsheet

This chapter gives documentation of the Vaadin Spreadsheet add-on, which provides a Microsoft Excel compatible spreadsheet component.

Chapter 20. Vaadin *TestBench*

This chapter gives the complete documentation of using the Vaadin TestBench tool for recording and executing user interface regression tests of Vaadin applications.

3. Supplementary Material

The Vaadin websites offer plenty of material that can help you understand what Vaadin is, what you can do with it, and how you can do it.

Demo Applications

The most important demo application for Vaadin is the Sampler, which demonstrates the use of all basic components and features. You can find it at <https://demo.vaadin.com/>.

Address Book Tutorial

The Address Book is a sample application accompanied with a tutorial that gives detailed step-by-step instructions for creating a real-life web application with Vaadin. You can find the tutorial from our website.

Developer Website

Vaadin Developer's Site in Github at <https://github.com/vaadin> provides various online resources for Vaadin Framework and some other products. You can find there the source code, issue system for reporting bugs and suggesting improvements, releases, activity timeline, and so forth.

You are more than welcome to contribute! You can find the contribution instructions in GitHub.

Online Documentation

You can find Vaadin documentation online at <https://vaadin.com/docs>. Lots of additional material, including technical HOWTOs, answers to Frequently Asked Questions, and other documentation is available at Vaadin web-site.

4. Support

Stuck with a problem? No need to lose your hair over it, the Vaadin Framework developer community and the Vaadin company offer support to all of your needs.

Community Support Forum

You can find the developer community forum at <https://vaadin.com/forum>. Please use the forum to discuss any problems you might encounter, wishes for features, and so on. The answer to your problems may already lie in the forum archives, so searching the discussions is always the best way to begin.

Report Bugs

If you have found a possible bug in Vaadin, the demo applications, or the documentation, please report it by filing an issue at <https://github.com/vaadin/framework/issues>. You may want to check the existing issues before filing a new one. You can make an issue to make a request for a new feature as well, or to suggest modifications to an existing feature.

Commercial Support

Vaadin Ltd. offers full commercial support and training services for the Vaadin Framework and related products. Read more about the commercial products at <https://vaadin.com/pro> for details.

5. Acknowledgements

Much of the book is the result of close work within the development team at Vaadin Ltd. Joonas Lehtinen, CEO and a founder of Vaadin Ltd, wrote the first outline of the book, which became the basis for the first two chapters. Since then, Marko Grönroos was primary author and editor for some time. Recently the development teams in Vaadin have taken over writing the content in this book.

The contributors are (in rough chronological order):

Marko Grönroos

Joonas Lehtinen

Jani Laakso	Jouni Koivumiita
Matti Tahvonen	Artur Signell
Marc Englund	Henri Sara
Jonatan Kronqvist	Mikael Grankvist
Teppo Kurki	Tomi Virtanen
Risto Yrjänä	John Ahlroos
Petter Holmström	Leif Åstrand
Guillermo Alvarez	Dmitrii Rogozin
Markus Koivisto	Elijah Motorny
Johannes Dahlström	Teemu Suo-Anttila
Sami Ekblad	Denis Anisimov
Manuel Carrasco Moñino	Pekka Hyvönen
Andrei Korzhevskii	Anssi Tuominen
Aleksi Hietanen	Stepan Zolotarev
Tien Nguyen	Henri Muurimaa

6. About Vaadin Ltd

Vaadin Ltd is a software company that helps developers make web apps their users love. Founded in 2000 with headquarters in Finland and offices in California and Germany, over 130 Vaadin experts help companies around the world in building applications on top of the Vaadin technologies.

Products include the Vaadin Framework, which this book is all about, as well as Vaadin Core Elements, a web component library for developers preferring to work in JavaScript. These free and open source products are supplemented with professional tools, support, training, and consulting services.

Chapter 1

Introduction

1.1. Overview	1
1.2. Example Application Walkthrough	3
1.3. Support for IDEs	5
1.4. Goals and Philosophy	6
1.5. Background	7

This chapter gives a brief introduction to software development with Vaadin. We also try to give some insight about the design philosophy behind Vaadin and its history.

1.1. Overview

Vaadin Framework is a Java web application development framework that is designed to make creation and maintenance of high quality web-based user interfaces easy. Vaadin supports two different programming models: server-side and client-side. The server-driven programming model is the more powerful one. It lets you forget the web and program user interfaces much like you would program a desktop application with conventional Java toolkits such as AWT, Swing, or SWT. But easier.

While traditional web programming is a fun way to spend your time learning new web technologies, you probably want to be productive and concentrate on the application logic. The server-side Vaadin framework takes care of managing the user interface in the browser and the AJAX communica-

tions between the browser and the server. With the Vaadin approach, you do not need to learn and deal directly with browser technologies, such as HTML or JavaScript.

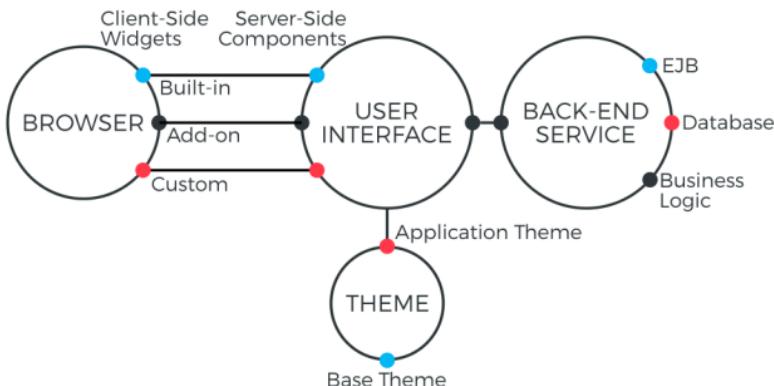


Figure 1.1. Vaadin application architecture

Figure 1.1. "Vaadin application architecture" illustrates the basic architectures of web applications made with Vaadin. The server-side application architecture consists of the *server-side framework* and a *client-side engine*. The engine runs in the browser as JavaScript code, rendering the user interface, and delivering user interaction to the server. The UI logic of an application runs as a Java Servlet in a Java application server.

As the client-side engine is executed as JavaScript in the browser, no browser plugins are needed for using applications made with Vaadin. This gives it an edge over frameworks based on Flash, Java Applets, or other plugins. Vaadin relies on the support of Google Web Toolkit for a wide range of browsers, so that the developer does not need to worry about browser support.

Because HTML, JavaScript, and other browser technologies are essentially invisible to the application logic, you can think of the web browser as only a thin client platform. A thin client displays the user interface and communicates user events to the server at a low level. The control logic of the user interface runs on a Java-based web server, together with your business logic. By contrast, a normal client-server architecture with a dedicated client application would include a lot of application

specific communications between the client and the server. Essentially removing the user interface tier from the application architecture makes our approach a very effective one.

Behind the server-driven development model, Vaadin makes the best use of AJAX (*Asynchronous JavaScript and XML*, see Section 4.2.3, "AJAX" for a description) techniques that make it possible to create Rich Internet Applications (RIA) that are as responsive and interactive as desktop applications.

In addition to the server-side Java application development, you can develop on the client-side by making new widgets in Java, and even pure client-side applications that run solely in the browser. The Vaadin client-side framework includes Google Web Toolkit (GWT), which provides a compiler from Java to the JavaScript that runs in the browser, as well a full-featured user interface framework. With this approach, Vaadin is pure Java on both sides.

Vaadin uses a client-side engine for rendering the user interface of a server-side application in the browser. All the client-server communications are hidden well under the hood. Vaadin is designed to be extensible, and you can indeed use any 3rd-party widgets easily, in addition to the component repertoire offered in Vaadin. In fact, you can find hundreds of add-ons in the Vaadin Directory.

Vaadin allows flexible separation between the appearance, structure, and interaction logic of the user interface. You can design the layouts either programmatically or declaratively, at the level of your choosing. The final appearance is defined in *themes* in CSS or Sass, as described in Chapter 9, *Themes*.

We hope that this is enough about the basic architecture and features of Vaadin for now. You can read more about it later in Chapter 4, *Architecture*, or jump straight to more practical things in Chapter 5, *Writing a Server-Side Web Application*.

1.2. Example Application Walkthrough

Let us follow the long tradition of first saying "Hello World!" when learning a new programming framework. First, using the primary server-side API.

```
import com.vaadin.server.VaadinRequest;
import com.vaadin.ui.Label;
import com.vaadin.ui.UI;

{@Title("My UI")
public class HelloWorld extends 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!"));

        // Have a clickable button
        content.addComponent(new Button("Push Me!",
            click -> Notification.show("Pushed!")));
    }
}
```

A Vaadin application has one or more *UIs* that extend the **com.vaadin.ui.UI** class. A UI is a part of the web page in which the Vaadin application runs. An application can have multiple UIs in the same page, especially in portals, or in different windows or tabs. A UI is associated with a user session, and a session is created for each user who uses the application. In the context of our Hello World UI, it is sufficient to know that the underlying session is created when the user first accesses the application by opening the page, and the `init()` method is invoked at that time.

The page title, which is shown in the caption of the browser window or tab, is defined with an annotation. The example uses a layout component as the root content of the UI, as that is the case with most Vaadin applications, which normally have more than one component. It then creates a new **Label** user interface component, which displays simple text, and sets the text to "Hello World!". The label is added to the layout.

The example also shows how to create a button and handle button click events. Event handling is described in Section 4.4, "Events and Listeners" and on the practical side in Section 5.4, "Handling Events with Listeners". In Java 8, you can implement listeners with lambda expressions, which simplifies the handler code significantly.

The result of the Hello World application, when opened in a browser, is shown in Figure 1.2, "Hello World Application".

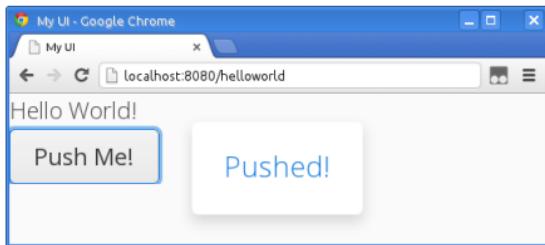


Figure 1.2. Hello World Application

To run a program, you need to package it as a web application WAR package and deploy it to a server, as explained in Section 5.9, "Deploying an Application". During development, you typically deploy to an application server integrated with the IDE.

1.3. Support for IDEs

While Vaadin is not bound to any specific IDE, and you can in fact easily use it without any IDE altogether, we provide special support for the Eclipse IDE, IntelliJ IDEA, and the NetBeans IDE, which have become the most used environment for Java development.

An official Vaadin plug-in is available for the Eclipse and NetBeans IDEs. It helps in:

- creating new Vaadin projects,
- creating custom themes,
- creating custom client-side widgets,
- downloading add-ons from the Vaadin directory, and
- easily upgrading to a newer version of the Vaadin library.

Availability of the features depends on the IDE. The ultimate edition of IntelliJ IDEA comes with built-in support for Vaadin.

Vaadin Designer is a commercial plug-in available for the Eclipse IDE and IntelliJ IDEA. It enables visual editing of declarative designs that you can use in your applications. See Chapter 8, *Vaadin Designer* for more information.

Using the Vaadin plug-in is the recommended way of installing Vaadin for development. Installing the IDEs and the plug-ins is covered in Chapter 2, *Installing the Development Toolchain*. The creation of a new Vaadin project with each IDE is covered in Chapter 3, *Creating a Vaadin Application*.

1.4. Goals and Philosophy

Simply put, Vaadin's ambition is to be the best possible tool when it comes to creating web user interfaces for business applications. It is easy to adopt, as it is designed to support both entry-level and advanced programmers, as well as usability experts and graphic designers.

When designing Vaadin, we have followed the philosophy inscribed in the following rules.

1.4.1. Right tool for the right purpose

Because our goals are high, the focus must be clear. Vaadin is designed for creating web applications. It is not designed for creating websites or advertisement demos. You may find, for example, JSP/JSF more suitable for such purposes.

1.4.2. Simplicity and maintainability

We have chosen to emphasize robustness, simplicity, and maintainability. This involves following the well-established best practices in user interface frameworks and ensuring that our implementation represents an ideal solution for its purpose without clutter or bloat.

1.4.3. Choice between declarative and dynamic UIs

The Web is inherently document-centered and very much bound to the declarative presentation of user interfaces. Vaadin allows for declarative designs of views, layouts, and even entire UIs. Vaadin Designer enables creating such designs

visually. Nevertheless, the programmatic approach by building the UIs from Java components frees the programmer from its limitations. To create highly dynamic views, it is more natural to create them by programming.

1.4.4. Tools should not limit your work

There should not be any limits on what you can do with the framework: if for some reason the user interface components do not support what you need to achieve, it must be easy to add new ones to your application. When you need to create new components, the role of the framework is critical: it makes it easy to create re-usable components that are easy to maintain.

1.5. Background

The Vaadin Framework was not written overnight. After working with web user interfaces since the beginning of the Web, a group of developers got together in 2000 to form IT Mill. The team had a desire to develop a new programming paradigm that would support the creation of real user interfaces for real applications using a real programming language.

The library was originally called Millstone Library. The first version was used in a large production application that IT Mill designed and implemented for an international pharmaceutical company. IT Mill made the application already in the year 2001 and it is still in use. Since then, the company has produced dozens of large business applications with the library and it has proven its ability to solve hard problems easily. Millstone 3 was released as open source in 2002.

Progress has often required hard decisions to avoid carrying unnecessary legacy burden far into the future. Nevertheless, our aim has always been to keep migrations easy.

1.5.1. Release 4 with Single-Page Rendering

The next generation of the library, IT Mill Toolkit 4, was released in 2006. It introduced an entirely new AJAX-based presentation engine. This allowed the development of AJAX applica-

tions without the need to worry about communications between the client and the server.

1.5.2. Release 5 Powered by GWT

IT Mill Toolkit 5, released initially at the end of 2007, took a significant step further into AJAX. The client-side rendering of the user interface was completely rewritten using GWT, the Google Web Toolkit.

IT Mill Toolkit 5 introduced many significant improvements both in the server-side API and in the functionality. Rewriting the Client-Side Engine with GWT allowed the use of Java both on the client and the server-side. The transition from JavaScript to GWT made the development and integration of custom components and customization of existing components much easier than before, and it also allows easy integration of existing GWT components. The adoption of GWT on the client-side did not, by itself, cause any changes in the server-side API, because GWT is a browser technology that is hidden well behind the API. Also theming was completely revised in IT Mill Toolkit 5.

The Release 5 was published under the Apache License 2, an unrestrictive open source license, to create faster expansion of the user base and to make the formation of a developer community possible.

1.5.3. Birth of Vaadin Release 6

IT Mill Toolkit was renamed as *Vaadin Framework*, or Vaadin in short, in spring 2009. Later IT Mill, the company, was also renamed as Vaadin Ltd. Vaadin means an adult female semi-domesticated mountain reindeer in Finnish.

With Vaadin 6, the number of developers using the framework exploded. Together with the release, the Vaadin Plugin for Eclipse was released, helping the creation of Vaadin projects. The introduction of Vaadin Directory in early 2010 gave it a further boost, as the number of available components multiplied almost overnight. Many of the originally experimental components have since then matured and are now used by thousands of developers. In 2013, we are seeing tremendous

growth in the ecosystem around Vaadin. The size of the user community, at least if measured by forum activity, has already gone past the competing server-side frameworks and even GWT.

1.5.4. The Major Revision with Vaadin 7

Vaadin 7 was a major revision that changed the Vaadin API much more than Vaadin 6 did. It became more web-oriented than Vaadin 6 was. We are doing everything we can to help Vaadin rise high in the web universe. Some of this work is easy and almost routine - fixing bugs and implementing features. But going higher also requires standing firmer. That was one of the aims of Vaadin 7 - redesigning the product so that the new architecture enables Vaadin to reach over many long-standing challenges.

Inclusion of the Google Web Toolkit in Vaadin 7 was a significant development, as it meant that Vaadin now provides support for GWT as well. When Google opened the GWT development in summer 2012, Vaadin (the company) joined the new GWT steering committee. As a member of the committee, Vaadin can work towards the success of GWT as a foundation of the Java web development community.

1.5.5. Vaadin Framework 8 with New Data Binding API

The biggest change in Vaadin Framework 8 is the complete modernization of the data binding API. Binding components to data sources is one of the core features of the Vaadin Framework, as it eliminates the need to explicitly shuffle data between components and data objects, typically beans. The old data model was designed in time before Java features such as generics. While the data model was improved over the years, it was fundamentally outdated and complex to use. The new data binding API works much more fluently in Java 8, especially with Java 8 features such as lambda expressions and streams. Consequently, to be able to fully use the new features of Java 8, we have raised the requirements from Java 6 to 8. The change should make Vaadin Framework up to date with the most current Java technologies used by developers.

Chapter 2

Installing the Development Toolchain

2.1. Overview	11
2.2. A Reference Toolchain	13
2.3. Installing Java SDK	15
2.4. Installing a Web Server	16
2.5. Installing the Eclipse IDE and Plugin	17
2.6. Installing the NetBeans IDE and Plugin	25
2.7. Installing and Configuring IntelliJ IDEA	28

This chapter gives practical instructions for installing the development tools.

2.1. Overview

You can develop Vaadin applications in essentially any development environment that has the Java SDK and deploys to a Java Servlet container. You can use Vaadin with any Java IDE or no IDE at all. Vaadin has special support for the Eclipse and NetBeans IDEs, as well as for IntelliJ IDEA.

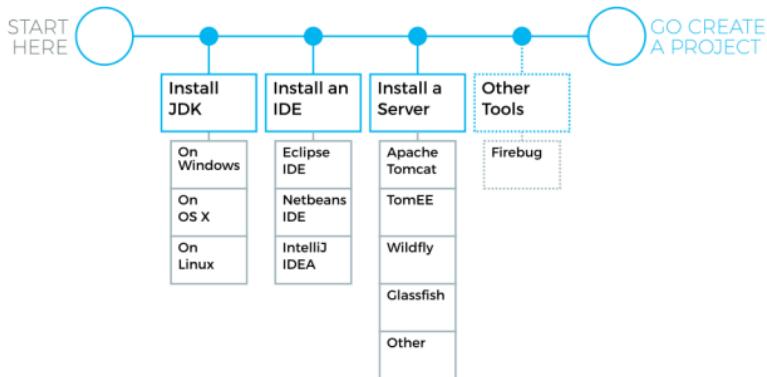


Figure 2.1. Vaadin installation steps

Managing Vaadin and other Java libraries can get tedious to do manually, so using a build system that manages dependencies automatically is advised. Vaadin is distributed in the Maven central repository, and can be used with any build or dependency management system that can access Maven repositories, such as Ivy or Gradle, in addition to Maven.

Vaadin has a multitude of installation options for different IDEs and dependency managers. You can also install it from an installation package:

- With the Eclipse IDE, use the Vaadin Plugin for Eclipse, as described in [Vaadin Plugin for Eclipse](#)
- With the Vaadin plugin for NetBeans IDE ([Section 3.5, "Creating a Project with the NetBeans IDE"](#)) or IntelliJ IDEA
- With Maven, Ivy, Gradle, or other Maven-compatible dependency manager, under Eclipse, NetBeans, IDEA, or using command-line, as described in [Section 3.7, "Creating a Project with Maven"](#)
- From installation package without dependency management, as described in [Section 3.8, "Vaadin Installation Package"](#)

2.2. A Reference Toolchain

This section presents a reference development environment. Vaadin supports a wide variety of tools, so you can use any IDE for writing the code, almost any Java web server for deploying the application, most web browsers for using it, and any operating system platform supported by Java.

In this example, we use the following toolchain:

- Windows, Linux, or Mac OS X
- Oracle Java SE 8
- Eclipse IDE for Java EE Developers
- Apache Tomcat 8.0 (Core)
- Google Chrome browser
- Vaadin Framework

The above reference toolchain is a good choice of tools, but you can use almost any tools you are comfortable with.

We recommend using Java 8 for Vaadin development, but you need to make sure that your entire toolchain supports it.

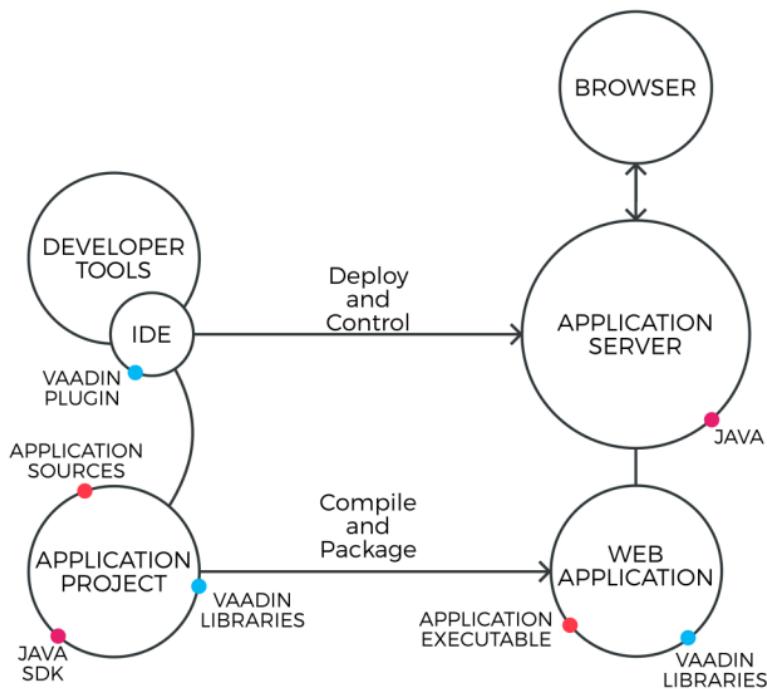


Figure 2.2. Development Toolchain and Process

Figure 2.2, “Development Toolchain and Process” illustrates the development toolchain. You develop your application as an Eclipse project. The project must include, in addition to your source code, the Vaadin libraries. It can also include project-specific themes.

You need to compile and deploy a project to a web container before you can use it. You can deploy a project through the Web Tools Platform (WTP) for Eclipse (included in the Eclipse EE package), which allows automatic deployment of web applications from Eclipse. You can also deploy a project manually, by creating a web application archive (WAR) and deploying it to the web container.

2.3. Installing Java SDK

A Java SDK is required by Vaadin and also by any of the Java IDEs. Vaadin Framework 8 requires Java 8. Java EE 7 is required for proper server push support with WebSockets.

2.3.1. Windows

Follow the following steps:

1. Download Oracle Java SE 8.0 from <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
2. Install the Java SDK by running the installer. The default options are fine.

2.3.2. Linux / UNIX

Most Linux systems either have JDK preinstalled or allow installing it through a package management system. Notice however that they have OpenJDK as the default Java implementation. While it is known to have worked with Vaadin Framework and possibly also with the development toolchain, we do not especially support it.

Depending on your OS distribution, you may have to download Java JDK 8:

1. Download Oracle Java SE 8.0 from <http://www.oracle.com/technetwork/java/javase/downloads/>
2. Decompress it under a suitable base directory, such as /opt. For example, for Java SDK, enter (either as root or with **sudo** in Linux):

```
# cd /opt  
# sh <path>/jdk-<version>.bin
```

and follow the instructions in the installer.

3. Set up the JAVA_HOME environment variable to point to the Java installation directory. Also, include the \$JAVA_HOME/bin in the PATH. How you do that varies by the UNIX variant. For example, in Linux and using the

Bash shell, you would add lines such as the following to the .bashrc or .profile script in your home directory:

```
export JAVA_HOME=/opt/jdk1.8.0_31  
export PATH=$PATH:$HOME/bin:$JAVA_HOME/bin
```

You could also make the setting system-wide in a file such as /etc/bash.bashrc, /etc/profile, or an equivalent file. If you install Apache Ant or Maven, you may also want to set up those in the path.

Settings done in a bashrc file require that you open a new shell window. Settings done in a profile file require that you log in into the system. You can, of course, also give the commands in the current shell.

2.4. Installing a Web Server

You can run Vaadin applications in any Java servlet container that supports at least Servlet API 3.0. Server push can benefit from using communication modes, such as WebSocket, enabled by features in some latest servers. For Java EE containers, at least Wildfly, Glassfish, and Apache TomEE Web Profile are recommended.

Some Java IDEs have server integration, so we describe installation of the server before the IDEs.

Some IDE bundles also include a development server; for example, NetBeans IDE includes GlassFish and Apache Tomcat.

You can also opt to install a development server from a Maven dependency and let the IDE control it through Maven executions.

2.4.1. Installing Apache Tomcat

Apache Tomcat is a lightweight Java web server suitable for both development and production. There are many ways to install it, but here we simply decompress the installation package.

Apache Tomcat should be installed with user permissions. During development, you will be running Eclipse or some

other IDE with user permissions, but deploying web applications to a Tomcat server that is installed system-wide requires administrator or root permissions.

1. Download the installation package:

Apache Tomcat 8.0 (Core Binary Distribution) from <http://tomcat.apache.org/>

2. Decompress Apache Tomcat package to a suitable target directory, such as C:\dev (Windows) or /opt (Linux or Mac OS X). The Apache Tomcat home directory will be C:\dev\apache-tomcat-8.0.x or /opt/apache-tomcat-8.0.x, respectively.

Do not start the server. If you use an IDE integration, the IDE will control starting and stopping the server.

2.5. Installing the Eclipse IDE and Plugin

If you are using the Eclipse IDE, using the Vaadin Plugin for Eclipse helps greatly. The plugin includes wizards for creating new Vaadin-based projects, themes, and client-side widgets and widget sets. Notice that you can also create Vaadin projects as Maven projects in Eclipse.

Using Eclipse IDE for Vaadin development requires installing the IDE itself and the Vaadin Plugin for Eclipse. You are advised to also configure a web server in Eclipse. You can then use the server for running the Vaadin applications that you create.

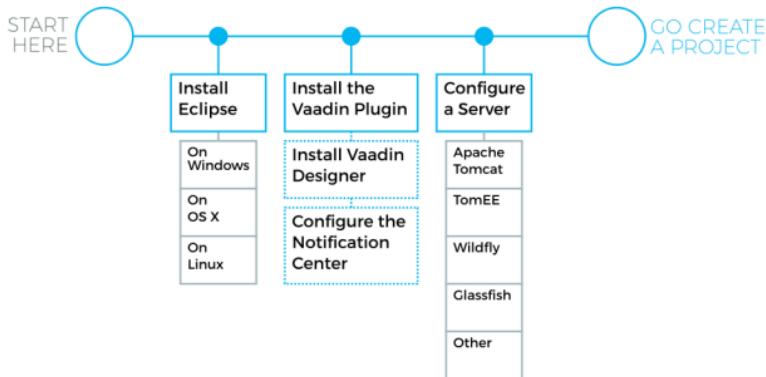


Figure 2.3. Installing the Eclipse IDE toolchain

Vaadin Designer is a visual design tool for professional developers. It allows for easy creation of declarative designs. It is also good as a sketching tool, as well as an easy way to learn about Vaadin components and layouts.

Once you have installed the Eclipse IDE and the plug-in, you can create a Vaadin application project as described in Section 3.4, "Creating and Running a Project in Eclipse".

2.5.1. Installing the Eclipse IDE

Windows

1. Download the Eclipse IDE for Java EE Developers from <http://www.eclipse.org/downloads/>
2. Decompress the Eclipse IDE package to a suitable directory. You are free to select any directory and to use any ZIP decompressor, but in this example we decompress the ZIP file by just double-clicking it and selecting "Extract all files" task from Windows compressed folder task. In our installation example, we use C:\dev as the target directory.

Eclipse is now installed in C:\dev\eclipse. You can start it from there by double clicking eclipse.exe.

Linux / OS X / UNIX

We recommend that you install Eclipse manually in Linux and other UNIX variants. They may have it available from a package repository, but using such an installation may cause problems with installing plug-ins.

You can install Eclipse as follows:

1. Download Eclipse IDE for Java EE Developers from <http://www.eclipse.org/downloads/>
2. Decompress the Eclipse package into a suitable base directory. It is important to make sure that there is no old Eclipse installation in the target directory. Installing a new version on top of an old one probably renders Eclipse unusable.
3. Eclipse should normally be installed as a regular user, which makes installation of plug-ins easier. Eclipse also stores some user settings in the installation directory.

To install the package, enter:

```
$ tar zxf <path>/eclipse-jee-<version>.tar.gz
```

This will extract the package to a subdirectory with the name `eclipse`.

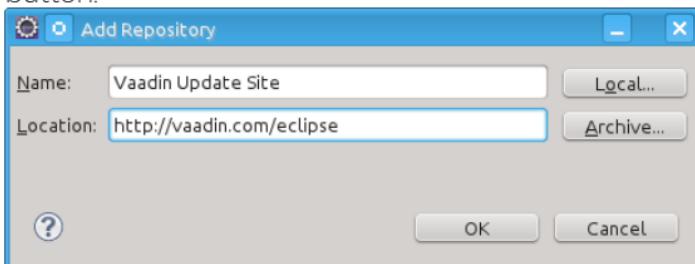
4. If you wish to enable starting Eclipse from command-line, you need to add the Eclipse installation directory to your system or user PATH, or make a symbolic link or script to point to the executable.

An alternative to the above procedure would be to use an Eclipse version available through the package management system of your operating system. It is, however, *not recommended*, because you will need write access to the Eclipse installation directory to install Eclipse plugins, and you may face incompatibility issues with Eclipse plugins installed by the package management of the operating system.

2.5.2. Installing the Vaadin Eclipse Plugin

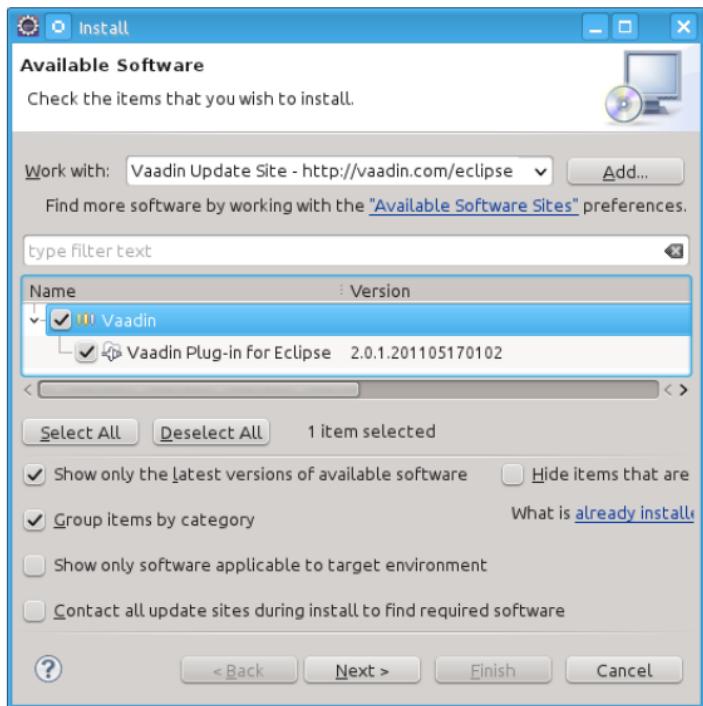
You can install the plugin as follows:

1. Select **Help > Install New Software...**.
2. Add the Vaadin plugin update site by clicking **Add...** button.



Enter a name such as "Vaadin Update Site" and the URL of the update site: <https://vaadin.com/eclipse>. If you want or need to use the latest unstable plugin, which is usually more compatible with development and beta releases of Vaadin Framework, you can use <https://vaadin.com/eclipse/experimental> and give it a distinctive name such as "Vaadin Experimental Site". Then click **OK**. The Vaadin site should now appear in the **Available Software** window.

3. Currently, if using the stable plugin, the **Group items by category** should be enabled. If using the experimental plugin, it should be disabled. This may change in future.
4. Select all the Vaadin plugins in the tree.



Then, click **Next**.

5. Review the installation details and click **Next**.
6. Accept or unaccept the license. Finally, click **Finish**.
7. After the plugin is installed, Eclipse will ask to restart itself. Click **Restart**.

More installation instructions for the Eclipse plugin can be found at <https://vaadin.com/eclipse>.

2.5.3. Notification Center

The notification center is a feature of the Vaadin Eclipse plug-in. It displays notifications about new Vaadin releases as well as news about upcoming events, such as webinars. The notification center can be connected to your Vaadin account.

The plug-in adds an indicator in the bottom right corner. The indicator shows whether or not there are any pending notifications.

ations. The indicator turns red when there are new notifications.

Clicking the tray icon opens up the pop-up, as shown in Figure 2.4, "Overview of the notification center".

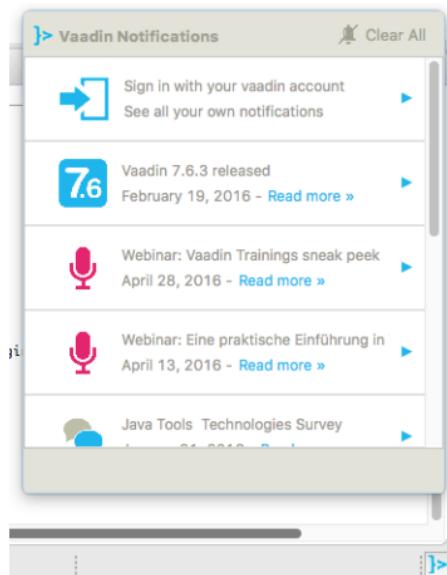


Figure 2.4. Overview of the notification center

By clicking a notification in the list, you can open it up.

Clicking on the **Clear All** icon in the main pop-up clears all notifications and marks them all as read.

Signing in

The notification center uses your Vaadin account to determine which notifications you have acknowledged as read. If you want to keep the notification center in sync with your Vaadin account, you can sign in. If you have read a notification on the site, it will be marked as read in the notification center and vice versa.

When you are not signed in, the top-most notification will be a notification that asks you to sign in.

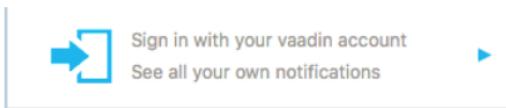


Figure 2.5. The sign-in notification

When you click the sign-in item, a dialog opens for signing in.

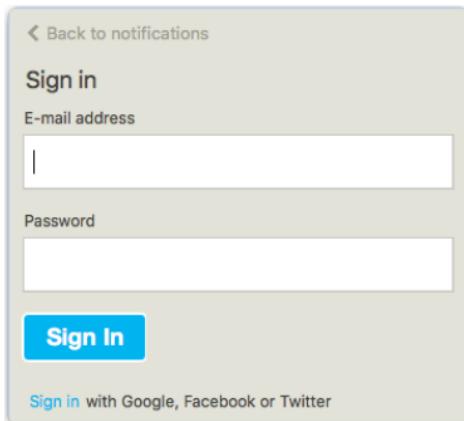


Figure 2.6. The sign-in dialog

You can then sign in with your Vaadin account.

If you do not have one, you can sign in using a Google, Facebook, or Twitter account instead.

First, click on the sign-in link. It opens a second dialog, as shown in Figure 2.7, "Sign-in authentication dialog".

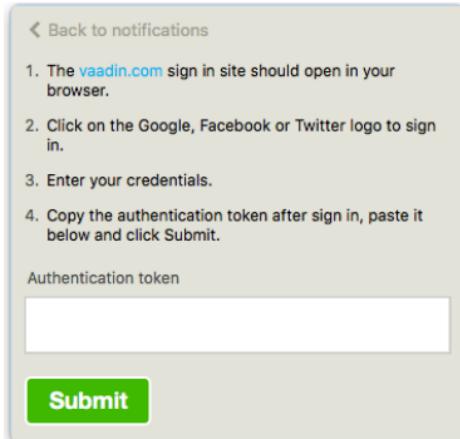


Figure 2.7. Sign-in authentication dialog

By following the vaadin.com link and logging in to the Vaadin website using your preferred account, you can then copy the authentication token from the resulting page. After that, you can paste the authorization token into the dialog and click **Submit** to log in.

Notification Settings

You can open the settings by selecting **Eclipse... ▾ Preferences**.

The options are as follows:

Enable automatic widgetset compilation

Compiles the widgetset of Maven-based projects whenever there are changes and the project is rebuilt.

Enable Vaadin pre-release archetypes

Adds pre-release (alpha/beta) archetypes to the archetype selection list when creating a new Vaadin project.

Enable

Disabling this disables all notifications.

Show popups

Disabling this stops pop-ups from appearing. The notifications can still be viewed by opening the notification center window.

Check for new Vaadin versions

Polls for new Vaadin versions once every 24h and gives a notification if there are new versions available.

Check for new notifications from vaadin.com

Polls the Vaadin site for notifications once every 4 hours.

2.5.4. Updating the Plugins

If you have automatic updates enabled in Eclipse (see **Window** \otimes **Preferences** \otimes **Install/Update** \otimes **Automatic Updates**), the Vaadin plugin will be updated automatically along with other plugins. Otherwise, you can update the Vaadin plugin manually as follows:

1. Select **Help** \otimes **Check for Updates**. Eclipse will contact the update sites of the installed software.
2. After the updates are installed, Eclipse will ask to restart itself. Click **Restart**.

Notice that updating the Vaadin plugin only updates the plugin and *not* the Vaadin libraries, which are project specific. See below for instructions for updating the libraries.

2.6. Installing the NetBeans IDE and Plugin

Vaadin offers official support for the NetBeans IDE. The Vaadin Plug-in for NetBeans supports creating Vaadin projects, updating Vaadin libraries, compiling widget sets, and more. It also allows directly downloading Vaadin add-ons from the Vaadin Directory.

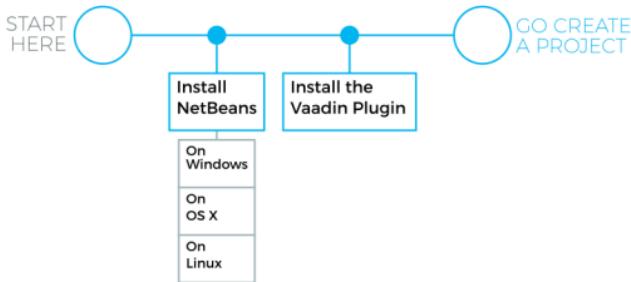


Figure 2.8. Installation of the NetBeans IDE toolchain

The installation bundle includes a web server, so you do not need that.

Once done with the installation, you can proceed to create a Vaadin project, as described in Section 3.5, “Creating a Project with the NetBeans IDE”.

2.6.1. Installing the NetBeans IDE

Download NetBeans IDE from the website at netbeans.org. We recommend using the *Java EE* download bundle, which includes support for Java EE, and also both the GlassFish and Tomcat application servers.

1. Run the installer
 - a. In OS X and Linux:


```
$ sh netbeans-<version>.sh
```
2. Select to install either GlassFish or Apache Tomcat, or both. GlassFish supports Java EE, which is required by Vaadin CDI and Vaadin Spring add-ons, while standard Tomcat does not support it. Click **Next**.
3. If you accept the license, click **Next**.
4. Choose installation folder and Java SDK.

In OS X and Linux, if you ran the installer with root permissions or can write to /opt, the /opt path is standard for such system-wide packages.

Click **Next**.

5. Choose the folder for installing the server.
6. Check the settings.

Click **Finish**. It takes a while to install the NetBeans IDE.

7. Finally, click **Done** to exit the installer.

You can now start NetBeans by starting the bin/netbeans from under the installation folder.

In Linux and OS X:

```
$ /opt/netbeans-8.1/bin/netbeans
```

You can now proceed to install the Vaadin Plug-in for NetBeans IDE.

2.6.2. Installing the Vaadin Plug-in for NetBeans IDE

You can install the plug-in from the NetBeans Plugin Portal Update Center as follows.

1. Select **Tools ▾ Plugins** from the NetBeans main menu.
2. Select the **Available Plugins** tab.
 - a. Type "Vaadin" in the **Search** box and press **Enter**.
 - b. Select the **Install** check box for the **Vaadin Plugin for NetBeans**.
 - c. Click **Install**.
3. In the plugin installation window, click **Next**.
4. Accept the license if choose to do so. Click **Install**.

5. The Vaadin Plugin is not signed, so you need to verify the certificate. Click **Continue**.

6. In the final step, select **Restart IDE now** and click **Finish**.

You can now proceed to create a Vaadin project, as described in Section 3.5, "Creating a Project with the NetBeans IDE".

The Vaadin Plug-in for NetBeans IDE can also be downloaded from the plug-in page at plugins.netbeans.org/plugin/50531/vaadin-plug-in-for-netbeans.

2.7. Installing and Configuring IntelliJ IDEA

With IntelliJ IDEA, you have two choices: use the commercial Ultimate Edition or the free Community Edition. In the following, we cover the installation and configuration of them both.

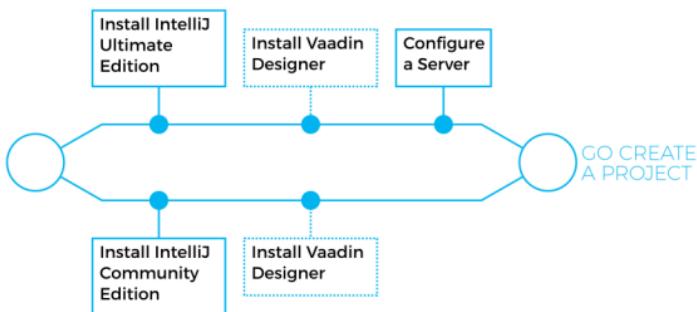


Figure 2.9. Installing the IntelliJ IDEA toolchain

The Ultimate Edition

The Ultimate Edition includes built-in support for creating Vaadin applications and running or debugging them in an integrated application server.

Community Edition

You can create a Vaadin application most easily with a Maven archetype and deploy it to a server using a Maven run/debug configuration.

You can get the both editions from the website at jetbrains.com/idea.

2.7.1. Installing the Ultimate Edition

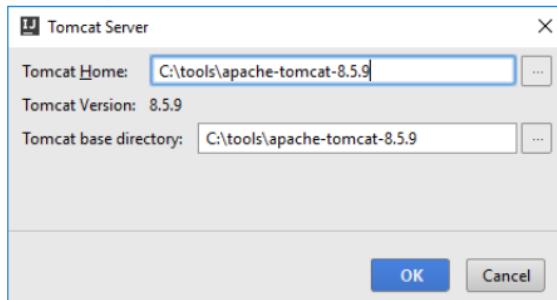
Follow the installation instructions given at the website.

Configuring an Application Server

To run a Vaadin application during development in the Ultimate Edition of IntelliJ IDEA, you first need to install and configure an application server that is integrated with the IDE. The edition includes integration with many commonly used application servers.

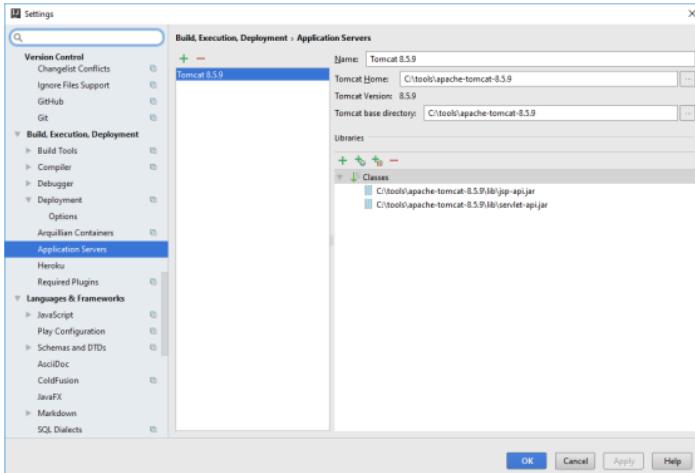
In the following, we configure Apache Tomcat:

1. Download and extract Tomcat installation package to a local directory, as instructed in Section 2.4.1, “Installing Apache Tomcat”.
2. Select **Configure** \otimes **Settings**.
3. Select **IDE Settings** \otimes **Application Servers**.
4. Click **+** and select **Tomcat Server** to add a Tomcat server, or any of the other supported servers. A WebSocket-enabled server, such as Glassfish or TomEE, is required for server push.
5. In the Tomcat Server dialog, specify the home directory for the server.



Click **OK**.

6. Review the application server settings page to check that it is OK.



Then, click **OK**.

Chapter 3

Creating a Vaadin Application

3.1. Overview	31
3.2. Vaadin Libraries	32
3.3. Overview of Maven Archetypes	34
3.4. Creating and Running a Project in Eclipse	35
3.5. Creating a Project with the NetBeans IDE	43
3.6. Creating a Project with IntelliJ IDEA	48
3.7. Creating a Project with Maven	50
3.8. Vaadin Installation Package	53
3.9. Using Vaadin with Scala	53

This chapter gives practical instructions for creating a Vaadin application project and deploying it to a server to run it. We also consider topics such as debugging.

The instructions are given separately for the Eclipse IDE, NetBeans, and IntelliJ IDEA.

3.1. Overview

Once you have installed a development environment, as described in the previous chapter, creating a Vaadin Framework project proceeds in the IDE that you have chosen.

The Vaadin Framework core library and all Vaadin add-ons are available through Maven, a commonly used build and dependency management system.

The recommended way to create a Vaadin application project is to use a Maven archetype. The archetypes contain all the needed dependencies, which Maven takes care of.

In this chapter, we:

- Give an overview of the Vaadin Framework libraries
- List the available Maven archetypes
- Give step-by-step instructions for creating a project in the Eclipse IDE, NetBeans IDE, and IntelliJ IDEA, as well as with command-line.

3.2. Vaadin Libraries

Vaadin Framework comes as a set of library JARs, of which some are optional or alternative ones, depending on whether you are developing server-side or client-side applications, whether you use add-on components, or use CSS or Sass themes.

vaadin-server-8.x.x.jar

The main library for developing server-side Vaadin applications, as described in Chapter 5, *Writing a Server-Side Web Application*. It requires the vaadin-shared and the vaadin-themes libraries. You can use the pre-built vaadin-client-compiled for server-side development, unless you need add-on components or custom widgets.

vaadin-themes-8.x.x.jar

Vaadin Framework built-in themes both as SCSS source files and precompiled CSS files. The library is required both for basic use with CSS themes and for compiling custom Sass themes.

vaadin-push-8.x.x.jar

The implementation of server push for Vaadin Framework. This is needed for web applications which use

server push (e.g. using the @Push annotation in a Servlet class).

vaadin-shared-8.x.x.jar

A shared library for server-side and client-side development. It is always needed.

vaadin-client-compiled-8.x.x.jar

A pre-compiled Vaadin Client-Side Engine (widget set) that includes all the basic built-in widgets in Vaadin. This library is not needed if you compile the application widget set with the Vaadin Client Compiler.

vaadin-client-8.x.x.jar

The client-side Vaadin Framework, including the basic GWT API and Vaadin-specific widgets and other additions. It is required when using the vaadin-client-compiler to compile client-side modules. It is not needed if you just use the server-side framework with the pre-compiled Client-Side Engine. You should not deploy it with a web application.

vaadin-client-compiler-8.x.x.jar

The Vaadin Client Compiler is a Java-to-JavaScript compiler that allows building client-side modules, such as the Client-Side Engine (widget set) required for server-side applications. The compiler is needed, for example, for compiling add-on components to the application widget set, as described in Chapter 17, *Using Vaadin Add-ons*.

For detailed information regarding the compiler, see Section 13.4, “Compiling a Client-Side Module”. Note that you should not deploy this library with a web application.

vaadin-compatibility-* -8.x.x.jar

The Vaadin Framework 7 compatibility packages contain the components and themes that are present in framework version 7, but not in version 8. These packages exist for making it easier to migrate from version 7 to 8. There is a compatibility package for everything except vaadin-client-compiler.

Some of the libraries depend on each other, for instance vaadin-shared is included as a dependency of vaadin-server.

The different ways to install the libraries are described in the subsequent sections.

Note that the vaadin-client-compiler and vaadin-client JARs should not be deployed with the web application. The Maven scope provided can be used. Some other libraries, such as vaadin-sass-compiler, are not needed in production deployment.

3.3. Overview of Maven Archetypes

Vaadin currently offers the following Maven archetypes for different kinds of projects:

vaadin-archetype-application

This is a single-module project for simple applications. It is good for quick demos and trying out Vaadin Framework. It is also useful when you are experienced with the framework and want to build all the aspects of the application yourself.

vaadin-archetype-application-multimodule

A complete Vaadin Framework application development setup. It features separate production and development profiles.

vaadin-archetype-application-example

An example CRUD web application using multi-module project setup.

vaadin-archetype-widget

A multi-module project for a new Vaadin Framework add-on. It has two modules: one for the add-on and another for a demo application.

vaadin-archetype-liferay-portlet

A portlet development setup for the open-source Liferay portal.

3.4. Creating and Running a Project in Eclipse

This section gives instructions for creating a new Eclipse project using the Vaadin Plugin. The task will include the following steps:

1. Create a new project
2. Write the source code
3. Configure and start web server
4. Open a web browser to use the web application

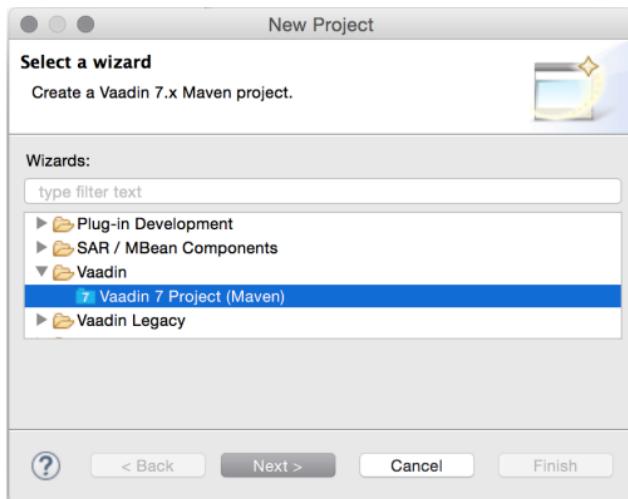
We also show how you can debug the application in the debug mode in Eclipse.

This walkthrough assumes that you have already installed the Eclipse IDE, the Vaadin Plugin, and a development server, as instructed in Section 2.5, “Installing the Eclipse IDE and Plugin”.

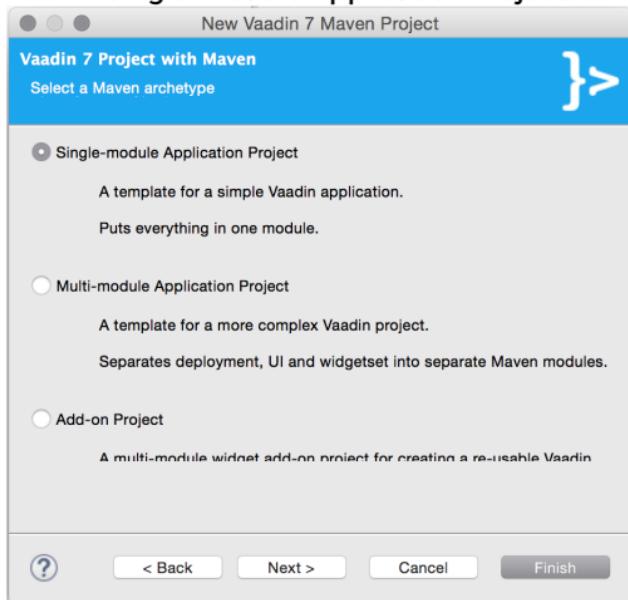
3.4.1. Creating a Maven Project

Let us create the first application project with the tools installed in the previous section. First, launch Eclipse and follow the following steps:

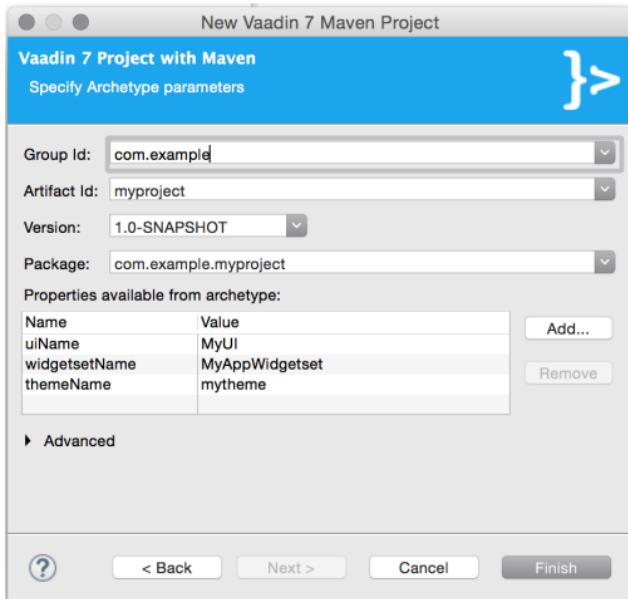
1. Start creating a new project by selecting from the menu **File** \otimes **New** \otimes **Project....**
2. In the **New Project** window that opens, select **Vaadin** \otimes **Vaadin 8 Project (Maven)** and click **Next**.



3. In the **Select a Maven archetype** step, you need to select the project type. To create a simple test project, select the **Single-module Application Project**.



4. In the **Specify archetype parameters** step, you need to give at least the **Group Id** and the **Artifact Id**. The default values should be good for the other settings.



Group Id

Give the project an organization-level identifier, for example, com.example. It is used as a prefix for your Java package names, and hence must be a valid Java package name itself.

Artifact Id

Give the project a name, for example, myproject. The artifact ID must be a valid Java sub-package name.

Version

Give the project a Maven compatible version number, for example, 1.0-SNAPSHOT. The version number should typically start with two or more integers separated with dots, and should not contain spaces.

Package

Give the base package name for the project, for example, com.example.myproject. It is by default generated from the group ID and the artifact ID.

Properties

Enter values for archetype-specific properties that control naming of various elements in the created project, such as the UI class name.

You can change the version later in the pom.xml.

Finally, click **Finish** to create the project.

3.4.2. Exploring the Project

After the **New Project** wizard exits, it has done all the work for you. A UI class skeleton has been written to the src directory. The project hierarchy shown in the Project Explorer is shown in Figure 3.1, "A new Vaadin project".

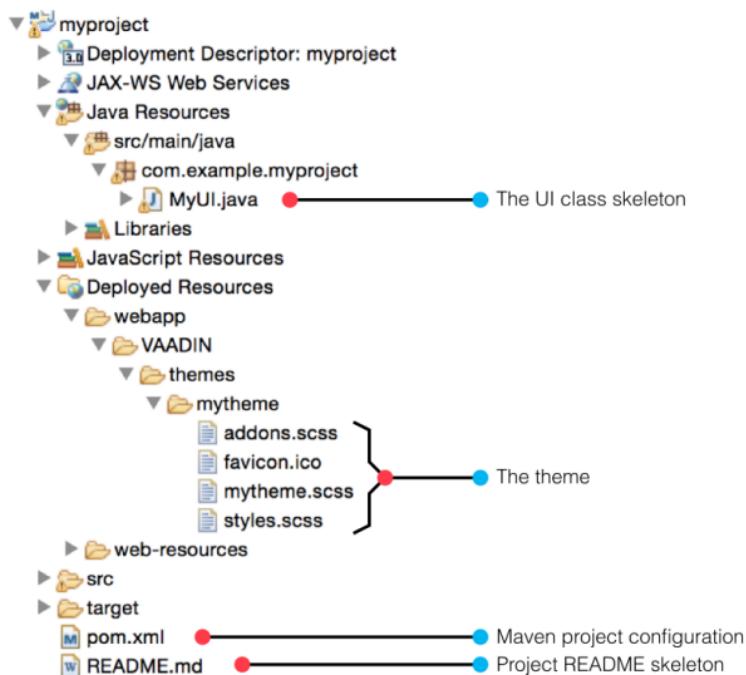


Figure 3.1. A new Vaadin project

The Vaadin libraries and other dependencies are managed by Maven. Notice that the libraries are not stored under the

project folder, even though they are listed in the **Java Resources** \otimes **Libraries** \otimes **Maven Dependencies** virtual folder.

The UI Class

The UI class created by the plug-in contains the following code:

```
package com.example.myproject;

import com.vaadin.ui.UI;
...

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

    @Override
    protected void init(VaadinRequest vaadinRequest) {
        final VerticalLayout layout = new VerticalLayout();

        final TextField name = new TextField();
        name.setCaption("Type your name here.");

        Button button = new Button("Click Me");
        button.addClickListener(e ->
            layout.addComponent(new Label("Thanks " + name.getValue()
                + ", it works!")));
        layout.addComponents(name, button);

        setContent(layout);
    }

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

3.4.3. Compiling the Theme

Before running the project for the first time, click the **Compile Vaadin Theme** button in the toolbar, as shown in Figure 3.2, “Compile Vaadin Theme”.



Figure 3.2. Compile Vaadin Theme

3.4.4. Coding Tips for Eclipse

Code Completion

One of the most useful features in Eclipse is *code completion*. Pressing **Ctrl+Space** in the editor will display a pop-up list of possible class name and method name completions, as shown in Figure 3.3, “Java Code Completion in Eclipse”, depending on the context of the cursor position.

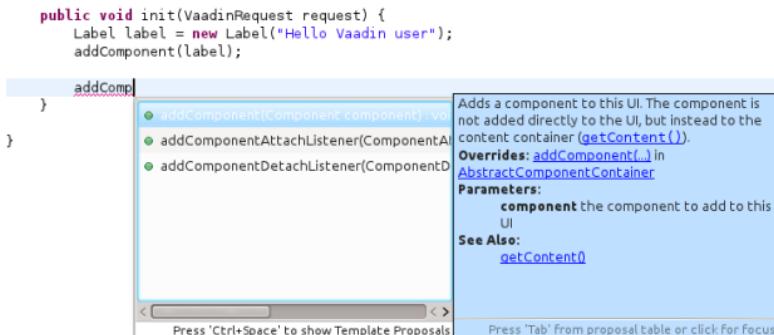


Figure 3.3. Java Code Completion in Eclipse

Generating Imports

To automatically add an import statement for a class, such as **Button**, simply press **Ctrl+Shift+O** or click the red error indicator on the left side of the editor window. If the class is available in multiple packages, a list of the alternatives is displayed, as shown in Figure 3.4, “Importing classes automatically”.

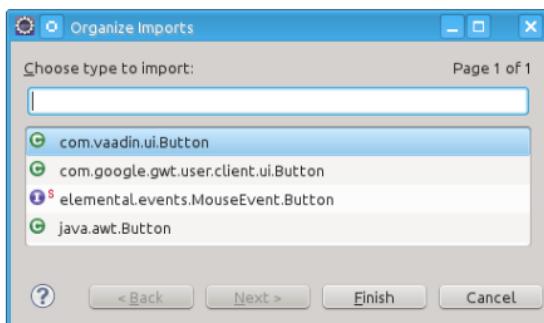


Figure 3.4. Importing classes automatically

For server-side Vaadin development, you should generally use the classes under the com.vaadin.ui or com.vaadin.server packages. You *can not use client-side classes (under com.vaadin.client) or GWT classes for server-side development.*

3.4.5. Setting Up and Starting the Web Server

Eclipse IDE for Java EE Developers has the Web Standard Tools package installed, which supports control of various web servers and automatic deployment of web content to the server when changes are made to a project.

Make sure that Tomcat was installed with user permissions. Configuration of the web server in Eclipse will fail if the user does not have write permissions to the configuration and deployment directories under the Tomcat installation directory.

Follow the following steps:

1. Switch to the **Servers** tab in the lower panel in Eclipse. List of servers should be empty after Eclipse is installed. Right-click on the empty area in the panel and select **New ▾ Server**.
2. Select **Apache ▾ Tomcat v8.0 Server** and set **Server's host name** as localhost, which should be the default. If you have only one Tomcat installed, **Server runtime** has only one choice. Click **Next**.
3. Add your project to the server by selecting it on the left and clicking **Add** to add it to the configured projects on the right. Click **Finish**.
4. The server and the project are now installed in Eclipse and are shown in the **Servers** tab. To start the server, right-click on the server and select **Debug**. To start the server in non-debug mode, select **Start**.
5. The server starts and the WebContent directory of the project is published to the server on <http://localhost:8080/myproject/>.

3.4.6. Running and Debugging

Starting your application is as easy as selecting **myproject** from the **Project Explorer** and then **Run** → **Debug As** → **Debug on Server**. Eclipse then opens the application in built-in web browser.

You can insert break points in the Java code by double-clicking on the left margin bar of the source code window. For example, if you insert a breakpoint in the `buttonClick()` method and click the **What is the time?** button, Eclipse will ask to switch to the Debug perspective. Debug perspective will show where the execution stopped at the breakpoint. You can examine and change the state of the application. To continue execution, select **Resume** from **Run** menu.

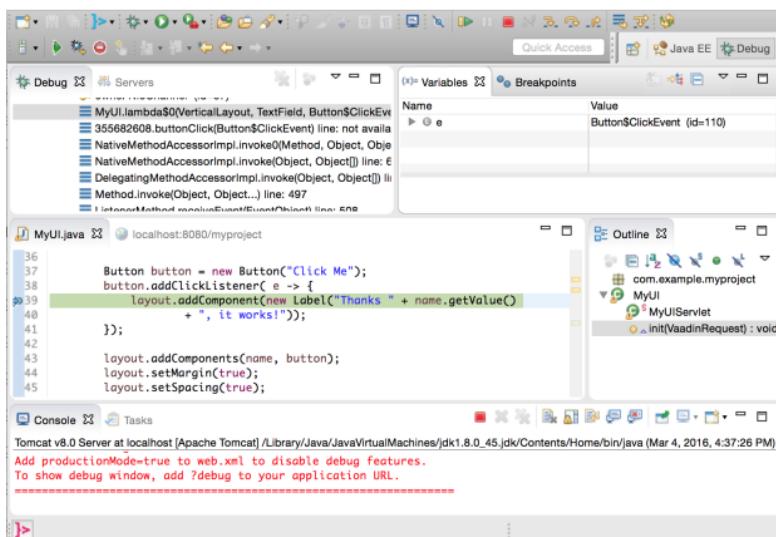


Figure 3.5. Debugging a Vaadin Application

Above, we described how to debug a server-side application. Debugging client-side applications and widgets is described in Section 13.6, “Debugging Client-Side Code”.

3.4.7. Updating the Vaadin Framework Libraries

Updating the Vaadin plugin does not update Vaadin Framework libraries. The libraries are project specific, as a different

version might be required for different projects, so you have to update them separately for each project.

1. Open the pom.xml in an editor in Eclipse.
2. Edit the vaadin.version property to set the Vaadin version.

Updating the libraries can take several minutes. You can see the progress in the Eclipse status bar. You can get more details about the progress by clicking the indicator.

3. *In Vaadin 7.6 and older:* if you have compiled the widget set for your project, recompile it by clicking the **Compile Vaadin Widgetset** button in the Eclipse toolbar.



4. Stop the integrated Tomcat (or other server) in Eclipse, clear its caches by right-clicking the server and selecting **Clean** as well as **Clean Tomcat Work Directory**, and restart it.

If you experience problems after updating the libraries, you can try using **Maven > Update Project**.

3.5. Creating a Project with the NetBeans IDE

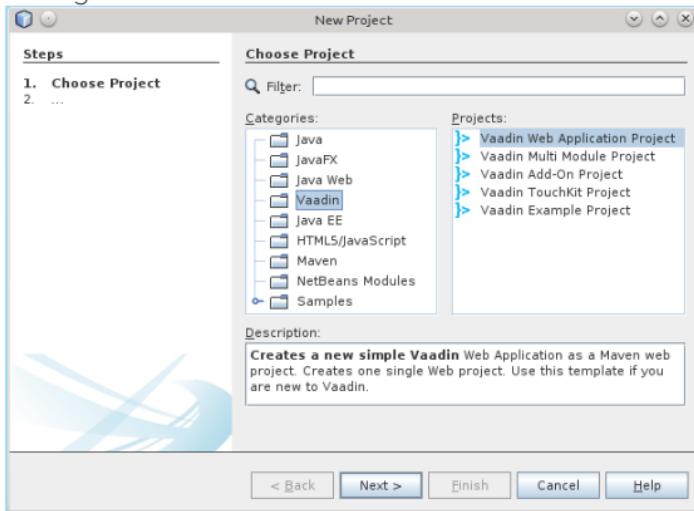
In the following, we walk you through the creation of a Vaadin project in NetBeans and show how to run it.

Installation of NetBeans and the Vaadin plugin is covered in Section 2.6, “Installing the NetBeans IDE and Plugin”.

Without the plugin, you can most easily create a Vaadin project as a Maven project using a Vaadin archetype. You can also create a Vaadin project as a regular web application project, but it requires many manual steps to install all the Vaadin libraries, create the UI class, configure the servlet, create theme, and so on.

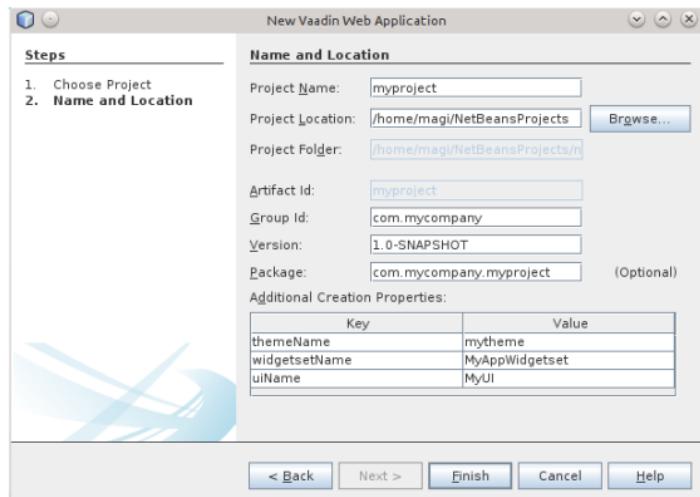
3.5.1. Creating a Project

1. Select **File > Net Project...** from the main menu or press **Ctrl+Shift+N**.
2. In the **New Project** window that opens, select the **Vaadin** category and one of the Vaadin archetypes from the right.



The archetypes are described in more detail in Section 3.3, "Overview of Maven Archetypes".

3. In the **Name and Location** step, enter the project parameters.



Project Name

A project name. The name must be a valid identifier that may only contain alphanumerics, minus, and underscore. It is appended to the group ID to obtain the Java package name for the sources.

Project Location

Path to the folder where the project is to be created.

Group Id

A Maven group ID for your project. It is normally your organization domain name in reverse order, such as com.example. The group ID is also used as a prefix for the Java source package, so it should be Java-compatible package name.

Version

Initial version of your application. The number must obey the Maven version numbering format.

Package

The Java package name to put sources in.

Additional Creation Properties

The properties control various names. They are specific to the archetype you chose.

Click **Finish**.

Creating the project can take a while as Maven loads all the needed dependencies.

3.5.2. Exploring the Project

The project wizard has done all the work for you: a UI class skeleton has been written to the src directory. The project hierarchy shown in the Project Explorer is shown in Figure 3.6. “A new Vaadin project in NetBeans”.

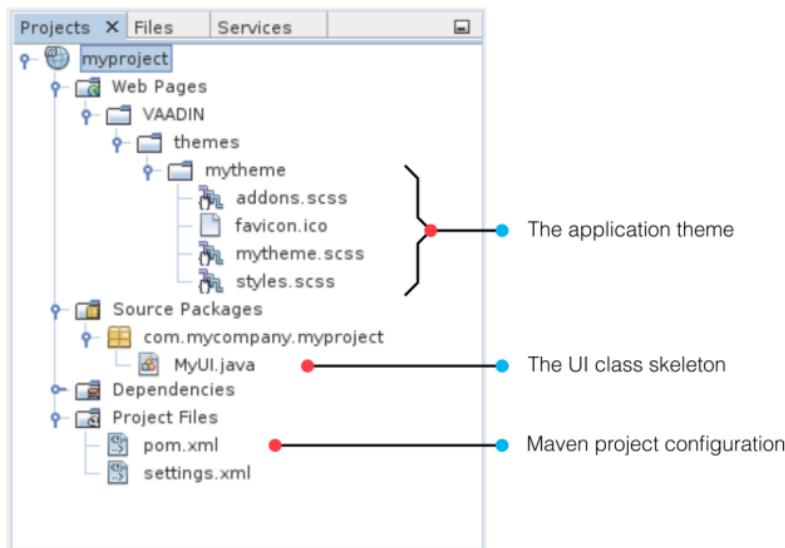


Figure 3.6. A new Vaadin project in NetBeans

mytheme

The theme of the UI. See Chapter 9, *Themes* for information about themes.

MyUI.java

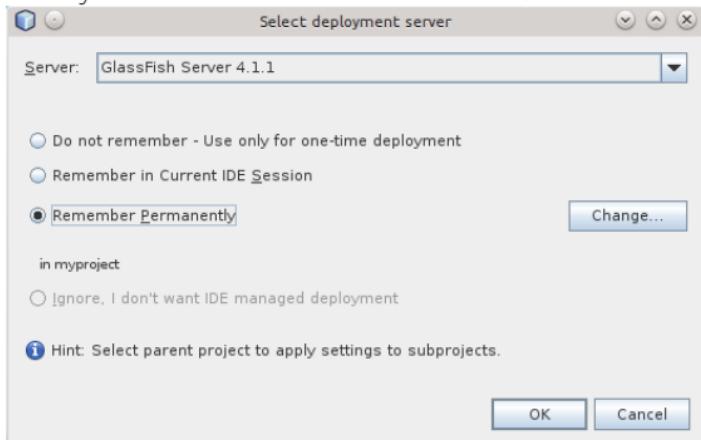
The UI class, which is the main entry-point of your application. See Chapter 5, *Writing a Server-Side Web Application* for information about the basic structure of Vaadin applications.

The Vaadin libraries and other dependencies are managed by Maven. Notice that the libraries are not stored under the project folder, even though they are listed in the **Java Resources** \otimes **Libraries** \otimes **Maven Dependencies** virtual folder.

3.5.3. Running the Application

Once created, you can run it in a server as follows.

1. In **Projects** tab, select the project and click in the **Run Project** button in the tool bar (or press **F6**).
2. In the **Select deployment server** window, select a server from the **Server** list. It should show either GlassFish or Apache Tomcat or both, depending on what you chose in NetBeans installation.



Also, select **Remember Permanently** if you want to use the same server also in future while developing applications.

Click **OK**.

The widget set will be compiled at this point, which may take a while.

If all goes well, NetBeans starts the server in port 8080 and, depending on your system configuration, launches the default browser to display the web application. If not, you can open it manually, for example, at <http://localhost:8080/myproject>.

The project name is used by default as the context path of the application.

Now when you edit the UI class in the source editor and save it, NetBeans will automatically redeploy the application. After it has finished after a few seconds, you can reload the application in the browser.

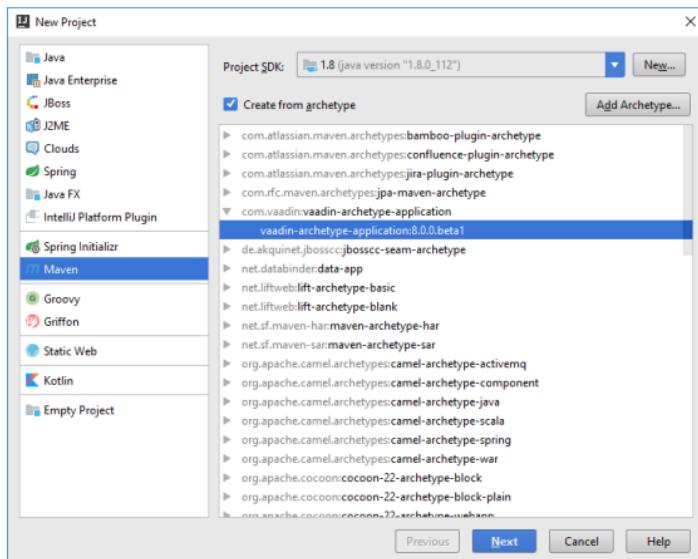
3.6. Creating a Project with IntelliJ IDEA

With both Community Edition and Ultimate Edition, you can create a Vaadin application most easily with a Maven archetype and deploy it to a server with a Maven run/debug configuration.

3.6.1. Creating a Maven Project

You can choose to create a Maven project in IntelliJ IDEA. You will not have the application server integration, but can deploy the application to an application server using a run/debug configuration.

1. Select New Project
2. In the **New Project** window, select Maven
3. Select the Java SDK to be used for the project. Vaadin requires at least Java 8.
4. Check **Create from archetype** checkbox
5. If the Vaadin archetype is not in the list, click **Add archetype**, enter **GroupId** com.vaadin, **ArtifactId** vaadin-archetype-application, and **Version** LATEST (or a specific version number).
Click **OK** in the dialog.
6. Select the archetype



Click **Next**.

7. Give a Maven **GroupId**, **ArtifactId**, and a **Version** for the project, or use the defaults.
8. Review the general Maven settings and settings for the new project. You may need to override the settings, especially if you are creating a Maven project for the first time.
9. Finish the wizard.

Creating the Maven project takes some time as Maven fetches the dependencies. Once done, the project is created and the Maven POM is opened in the editor.

For more detailed instructions, refer to <https://www.jetbrains.com/help/idea/>

Compiling the Project

To compile a Vaadin application using Maven, you can define a run/debug configuration to execute a goal such as package to build the deployable WAR package. It will also compile the

widget set and theme, if necessary. See Section 3.7.2, “Compiling and Running the Application” for more details.

Compilation is included in the following instructions for deploying the application.

Deploying to a Server

There exists Maven plugins for deploying to various application servers. For example, to deploy to Apache Tomcat, you can to configure the tomcat-maven-plugin and then execute the tomcat:deploy goal. See the documentation of the plugin that you use for more details. If no Maven plugin exists for a particular server, you can always use some lower-level method to deploy the application, such as running an Ant task.

In the following, we create a run/debug configuration to build, deploy, and launch a Vaadin Maven application on the light-weight Jetty web server.

1. Select **Run ▾ Edit Configurations**.
2. Click **+** and select **Maven** to create a new Maven run/debug configuration.
3. Enter a **Name** for the run configuration. For the **Command line**, enter `"package jetty:run"` to first compile and package the project, and then launch Jetty to run it.
Click **OK**.
4. Select the run configuration in the toolbar and click the **Run** button beside it.

Compiling the project takes some time on the first time, as it compiles the widget set and theme. Once the run console pane informs that Jetty Server has been started, you can open the browser at the default URL `http://localhost:8080/`.

3.7. Creating a Project with Maven

In previous sections, we looked into creating a Vaadin Maven project in different IDEs. In this section, we look how to create

such a project on command-line. You can then import such a project in your IDE.

In addition to regular Maven, you can use any Maven-compatible build or dependency management system, such as Ivy or Gradle. For Gradle, see the Gradle Vaadin Plugin.

For an interactive guide, see the instructions at vaadin.com/maven. It automatically generates you the command to create a new project based on archetype selection. It can also generate dependency declarations for Vaadin dependencies.

3.7.1. Working from Command-Line

You can create a new Maven project with the following command (given in one line):

```
$ mvn archetype:generate \
-DarchetypeGroupId=com.vaadin \
-DarchetypeArtifactId=vaadin-archetype-application \
-DarchetypeVersion=8.x.x \
-DgroupId=com.pany \
-DartifactId=project-name \
-Dversion=0.1 \
-Dpackaging=war
```

The parameters are as follows:

archetypeGroupId

The group ID of the archetype is com.vaadin for Vaadin archetypes.

archetypeArtifactId

The archetype ID. See the list of available archetypes in Section 3.3, "Overview of Maven Archetypes".

archetypeVersion

Version of the archetype to use. For prerelease versions it should be the exact version number, such as 8.0.0.beta2.

groupId

A Maven group ID for your project. It is normally your organization domain name in reverse order, such as

com.example. The group ID is also used as a prefix for the Java package in the sources, so it should be Java compatible - only alphanumerics and an underscore.

artifactId

Identifier of the artifact, that is, your project. The identifier may contain alphanumerics, minus, and underscore. It is appended to the group ID to obtain the Java package name for the sources. For example, if the group ID is com.example and artifact ID is myproject, the project sources would be placed in com.example.myproject package.

version

Initial version number of your application. The number must obey the Maven version numbering format.

packaging

How will the project be packaged. It is normally war.

Creating a project can take a while as Maven fetches all the dependencies.

3.7.2. Compiling and Running the Application

Before the application can be deployed, it must be compiled and packaged as a WAR package. You can do this with the package goal as follows:

```
$ mvn package
```

The location of the resulting WAR package should be displayed in the command output. You can then deploy it to your favorite application server.

The easiest way to run Vaadin applications with Maven is to use the light-weight Jetty web server. After compiling the package, all you need to do is type:

```
$ mvn jetty:run
```

The special goal starts the Jetty server in port 8080 and deploys the application. You can then open it in a web browser at <http://localhost:8080/project-name>.

3.7.3. Using Add-ons

If you use Vaadin add-ons from the Vaadin Directory, you need to add them as dependencies in the project POM. The instructions are given in Section 17.2, “Using Add-ons in a Maven Project”.

In projects that use Vaadin 7.6 or older, you need to compile the widget set manually. See the add-on usage instructions.

3.8. Vaadin Installation Package

While the recommended way to create a Vaadin project and install the libraries is to use an IDE plugin or a dependency management system, such as Maven, Vaadin is also available as a ZIP distribution package.

You can download the newest Vaadin installation package from the download page at <https://vaadin.com/download/>. Please use a ZIP decompression utility available in your operating system to extract the files from the ZIP package.

3.9. Using Vaadin with Scala

You can use Vaadin with any JVM compatible language, such as Scala or Groovy. There are, however, some caveats related to libraries and project set-up. In the following, we give instructions for creating a Scala UI in Eclipse, with the Scala IDE for Eclipse and the Vaadin Plugin for Eclipse.

1. Install the Scala IDE for Eclipse, either from an Eclipse update site or as a bundled Eclipse distribution.
2. Open an existing Vaadin Java project or create a new one as outlined in Section 3.4, “Creating and Running a Project in Eclipse”. You can delete the UI class created by the wizard.
3. Switch to the Scala perspective by clicking the perspective in the upper-right corner of the Eclipse window.
4. Right-click on the project folder in **Project Explorer** and select **Configure** \otimes **Add Scala Nature**.

5. The web application needs scala-library.jar in its class path. If using Scala IDE, you can copy it from somewhere under your Eclipse installation to the class path of the web application, that is, either to the WebContent/WEB-INF/lib folder in the project or to the library path of the application server. If copying outside Eclipse to a project, refresh the project by selecting it and pressing **F5**.

You could also get it with a Maven dependency, just make sure that the version is same as what the Scala IDE uses.

You should now be able to create a Scala UI class, such as the following:

```
@Theme("mytheme")
class MyScalaUI extends UI {
    override def init(request: VaadinRequest) = {
        val content: VerticalLayout = new VerticalLayout
        setContent(content)

        val label: Label = new Label("Hello, world!")
        content.addComponent(label)

        // Handle user interaction
        content.addComponent(new Button("Click Me!",
            new ClickListener {
                override def buttonClick(event: ClickEvent) =
                    Notification.show("The time is " + new Date())
            })
        )
    }
}
```

Eclipse and Scala IDE should be able to import the Vaadin classes automatically when you press **Ctrl+Shift+O**.

You need to define the Scala UI class either in a servlet class (in Servlet 3.0 project) or in a web.xml deployment descriptor, just like described in Section 3.4.2, “Exploring the Project” for Java UIs.

Chapter 4

Architecture

4.1. Overview	55
4.2. Technological Background	60
4.3. Client-Side Engine	64
4.4. Events and Listeners	66

In Chapter 1, *Introduction*, we gave a short introduction to the general architecture of Vaadin. This chapter looks deeper into the architecture at a more technical level.

4.1. Overview

Vaadin Framework provides two development models for web applications: for the client-side (the browser) and for the server-side. The server-driven development model is the more powerful one, allowing application development solely on the server-side, by utilizing an AJAX-based Vaadin Client-Side Engine that renders the user interface in the browser. The client-side model allows developing widgets and applications in Java, which are compiled to JavaScript and executed in the browser. The two models can share their UI widgets, themes, and back-end code and services, and can be mixed together easily.

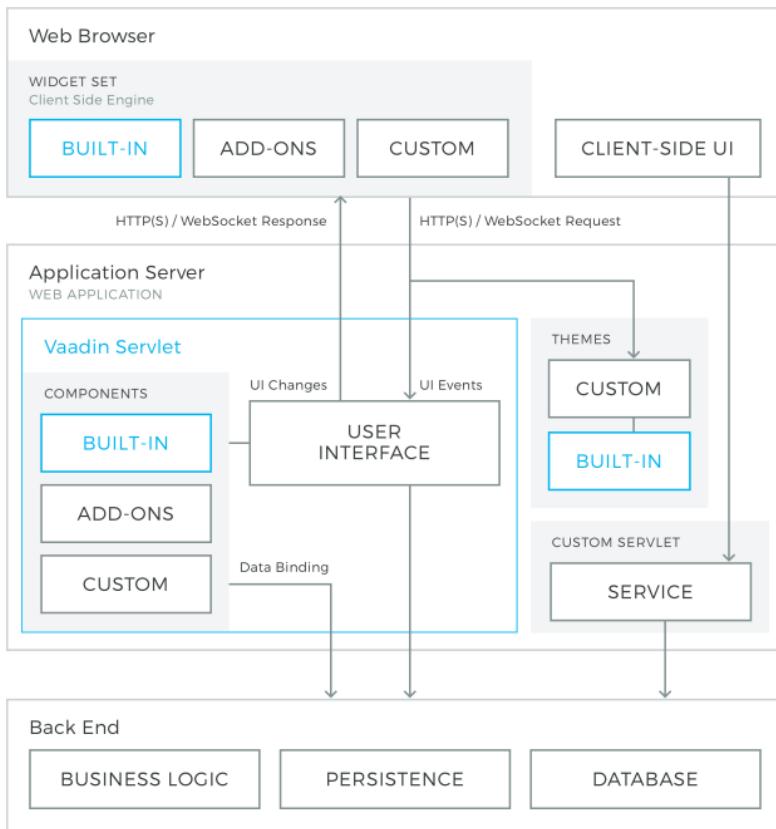


Figure 4.1. Vaadin runtime architecture

Figure 4.1. “Vaadin runtime architecture” gives a basic illustration of the client-side and server-side communications, in a running situation where the page with the client-side code (engine or application) has been initially loaded in the browser.

Vaadin Framework consists of a *server-side API*, a *client-side API*, a horde of *user interface components/widgets* on the both sides, *themes* for controlling the appearance, and a *data model* that allows binding the server-side components directly to data. For client-side development, it includes the Vaadin Compiler, which allows compiling Java to JavaScript.

A server-side Vaadin application runs as a servlet in a Java web server, serving HTTP requests. The **VaadinServlet** is nor-

mally used as the servlet class. The servlet receives client requests and interprets them as events for a particular user session. Events are associated with user interface components and delivered to the event listeners defined in the application. If the UI logic makes changes to the server-side user interface components, the servlet renders them in the web browser by generating a response. The client-side engine running in the browser receives the responses and uses them to make any necessary changes to the page in the browser.

The major parts of the server-driven development architecture and their function are as follows:

User Interface

Vaadin applications provide a user interface for the user to interface with the business logic and data of the application. At technical level, the UI is realized as a *UI* class that extends **com.vaadin.ui.UI**. Its main task is to create the initial user interface out of UI components and set up event listeners to handle user input. The UI can then be loaded in the browser using an URL, or can be embedded to any HTML page. For detailed information about implementing a **UI**, see Chapter 5, *Writing a Server-Side Web Application*.

Please note that the term "UI" is used throughout this book to refer both to the general UI concept as well as the technical UI class concept.

User Interface Components/Widgets

The user interface of a Vaadin application consists of components that are created and laid out by the application. Each server-side component has a client-side counterpart, a *widget*, by which it is rendered in the browser and with which the user interacts. The client-side widgets can also be used by client-side applications. The server-side components relay these events to the application logic. Field components that have a value, which the user can view or edit, can be bound to a data source (see below). For a more detailed description of the UI component architecture, see Chapter 6, *User Interface Components*.

Client-Side Engine

The Client-Side Engine of Vaadin manages the rendering of the UI in the web browser by employing various client-side *widgets*, counterparts of the server-side components. It communicates user interaction to the server-side, and then again renders the changes in the UI. The communications are made using asynchronous HTTP or HTTPS requests. See Section 4.3, “Client-Side Engine”.

Vaadin Servlet

Server-side Vaadin applications work on top of the Java Servlet API (see Section 4.2.5, “Java Servlets”). The Vaadin servlet, or more exactly the **VaadinServlet** class, receives requests from different clients, determines which user session they belong to by tracking the sessions with cookies, and delegates the requests to their corresponding sessions. You can customize the Vaadin servlet by extending it.

Themes

Vaadin makes a separation between the appearance and component structure of the user interface. While the UI logic is handled as Java code, the presentation is defined in *themes* as CSS or Sass. Vaadin provides a number of default themes. User themes can, in addition to style sheets, include HTML templates that define custom layouts and other resources, such as images and fonts. Themes are discussed in detail in Chapter 9, *Themes*.

Events

Interaction with user interface components creates events, which are first processed on the client-side by the widgets, then passed all the way through the HTTP server, Vaadin servlet, and the user interface components to the event listeners defined in the application. See Section 4.4, “Events and Listeners”.

Server Push

In addition to the event-driven programming model, Vaadin supports server push, where the UI changes are pushed directly from the server to the client without a

client request or an event. This makes it possible to update UIs immediately from other threads and other UIs, without having to wait for a request. See Section 11.15, “Server Push”.

Data Binding

In addition to the user interface model, Vaadin provides a *data binding* API for associating data presented in field components, such as text fields, check boxes and selection components, with a data source. Using data binding, the user interface components can update the application data directly, often without the need for any control code. For example, you can bind a data grid component to a backend query response. For a complete overview of the data binding model, please refer to Chapter 10, *Binding Components to Data*.

Client-Side Applications

In addition to server-side web applications, Vaadin supports client-side application modules, which run in the browser. Client-side modules can use the same widgets, themes, and back-end services as server-side Vaadin applications. They are useful when you have a need for highly responsive UI logic, such as for games or for serving a large number of clients with possibly stateless server-side code, and for various other purposes, such as offering an off-line mode for server-side applications. Please see Chapter 14, *Client-Side Applications* for further details.

Back-end

Vaadin is meant for building user interfaces, and it is recommended that other application layers should be kept separate from the UI. The business logic can run in the same servlet as the UI code, usually separated at least by a Java API, possibly as EJBs, or distributed to a remote back-end service. The data storage is usually distributed to a database management system, and is typically accessed through a persistence solution, such as JPA.

4.2. Technological Background

This section provides an introduction to the various technologies and designs, which Vaadin Framework is based on. This knowledge is not necessary for using Vaadin Framework, but provides some background if you need to make low-level extensions to the framework.

4.2.1. HTML and JavaScript

The World Wide Web, with all its websites and most of the web applications, is based on the use of the Hypertext Markup Language (HTML). HTML defines the structure and formatting of web pages, and allows inclusion of graphics and other resources. It is based on a hierarchy of elements marked with start and end tags, such as `<div> ... </div>`. Vaadin Framework uses HTML version 5, although conservatively, to the extent supported by the major browsers, and their currently most widely used versions.

JavaScript, on the other hand, is a programming language for embedding programs in HTML pages. JavaScript programs can manipulate a HTML page through the Document Object Model (DOM) of the page. They can also handle user interaction events. The Client-Side Engine of the framework and its client-side widgets do exactly this, although it is actually programmed in Java, which is compiled to JavaScript with the Vaadin Client Compiler.

Vaadin Framework largely hides the use of HTML, allowing you to concentrate on the UI component structure and logic. In server-side development, the UI is developed in Java using UI components and rendered by the client-side engine as HTML, but it is possible to use HTML templates for defining the layout, as well as HTML formatting in many text elements. Also when developing client-side widgets and UIs, the built-in widgets in the framework hide most of HTML DOM manipulation.

4.2.2. Styling with CSS and Sass

While HTML defines the content and structure of a web page, *Cascading Style Sheet* (CSS) is a language for defining the

visual style, such as colors, text sizes, and margins. CSS is based on a set of rules that are matched with the HTML structure by the browser. The properties defined in the rules determine the visual appearance of the matching HTML elements.

```
/* Define the color of labels in my view */
.myview .v-label {
    color: blue;
}
```

Sass, or *Syntactically Awesome Stylesheets*, is an extension of the CSS language, which allows the use of variables, nesting, and many other syntactic features that make the use of CSS easier and clearer. Sass has two alternative formats, SCSS, which is a superset of the syntax of CSS3, and an older indented syntax, which is more concise. The Vaadin Sass compiler supports the SCSS syntax.

Vaadin Framework handles styling with *themes* defined with CSS or Sass, and associated images, fonts, and other resources. Vaadin themes are specifically written in Sass. In development mode, Sass files are compiled automatically to CSS. For production use, you compile the Sass files to CSS with the included compiler. The use of themes is documented in detail in Chapter 9, *Themes*, which also gives an introduction to CSS and Sass.

4.2.3. AJAX

AJAX, short for Asynchronous JavaScript and XML, is a technique for developing web applications with responsive user interaction, similar to traditional desktop applications. Conventional web applications, be they JavaScript-enabled or not, can get new page content from the server only by loading an entire new page. AJAX-enabled pages, on the other hand, handle the user interaction in JavaScript, send a request to the server asynchronously (without reloading the page), receive updated content in the response, and modify the page accordingly. This way, only small parts of the page data need to be loaded. This goal is achieved by the use of a certain set of technologies: HTML, CSS, DOM, JavaScript, and the XMLHttpRequest API in JavaScript. XML is just one way to serialize data between the client and the server, and in Vaadin it is serialized with the more efficient JSON.

The asynchronous requests used in AJAX are made possible by the XMLHttpRequest class in JavaScript. The API feature is available in all major browsers and is under way to become a W3C standard.

The communication of complex data between the browser and the server requires some sort of *serialization* (or *marshalling*) of data objects. The Vaadin servlet and the client-side engine handle the serialization of shared state objects from the server-side components to the client-side widgets, as well as serialization of RPC calls between the widgets and the server-side components.

4.2.4. GWT

The client-side of Vaadin Framework is based on GWT. Its purpose is to make it possible to develop web user interfaces that run in the browser easily with Java instead of JavaScript. Client-side modules are developed with Java and compiled into JavaScript with the Vaadin Compiler, which is an extension of the GWT Compiler. The client-side framework also hides much of the HTML DOM manipulation and enables handling browser events in Java.

GWT is essentially a client-side technology, normally used to develop user interface logic in the web browser. Pure client-side modules still need to communicate with a server using RPC calls and by serializing any data. The server-driven development mode in the framework effectively hides all the client-server communications and allows handling user interaction logic in a server-side application. This makes the architecture of an AJAX-based web application much simpler. Nevertheless, Vaadin Framework also allows developing pure client-side applications, as described in Chapter 14, *Client-Side Applications*.

See Section 4.3, “Client-Side Engine” for a description of how the client-side framework based on GWT is used in the Client-Side Engine of Vaadin Framework. Chapter 13, *Client-Side Vaadin Development* provides information about the client-side development, and Chapter 16, *Integrating with the Server-Side* about the integration of client-side widgets with the server-side components.

4.2.5. Java Servlets

A Java Servlet is a class that is executed in a Java web server (a *Servlet container*) to extend the capabilities of the server. In practice, it is normally a part of a *web application*, which can contain HTML pages to provide static content, and JavaServer Pages (JSP) and Java Servlets to provide dynamic content. This is illustrated in Figure 4.2, "Java Web Applications and Servlets".

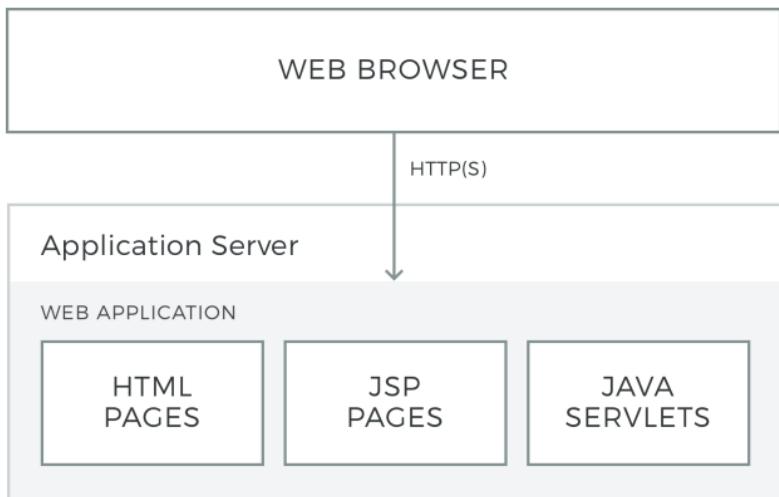


Figure 4.2. Java Web Applications and Servlets

Web applications are usually packaged and deployed to a server as *WAR* (*Web application ARchive*) files, which are Java JAR packages, which in turn are ZIP compressed packages. The web application is defined in a deployment descriptor, which defines the servlet classes and also the mappings from request URL paths to the servlets.

The servlets are Java classes that handle HTTP requests passed to them by the server through the *Java Servlet API*. They can generate HTML or other content as a response. JSP pages, on the other hand, are HTML pages, which allow including Java source code embedded in the pages. They are actually translated to Java source files by the container and then compiled to servlets.

The UIs of server-side Vaadin applications run as servlets. They are wrapped inside a **VaadinServlet** servlet class, which handles session tracking and other tasks. On the initial request, it returns an HTML loader page and then mostly JSON responses to synchronize the widgets and their server-side counterparts. It also serves various resources, such as themes. The server-side UIs are implemented as classes extending the **UI** class, as described in Chapter 5, *Writing a Server-Side Web Application*. The class is given as a parameter to the Vaadin Servlet in the deployment descriptor.

The Client-Side Engine of Vaadin Framework as well as any client-side extension are loaded to the browser as static JavaScript files. The client-side engine, or widget set in technical terms, needs to be located under the VAADIN/widgetsets path in the web application. It is normally automatically compiled to include the default widget set, as well as any installed add-ons and custom widgets.

4.3. Client-Side Engine

The user interface of a server-side Vaadin application is rendered in the browser by the Vaadin Client-Side Engine. It is loaded in the browser when the page with the Vaadin UI is opened. The server-side UI components are rendered using *widgets* (as they are called in GWT) on the client-side. The client-side engine is illustrated in Figure 4.3, "Vaadin Client-Side Engine".



Figure 4.3. Vaadin Client-Side Engine

The client-side framework includes two kinds of built-in widgets: GWT widgets and Vaadin-specific widgets. The two widget collections have significant overlap, where the Vaadin widgets provide a bit different features than the GWT widgets. In addition, many add-on widgets and their server-side counterparts exist, and you can easily download and install them, as described in Chapter 17, *Using Vaadin Add-ons*. You can also develop your own widgets, as described in Chapter 13, *Client-Side Vaadin Development*.

The rendering with widgets, as well as the communication to the server-side, is handled in the **ApplicationConnection**. Connecting the widgets with their server-side counterparts is done in *connectors*, and there is one for each widget that has

a server-side counterpart. The framework handles serialization of component state transparently, and includes an RPC mechanism between the two sides. Integration of widgets with their server-side counterpart components is described in Chapter 16, *Integrating with the Server-Side*.

4.4. Events and Listeners

Vaadin Framework offers an event-driven programming model for handling user interaction. When a user does something in the user interface, such as clicks a button or selects an item, the application needs to know about it. Many Java-based user interface frameworks follow the *Event-Listener pattern* (also known as the Observer design pattern) to communicate user input to the application logic. So does Vaadin Framework. The design pattern involves two kinds of elements: an object that generates ("fires" or "emits") events and a number of listeners that listen for the events. When such an event occurs, the object sends a notification about it to all the listeners. In a typical case, there is only one listener.

Events can serve many kinds of purposes. In Vaadin Framework, the usual purpose of events is handling user interaction in a user interface. Session management can require special events, such as time-out, in which case the event would actually be the lack of user interaction. Time-out is a special case of timed or scheduled events, where an event occurs at a specific date and time or when a set time has passed.

To receive events of a particular type, an application must register a listener object with the event source. The listeners are registered in the components with an `add*Listener()` method (with a method name specific to the listener).

Most components that have related events define their own event class and the corresponding listener class. For example, the **Button** has **Button.ClickEvent** events, which can be listened to through the **Button.ClickListener** functional interface.

In the following, we assign a button click listener using a lambda expression.

```
final Button button = new Button("Push it!");
```

```
button.addClickListener(event ->  
    button.setCaption("You pushed it!"));
```

Figure 4.4, "Class Diagram of a Button Click Listener" illustrates the case where an application-specific class inherits the **Button.ClickListener** interface to be able to listen for button click events. The application must instantiate the listener class and register it with addClickListener(). When an event occurs, an event object is instantiated, in this case a **Button.ClickEvent**. The event object knows the related UI component, in this case the **Button**.

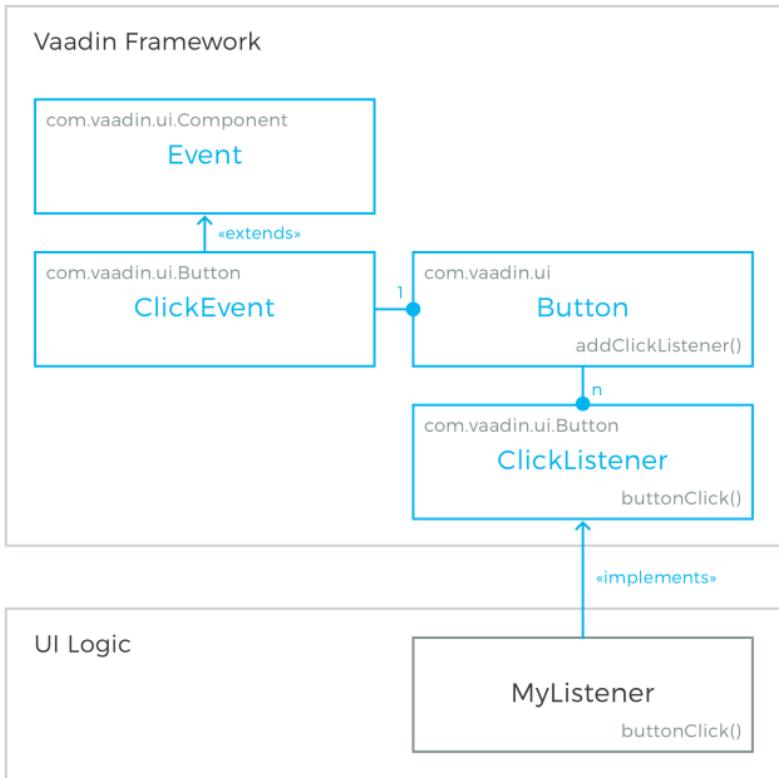


Figure 4.4. Class Diagram of a Button Click Listener

Section 5.4, "Handling Events with Listeners" goes into details of handling events in practice.

Chapter 5

Writing a Server-Side Web Application

5.1. Overview	69
5.2. Building the UI	74
5.3. Designing UIs Declaratively	79
5.4. Handling Events with Listeners	86
5.5. Images and Other Resources	87
5.6. Handling Errors	92
5.7. Notifications	95
5.8. Application Lifecycle	99
5.9. Deploying an Application	107

This chapter provides the fundamentals of server-side web application development with Vaadin, concentrating on the basic elements of an application from a practical point-of-view.

5.1. Overview

A Vaadin Framework application runs as a Java Servlet in a servlet container. The Java Servlet API is, however, hidden behind the framework. The user interface of the application is implemented as a *UI* class, which needs to create and

manage the user interface components that make up the user interface. User input is handled with event listeners, although it is also possible to bind the user interface components directly to data. The visual style of the application is defined in themes as CSS or Sass. Icons, other images, and downloadable files are handled as *resources*, which can be external or served by the application server or the application itself.

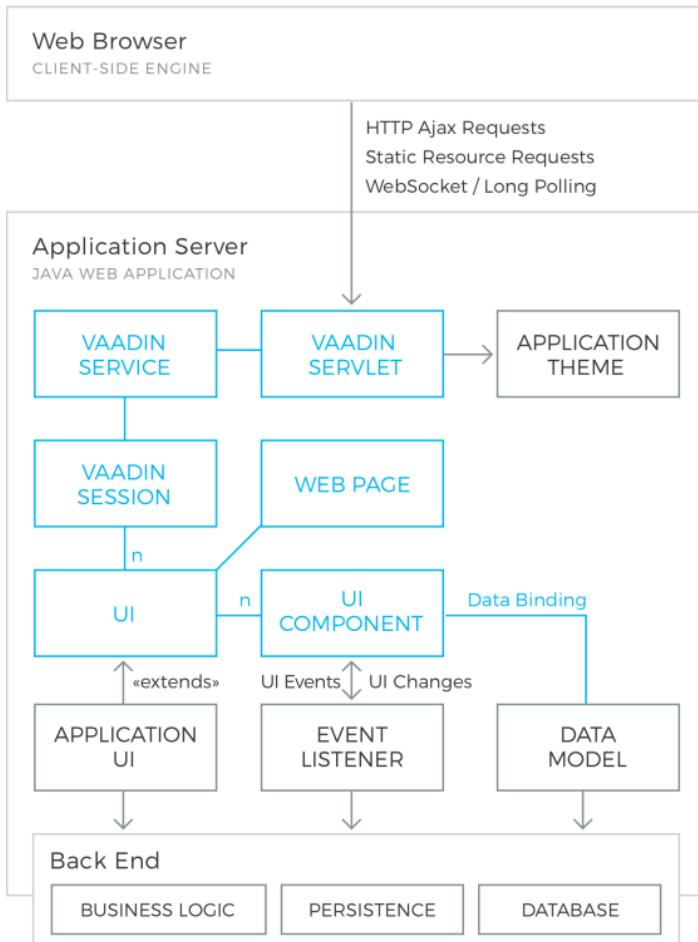


Figure 5.1. Vaadin Framework Application Architecture

Figure 5.1, "Vaadin Framework Application Architecture" illustrates the basic architecture of an application made with the

Vaadin Framework, with all the major elements, which are introduced below and discussed in detail in this chapter.

First of all, a Vaadin Framework application must have one or more UI classes that extend the abstract **com.vaadin.ui.UI** class and implement the init() method. A custom theme can be defined as an annotation for the UI.

```
@Theme("hellotheme")
public class HelloWorld extends UI {
    protected void init(VaadinRequest request) {
        ... initialization code goes here ...
    }
}
```

A UI is a viewport to the application running in a web page. A web page can actually have multiple such UIs within it. Such situation is typical especially with portlets in a portal. An application can run in multiple browser windows, each having a distinct **UI** instance. The UIs of an application can be the same UI class or different.

Vaadin Framework handles servlet requests internally and associates the requests with user sessions and a UI state. Because of this, you can develop applications with Vaadin Framework much like you would develop desktop applications.

The most important task in the initialization is the creation of the initial user interface. This, and the deployment of a UI as a Java Servlet in the Servlet container, as described in Section 5.9, “Deploying an Application”, are the minimal requirements for an application.

Below is a short overview of the other basic elements of an application besides UI:

UI

A *UI* represents an HTML fragment in which a Vaadin application runs in a web page. It typically fills the entire page, but can also be just a part of a page. You normally develop an application with Vaadin Framework by extending the **UI** class and adding content to it. A UI is essentially a viewport connected to a user session of an

application, and you can have many such views, especially in a multi-window application. Normally, when the user opens a new page with the URL of the UI, a new **UI** (and the associated **Page** object) is automatically created for it. All of them share the same user session.

The current UI object can be accessed globally with `UI.getCurrent()`. The static method returns the thread-local UI instance for the currently processed request .

Page

A **UI** is associated with a **Page** object that represents the web page as well as the browser window in which the UI runs.

The **Page** object for the currently processed request can be accessed globally from a Vaadin application with `Page.getCurrent()`. This is equivalent to calling `UI.getCurrent().getPage()`.

Vaadin Session

A **VaadinSession** object represents a user session with one or more UIs open in the application. A session starts when a user first opens a UI of a Vaadin application, and closes when the session expires in the server or when it is closed explicitly.

User Interface Components

The user interface consists of components that are created by the application. They are laid out hierarchically using special *layout components*, with a content root layout at the top of the hierarchy. User interaction with the components causes *events* related to the component, which the application can handle. *Field components* are intended for inputting values and can be directly bound to data using the data model of the framework. You can make your own user interface components through either inheritance or composition. For a thorough reference of user interface components, see Chapter 6, *User Interface Components*, for layout components, see Chapter 7, *Managing Layout*, and for compositing components, see Section 6.27, "Composition with **CustomComponent**".

Events and Listeners

Vaadin Framework follows an event-driven programming paradigm, in which events, and listeners that handle the events, are the basis of handling user interaction in an application (although also server push is possible as described in Section 11.15, “Server Push”). Section 4.4, “Events and Listeners” gave an introduction to events and listeners from an architectural point-of-view, while Section 5.4, “Handling Events with Listeners” later in this chapter takes a more practical view.

Resources

A user interface can display images or have links to web pages or downloadable documents. These are handled as *resources*, which can be external or provided by the web server or the application itself. Section 5.5, “Images and Other Resources” gives a practical overview of the different types of resources.

Themes

The presentation and logic of the user interface are separated. While the UI logic is handled as Java code, the presentation is defined in *themes* as CSS or SCSS. Vaadin includes some built-in themes. User-defined themes can, in addition to style sheets, include HTML templates that define custom layouts and other theme resources, such as images. Themes are discussed in detail in Chapter 9, *Themes*, custom layouts in Section 7.14, “Custom Layouts”, and theme resources in Section 5.5.4, “Theme Resources”.

Data Binding

With data binding, any field component in Vaadin Framework can be bound to the properties of business objects such as JavaBeans and grouped together as forms. The components can get their values from and update user input to the data model directly, without the need for any control code. Similarly, any select component can be bound to a *data provider*, fetching its items from a Java Collection or a backend such as an SQL database. For a complete overview of data binding in Vaadin, please refer to Chapter 10, *Binding Components to Data*.

5.2. Building the UI

Vaadin Framework user interfaces are built hierarchically from components, so that the leaf components are contained within layout components and other component containers. Building the hierarchy starts from the top (or bottom - whichever way you like to think about it), from the **UI** class of the application. You normally set a layout component as the content of the UI and fill it with other components.

```
public class MyHierarchicalUI extends UI {  
    @Override  
    protected void init(VaadinRequest request) {  
        // The root of the component hierarchy  
        VerticalLayout content = new VerticalLayout();  
        content.setSizeFull(); // Use entire window  
        setContent(content); // Attach to the UI  
  
        // Add some component  
        content.addComponent(new Label("<b>Hello</b> - How are you?",  
            ContentMode.HTML));  
  
        Grid<Person> grid = new Grid<>();  
        grid.setCaption("My Grid");  
        grid.setItems(GridExample.generateContent());  
        grid.setSizeFull();  
        content.addComponent(grid);  
        content.setExpandRatio(grid, 1); // Expand to fill  
    }  
}
```

The component hierarchy is illustrated in Figure 5.2, "Schematic diagram of the UI".

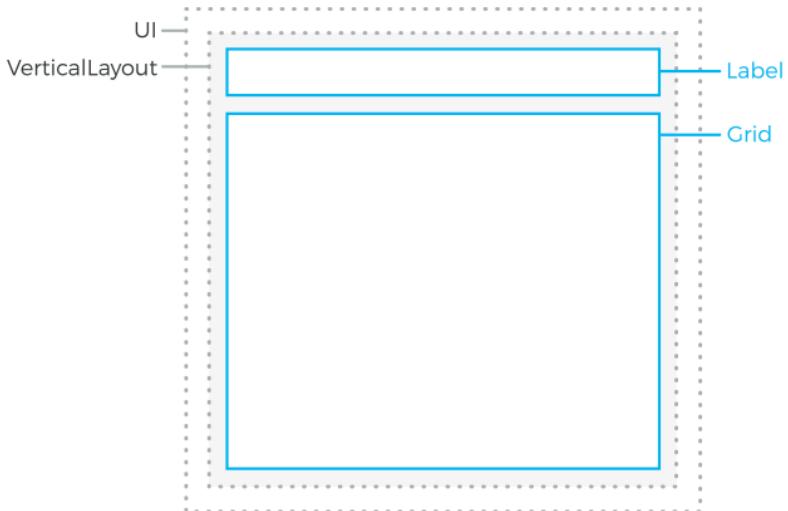


Figure 5.2. Schematic diagram of the UI

The actual UI is shown in Figure 5.3, "Simple hierarchical UI".

The screenshot shows a user interface with a greeting message and a data grid. The message 'Hello! - How are you?' is displayed above a table titled 'My Grid'. The table has columns for Name, City, and Year. The data rows are:

Name	City	Year
Charles Darwin	Oxford	1994
Charles Adams	London	1884
Isaac Lovelace	Oxford	1928
Charles Newton	Oxford	1818
Charles Adams	London	1883

Figure 5.3. Simple hierarchical UI

Instead of building the layout in Java, you can also use a declarative design, as described later in Section 5.3, "Designing UIs Declaratively". The examples given for the declarative layouts give exactly the same UI layout as built from the components above. The easiest way to create declarative designs is to use Vaadin Designer.

The built-in components are described in Chapter 6, *User Interface Components* and the layout components in Chapter 7, *Managing Layout*.

The example application described above just is, it does not do anything. User interaction is handled with event listeners, as described a bit later in Section 5.4, “Handling Events with Listeners”.

5.2.1. Application Architecture

Once your application grows beyond a dozen or so lines, which is usually quite soon, you need to start considering the application architecture more closely. You are free to use any object-oriented techniques available in Java to organize your code in methods, classes, packages, and libraries. An architecture defines how these modules communicate together and what sort of dependencies they have between them. It also defines the scope of the application. The scope of this book, however, only gives a possibility to mention some of the most common architectural patterns in Vaadin applications.

The subsequent sections describe some basic application patterns. For more information about common architectures, see Section 11.9, “Advanced Application Architectures”, which discusses layered architectures, the Model-View-Presenter (MVP) pattern, and so forth.

5.2.2. Compositing Components

User interfaces typically contain many user interface components in a layout hierarchy. Vaadin provides many layout components for laying contained components vertically, horizontally, in a grid, and in many other ways. You can extend layout components to create composite components.

```
class MyView extends VerticalLayout {  
    TextField entry = new TextField("Enter this");  
    Label display = new Label("See this");  
    Button click = new Button("Click This");  
  
    public MyView() {  
        addComponent(entry);  
        addComponent(display);  
    }  
}
```

```
addComponent(click);

setSizeFull();
addStyleName("myview");
}

}
```

```
// Create an instance of MyView
Layout myview = new MyView();
```

While extending layouts is an easy way to make component composition, it is a good practice to encapsulate implementation details, such as the exact layout component used. Otherwise, the users of such a composite could begin to rely on such implementation details, which would make changes harder. For this purpose, Vaadin has a special **CustomComponent** wrapper, which hides the content representation.

```
class MyView extends CustomComponent {
    TextField entry = new TextField("Enter this");
    Label display = new Label("See this");
    Button click = new Button("Click This");

    public MyView() {
        Layout layout = new VerticalLayout();

        layout.addComponent(entry);
        layout.addComponent(display);
        layout.addComponent(click);

        setCompositionRoot(layout);
        setSizeFull();
    }
}
```

```
// Create an instance of MyView
MyView myview = new MyView();
```

For a more detailed description of the **CustomComponent**, see Section 6.27, "Composition with **CustomComponent**".

5.2.3. View Navigation

While the simplest applications have just one *view* (or *screen*), most of them often require several. Even in a single view, you often want to have sub-views, for example to display different

content. Figure 5.4, "Navigation Between Views" illustrates a typical navigation between different top-level views of an application, and a main view with sub-views.

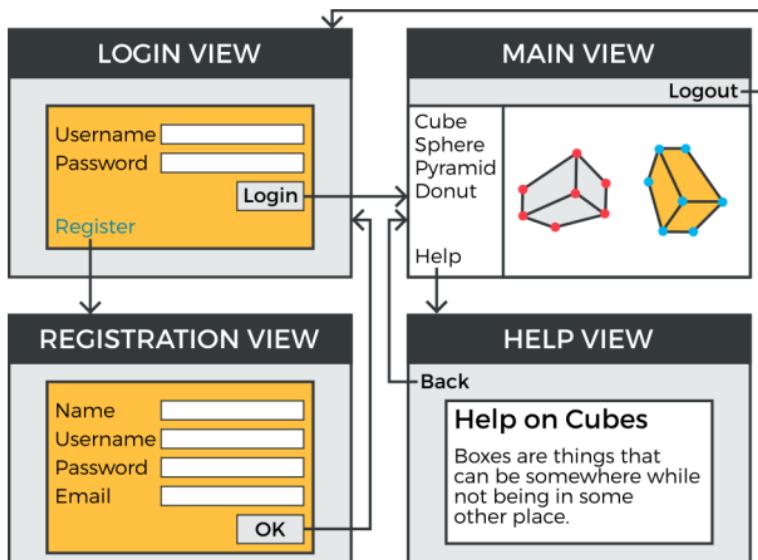


Figure 5.4. Navigation Between Views

The **Navigator** described in Section 11.8, "Navigating in an Application" is a view manager that provides a flexible way to navigate between views and sub-views, while managing the URI fragment in the page URL to allow bookmarking, linking, and going back in the browser history.

Often Vaadin application views are part of something bigger. In such cases, you may need to integrate the Vaadin applications with the other website. You can use the embedding techniques described in Section 11.2, "Embedding UIs in Web Pages".

5.2.4. Accessing UI, Page, Session, and Service

You can get the UI and the page to which a component is attached to with `getUI()` and `getPage()`.

However, the values are null until the component is attached to the UI, and typically, when you need it in constructors, it is

not. It is therefore preferable to access the current UI, page, session, and service objects from anywhere in the application using the static getCurrent() methods in the respective **UI**, **Page**, **VaadinSession**, and **VaadinService** classes.

```
// Set the default locale of the UI  
UI.getCurrent().setLocale(new Locale("en"));  
  
// Set the page title (window or tab caption)  
Page.getCurrent().setTitle("My Page");  
  
// Set a session attribute  
VaadinSession.getCurrent().setAttribute("myattrib", "hello");  
  
// Access the HTTP service parameters  
File baseDir = VaadinService.getCurrent().getBaseDirectory();
```

You can get the page and the session also from a **UI** with getPage() and getSession() and the service from **VaadinSession** with getService().

The static methods use the built-in ThreadLocal support in the classes.

5.3. Designing UIs Declaratively

Declarative definition of composites and even entire UIs makes it easy for developers and especially graphical designers to work on visual designs without any coding. Designs can be modified even while the application is running, as can be the associated themes. A design is a representation of a component hierarchy, which can be accessed from Java code to implement dynamic UI logic, as well as data binding.

For example, considering the following layout in Java:

```
VerticalLayout vertical = new VerticalLayout();  
vertical.addComponent(new TextField("Name"));  
vertical.addComponent(new TextField("Street address"));  
vertical.addComponent(new TextField("Postal code"));  
layout.addComponent(vertical);
```

You could define it declaratively with the following equivalent design:

```
<vaadin-vertical-layout>  
  <vaadin-text-field caption="Name"/>
```

```
<vaadin-text-field caption="Street address"/>
<vaadin-text-field caption="Postal code"/>
</vaadin-vertical-layout>
```

Declarative designs can be crafted by hand, but are most conveniently created with the Vaadin Designer.

In the following, we first go through the syntax of the declarative design files, and then see how to use them in applications by binding them to data and handling user interaction events.

5.3.1. Declarative Syntax

A design is an HTML document with custom elements for representing components and their configuration. A design has a single root component inside the HTML body element. Enclosing `<html>`, `<head>`, and `<body>` are optional, but necessary if you need to make namespace definitions for custom components. Other regular HTML elements may not be used in the file, except inside components that specifically accept HTML content.

In a design, each nested element corresponds to a Vaadin component in a component tree. Components can have explicitly given IDs to enable binding them to variables in the Java code, as well as optional attributes.

```
<!DOCTYPE html>
<html>
<body>
  <vaadin-vertical-layout size-full>
    <!-- Label with HTML content -->
    <vaadin-label><b>Hello!</b> -
      How are you?</vaadin-label>

    <vaadin-grid _id="mygrid" caption="My Grid"
      size-full :expand/>
  </vaadin-vertical-layout>
</body>
</html>
```

The DOCTYPE is not required, neither is the `<html>`, or `<body>` elements. Nevertheless, there may only be one design root element.

The above design defines the same UI layout as done earlier with Java code, and illustrated in Figure 5.3, "Simple hierarchical UI".

5.3.2. Component Elements

HTML elements of the declarative syntax are directly mapped to Vaadin components according to their Java class names. The tag of a component element has a namespace prefix separated by a dash. Vaadin core components, which are defined in the com.vaadin.ui package, have vaadin- prefix. The rest of an element tag is determined from the Java class name of the component, by making it lower-case, while adding a dash (-) before every previously upper-case letter as a word separator. For example, **ComboBox** component has declarative element tag vaadin-combo-box.

Component Prefix to Package Mapping

You can use any components in a design: components extending Vaadin components, composite components, and add-on components. To do so, you need to define a mapping from an element prefix to the Java package of the component. The prefix is used as a sort of a namespace.

The mappings are defined in <meta name="package-mapping" ...> elements in the HTML head. A *content* attribute defines a mapping, in notation with a prefix separated from the corresponding Java package name with a colon, such as my:com.example.myapp.

For example, consider that you have the following composite class **com.example.myapp.ExampleComponent**:

```
package com.example.myapp;

public class ExampleComponent extends CustomComponent {
    public ExampleComponent() {
        setCompositionRoot(new Label("I am an example."));
    }
}
```

You would make the package prefix mapping and then use the component as follows:

```
<!DOCTYPE html>
<html>
<head>
  <meta name="package-mapping"
        content="my:com.example.myapp" />
</head>

<body>
  <vaadin-vertical-layout>
    <vaadin-label><b>Hello!</b> -
      How are you?</vaadin-label>

    <!-- Use it here -->
    <my-example-component>
  </vaadin-vertical-layout>
</body>
</html>
```

Inline Content and Data

The element content can be used for certain default attributes, such as a button caption. For example:

```
<vaadin-button><b>OK</b></vaadin-button>
```

Some components, such as selection components, allow defining inline data within the element. For example:

```
<vaadin-native-select>
  <option>Mercury</option>
  <option>Venus</option>
  <option selected>Earth</option>
</vaadin-native-select>
```

The declarative syntax of each component type is described in the JavaDoc API documentation of Vaadin.

5.3.3. Component Attributes

Attribute-to-Property Mapping

Component properties are directly mapped to the attributes of the HTML elements according to the names of the properties. Attributes are written in lower-case letters and dash is used for word separation instead of upper-case letters in the

Java methods, so that placeholder attribute is equivalent to setPlaceholder().

For example, the *caption* property, which you can set with setCaption(), is represented as caption attribute. You can find the component properties by the setter methods in the JavaDoc API documentation of the component classes.

```
<vaadin-text-field caption="Name" placeholder="Enter Name"/>
```

Attribute Values

Attribute parameters must be enclosed in quotes and the value given as a string must be convertible to the type of the property (string, integer, boolean, or enumeration). Object types are not supported.

Some attribute names are given by a shorthand. For example, *alternateText* property of the **Image** component, which you would set with setAlternateText(), is given as the alt attribute.

Boolean values must be either true or false. The value can be omitted, in which case true is assumed. For example, the enabled attribute is boolean and has default value "true", so enabled="true" and enabled are equivalent.

```
<vaadin-button enabled="false">OK</vaadin-button>
```

Parent Component Settings

Certain settings, such as a component's alignment in a layout, are not done in the component itself, but in the layout. Attributes prefixed with colon (:) are passed to the containing component, with the component as a target parameter. For example, :expand="1" given for a component c is equivalent to calling setExpandRatio(c, 1) for the containing layout.

```
<vaadin-vertical-layout size-full>
  <!-- Align right in the containing layout -->
  <vaadin-label width-auto :right>Hello!</vaadin-label>

  <!-- Expands to take up all remaining vertical space -->
  <vaadin-horizontal-layout size-full :expand>
    <!-- Automatic width - shrinks horizontally -->
    <vaadin-radio-button-group width-auto height-full/>

    <!-- Expands horizontally to take remaining space -->
```

```
<vaadin-grid size-full :expand/>
</vaadin-horizontal-layout>
</vaadin-vertical-layout>
```

5.3.4. Component Identifiers

Components can be identified by either an identifier or a caption. There are two types of identifiers: page-global and local. This allows accessing them from Java code and binding them to components, as described later in Section 5.3.5, "Using Designs in Code".

The id attribute can be used to define a page-global identifier, which must be unique within the page. Another design or UI shown simultaneously in the same page may not have components sharing the same ID. Using global identifiers is therefore not recommended, except in special cases where uniqueness is ensured.

The _id attribute defines a local identifier used only within the design. This is the recommended way to identifying components.

```
<vaadin-grid _id="mygrid" caption="My Grid"/>
```

5.3.5. Using Designs in Code

The main use of declarative designs is in building application views, sub-views, dialogs, and forms through composition. The two main tasks are filling the designs with application data and handling user interaction events.

Binding to a Design Root

You can bind any component container as the root component of a design with the **@DesignRoot** annotation. The class must match or extend the class of the root element in the design.

The member variables are automatically initialized from the design according to the component identifiers (see Section 5.3.4, "Component Identifiers"), which must match the variable names.

For example, the following class could be used to bind the design given earlier.

```
@DesignRoot
public class MyViewDesign extends VerticalLayout {
    RadioButtonGroup<String> myRadioButtonGroup;
    Grid<String> myGrid;

    public MyViewDesign() {
        Design.read("MyDeclarativeUI.html", this);

        // Show some (example) data
        myCheckBoxGroup.setItems("Venus", "Earth", "Mars");
        myGrid.setItems(
            GridExample.generateContent());

        // Some interaction
        myCheckBoxGroup.addValueChangeListener(event ->
            Notification.show("Selected " +
                event.getValue());
    }
}
```

The design root class must match or extend the root element class of the design. For example, earlier we had `<vaadin-vertical-layout>` element in the HTML file, which can be bound to a class extending **VerticalLayout**.

Using a Design

The fact that a component is defined declaratively is not visible in its API, so you can create and use such it just like any other component.

For example, to use the previously defined design root component as the content of the entire UI:

```
public class DeclarativeViewUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        setContent(new MyViewDesign());
    }
}
```

Designs in View Navigation

To use a design in view navigation, as described in Section 11.8, “Navigating in an Application”, you just need to implement the View interface.

```
@DesignRoot
public class MainView extends VerticalLayout
    implements View {
    public MainView() {
        Design.read(this);
        ...
    }
    ...
}

// Use the view by precreating it
navigator.addView(MAINVIEW, new MainView());
```

See Section 11.8.3, “Handling URI Fragment Path” for a complete example.

5.4. Handling Events with Listeners

Let us put into practice what we learned of event handling in Section 4.4, “Events and Listeners”. You can implement listener interfaces by directly using lambda expressions, method references or anonymous classes.

For example, in the following, we use a lambda expression to handle button click events in the constructor:

```
layout.addComponent(new Button("Click Me!",
    event -> event.getButton().setCaption("You made click!")));
```

Directing events to handler methods is easy with method references:

```
public class Buttons extends CustomComponent {
    public Buttons() {
        setCompositionRoot(new HorizontalLayout(
            new Button("OK", this::ok),
            new Button("Cancel", this::cancel)));
    }

    private void ok(ClickEvent event) {
```

```
    event.getButton().setCaption ("OK!");
}

private void cancel(ClickEvent event) {
    event.getButton().setCaption ("Not OK!");
}
}
```

5.4.1. Using Anonymous Classes

The following example defines an anonymous class that inherits the **Button.ClickListener** interface.

```
// Have a component that fires click events
Button button = new Button("Click Me!");

// Handle the events with an anonymous class
button.addClickListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        button.setCaption("You made me click!");
    }
});
```

Most components allow passing a listener to the constructor. Note that to be able to access the component from the anonymous listener class, you must have a reference to the component that is declared before the constructor is executed, for example as a member variable in the outer class. You can also get a reference to the component from the event object:

```
final Button button = new Button("Click It!");
new Button.ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        event.getButton().setCaption("Done!");
    }
});
```

5.5. Images and Other Resources

Web applications can display various *resources*, such as images, other embedded content, or downloadable files, that the browser has to load from the server. Image resources are typically displayed with the **Image** component or as compon-

ent icons. Embedded browser frames can be displayed with **BrowserFrame**, and other content with the **Embedded** component, as described in Section 6.29, "Embedded Resources". Downloadable files are usually provided by clicking a **Link** or using the **FileDownloader** extension.

There are several ways to how such resources can be provided by the web server. Static resources can be provided without having to ask for them from the application. For dynamic resources, the user application must be able to create them dynamically. The resource request interfaces in Vaadin allow applications to both refer to static resources as well as dynamically create them. The dynamic creation includes the **StreamResource** class and the RequestHandler described in Section 11.4, "Request Handlers".

Vaadin also provides low-level facilities for retrieving the URI and other parameters of a HTTP request. We will first look into how applications can provide various kinds of resources and then look into low-level interfaces for handling URIs and parameters to provide resources and functionalities.

Notice that using request handlers to create "pages" is not normally meaningful in Vaadin or in AJAX applications generally. Please see Section 4.2.3, "AJAX" for a detailed explanation.

5.5.1. Resource Interfaces and Classes

The resource classes in Vaadin are grouped under two interfaces: a generic **Resource** interface and a more specific **ConnectorResource** interface for resources provided by the servlet.

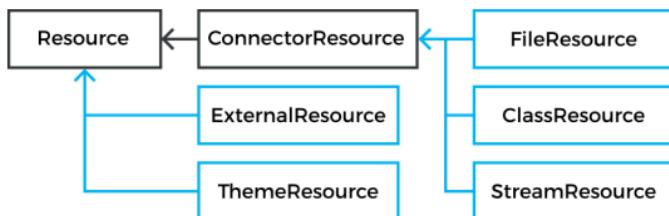


Figure 5.5. Resource Interface and Class Diagram

5.5.2. File Resources

File resources are files stored anywhere in the file system. As such, they can not be retrieved by a regular URL from the server, but need to be requested through the Vaadin servlet. The use of file resources is typically necessary for persistent user data that is not packaged in the web application, which would not be persistent over redeployments.

A file object that can be accessed as a file resource is defined with the standard **java.io.File** class. You can create the file either with an absolute or relative path, but the base path of the relative path depends on the installation of the web server. For example, with Apache Tomcat, the default current directory would be the installation path of Tomcat.

In the following example, we provide an image resource from a file stored in the web application. Notice that the image is stored under the WEB-INF folder, which is a special folder that is never accessible using an URL, unlike the other folders of a web application. This is a security solution - another would be to store the resource elsewhere in the file system.

```
// Find the application directory
String basepath = VaadinService.getCurrent()
    .getBaseDirectory().getAbsolutePath();

// Image as a file resource
FileResource resource = new FileResource(new File(basepath +
    "/WEB-INF/images/image.png"));

// Show the image in the application
Image image = new Image("Image from file", resource);

// Let the user view the file in browser or download it
Link link = new Link("Link to the image file", resource);
```

In a Maven based Vaadin project the image file should be located inside src/main/webapp/WEB-INF/images/image.png.

5.5.3. Class Loader Resources

The **ClassResource** allows resources to be loaded from the class path using Java Class Loader. Normally, the relevant class path entry is the WEB-INF/classes folder under the web applic-

ation, where the Java compilation should compile the Java classes and copy other files from the source tree.

The one-line example below loads an image resource from the application package and displays it in an **Image** component.

```
layout.addComponent(new Image(null,  
    new ClassResource("smiley.jpg")));
```

5.5.4. Theme Resources

Theme resources of **ThemeResource** class are files, typically images, included in a theme. A theme is located with the path VAADIN/themes/themename in a web application. The name of a theme resource is given as the parameter for the constructor, with a path relative to the theme folder.

```
// A theme resource in the current theme ("mytheme")  
// Located in: VAADIN/themes/mytheme/img/themeimage.png  
ThemeResource resource = new ThemeResource("img/themeimage.png");  
  
// Use the resource  
Image image = new Image("My Theme Image", resource);
```

To use theme resources, you must set the theme for the UI. See Chapter 9, *Themes* for more information regarding themes.

5.5.5. Stream Resources

Stream resources allow creating dynamic resource content. Charts are typical examples of dynamic images. To define a stream resource, you need to implement the **StreamResource.StreamSource** interface and its `getStream()` method. The method needs to return an **InputStream** from which the stream can be read.

The following example demonstrates the creation of a simple image in PNG image format.

```
import java.awt.image.*;  
  
public class MyImageSource implements StreamSource {  
    ByteArrayOutputStream imagebuffer = null;  
    int reloads = 0;  
  
    // This method generates the stream contents
```

```

public InputStream getStream () {
    // Create an image
    BufferedImage image = new BufferedImage (400, 400,
        BufferedImage.TYPE_INT_RGB);
    Graphics2D drawable = image.createGraphics();

    // Draw something static
    drawable.setStroke(new BasicStroke(5));
    drawable.setColor(Color.WHITE);
    drawable.fillRect(0, 0, 400, 400);
    drawable.setColor(Color.BLACK);
    drawable.drawOval(50, 50, 300, 300);

    // Draw something dynamic
    drawable.setFont(new Font("Montserrat",
        Font.PLAIN, 48));
    drawable.drawString("Reloads=" + reloads, 75, 216);
    reloads++;
    drawable.setColor(new Color(0, 165, 235));
    int x= (int) (200-10 + 150*Math.sin(reloads * 0.3));
    int y= (int) (200-10 + 150*Math.cos(reloads * 0.3));
    drawable.fillOval(x, y, 20, 20);

    try {
        // Write the image to a buffer
        imagebuffer = new ByteArrayOutputStream();
        ImageIO.write(image, "png", imagebuffer);

        // Return a stream from the buffer
        return new ByteArrayInputStream(
            imagebuffer.toByteArray());
    } catch (IOException e) {
        return null;
    }
}
}

```

The content of the generated image is dynamic, as it updates the reloads counter with every call. The **ImageIO.write()** method writes the image to an output stream, while we had to return an input stream, so we stored the image contents to a temporary buffer.

Below we display the image with the **Image** component.

```

// Create an instance of our stream source.
StreamSource imagesource = new MyImageSource();

// Create a resource that uses the stream source and give it
// a name. The constructor will automatically register the
// resource in the application.
StreamResource resource =
    new StreamResource(imagesource, "myimage.png");

```

```
// Create an image component that gets its contents  
// from the resource.  
layout.addComponent(new Image("Image title", resource));
```

The resulting image is shown in Figure 5.6, "A stream resource".

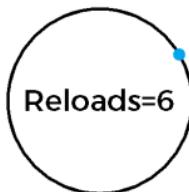


Figure 5.6. A stream resource

Another way to create dynamic content is a request handler, described in Section 11.4, "Request Handlers".

5.6. Handling Errors

5.6.1. Error Indicator and Message

All components have a built-in error indicator that is turned on if validating the component fails, and can be set explicitly with `setComponentError()`. Usually, the error indicator is placed right of the component caption. The error indicator is part of the component caption, so its placement is usually managed by the layout in which the component is contained, but some components handle it themselves. Hovering the mouse pointer over the field displays the error message.

```
textfield.setComponentError(new UserError("Bad value"));  
button.setComponentError(new UserError("Bad click"));
```

The result is shown in Figure 5.7, "Error Indicator Active".



Figure 5.7. Error Indicator Active

5.6.2. Connection Fault

If the connection to the server is lost, Vaadin application shows a "lost connection" notification and tries to restore the connection. After several retries, an error message is shown. You can customize the messages, timeouts, and the number of reconnect attempts in the **ReconnectDialogConfiguration** object, which you can access from your **UI** with `getReconnectDialogConfiguration()`.

5.6.3. Customizing System Messages

System messages are notifications that indicate a major invalid state that usually requires restarting the application. Session timeout is perhaps the most typical such state.

System messages are strings managed in the **SystemMessages** class. Each message has four properties: a short caption, the actual message, a URL to which to redirect after displaying the message, and property indicating whether the notification is enabled.

You can override the default system messages by setting the `SystemMessagesProvider` in the **VaadinService**. You need to implement the `getSystemMessages()` method, which should return a **SystemMessages** object. The easiest way to customize the messages is to use a **CustomizedSystemMessages** object.

You can set the system message provider in the `servletInitialized()` method of a custom servlet class, for example as follows:

```
getService().setSystemMessagesProvider(  
    new SystemMessagesProvider() {  
        @Override  
        public SystemMessages getSystemMessages(  
            SystemMessagesInfo systemMessagesInfo) {  
            CustomizedSystemMessages messages =  
                new CustomizedSystemMessages();  
            messages.setCommunicationErrorCaption("Comm Err");  
            messages.setCommunicationErrorMessage("This is bad.");  
            messages.setCommunicationErrorNotificationEnabled(true);  
            messages.setCommunicationErrorURL("http://vaadin.com/");  
            return messages;  
        }  
    });
```

See Section 5.8.2, “Vaadin Servlet, Portlet, and Service” for information about customizing Vaadin servlets.

5.6.4. Handling Uncaught Exceptions

Handling events can result in exceptions either in the application logic or in the framework itself, but some of them may not be caught properly by the application. Any such exceptions are eventually caught by the framework. It delegates the exceptions to the **DefaultErrorHandler**, which displays the error as a component error, that is, with a small red "!" - sign (depending on the theme). If the user hovers the mouse pointer over it, the entire backtrace of the exception is shown in a large tooltip box.

You can customize the default error handling by implementing a custom ErrorHandler and enabling it with setErrorHandler() in any of the components in the component hierarchy, including the **UI**, or in the **VaadinSession** object. You can either implement the ErrorHandler or extend the **DefaultErrorHandler**. In the following example, we modify the behavior of the default handler.

```
// Here's some code that produces an uncaught exception
final VerticalLayout layout = new VerticalLayout();
final Button button = new Button("Click Me!", event ->
    ((String)null).length()); // Null-pointer exception
layout.addComponent(button);

// Configure the error handler for the UI
UI.getCurrent().setErrorHandler(new DefaultErrorHandler() {
    @Override
    public void error(com.vaadin.server.ErrorEvent event) {
        // Find the final cause
        String cause = "<b>The click failed because:</b><br/>";
        for (Throwable t = event.getThrowable(); t != null;
            t = t.getCause())
            if (t.getCause() == null) // We're at final cause
                cause += t.getClass().getName() + "<br/>";

        // Display the error message in a custom fashion
        layout.addComponent(new Label(cause, ContentMode.HTML));

        // Do the default error handling (optional)
        doDefault(event);
    }
});
```

The above example also demonstrates how to dig up the final cause from the cause stack.

When extending **DefaultErrorHandler**, you can call `doDefault()` as was done above to run the default error handling, such as set the component error for the component where the exception was thrown. See the source code of the implementation for more details. You can call `findAbstractComponent(event)` to find the component that caused the error. If the error is not associated with a component, it returns null.

5.7. Notifications

Notifications are error or information boxes that appear briefly, typically at the center of the screen. A notification box has a caption and an optional description and icon. The box stays on the screen either for a preset time or until the user clicks it. The notification type defines the default appearance and behaviour of a notification.

There are two ways to create a notification. The easiest is to use a static shorthand `Notification.show()` method, which takes the caption of the notification as a parameter, and an optional description and notification type, and displays it in the current page.

```
Notification.show("This is the caption",
  "This is the description",
  Notification.Type.HUMANIZED_MESSAGE);
```

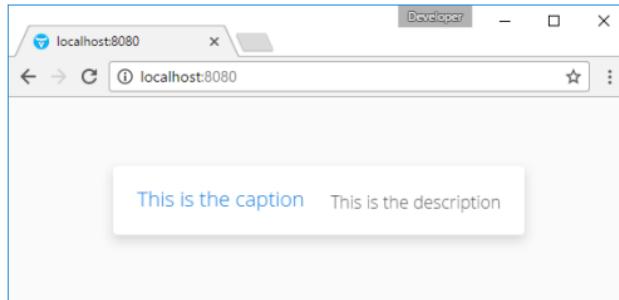


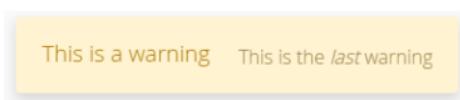
Figure 5.8. Notification

For more control, you can create a **Notification** object. Different constructors exist for taking just the caption, and optionally the description, notification type, and whether HTML is allowed

or not. Notifications are shown in a **Page**, typically the current page.

```
new Notification("This is a warning",
  "This is the <i>last</i> warning",
  Notification.Type.WARNING_MESSAGE, true)
.show(Page.getCurrent());
```

The caption and description are by default written on the same line. If you want to have a line break between them, use the HTML line break markup "`
`" if HTML is enabled, or "`\n`" if not. HTML is disabled by default, but can be enabled with `setHtmlContentAllowed(true)`. When enabled, you can use any HTML markup in the caption and description of a notification. If it is in any way possible to get the notification content from user input, you should either disallow HTML or sanitize the content carefully, as noted in Section 11.7.1, "Sanitizing User Input to Prevent Cross-Site Scripting".



This is a warning This is the /last warning

Figure 5.9. Notification with HTML Formatting

5.7.1. Notification Type

The notification type defines the overall default style and behaviour of a notification. If no notification type is given, the "humanized" type is used as the default. The notification types, listed below, are defined in the **Notification.Type** class.

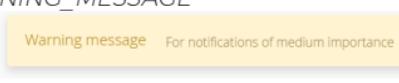
`HUMANIZED_MESSAGE`



Humanized message For minimal annoyance

A user-friendly message that does not annoy too much: it does not require confirmation by clicking and disappears quickly. It is centered and has a neutral gray color.

`WARNING_MESSAGE`



Warning message For notifications of medium importance

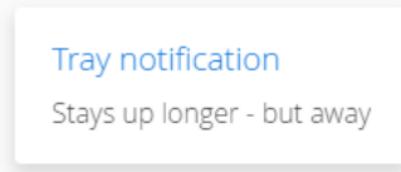
Warnings are messages of medium importance. They are displayed with colors that are neither neutral nor too distractible. A warning is displayed for 1.5 seconds, but the user can click the message box to dismiss it. The user can continue to interact with the application while the warning is displayed.

ERROR_MESSAGE



Error messages are notifications that require the highest user attention, with alert colors, and they require the user to click the message to dismiss it. The error message box does not itself include an instruction to click the message, although the close box in the upper right corner indicates it visually. Unlike with other notifications, the user can not interact with the application while the error message is displayed.

TRAY_NOTIFICATION



Tray notifications are displayed in the "system tray" area, that is, in the lower-right corner of the browser view. As they do not usually obscure any user interface, they are displayed longer than humanized or warning messages, 3 seconds by default. The user can continue to interact with the application normally while the tray notification is displayed.

5.7.2. Customizing Notifications

All of the features of specific notification types can be controlled with the **Notification** properties. Once configured, you need to show it in the current page.

```
// Notification with default settings for a warning
Notification notif = new Notification(
    "Warning",
    "Area of reindeer husbandry",
    Notification.TYPE_WARNING_MESSAGE);

// Customize it
notif.setDelayMsec(20000);
notif.setPosition(Position.BOTTOM_RIGHT);
notif.setStyleName("mystyle");
notif.setIcon(new ThemeResource("img/reindeer.png"));

// Show it in the page
notif.show(Page.getCurrent());
```

The `setPosition()` method allows setting the positioning of the notification. The position can be specified by any of the constants defined in the **Position** enum.

The `setDelayMSec()` allows setting the time for how long the notification is displayed in milliseconds. Parameter value -1 means that the message is displayed until the user clicks the message box. It also prevents interaction with other parts of the application window, which is the default behaviour for error notifications. It does not, however, add a close box that the error notification has.

5.7.3. Styling with CSS

```
.v-Notification {}
.popoverContent {}
.gwt-HTML {}
  h1 {}
  p {}
```

The notification box is a floating div element under the body element of the page. It has an overall v-Notification style. The content is wrapped inside an element with `popoverContent` style. The caption is enclosed within an `h1` element and the description in a `p` element.

To customize it, add a style for the **Notification** object with `setStyleName("mystyle")`, and make the settings in the theme, for example as follows:

```
.v-Notification.mystyle {  
    background: #FFFF00;  
    border: 10px solid #C00000;  
    color: black;  
}
```

The result is shown, with the icon set earlier in the customization example, in Figure 5.10, "A Styled Notification".



Figure 5.10. A Styled Notification

5.8. Application Lifecycle

In this section, we look into more technical details of application deployment, user sessions, and UI instance lifecycle. These details are not generally needed for writing Vaadin applications, but may be useful for understanding how they actually work and, especially, in what circumstances their execution ends.

5.8.1. Deployment

Before a Vaadin application can be used, it has to be deployed to a Java web server, as described in Section 5.9, "Deploying an Application". Deploying reads the servlet classes annotated with the @WebServlet annotation or the web.xml deployment descriptor in the application to register servlets for specific URL paths and loads the classes. Deployment does not yet normally run any code in the application, although static blocks in classes are executed when they are loaded.

Undeploying and Redeploying

Applications are undeployed when the server shuts down, during redeployment, and when they are explicitly un-

deployed. Undeploying a server-side Vaadin application ends its execution, all application classes are unloaded, and the heap space allocated by the application is freed for garbage-collection.

If any user sessions are open at this point, the client-side state of the UIs is left hanging and an Out of Sync error is displayed on the next server request.

Redeployment and Serialization

Some servers, such as Tomcat, support *hot deployment*, where the classes are reloaded while preserving the memory state of the application. This is done by serializing the application state and then deserializing it after the classes are reloaded. This is, in fact, done with the basic Eclipse setup with Tomcat and if a UI is marked as **@PreserveOnRefresh**, you may actually need to give the ?restartApplication URL parameter to force it to restart when you reload the page. Tools such as JRebel go even further by reloading the code in place without need for serialization. The server can also serialize the application state when shutting down and restarting, thereby preserving sessions over restarts.

Serialization requires that the applications are *Serializable*, that is, all classes implement the Serializable interface. All Vaadin classes do. If you extend them or implement interfaces, you can provide an optional serialization key, which is automatically generated by Eclipse if you use it. Serialization is also used for clustering and cloud computing.

5.8.2. Vaadin Servlet, Portlet, and Service

The **VaadinServlet**, or **VaadinPortlet** in a portal, receives all server requests mapped to it by its URL, as defined in the deployment configuration, and associates them with sessions. The sessions further associate the requests with particular UIs.

When servicing requests, the Vaadin servlet or portlet handles all tasks common to both servlets and portlets in a **VaadinService**. It manages sessions, gives access to the deployment configuration information, handles system messages, and does various other tasks. Any further servlet or portlet specific

tasks are handled in the corresponding **VaadinServletService** or **VaadinPortletService**. The service acts as the primary low-level customization layer for processing requests.

Customizing Vaadin Servlet

Many common configuration tasks need to be done in the servlet class, which you already have if you are using the @WebServlet annotation for Servlet 3.0 to deploy the application. You can handle most customization by overriding the `servletInitialized()` method, where the **VaadinService** object is available with `getService()` (it would not be available in a constructor). You should always call `super.servletInitialized()` in the beginning.

```
public class MyServlet extends VaadinServlet {  
    @Override  
    protected void servletInitialized()  
        throws ServletException {  
        super.servletInitialized();  
        ...  
    }  
}
```

To add custom functionality around request handling, you can override the `service()` method.

Customizing Vaadin Service

To customize **VaadinService**, you first need to extend the **VaadinServlet** or - **Portlet** class and override the `createServletService()` to create a custom service object.

5.8.3. User Session

A user session begins when a user first makes a request to a Vaadin servlet or portlet by opening the URL for a particular **UI**. All server requests belonging to a particular UI class are processed by the **VaadinServlet** or **VaadinPortlet** class. When a new client connects, it creates a new user session, represented by an instance of **VaadinSession**. Sessions are tracked using cookies stored in the browser.

You can obtain the **VaadinSession** of a **UI** with `getSession()` or globally with `VaadinSession.getCurrent()`. It also provides access

to the lower-level session objects, HttpSession and PortletSession, through a **WrappedSession**. You can also access the deployment configuration through **VaadinSession**, as described in Section 5.9.7, “Deployment Configuration”.

A session ends after the last **UI** instance expires or is closed, as described later.

Handling Session Initialization and Destruction

You can handle session initialization and destruction by implementing a SessionInitListener or SessionDestroyListener, respectively, to the **VaadinService**. You can do that best by extending **VaadinServlet** and overriding the servletInitialized() method, as outlined in Section 5.8.2, “Vaadin Servlet, Portlet, and Service”.

```
public class MyServlet extends VaadinServlet
    implements SessionInitListener, SessionDestroyListener {

    @Override
    protected void servletInitialized() throws ServletException {
        super.servletInitialized();
        getService().addSessionInitListener(this);
        getService().addSessionDestroyListener(this);
    }

    @Override
    public void sessionInit(SessionInitEvent event)
        throws ServiceException {
        // Do session start stuff here
    }

    @Override
    public void sessionDestroy(SessionDestroyEvent event) {
        // Do session end stuff here
    }
}
```

5.8.4. Loading a UI

When a browser first accesses a URL mapped to the servlet of a particular UI class, the Vaadin servlet generates a loader page. The page loads the client-side engine (widget set), which in turn loads the UI in a separate request to the Vaadin servlet.

A **UI** instance is created when the client-side engine makes its first request. The servlet creates the UIs using a **UIProvider** registered in the **VaadinSession** instance. A session has at

least a **DefaultUIProvider** for managing UIs opened by the user. If the application lets the user open popup windows with a **BrowserWindowOpener**, each of them has a dedicated special UI provider.

Once a new UI is created, its `init()` method is called. The method gets the request as a **VaadinRequest**.

Customizing the Loader Page

The HTML content of the loader page is generated as an HTML DOM object, which can be customized by implementing a `BootstrapListener` that modifies the DOM object. To do so, you need to extend the **VaadinServlet** and add a `SessionInitListener` to the service object, as outlined in Section 5.8.3, "User Session". You can then add the bootstrap listener to a session with `addBootstrapListener()` when the session is initialized.

Loading the widget set is handled in the loader page with functions defined in a separate `vaadinBootstrap.js` script.

You can also use entirely custom loader code, such as in a static HTML page, as described in Section 11.2, "Embedding UIs in Web Pages".

Custom UI Providers

You can create UI objects dynamically according to their request parameters, such as the URL path, by defining a custom `UIProvider`. You need to add custom UI providers to the session object which calls them. The providers are chained so that they are requested starting from the one added last, until one returns a UI (otherwise they return null). You can add a UI provider to a session most conveniently by implementing a custom servlet and adding the UI provider to sessions in a `SessionInitListener`.

Preserving UI on Refresh

Reloading a page in the browser normally spawns a new **UI** instance and the old UI is left hanging, until cleaned up after a while. This can be undesired as it resets the UI state for the user. To preserve the UI, you can use the **@PreserveOnRefresh**

annotation for the UI class. You can also use a **UIProvider** with a custom implementation of `isUiPreserved()`.

```
@PreserveOnRefresh  
public class MyUI extends UI {
```

Adding the `?restartApplication` parameter in the URL tells the Vaadin servlet to create a new **UI** instance when loading the page, thereby overriding the **@PreserveOnRefresh**. This is often necessary when developing such a UI in Eclipse, when you need to restart it after redeploying, because Eclipse likes to persist the application state between redeployments. If you also include a URI fragment, the parameter should be given before the fragment.

5.8.5. UI Expiration

UI instances are cleaned up if no communication is received from them after some time. If no other server requests are made, the client-side sends keep-alive heartbeat requests. A UI is kept alive for as long as requests or heartbeats are received from it. It expires if three consecutive heartbeats are missed.

The heartbeats occur at an interval of 5 minutes, which can be changed with the `heartbeatInterval` parameter of the servlet. You can configure the parameter in **@VaadinServletConfiguration** or in `web.xml` as described in Section 5.9.6, "Other Servlet Configuration Parameters".

When the UI cleanup happens, a **DetachEvent** is sent to all **DetachListener#s added to the UI**. When the `[classname]#UI` is detached from the session, `detach()` is called for it.

5.8.6. Closing UIs Explicitly

You can explicitly close a UI with `close()`. The method marks the UI to be detached from the session after processing the current request. Therefore, the method does not invalidate the UI instance immediately and the response is sent as usual.

Detaching a UI does not close the page or browser window in which the UI is running and further server request will cause

error. Typically, you either want to close the window, reload it, or redirect it to another URL. If the page is a regular browser window or tab, browsers generally do not allow closing them programmatically, but redirection is possible. You can redirect the window to another URL with `setLocation()`, as is done in the examples in Section 5.8.8, "Closing a Session". You can close popup windows by making JavaScript `close()` call for them, as described in Section 11.1.2, "Closing Popup Windows".

If you close other UI than the one associated with the current request, they will not be detached at the end of the current request, but after next request from the particular UI. You can make that occur quicker by making the UI heartbeat faster or immediately by using server push.

5.8.7. Session Expiration

A session is kept alive by server requests caused by user interaction with the application as well as the heartbeat monitoring of the UIs. Once all UIs have expired, the session still remains. It is cleaned up from the server when the session timeout configured in the web application expires.

If there are active UIs in an application, their heartbeat keeps the session alive indefinitely. You may want to have the sessions timeout if the user is inactive long enough, which is the original purpose of the session timeout setting. If the `idleSessions` parameter of the servlet is set to true in the web.xml, as described in Section 5.9.4, "Using a web.xml Deployment Descriptor", the session and all of its UIs are closed when the timeout specified by the `session-timeout` parameter of the servlet expires after the last non-heartbeat request. Once the session is gone, the browser will show an Out Of Sync error on the next server request. To avoid the ugly message, you may want to set a redirect URL for the UIs, as described in Section 5.6.3, "Customizing System Messages".

The related configuration parameters are described in Section 5.9.6, "Other Servlet Configuration Parameters".

You can handle session expiration on the server-side with a `SessionDestroyListener`, as described in Section 5.8.3, "User Session".

5.8.8. Closing a Session

You can close a session by calling `close()` on the **VaadinSession**. It is typically used when logging a user out and the session and all the UIs belonging to the session should be closed. The session is closed immediately and any objects related to it are not available after calling the method.

When closing the session from a UI, you typically want to redirect the user to another URL. You can do the redirect using the `setLocation()` method in **Page**. This needs to be done before closing the session, as the UI or page are not available after that. In the following example, we display a logout button, which closes the user session.

```
public class MyUI extends UI {  
    @Override  
    protected void init(VaadinRequest request) {  
        setContent(new Button("Logout", event -> {  
            // Redirect this page immediately  
            getPage().setLocation("/myapp/logout.html");  
  
            // Close the session  
            getSession().close();  
        }));  
  
        // Notice quickly if other UIs are closed  
        setPollInterval(3000);  
    }  
}
```

This is not enough. When a session is closed from one UI, any other UIs attached to it are left hanging. When the client-side engine notices that a UI and the session are gone on the server-side, it displays a "Session Expired" message and, by default, reloads the UI when the message is clicked. You can customize the message and the redirect URL in the system messages.

It is described in Section 5.6.3, "Customizing System Messages".

The client-side engine notices the expiration when user interaction causes a server request to be made or when the keep-alive heartbeat occurs. To make the UIs detect the situation faster, you need to make the heart beat faster, as was done in the example above. You can also use server push to close

the other UIs immediately, as is done in the following example. Access to the UIs must be synchronized as described in Section 11.15, "Server Push".

```
@Push
public class MyPushyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        setContent(new Button("Logout", event -> {
            for (UI ui: VaadinSession.getCurrent().getUIs()) {
                ui.access(() -> {
                    // Redirect from the page
                    ui.getPage().setLocation("/logout.html");
                });
            }
        }));
    }
}
```

In the above example, we assume that all UIs in the session have push enabled and that they should be redirected; popups you might want to close instead of redirecting. It is not necessary to call `close()` for them individually, as we close the entire session afterwards.

5.9. Deploying an Application

Vaadin Framework applications are deployed as *Java web applications*, which can contain a number of servlets, each of which can be a Vaadin application or some other servlet, and static resources such as HTML files. Such a web application is normally packaged as a WAR (Web application ARchive) file, which can be deployed to a Java application server (or a servlet container to be exact). A WAR file, which has the `.war` extension, is a subtype of JAR (Java ARchive), and like a regular JAR, is a ZIP-compressed file with a special content structure.

For a detailed tutorial on how web applications are packaged, please refer to any Java book that discusses Java Servlets.

In the Java Servlet parlance, a "web application" means a collection of Java servlets or portlets, JSP and static HTML pages, and various other resources that form an application. Such a Java web application is typically packaged as a WAR package for deployment. Server-side Vaadin UIs run as servlets within such a Java web application. There exists also other

kinds of web applications. To avoid confusion with the general meaning of "web application", we often refer to Java web applications with the slight misnomer "WAR" in this book.

5.9.1. Creating Deployable WAR in Eclipse

To deploy an application to a web server, you need to create a WAR package. Here we give the instructions for Eclipse.

1. Select **File → Export** and then **Web → WAR File**. Or, right-click the project in the Project Explorer and select **Web → WAR File**.
2. Select the **Web project** to export. Enter **Destination** file name (.war).
3. Make any other settings in the dialog, and click **Finish**.

5.9.2. Web Application Contents

The following files are required in a web application in order to run it.

WEB-INF/web.xml (optional with Servlet 3.0)

This is the web application descriptor that defines how the application is organized, that is, what servlets and such it has. You can refer to any Java book about the contents of this file. It is not needed if you define the Vaadin servlet with the **@WebServlet** annotation in Servlet API 3.0.

WEB-INF/lib/*jar

These are the Vaadin libraries and their dependencies. They can be found in the installation package or as loaded by a dependency management system such as Maven.

Your UI classes

You must include your UI classes either in a JAR file in WEB-INF/lib or as classes in WEB-INF/classes

Your own theme files (OPTIONAL)

If your application uses a special theme (look and feel), you must include it in VAADIN/themes/themename directory.

Widget sets (OPTIONAL)

If your application uses add-ons or custom widgets, they must be compiled to the VAADIN/widgetset/directory. When using add-ons, this is done automatically in Maven projects. See Section 17.2, "Using Add-ons in a Maven Project" for more information.

5.9.3. Web Servlet Class

When using the Servlet 3.0 API, you normally declare the Vaadin servlet classes with the @WebServlet annotation. The Vaadin UI associated with the servlet and other Vaadin-specific parameters are declared with a separate @VaadinServletConfiguration annotation.

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

The Vaadin Plugin for Eclipse creates the servlet class as a static inner class of the UI class. Normally, you may want to have it as a separate regular class.

The *value* parameter is the URL pattern for mapping request URLs to the servlet, as described in Section 5.9.5, "Servlet Mapping with URL Patterns". The *ui* parameter is the UI class. Production mode is disabled by default, which enabled on-the-fly theme compilation, debug window, and other such development features. See the subsequent sections for details on the different servlet and Vaadin configuration parameters.

You can also use a web.xml deployment descriptor in Servlet 3.0 projects.

5.9.4. Using a web.xml Deployment Descriptor

A deployment descriptor is an XML file with the name web.xml in the WEB-INF sub-directory of a web application. It is a standard component in Java EE describing how a web application should be deployed. The descriptor is not required with Servlet API 3.0, where you can also define servlets with the **@WebServlet** annotation as described earlier, as web fragments, or programmatically. You can use both a web.xml and WebServlet in the same application. Settings in the web.xml override the ones given in annotations.

The following example shows the basic contents of a deployment descriptor. You simply specify the UI class with the *UI* parameter for the **com.vaadin.server.VaadinServlet**. The servlet is then mapped to a URL path in a standard way for Java Servlets.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    id="WebApp_ID" version="3.0"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

    <servlet>
        <servlet-name>myservlet</servlet-name>
        <servlet-class>
            com.vaadin.server.VaadinServlet
        </servlet-class>

        <init-param>
            <param-name>UI</param-name>
            <param-value>com.ex.myprj.MyUI</param-value>
        </init-param>

        <!-- If not using the default widget set-->
        <init-param>
            <param-name>widgetset</param-name>
            <param-value>com.ex.myprj.AppWidgetSet</param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>myservlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

The descriptor defines a servlet with the name myservlet. The servlet class, **com.vaadin.server.VaadinServlet**, is provided by Vaadin framework and is normally the same for all Vaadin projects. For some purposes, you may need to use a custom servlet class that extends the **VaadinServlet**. The class name must include the full package path.

Widget Set

The widget set is normally defined and compiled automatically in Maven projects. It may be necessary to define it manually in some cases, such as when developing custom widgets or if you need to include special rules in the widget set definition file (.gwt.xml module descriptor).

The widget set of a UI can be defined with the **@WidgetSet** annotation for the UI class.

```
@WidgetSet("com.example.myproject.MyWidgetSet")
class MyUI extends UI {
```

...

You can also define it with the *widgetset* init parameter for the servlet.

The name of a widget set is technically a Java class name with the same path as the widget set definition file, but without the .gwt.xml extension.

If a widget set is not specified, the default is used. In a project that does not use add-ons or custom widgets, the **com.vaadin.DefaultWidgetSet** is used. It contains all the widgets for the built-in Vaadin components. When using add-ons, the Vaadin Maven Plugin automatically defines an **AppWidgetSet** that includes all the add-on widget sets.

The widget set must be compiled, as described in Chapter 17, *Using Vaadin Add-ons* (for add-ons) or Section 13.4, “Compiling a Client-Side Module” (for custom widgets and client-side modules), and properly deployed with the application.

5.9.5. Servlet Mapping with URL Patterns

The servlet needs to be mapped to an URL path, which requests it is to handle.

With **@WebServlet** annotation for the servlet class:

```
@WebServlet(value = "/*", asyncSupported = true)
```

The URL pattern is defined in the above examples as `/*`. This matches any URL under the project context. We defined above the project context as `myproject` so the URL for the page of the UI will be `http://localhost:8080/myproject/`.

Mapping Sub-Paths

If an application has multiple UIs or servlets, they have to be given different paths in the URL, matched by a different URL pattern. Also, you may need to have statically served content under some path. Having an URL pattern `/myui/*` would match a URL such as `http://localhost:8080/myproject/myui/`. Notice that the slash and the asterisk *must* be included at the end of the pattern. In such case, you also need to map URLs with `/VAADIN/*` to a servlet (unless you are serving it statically as noted below).

With a **@WebServlet** annotation for a servlet class, you can define multiple mappings as a list enclosed in curly braces as follows:

```
@WebServlet(value = {"/myui/*", "/VAADIN/*"},  
           asyncSupported = true)
```

If you have multiple servlets, you should specify only one `/VAADIN/*` mapping. It does not matter which servlet you map the pattern to, as long as it is a Vaadin servlet.

You do not have to provide the above `/VAADIN/*` mapping if you serve both the widget sets and (custom and default) themes statically in the `/VAADIN` directory in the web application. The mapping simply allows serving them dynamically from the Vaadin JAR. Serving them statically is recommended for production environments as it is faster. If you serve the content from within the same web application, you may not

have the root pattern /* for the Vaadin servlet, as then all the requests would be mapped to the servlet.

5.9.6. Other Servlet Configuration Parameters

The servlet class or deployment descriptor can have many parameters and options that control the execution of a servlet. You can find complete documentation of the basic servlet parameters in the appropriate Java Servlet Specification.

@VaadinServletConfiguration accepts a number of special parameters, as described below.

In a web.xml, you can set most parameters either as a <context-param> for the entire web application, in which case they apply to all Vaadin servlets, or as an <init-param> for an individual servlet. If both are defined, servlet parameters override context parameters.

Production Mode

By default, Vaadin applications run in *debug mode* (or *development mode*), which should be used during development. This enables various debugging features. For production use, you should have the productionMode=true setting in the **@VaadinServletConfiguration**.

The parameter and the debug and production modes are described in more detail in Section 11.3, “Debug Mode and Window”.

Custom UI Provider

Vaadin normally uses the **DefaultUIProvider** for creating **UI** class instances. If you need to use a custom UI provider, you can define its class with the *UIProvider* parameter. The provider is registered in the **VaadinSession**.

The parameter is logically associated with a particular servlet, but can be defined in the context as well.

UI Heartbeat

Vaadin monitors UIs by using a heartbeat, as explained in Section 5.8.5, “UI Expiration”. If the user closes the browser window of a Vaadin application or navigates to another page, the Client-Side Engine running in the page stops sending heartbeat to the server, and the server eventually cleans up the **UI** instance.

The interval of the heartbeat requests can be specified in seconds with the *heartbeatInterval* parameter either as a context parameter for the entire web application or an init parameter for the individual servlet. The default value is 300 seconds (5 minutes).

Session Timeout After User Inactivity

In normal servlet operation, the session timeout defines the allowed time of inactivity after which the server should clean up the session. The inactivity is measured from the last server request. Different servlet containers use varying defaults for timeouts, such as 30 minutes for Apache Tomcat. There is no way to programmatically set the global session timeout, but you can set it in the deployment descriptor with:

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

The session timeout should be longer than the heartbeat interval or otherwise sessions are closed before the heartbeat can keep them alive. As the session expiration leaves the UIs in a state where they assume that the session still exists, this would cause an Out Of Sync error notification in the browser.

However, having a shorter heartbeat interval than the session timeout, which is the normal case, prevents the sessions from expiring. If the *closeldleSessions* parameter for the servlet is enabled (disabled by default), Vaadin closes the UIs and the session after the time specified in the *session-timeout* init parameter expires after the last non-heartbeat request.

Push Mode

You can enable server push, as described in Section 11.15, "Server Push", for a UI either with a `@Push` annotation for the UI or in the descriptor. The push mode is defined with a `pushmode` init parameter. The automatic mode pushes changes to the browser automatically after `access()` finishes. With manual mode, you need to do the push explicitly with `push()`. You can enable asynchronous processing with the `async-supported` init parameter.

Cross-Site Request Forgery Prevention

Vaadin uses a protection mechanism to prevent malicious cross-site request forgery (XSRF or CSRF), also called one-click attacks or session riding, which is a security exploit for executing unauthorized commands in a web server. This protection is normally enabled. However, it prevents some forms of testing of Vaadin applications, such as with JMeter. In such cases, you can disable the protection by setting the `disable-xsrf-protection` context parameter to true.

5.9.7. Deployment Configuration

The Vaadin-specific parameters defined in the deployment configuration are available from the `DeploymentConfiguration` object managed by the `VaadinSession`.

```
DeploymentConfiguration conf =  
    getSession().getConfiguration();  
  
// Heartbeat interval in seconds  
int heartbeatInterval = conf.getHeartbeatInterval();
```

Parameters defined in the Java Servlet definition, such as the session timeout, are available from the low-level `HttpSession` or `PortletSession` object, which are wrapped in a `WrappedSession` in Vaadin. You can access the low-level session wrapper with `getSession()` of the `VaadinSession`.

```
WrappedSession session = getSession().getSession();  
int sessionTimeout = session.getMaxInactiveInterval();
```

You can also access other `HttpSession` and `PortletSession` session properties through the interface, such as set and read

session attributes that are shared by all servlets belonging to a particular servlet or portlet session.

Chapter 6

User Interface Components

6.1. Overview	118
6.2. Interfaces and Abstractions	121
6.3. Common Component Features	123
6.4. Field Components	137
6.5. Selection Components	141
6.6. Component Extensions	145
6.7. Label	146
6.8. Link	149
6.9. TextField	152
6.10. TextArea	156
6.11. PasswordField	158
6.12. RichTextArea	159
6.13. Date Input with DateField	161
6.14. Button	167
6.15. CheckBox	169
6.16. ComboBox	170
6.17. ListSelect	173
6.18. NativeSelect	174
6.19. CheckBoxGroup and RadioButtonGroup	175
6.20. TwinColSelect	177
6.21. Grid	179
6.22. MenuBar	196
6.23. Upload	199
6.24. ProgressBar	202

6.25. Slider	203
6.26. PopupView	205
6.27. Composition with CustomComponent	207
6.28. Composite Fields with CustomField	209
6.29. Embedded Resources	210

This chapter provides an overview and a detailed description of all non-layout components in Vaadin.

6.1. Overview

Vaadin provides a comprehensive set of user interface components and allows you to define custom components.

The component hierarchy of Vaadin is presented in the next four diagrams:

- Figure 6.1, "Basic Components" illustrates the inheritance hierarchy of the UI component classes and interfaces not bound directly to business objects.
- Figure 6.2, "Field Components" presents fields bound to single values
- Figure 6.3, "Selection Components" presents components that show a list of data and allow selection
- Figure 6.4, "Layouts and Component Containers" layouts and other component containers

Interfaces are displayed with a dotted outline, abstract classes in gray, and regular classes in blue.

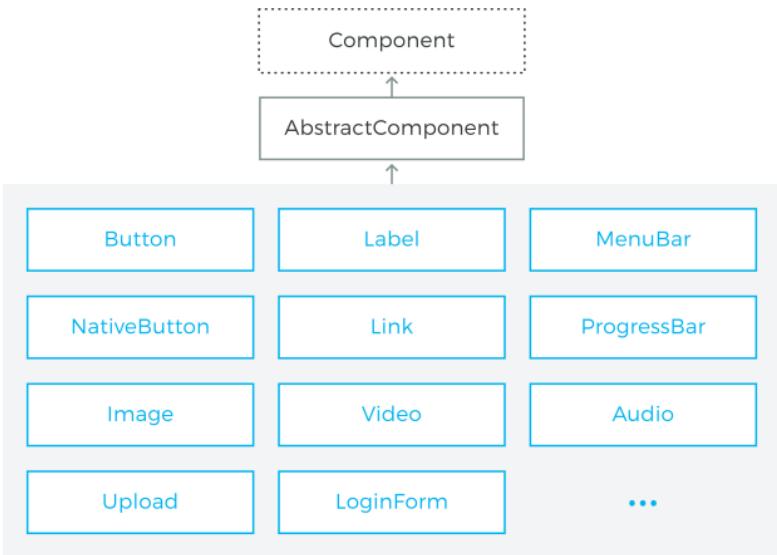


Figure 6.1. Basic Components

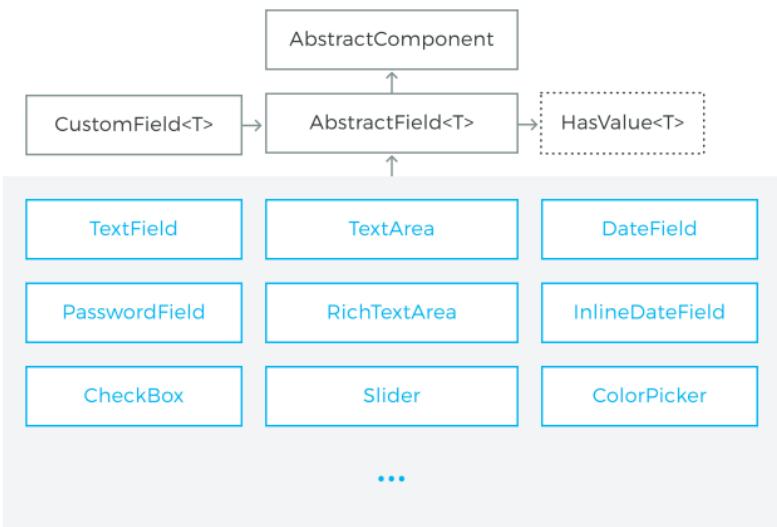


Figure 6.2. Field Components

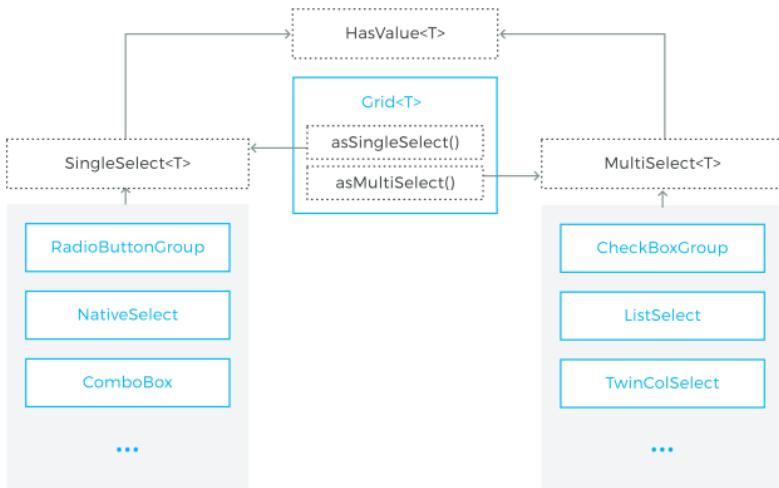


Figure 6.3. Selection Components

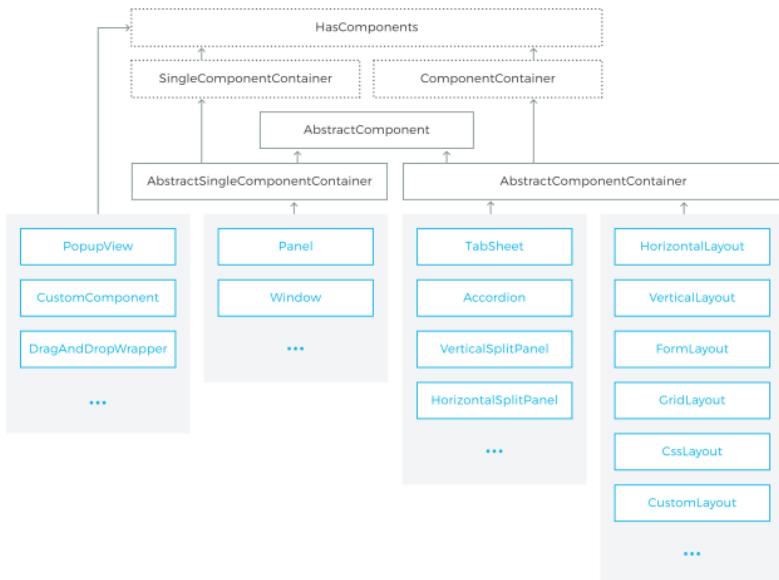


Figure 6.4. Layouts and Component Containers

The **Component** is interface implemented by all components. In practice, all components extend the **AbstractComponent** abstract class.

There are three more specific types of components.

Field Components

allow user to edit a value in the UI. All extend **Abstract-Field**. Field components are described in detail in Section 6.4, "Field Components".

Selection Component

show a list of data that the user can select from. All extend **AbstractListing**. Selection components are described in detail in Section 6.5, "Selection Components".

Layouts and Component Containers

Components that can contain other components. All layouts and containers implement the HasComponents interface. Layout components are described in detail in Chapter 7, *Managing Layout*.

You can browse the built-in UI components of Vaadin library in the Sampler application of the Vaadin Demo. The Sampler shows a description, JavaDoc documentation, and a code samples for each of the components.

In addition to the built-in components, many components are available as add-ons, either from the Vaadin Directory or from independent sources. Both commercial and free components exist. The installation of add-ons is described in Chapter 17, *Using Vaadin Add-ons*.

6.2. Interfaces and Abstractions

Vaadin user interface components are built on a skeleton of interfaces and abstract classes that define and implement the features common to all components and the basic logic how the component states are serialized between the server and the client.

This section gives details on the basic component interfaces and abstractions. The layout and other component container abstractions are described in Chapter 7, *Managing Layout*. The interfaces that define the Vaadin data model are described in Chapter 10, *Binding Components to Data*.

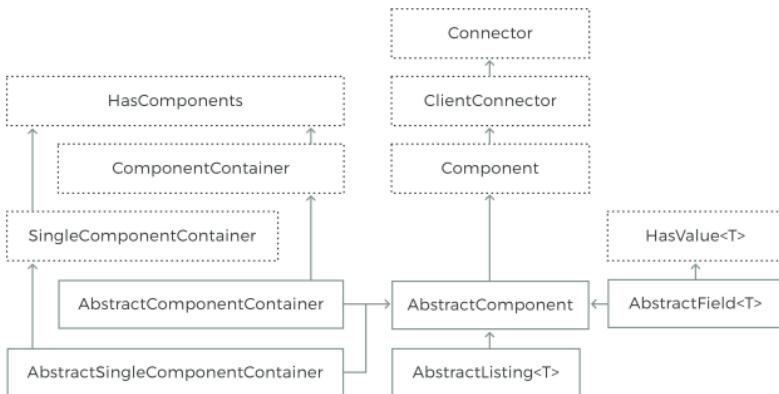


Figure 6.5. Component interfaces and abstractions

All components are connectors that connect to the client-side widgets.

In addition to the interfaces defined within the Vaadin framework, all components implement the **java.io.Serializable** interface to allow serialization. Serialization is needed in many clustering and cloud computing solutions.

6.2.1. Component Interface

The Component interface is paired with the **AbstractComponent** class, which implements all the methods defined in the interface.

Component Tree Management

Components are laid out in the user interface hierarchically. The layout is managed by layout components, or more generally components that implement the **ComponentContainer** interface. Such a container is the parent of the contained components.

The `getParent()` method allows retrieving the parent component of a component. While there is a `setParent()`, you rarely need it as you usually add components with the `addComponent()` method of the **ComponentContainer** interface, which automatically sets the parent.

A component does not know its parent when the component is still being created, so you can not refer to the parent in the constructor with getParent().

Attaching a component to an UI triggers a call to its attach() method. Correspondingly, removing a component from a container triggers calling the detach() method. If the parent of an added component is already connected to the UI, the attach() is called immediately from setParent().

```
public class AttachExample extends CustomComponent {  
    public AttachExample() {  
    }  
  
    @Override  
    public void attach() {  
        super.attach(); // Must call.  
  
        // Now we know who ultimately owns us.  
        ClassResource r = new ClassResource("smiley.jpg");  
        Image image = new Image("Image:", r);  
        setCompositionRoot(image);  
    }  
}
```

The attachment logic is implemented in **AbstractComponent**, as described in Section 6.2.2, “**AbstractComponent**”.

6.2.2. AbstractComponent

AbstractComponent is the base class for all user interface components. It is the (only) implementation of the **Component** interface, implementing all the methods defined in the interface. When creating a new component, you should extend **AbstractComponent** or one of its subclasses.

6.3. Common Component Features

The component base classes and interfaces provide a large number of features. Let us look at some of the most commonly needed features. Features not documented here can be found from the Java API Reference.

The interface defines a number of properties, which you can retrieve or manipulate with the corresponding setters and getters.

6.3.1. Caption

A caption is an explanatory textual label accompanying a user interface component, usually shown above, left of, or inside the component. The contents of a caption are automatically quoted, so no raw HTML can be rendered in a caption.

The caption text can usually be given as the first parameter of a constructor of a component or with setCaption().

```
// New text field with caption "Name"  
TextField name = new TextField("Name");  
layout.addComponent(name);
```

The caption of a component is, by default, managed and displayed by the layout component or component container inside which the component is placed. For example, the **VerticalLayout** component shows the captions left-aligned above the contained components, while the **FormLayout** component shows the captions on the left side of the vertically laid components, with the captions and their associated components left-aligned in their own columns. The **Custom-Component** does not manage the caption of its composition root, so if the root component has a caption, it will not be rendered.

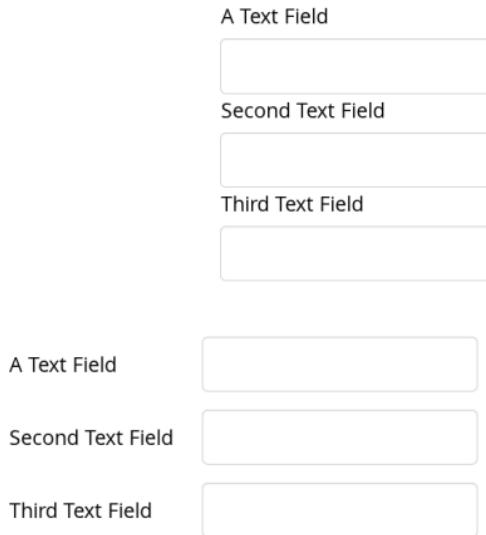


Figure 6.6. Caption Management by VerticalLayout and FormLayout.

Some components, such as **Button** and **Panel**, manage the caption themselves and display it inside the component.

Icon (see Section 6.3.4, “Icon”) is closely related to caption and is usually displayed horizontally before or after it, depending on the component and the containing layout. Also the required indicator in field components is usually shown before or after the caption.

An alternative way to implement a caption is to use another component as the caption, typically a **Label**, a **TextField**, or a **Panel**. A **Label**, for example, allows highlighting a shortcut key with HTML markup or to bind the caption to a data source. The **Panel** provides an easy way to add both a caption and a border around a component.

CSS Style Rules

```
.v-caption {}  
.v-captiontext {}  
.v-required-field-indicator {}
```

A caption is be rendered inside an HTML element that has the v-caption CSS style class. The containing layout may enclose a caption inside other caption-related elements.

Some layouts put the caption text in a v-captiontext element. An optional required indicator in field components is contained in a separate element with v-required-field-indicator style.

6.3.2. Description and Tooltips

All components (that inherit **AbstractComponent**) have a description separate from their caption. The description is usually shown as a tooltip that appears when the mouse pointer hovers over the component for a short time.

You can set the description with setDescription() and retrieve with getDescription().

```
Button button = new Button("A Button");
button.setDescription("This is the tooltip");
```

The tooltip is shown in Figure 6.7. "Component Description as a Tooltip".



Figure 6.7. Component Description as a Tooltip

A description is rendered as a tooltip in most components.

When a component error has been set with setComponentError(), the error is usually also displayed in the tooltip, below the description. Components that are in error state will also display the error indicator. See Section 5.6.1, "Error Indicator and Message".

The description is actually not plain text, but you can use HTML tags to format it. Such a rich text description can contain any HTML elements, including images.

```
button.setDescription(  
    "<h2><img src=\"..\VAADIN/themes/sampler/\"+  
     \"icons/comment_yellow.gif\"/>"+  
    "A richtext tooltip</h2>"+  
    "<ul>"+  
    " <li>Use rich formatting with HTML</li>"+  
    " <li>Include images from themes</li>"+  
    " <li>etc.</li>"+  
    "</ul>");
```

The result is shown in Figure 6.8, "A Rich Text Tooltip".

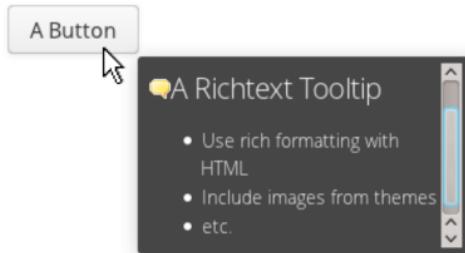


Figure 6.8. A Rich Text Tooltip

6.3.3. Enabled

The *enabled* property controls whether the user can actually use the component. A disabled component is visible, but grayed to indicate the disabled state.

Components are always enabled by default. You can disable a component with `setEnabled(false)`.

```
Button enabled = new Button("Enabled");  
enabled.setEnabled(true); // The default  
layout.addComponent(enabled);
```

```
Button disabled = new Button("Disabled");  
disabled.setEnabled(false);  
layout.addComponent(disabled);
```

Figure 6.9, "An Enabled and Disabled **Button**" shows the enabled and disabled buttons.



Figure 6.9. An Enabled and Disabled Button

A disabled component is automatically put in read-only like state. No client interaction with a disabled component is sent to the server and, as an important security feature, the server-side components do not receive state updates from the client in the disabled state. This feature exists in all built-in components in the Framework meaning all client to server RPC calls are ignored for disabled components.

CSS Style Rules

Disabled components have the `v-disabled` CSS style in addition to the component-specific style. To match a component with both the styles, you have to join the style class names with a dot as done in the example below.

```
.v-textfield.v-disabled {  
    border: dotted;  
}
```

This would make the border of all disabled text fields dotted.

In the Valo theme, the opacity of disabled components is specified with the `$v-disabled-opacity` parameter.

6.3.4. Icon

An icon is an explanatory graphical label accompanying a user interface component, usually shown above, left of, or inside the component. Icon is closely related to caption (see Section 6.3.1, “Caption”) and is usually displayed horizontally before or after it, depending on the component and the containing layout.

The icon of a component can be set with the `setIcon()` method. The image is provided as a resource, perhaps most typically a **ThemeResource**.

```
// Component with an icon from a custom theme  
TextField name = new TextField("Name");
```

```
name.setIcon(new ThemeResource("icons/user.png"));
layout.addComponent(name);

// Component with an icon from another theme ('runo')
Button ok = new Button("OK");
ok.setIcon(new ThemeResource("../runo/icons/16/ok.png"));
layout.addComponent(ok);
```

The icon of a component is, by default, managed and displayed by the layout component or component container in which the component is placed. For example, the **VerticalLayout** component shows the icons left-aligned above the contained components, while the **FormLayout** component shows the icons on the left side of the vertically laid components, with the icons and their associated components left-aligned in their own columns. The **CustomComponent** does not manage the icon of its composition root, so if the root component has an icon, it will not be rendered.



Figure 6.10. Displaying an Icon from a Theme Resource.

Some components, such as **Button** and **Panel**, manage the icon themselves and display it inside the component.

In addition to image resources, you can use *font icons*, which are icons included in special fonts, but which are handled as special resources. See Section 9.8, “Font Icons” for more details.

CSS Style Rules

An icon will be rendered inside an HTML element that has the v-icon CSS style class. The containing layout may enclose an icon and a caption inside elements related to the caption, such as v-caption.

6.3.5. Locale

The locale property defines the country and language used in a component. You can use the locale information in conjunction with an internationalization scheme to acquire local-

ized resources. Some components, such as **TextField**, use the locale for component localization.

You can set the locale of a component (or the application) with `setLocale()` as follows:

```
// Component for which the locale is meaningful
InlineDateField date = new InlineDateField("Datum");

// German language specified with ISO 639-1 language
// code and ISO 3166-1 alpha-2 country code.
date.setLocale(new Locale("de", "DE"));

date.setResolution(Resolution.DAY);
layout.addComponent(date);
```

The resulting date field is shown in Figure 6.11, "Set locale for **InlineDateField**".



Figure 6.11. Set locale for **InlineDateField**

Getting the Locale

You can get the locale of a component with `getLocale()`. If the locale is undefined for a component, that is, not explicitly set, the locale of the parent component is used. If none of the parent components have a locale set, the locale of the UI is

used, and if that is not set, the default system locale is set, as given by `Locale.getDefault()`.

The `getLocale()` returns null if the component is not yet attached to the UI, which is usually the case in most constructors, so it is a bit awkward to use it for internationalization. You can get the locale in `attach()`, as shown in the following example:

```
Button cancel = new Button() {
    @Override
    public void attach() {
        super.attach();
        ResourceBundle bundle = ResourceBundle.getBundle(
            MyAppCaptions.class.getName(), getLocale());
        setCaption(bundle.getString(MyAppCaptions.CancelKey));
    }
};
layout.addComponent(cancel);
```

However, it is normally a better practice to use the locale of the current UI to get the localized resource right when the component is created.

```
// Captions are stored in MyAppCaptions resource bundle
// and the UI object is known in this context.
ResourceBundle bundle =
    ResourceBundle.getBundle(MyAppCaptions.class.getName(),
        UI.getCurrent().getLocale());

// Get a localized resource from the bundle
Button cancel =
    new Button(bundle.getString(MyAppCaptions.CancelKey));
layout.addComponent(cancel);
```

Selecting a Locale

A common task in many applications is selecting a locale. The locale can be set for the **UI** or single **Component**. By default each component uses the locale from the **UI** it has been attached to. Setting a locale to a **Component** only applies the locale to that component and its children. Note, that updating the locale for a component does not update its children, thus any child component that uses the locale should be updated manually.

6.3.6. Read-Only

The property defines whether the value of a component can be changed. The property is only applicable to components implementing the `HasValue` interface.

```
TextField readwrite = new TextField("Read-Write");
readwrite.setValue("You can change this");
readwrite.setReadOnly(false); // The default
```

```
TextField readonly = new TextField("Read-Only");
readonly.setValue("You can't touch this");
readonly.setReadOnly(true);
```

The resulting read-only text field is shown in Figure 6.12, "A read-only component".

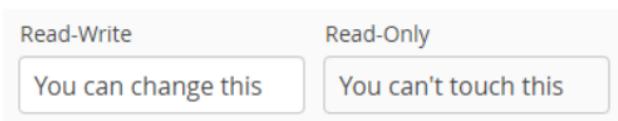


Figure 6.12. A read-only component

Notice that the value of a selection component is the selection, not its items. A read-only selection component doesn't therefore allow its selection to be changed, but other changes are possible. For example, if you have a `Grid` with a read-only selection model in editable mode, its contained fields and the underlying data model can still be edited, and the user could sort it or reorder the columns.

Client-side state modifications will not be communicated to the server-side and, more importantly, server-side field components will not accept changes to the value of a read-only `HasValue` component. The latter is an important security feature, because a malicious user can not fabricate state changes in a read-only field.

Also notice that while the read-only status applies automatically to the value of a field, it does not apply to other component variables. A read-only component can accept some other state changes from the client-side and some of such changes could be acceptable, such as change in the scroll bar position

of a **ListSelect**. Custom components should check the read-only state for variables bound to business data.

CSS Style Rules

Setting a normally editable component to read-only state can change its appearance to disallow editing the value. In addition to CSS styling, also the HTML structure can change. For example, **TextField** loses the edit box and appears much like a **Label**.

A read-only component will have the `v-readonly` style. The following CSS rule would make the text in all read-only **TextField** components appear in italic.

```
.v-textfield.v-readonly {  
    font-style: italic;  
}
```

6.3.7. Style Name

The `style name` property defines one or more custom CSS style class names for the component. The `getStyleNames()` returns the current style names as a space-separated list. The `setStyleName()` replaces all the styles with the given style name or a space-separated list of style names. You can also add and remove individual style names with `addStyleName()` and `removeStyleName()`. A style name must be a valid CSS style name.

```
Label label = new Label("This text has a lot of style");  
label.addStyleName("mystyle");  
layout.addComponent(label);
```

The style name will appear in the component's HTML element in two forms: literally as given and prefixed with the component-specific style name. For example, if you add a style name `mystyle` to a **Button**, the component would get both `mystyle` and `v-button-mystyle` styles. Neither form may conflict with built-in style names of Vaadin. For example, `focus` style would conflict with a built-in style of the same name, and an `content` style for a **Panel** component would conflict with the built-in `v-panel-content` style.

The following CSS rule would apply the style to any component that has the `mystyle` style.

```
.mystyle {  
    font-family: fantasy;  
    font-style: italic;  
    font-size: 25px;  
    font-weight: bolder;  
    line-height: 30px;  
}
```

The resulting styled component is shown in Figure 6.13.
"Component with a custom style"

This text has a lot of style

Figure 6.13. Component with a custom style

6.3.8. Visible

Components can be hidden by setting the *visible* property to *false*. Also the caption, icon and any other component features are made hidden. Hidden components are not just invisible, but their content is not communicated to the browser at all. That is, they are not made invisible cosmetically with only CSS rules. This feature is important for security if you have components that contain security-critical information that must only be shown in specific application states.

```
TextField invisible = new TextField("No-see-um");  
invisible.setValue("You can't see this!");  
invisible.setVisible(false);  
layout.addComponent(invisible);
```

If you need to make a component only cosmetically invisible, you should use a custom theme to set it display: none style. This is mainly useful for some special components that have effects even when made invisible in CSS. If the hidden component has undefined size and is enclosed in a layout that also has undefined size, the containing layout will collapse when the component disappears. If you want to have the component keep its size, you have to make it invisible by setting all its font and other attributes to be transparent. In such cases, the invisible content of the component can be made visible easily in the browser.

6.3.9. Sizing Components

Vaadin components are sizeable; not in the sense that they were fairly large or that the number of the components and their features are sizeable, but in the sense that you can make them fairly large on the screen if you like, or small or whatever size.

The **Sizeable** interface, shared by all components, provides a number of manipulation methods and constants for setting the height and width of a component in absolute or relative units, or for leaving the size undefined.

The size of a component can be set with `setWidth()` and `setHeight()` methods. The methods take the size as a floating-point value. You need to give the unit of the measure as the second parameter for the above methods.

```
mycomponent.setWidth(100, Unit.PERCENTAGE);
mycomponent.setWidth(400, Unit.PIXELS);
```

Alternatively, you can specify the size as a string. The format of such a string must follow the HTML/CSS standards for specifying measures.

```
mycomponent.setWidth("100%");
mycomponent.setHeight("400px");
```

The "100%" percentage value makes the component take all available size in the particular direction. You can also use the shorthand method `setSizeFull()` to set the size to 100% in both directions.

The size can be *undefined* in either or both dimensions, which means that the component will take the minimum necessary space. Most components have *undefined* size by default, but some layouts have full size in horizontal direction. You can set the height, width, or both as *undefined* with the methods `setWidthUndefined()`, `setHeightUndefined()`, and `setSizeUndefined()`, respectively.

Always keep in mind that *a layout with undefined size may not contain components with defined relative size*, such as

"full size", except in some special cases. See Section 7.13.1, "Layout Size" for details.

If a component inside **HorizontalLayout** or **VerticalLayout** has full size in the namesake direction of the layout, the component will expand to take all available space not needed by the other components. See Section 7.13.1, "Layout Size" for details.

6.3.10. Managing Input Focus

When the user clicks on a component, the component gets the *input focus*, which is indicated by highlighting according to style definitions. If the component allows inputting text, the focus and insertion point are indicated by a cursor. Pressing the Tab key moves the focus to the component next in the *focus order*.

Focusing is supported by all **AbstractField** and **AbstractListing** components and also by components such as **Button**, **Upload**, and **TabSheet**.

The focus order or *tab index* of a component is defined as a positive integer value, which you can set with `setTabIndex()` and get with `getTabIndex()`. The tab index is managed in the context of the page in which the components are contained. The focus order can therefore jump between two any lower-level component containers, such as sub-windows or panels.

The default focus order is determined by the natural hierarchical order of components in the order in which they were added under their parents. The default tab index is 0 (zero).

Giving a negative integer as the tab index removes the component from the focus order entirely.

CSS Style Rules

The component having the focus will have an additional style class with the `-focus` suffix. For example, a **TextField**, which normally has the `v-textfield` style, would additionally have the `v-textfield-focus` style.

For example, the following would make a text field blue when it has focus.

```
.v-textfield-focus {  
    background: lightblue;  
}
```

6.4. Field Components

Fields are components that have a value that the user can change through the user interface. Figure 6.14, "Field Components" illustrates the inheritance relationships and the important interfaces and base classes.

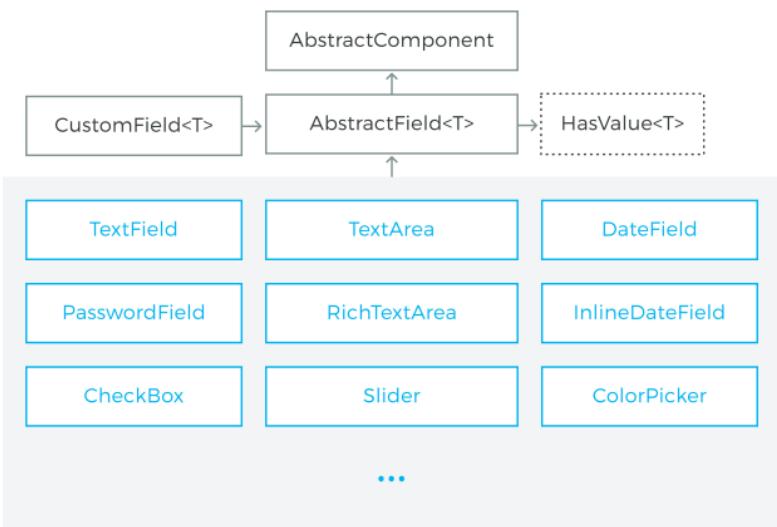


Figure 6.14. Field Components

Field components are built upon the framework defined in the **HasValue** interface. **AbstractField** is the base class for all field components, except those components that allow the user to select a value. (see Section 6.5, "Selection Components").

In addition to the component features inherited from **AbstractComponent**, it implements the features defined in the **HasValue** and **Component.Focusable** interfaces.

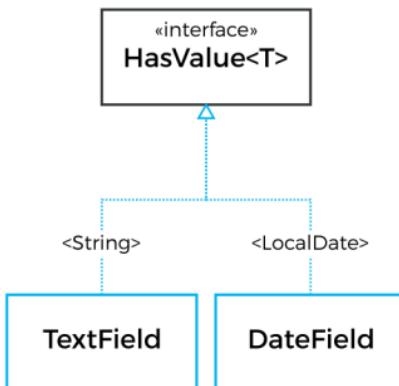


Figure 6.15. Field components having values

The description of the `HasValue` interface and field components extending [classname]`#AbstractField` is broken down in the following sections.

6.4.1. The `HasValue` Interface

The `HasValue` interface marks a component that has a user editable value. The type parameter in the interface is the type of the value that the component is editing.

You can set the value with the `setValue()` and read it with the `getValue()` method defined in the **HasValue** interface.

The **HasValue** interface defines a number of properties, which you can access with the corresponding setters and getters.

`readOnly`

Set the component to be read-only, meaning that the value is not editable.

`requiredIndicatorVisible`

When enabled, a required indicator (the asterisk * character) is displayed on the left, above, or right the field, depending on the containing layout and whether the field has a caption. When the component is used in a form (see the section called "Validation"), it can be set to be required, which will automatically show the required indicator, and validate that the value is not

empty. Without validation, the required indicator is merely a visual guide.

emptyValue

The initial empty value of the component.

clear

Clears the value to the empty value.

6.4.2. Handling Value Changes

HasValue provides addValueChangeListener method for listening to changes to the field value. This method returns a **Registration** object that can be used to later remove the added listener if necessary.

```
TextField textField = new TextField();
Label echo = new Label();

textField.addValueChangeListener(event -> {
    String origin = event.isUserOriginated()
        ? "user"
        : "application";
    String message = origin
        + " entered the following: "
        + event.getValue();
    Notification.show(message);
});
```

6.4.3. Binding Fields to Data

Fields can be grouped into *forms* and coupled with business data objects with the **Binder** class. When a field is bound to a property using **Binder**, it gets its default value from the property, and is stored to the property either manually via the Binder.save method, or automatically every time the value changes.

```
class Person {
    private String name;
    public String getName() { /* */ }
    public void setName(String) { /* */ }
}
```

```
TextField nameField = new TextField();
```

```
Binder<Person> binder = new Binder<>();

// Bind nameField to the Person.name property
// by specifying its getter and setter
binder.bind(nameField, Person::getName, Person::setName);

// Bind an actual concrete Person instance.
// After this, whenever the user changes the value
// of nameField, p.setName is automatically called.
Person p = new Person();
binder.bind(p);
```

For more information on data binding, see Section 10.3, “Binding Data to Forms”

6.4.4. Validating Field Values

User input may be syntactically or semantically invalid. **Binder** allows adding a chain of one or more *validators* for automatically checking the validity of the input before storing it to the data object. You can add validators to fields by calling the `withValidator` method on the Binding object returned by `Binder.forField`. There are several built-in validators in the Framework, such as the **StringLengthValidator** used below.

```
binder.forField(nameField)
    .withValidator(new StringLengthValidator(
        "Name must be between 2 and 20 characters long",
        2, 20))
    .bind(Person::getName, Person::setName);
```

Failed validation is by default indicated with the error indicator of the field, described in Section 5.6.1, “Error Indicator and Message”. Hovering mouse on the field displays the error message returned by the validator. If any value in a set of bound fields fails validation, none of the field values are saved into the bound property until the validation passes.

Implementing Custom Validators

Validators implement the Validator interface that simply extends `java.util.function.Function`, returning a special type called Result. This return type represents the validation outcome: whether or not the given input was valid.

```
class MyValidator implements Validator<String> {
    @Override
    public Result<String> apply(String value, ValueContext context) {
```

```

    if(input.length() == 6) {
        return Result.ok(input);
    } else {
        return Result.error(
            "Must be exactly six characters long");
    }
}

```

Because `Result.ok` takes the valid value as an argument, a validator can also do some sanitization on valid inputs, such as removing leading and trailing whitespace from a string. Since `Validator` is a functional interface, you can often simply write a lambda expression instead of a full class declaration. There is also an `withValidator` overload that creates a validator from a boolean function and an error message.

```

binder.forField(nameField)
    .withValidator(name -> name.length() < 20,
        "Name must be less than 20 characters long")
    .bind(Person::getName, Person::setName);

```

6.4.5. Converting Field Values

Field values are always of some particular type. For example, **TextField** allows editing **String** values. When bound to a data source, the type of the source property can be something different, say an **Integer**. *Converters* are used for converting the values between the presentation and the model. Their usage is described in the section called “Conversion”.

6.5. Selection Components

For a better overview on how selection works, see [link:Section 10.5, “Selecting Items”](#).

Vaadin offers many alternative ways for selecting one or more items. The core library includes the following selection components, all based on either **AbstractSingleSelect** or **AbstractMultiSelect** class:

ComboBox (Section 6.16, “**ComboBox**”)

A drop-down list with a text box, where the user can type text to find matching items and select one item. The component also provides a placeholder and the user can enter new items.

ListSelect (Section 6.17, “**ListSelect**”)

A vertical list box for selecting items in multiple selection mode.

NativeSelect (Section 6.18, “**NativeSelect**”)

Provides selection using the native selection component of the browser, typically a drop-down list for single selection and a multi-line list in multiselect mode. This uses the `<select>` element in HTML.

RadioButtonGroup (Section 6.19, “**CheckBoxGroup** and **RadioGroup**”)

Shows the items as a vertically arranged group of radio buttons in single selection mode.

CheckBoxGroup (Section 6.19, “**CheckBoxGroup** and **RadioGroup**”)

Shows the items as a vertically arranged group of check boxes in multiple selection mode.

TwinColSelect (Section 6.20, “**TwinColSelect**”)

Shows two list boxes side by side where the user can select items from a list of available items and move them to a list of selected items using control buttons.

In addition, the **Grid** component allows user selection.

6.5.1. Binding Selection Components to Data

The selection components are typically bound to list of items obtained from backend system. You can give the list of items in the constructor or set it with `setItems()`. Read more in Chapter 10, *Binding Components to Data*.

You can get the current selection as the value of the selection component using `getValue` defined in `HasValue` interface. Selection changes are handled as value change or selection events, as is described later.

6.5.2. Item Captions

The items are typically a strings, in which case they can be used as the caption, but can be any object type. We could as well have given `Planet` instances for the items and use cap-

tions generated based on them `setItemCaptionGenerator()` method.

// List of Planet objects

```
List<Planet> planets = new ArrayList<>();
planets.add(new Planet("Mercury"));
planets.add(new Planet("Venus"));
planets.add(new Planet("Earth"));
```

// Create a selection component

```
ComboBox<Planet> select = new ComboBox<>("My Select");
```

// Add an items to the ComboBox

```
select.setItems(planets);
```

```
select.setItemCaptionGenerator(Planet::getName);
```

// Select the first

```
select.setSelectedItem(planets.get(0));
```

// or

```
// select.setValue(planets.get(0));
```

In addition to a caption, an item can have an icon. The icon is set with `setItemIconGenerator()`.

Typical use cases for captions are:

Using the items as the caption: the caption is retrieved with `toString()` method from the item. This is useful for simple objects like String or Integers, but also for objects that have human readable output for `toString()` .

Using a field of a item as caption: the caption is retrieved using the `ItemCaptionGenerator` typically given as a lambda or a method reference.

6.5.3. Item Icons

You can set an icon for each item with `setItemIconGenerator()`, in a fashion similar to captions. Notice, however, that icons are not supported in **NativeSelect**, **TwinColSelect**, and some other selection components and modes. This is because HTML does not support images inside the native select elements.

6.5.4. Getting and Setting Selection

For a better overview on how selection works, see link:Section 10.5, "Selecting Items".

You can get selected the item with `getValue()` of the `HasValue` interface that returns either a single selected item (case of `SingleSelect`) or a collection of selected items (case of `MultiSelect`). You can select an item with the corresponding `setValue()` method.

The **ComboBox** and **NativeSelect** will show empty selection when no actual item is selected.

6.5.5. Handling Selection Events

You can access the currently selected item with the `getValue()` (`SingleSelect`) or `getSelectedItems()` (`MultiSelect`) method of the component. Also, when handling selection events with a **SelectionListener**, the **SelectionEvent** will have the selected items of the event. Single- and Multiselect components have their own more specific listener and event types, `SingleSelectionListener` for **SingleSelectionEvent** and `MultiSelectionListener` for **MultiSelectionEvent** respectively. Both can be added with the `addSelectionListener` method.

```
// Create a selection component with some items
ComboBox<String> select = new ComboBox<>("My Select");
select.setItems("Io", "Europa", "Ganymedes", "Callisto");

// Handle selection event
select.addSelectionListener(event ->
    layout.addComponent(new Label("Selected " +
        event.getSelectedItem().orElse("none")));
)
```

The result of user interaction is shown in Figure 6.16, "Selected Item".

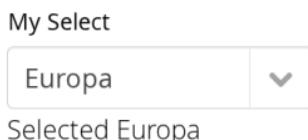


Figure 6.16. Selected Item

6.5.6. Multiple Selection

For a better overview on how selection works, see link:Section 10.5, "Selecting Items".

Some selection components, such as **CheckBoxGroup**, **ListSelect** and **TwinColSelect** are multiselect components, they extend **AbstractMultiSelect** class.

Multiselect components use the MultiSelect interface which extends HasValue. This provides more fine grained API for selection. You can get and set the selection with the `getSelectedItems()` and `select()` methods.

A change in the selection will trigger a **SelectionEvent**, which you can handle with a **SelectionListener**. The following example shows how to handle selection changes with a listener.

```
// A selection component with some items
ListSelect<String> select = new ListSelect<>("My Selection");
select.setItems("Mercury", "Venus", "Earth",
    "Mars", "Jupiter", "Saturn", "Uranus", "Neptune");

// Feedback on value changes
select.addSelectionListener(event -> {
    // Some feedback
    layout.addComponent(new Label("Selected: " +
        event.getNewSelection()));
});

});
```

6.6. Component Extensions

Components and UIs can have extensions which are attached to the component dynamically. Especially, many add-ons are extensions.

How a component is extended depends on the extension. Typically, they have an `extend()` method that takes the component to be extended as the parameter.

```
TextField tf = new TextField("Hello");
layout.addComponent(tf);
```

```
// Add a simple extension
new CapsLockWarning().extend(tf);
```

```
// Add an extension that requires some parameters
CSVValidator validator = new CSVValidator();
validator.setRegExp("[0-9]*");
validator.setError Message("Must be a number");
validator.extend(tf);
```

Development of custom extensions is described in Section 16.7, "Component and UI Extensions".

6.7. Label

Label component displays non-editable text. This text can be used for short simple labels or for displaying long text, such as paragraphs. The text can be formatted in HTML or as pre-formatted text, depending on the *content mode* of the label.

You can give the label text most conveniently in the constructor, as is done in the following. Label has undefined default width, so it will only take the space it needs.

```
// A container that is 100% wide by default
VerticalLayout layout = new VerticalLayout();

// label will only take the space it needs
Label label = new Label("Labeling can be dangerous");
layout.addComponent(label);
```

You can get and set the text of a **Label** with the `getValue()` and `setValue()` methods.

```
// Get the label's text to initialize a field
TextField editor = new TextField(null, // No caption
                                label.getValue());

// Change the label's text
editor.addValueChangeListener(event -> label.setValue(event.getValue()));
```

Even though **Label** is text and is often used as a caption, it is a normal component and therefore also has a caption that you can set with `setCaption()`. As with most other components, the caption is managed by the containing layout.

6.7.1. Text Width and Wrapping

Label has undefined default width, so it will only take the space it needs. If the width of the label's text exceeds the

available width for the label in the layout, the text overflows, and normally, gets truncated.

```
// A container with a defined width.  
Panel panel = new Panel("Panel Containing a Label");  
panel.setWidth("300px");  
  
panel.setContent(  
    new Label("This is a Label inside a Panel. There is " +  
        "enough text in the label to make the text " +  
        "get truncated when it exceeds the width of the panel."));
```

As the size of the **Panel** in the above example is fixed and the width of **Label** is the default undefined, the **Label** will overflow the layout horizontally and be truncated.

Setting **Label** to defined width will cause it to not overflow the layout, but to wrap to the next line.

6.7.2. Content Mode

The content of a label is formatted depending on a *content mode*. By default, the text is assumed to be plain text and any contained HTML-specific characters will be quoted appropriately to allow rendering the contents of a label in HTML in a web browser. The content mode can be set in the constructor or with `setContentMode()`, and can have the values defined in the **ContentMode** enumeration type in `com.vaadin.shared.ui.label` package:

TEXT

The default content mode where the label contains only plain text. All characters are allowed, including the special <, >, and & characters in HTML, which are quoted properly in HTML while rendering the component. This is the default mode.

PREFORMATTED

Content mode where the label contains preformatted text. It will be, by default, rendered with a fixed-width typewriter font. Preformatted text can contain line breaks, written in Java with the \n escape sequence for a newline character (ASCII 0x0a), or tabulator characters written with \t (ASCII 0x09).

HTML

Content mode where the label contains HTML.

Please note the following security and validity warnings regarding the HTML content mode.



Cross-Site Scripting Warning

Having **Label** in HTML content mode allows pure HTML content. If the content comes from user input, you should always carefully sanitize it to prevent cross-site scripting (XSS) attacks. Please see Section 11.7.1, "Sanitizing User Input to Prevent Cross-Site Scripting".

Also, the validity of the HTML content is not checked when rendering the component and any errors can result in an error in the browser. If the content comes from an uncertain source, you should always validate it before displaying it in the component.

The following example demonstrates the use of **Label** in different modes.

```
Label textLabel = new Label(  
    "Text where formatting characters, such as \\n, " +  
    "and HTML, such as <b>here</b>, are quoted.",  
    ContentMode.TEXT);
```

```
Label preLabel = new Label(  
    "Preformatted text is shown in an HTML <pre> tag.\n" +  
    "Formatting such as\\n" +  
    " * newlines\\n" +  
    " * whitespace\\n" +  
    "and such are preserved. HTML tags. \\n"+  
    "such as <b>bold</b>, are quoted.",  
    ContentMode.PREFORMATTED);
```

```
Label htmlLabel = new Label(  
    "In HTML mode, all HTML formatting tags, such as \\n" +  
    "<ul>" +  
    " * <li><b>bold</b></li>" +  
    " * <li>itemized lists</li>" +  
    " * <li>etc.</li>" +  
    "</ul>" +  
    "are preserved.",  
    ContentMode.HTML);
```

The rendering will look as shown in Figure 6.17, "Label Content Modes".

TEXT	Text where formatting characters, such as \n, and HTML, such as here, are quoted.
PREFORMATTED	<p>Preformatted text is shown in an HTML <pre> tag.</p> <p>Formatting such as</p> <ul style="list-style-type: none">* newlines* whitespace <p>and such are preserved. HTML tags, such as bold, are quoted.</p>
HTML	<p>In HTML mode, all HTML formatting tags, such as</p> <ul style="list-style-type: none">• bold• itemized lists• etc. <p>are preserved.</p>

Figure 6.17. Label Content Modes

6.7.3. CSS Style Rules

```
.v-label {}  
pre {} /* In PREFORMATTED content mode */
```

The **Label** component has a v-label overall style. In the *PREFORMATTED* content mode, the text is wrapped inside a `<pre>` element.

6.8. Link

The **Link** component allows making hyperlinks. References to locations are represented as resource objects, explained in Section 5.5, "Images and Other Resources". The **Link** is a regular HTML hyperlink, that is, an `<a href>` anchor element that is handled natively by the browser. Unlike when clicking a **Button**, clicking a **Link** does not cause an event on the server-side.

Links to an arbitrary URL can be made by using an **ExternalResource** as follows:

// Textual link

```
Link link = new Link("Click Me!",  
    new ExternalResource("http://vaadin.com/"));
```

You can use setIcon() to make image links as follows:

// Image link

```
Link iconic = new Link(null,  
    new ExternalResource("http://vaadin.com/"));  
iconic.setIcon(new ThemeResource("img/nicubunu_Chain.png"));
```

// Image + caption

```
Link combo = new Link("To appease both literal and visual",  
    new ExternalResource("http://vaadin.com/"));  
combo.setIcon(new ThemeResource("img/nicubunu_Chain.png"));
```

The resulting links are shown in Figure 6.18, "Link Example". You could add a "display: block" style for the icon element to place the caption below it.



Figure 6.18. Link Example

6.8.1. Opening a New Window

With the simple constructor used in the above example, the resource is opened in the current window. Using the constructor that takes the target window as a parameter, or by setting the target window with setTargetName(), you can open the resource in another window, such as a popup browser window/tab. As the target name is an HTML target string managed by the browser, the target can be any window, including windows not managed by the application itself. You can use the special underscored target names, such as _blank to open the link to a new browser window or tab.

// Hyperlink to a given URL

```
Link link = new Link("Take me a away to a faraway land",
```

```
new ExternalResource("http://vaadin.com/"));

// Open the URL in a new window/tab
link.setTargetName("_blank");

// Indicate visually that it opens in a new window/tab
link.setIcon(new ThemeResource("icons/external-link.png"));
link.addStyleName("icon-after-caption");
```

6.8.2. Opening as a Pop-Up Window

With the `_blank` target, a normal new browser window is opened. If you wish to open it in a popup window (or tab), you need to give a size for the window with `setTargetWidth()` and `setTargetHeight()`. You can control the window border style with `setTargetBorder()`, which takes any of the defined border styles `BorderStyle.DEFAULT`, `BorderStyle.MINIMAL`, and `BorderStyle.NONE`. The exact result depends on the browser.

```
// Open the URL in a popup
link.setTargetName("_blank");
link.setTargetBorder(BorderStyle.NONE);
link.setTargetHeight(300);
link.setTargetWidth(400);
```

6.8.3. Alternatives

In addition to the **Link** component, Vaadin allows alternative ways to make hyperlinks. Also, you can make hyperlinks (or any other HTML) in a **Label** in HTML content mode.

The **Button** component has a `ValoTheme.BUTTON_LINK` style name that makes it look like a hyperlink, while handling clicks in a server-side click listener instead of in the browser. However, browsers do not generally allow opening new windows after server side round trip, so for such tasks you need to use the **BrowserWindowOpener** extension described in Section 11.1.1, "Opening Pop-up Windows"

6.8.4. CSS Style Rules

```
.v-link {}
a {}
.v-icon {}
span {}
```

The overall style for the **Link** component is v-link. The root element contains the <a href> hyperlink anchor. Inside the anchor are the icon, with v-icon style, and the caption in a text span.

Hyperlink anchors have a number of *pseudo-classes* that are active at different times. An unvisited link has a:link class and a visited link a:visited. When the mouse pointer hovers over the link, it will have a:hover, and when the mouse button is being pressed over the link, the a:active class. When combining the pseudo-classes in a selector, please notice that a:hover must come after an a:link and a:visited, and a:active after the a:hover.

Icon Position

Normally, the link icon is before the caption. You can have it right of the caption by reversing the text direction in the containing element.

```
/* Position icon right of the link caption. */
.icon-after-caption {
    direction: rtl;
}

/* Add some padding around the icon. */
.icon-after-caption .v-icon {
    padding: 0 3px;
}
```

The resulting link is shown in Figure 6.19, "Link That Opens a New Window".

[Take me a away to a faraway land](#) ↗

Figure 6.19. Link That Opens a New Window

6.9. TextField

TextField is one of the most commonly used user interface components. It is a field that allows entering textual values with keyboard.

The following example creates a simple text field:

```
// Create a text field  
TextField tf = new TextField("A Field");  
  
// Put some initial content in it  
tf.setValue("Stuff in the field");
```

The result is shown in Figure 6.20, “**TextField** Example”.

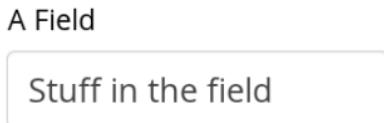


Figure 6.20. TextField Example

Value changes are handled by adding a listener using `addValueChangeListener()` method, as in most other fields. The value can be acquired with `getValue()` directly from the text field or from the parameter passed to the event listener.

```
// Handle changes in the value  
tf.addValueChangeListener(event ->  
    // Do something with the value  
    Notification.show("Value is: " + event.getValue()));
```

TextField edits **String** values, but you can use **Binder** to bind it to any property type that has a proper converter, as described in the section called “Conversion”.

Much of the API of **TextField** is defined in **AbstractTextField**, which allows different kinds of text input fields, which do not share all the features of the single-line text fields.

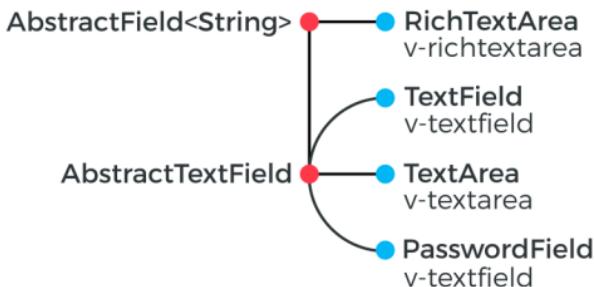


Figure 6.21. Text field class relationships

6.9.1. String Length

The `setMaxLength()` method sets the maximum length of the input string so that the browser prevents the user from entering a longer one. As a security feature, the input value is automatically truncated on the server-side, as the maximum length setting could be bypassed on the client-side. The maximum length property is defined at **AbstractTextField** level.

Notice that the maximum length setting does not affect the width of the field. You can set the width with `setWidth()`, as with other components. Using `em` widths is recommended to better approximate the proper width in relation to the size of the used font, but the `em` width is not exactly the width of a letter and varies by browser and operating system. There is no standard way in HTML for setting the width exactly to a number of letters (in a monospaced font).

6.9.2. Configuring the Granularity of Value Change Events

Often you want to control how frequently **TextField** value changes are transmitted to the server. Sometimes the changes should be sent only after the field loses focus. In the other extreme, it can sometimes be useful to receive events every time the user presses a key.

The *value change event mode* defines how quickly the changes are transmitted to the server and cause a server-side event. Lazier change events allow sending larger changes in one event if the user is typing fast, thereby reducing server requests.

You can set the text change event mode of a **TextField** with `setValueChangeMode()`. The allowed modes are defined in **ValueChangeMode** enum and are as follows:

`ValueChangeMode.LAZY(default)`

An event is triggered when there is a pause in editing the text. The length of the pause can be modified with `setValueChangeTimeout()`.

This is the default mode.

ValueChangeMode.TIMEOUT

A text change in the user interface causes the event to be communicated to the application after a timeout period. If more changes are made during this period, the event sent to the server-side includes the changes made up to the last change. The length of the timeout can be set with `setValueChangeTimeout()`.

ValueChangeMode.EAGER

The **ValueChangeEvent** is triggered immediately for every change in the text content, typically caused by a key press. The requests are separate and are processed sequentially one after another. Change events are nevertheless communicated asynchronously to the server, so further input can be typed while event requests are being processed.

ValueChangeMode.BLUR

The **ValueChangeEvent** is fired, after the field has lost focus.

```
// Text field with maximum length
TextField tf = new TextField("My Eventful Field");
tf.setValue("Initial content");
tf.setMaxLength(20);

// Counter for input length
Label counter = new Label();
counter.setValue(tf.getValue().length() +
    " of " + tf.getMaxLength());

// Display the current length interactively in the counter
tf.addValueChangeListener(event -> {
    int len = event.getValue().length();
    counter.setValue(len + " of " + tf.getMaxLength());
});

tf.setValueChangeMode(ValueChangeMode.EAGER);
```

The result is shown in Figure 6.22, "Text Change Events".

Figure 6.22. Text Change Events

6.9.3. CSS Style Rules

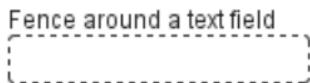
```
.v-textfield { }
```

The HTML structure of **TextField** is extremely simple, consisting only of an element with the v-textfield style.

For example, the following custom style uses dashed border:

```
.v-textfield-dashing {  
    border: thin dashed;  
}
```

The result is shown in Figure 6.23, "Styling TextField with CSS".

**Figure 6.23. Styling TextField with CSS**

The style name for **TextField** is also used in several components that contain a text input field, even if the text input is not an actual **TextField**. This ensures that the style of different text input boxes is similar.

6.10. TextArea

TextArea is a multi-line version of the **TextField** component described in Section 6.9, "**TextField**".

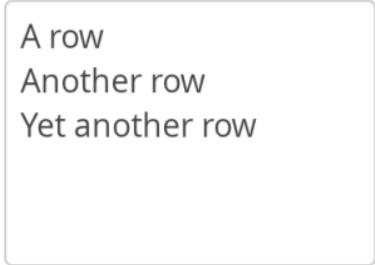
The following example creates a simple text area:

```
// Create the area  
TextArea area = new TextArea("Big Area");  
  
// Put some content in it  
area.setValue("A row\n"+
```

```
"Another row\n"+  
"Yet another row");
```

The result is shown in Figure 6.24, "TextArea Example".

Big Area



```
A row  
Another row  
Yet another row
```

Figure 6.24. TextArea Example

You can set the number of visible rows with `setRows()` or use the regular `setHeight()` to define the height in other units. If the actual number of rows exceeds the number, a vertical scrollbar will appear. Setting the height with `setRows()` leaves space for a horizontal scrollbar, so the actual number of visible rows may be one higher if the scrollbar is not visible.

You can set the width with the regular `setWidth()` method. Setting the size with the `em` unit, which is relative to the used font size, is recommended.

6.10.1. Word Wrap

The `setWordwrap()` sets whether long lines are wrapped (`true` - default) when the line length reaches the width of the writing area. If the word wrap is disabled (`false`), a vertical scrollbar will appear instead. The word wrap is only a visual feature and wrapping a long line does not insert line break characters in the field value: shortening a wrapped line will undo the wrapping.

```
TextArea area1 = new TextArea("Wrapping");  
area1.setWordWrap(true); // The default  
area1.setValue("A quick brown fox jumps over the lazy dog");
```

```
TextArea area2 = new TextArea("Nonwrapping");  
area2.setWordWrap(false);
```

```
area2.setValue("Victor jagt zw&ouml;lfe Boxk&auml;mpfer quer "+  
"&uuml;ber den Sylter Deich");
```

The result is shown in Figure 6.25, "Word Wrap in **TextArea**".

Wrapping

```
A quick brown fox  
jumps over the lazy  
dog
```

Nonwrapping

```
Victor jagt zwölf Boxkä
```

Figure 6.25. Word Wrap in **TextArea**

6.10.2. CSS Style Rules

```
.v-textarea { }
```

The HTML structure of **TextArea** is extremely simple, consisting only of an element with v-textarea style.

6.11. PasswordField

The **PasswordField** is a variant of **TextField** that hides the typed input from visual inspection.

```
PasswordField tf = new PasswordField("Keep it secret");
```

The result is shown in Figure 6.26, "**PasswordField**".

Keep it secret

```
.....|
```

Figure 6.26. **PasswordField**

You should note that the **PasswordField** hides the input only from "over the shoulder" visual observation. Unless the server connection is encrypted with a secure connection, such as HTTPS, the input is transmitted in clear text and may be inter-

cepted by anyone with low-level access to the network. Also phishing attacks that intercept the input in the browser may be possible by exploiting JavaScript execution security holes in the browser.

6.11.1. CSS Style Rules

```
.v-textfield { }
```

The **PasswordField** does not have its own CSS style name but uses the same v-textfield style as the regular **TextField**. See Section 6.9.3, “CSS Style Rules” for information on styling it.

6.12. RichTextArea

The **RichTextArea** field allows entering or editing formatted text. The toolbar provides all basic editing functionalities. The text content of **RichTextArea** is represented in HTML format. **RichTextArea** inherits **TextField** and does not add any API functionality over it. You can add new functionality by extending the client-side components **VRichTextArea** and **VRichTextToolbar**.

As with **TextField**, the textual content of the rich text area is the **Property** of the field and can be set with `setValue()` and read with `getValue()`.

```
// Create a rich text area
final RichTextArea rtarea = new RichTextArea();
rtarea.setCaption("My Rich Text Area");

// Set initial content as HTML
rtarea.setValue("<h1>Hello</h1>\n" +
    "<p>This rich text area contains some text.</p>");
```

My Rich Textarea

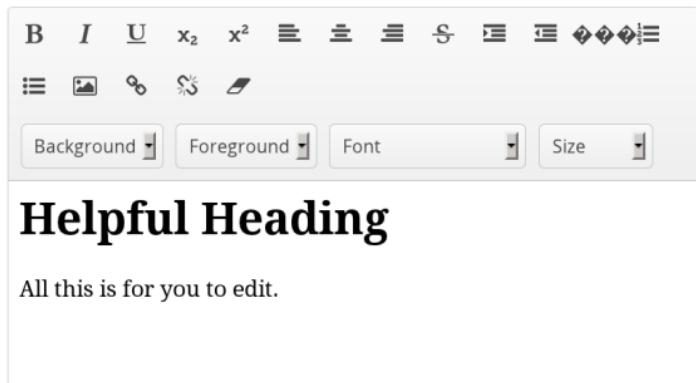


Figure 6.27. Rich Text Area Component

Above, we used context-specific tags such as `<h1>` in the initial HTML content. The rich text area component does not allow creating such tags, only formatting tags, but it does preserve them unless the user edits them away. Any non-visible whitespace such as the new line character (`\n`) are removed from the content. For example, the value set above will be as follows when read from the field with `getValue()`:

```
<h1>Hello</h1> <p>This rich text area contains some text.</p>
```



Cross-Site Scripting Warning

The user input from a **RichTextArea** is transmitted as HTML from the browser to server-side and is not sanitized. As the entire purpose of the **RichTextArea** component is to allow input of formatted text, you can not sanitize it just by removing all HTML tags. Also many attributes, such as `style`, should pass through the sanitization.

See Section 11.7.1, "Sanitizing User Input to Prevent Cross-Site Scripting" for more details on Cross-Site scripting vulnerabilities and sanitization of user input.

6.12.1. Localizing RichTextArea Toolbars

The rich text area is one of the few components in Vaadin that contain textual labels. The selection boxes in the toolbar are in English and currently can not be localized in any other way than by inheriting or reimplementing the client-side **VRichTextToolbar** widget. The buttons can be localized simply with CSS by downloading a copy of the toolbar background image, editing it, and replacing the default toolbar. The toolbar is a single image file from which the individual button icons are picked, so the order of the icons is different from the rendered. The image file depends on the client-side implementation of the toolbar.

6.12.2. CSS Style Rules

```
.v-richtextarea {}  
.v-richtextarea .gwt-RichTextToolbar {}  
.v-richtextarea .gwt-RichTextArea {}
```

The rich text area consists of two main parts: the toolbar with overall style `.gwt-RichTextToolbar` and the editor area with style `.gwt-RichTextArea`. The editor area obviously contains all the elements and their styles that the HTML content contains. The toolbar contains buttons and drop-down list boxes with the following respective style names:

```
.gwt-ToggleButton {}  
.gwt-ListBox {}
```

6.13. Date Input with DateField

The **DateField** component provides the means to display and input dates. The field comes in two variations: **DateField**, with a numeric input box and a popup calendar view, and **InlineDateField**, with the calendar view always visible.

The example below illustrates the use of the **DateField**. We set the initial date of the date field to current date by using the factory method now of the **java.time.LocalDate** class.

```
// Create a DateField with the default style  
DateField date = new DateField();
```

```
// Set the date to present  
date.setValue(LocalDate.now());
```

The result is shown in Figure 6.28, “**DateField** for Selecting a Date”.

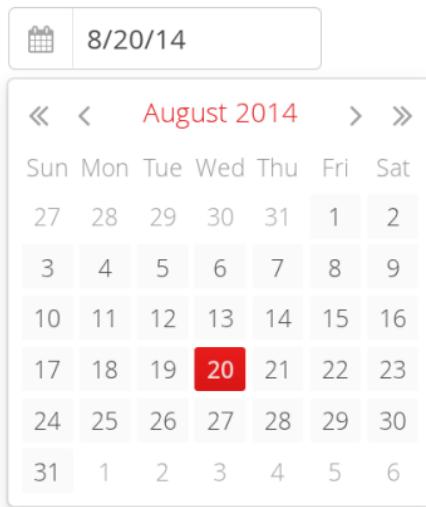


Figure 6.28. DateField for Selecting a Date

6.13.1. DateField

The **DateField** provides date input using a text box. Clicking the handle right of the date opens a popup view for selecting the year, month, and day. The Down key also opens the popup. Once opened, the user can navigate the calendar using the cursor keys.

The date selected from the popup is displayed in the text box according to the default date and format of the current locale, or as specified with `setDateFormat()`. The same format definitions are used for parsing user input.

Date Format

The date is normally displayed according to the default format for the current locale (see Section 6.3.5, “Locale”). You can specify a custom format with `setDateFormat()`. It takes a format

string that follows the format of the `java.time.format.DateTimeFormatter` in Java.

```
// Display only year, month, and day in ISO format  
date.setDateFormat("yyyy-MM-dd");
```

```
// Display the correct format as placeholder  
date.setPlaceholder("yyyy-mm-dd");
```

The result is shown in Figure 6.29, "Custom Date Format for **DateField**".

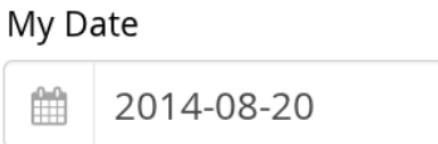


Figure 6.29. Custom Date Format for DateField

The same format specification is also used for parsing user-input date, as described later.

Handling Malformed User Input

A user can easily input a malformed or otherwise invalid date. **DateField** has two validation layers: first on the client-side and then on the server-side.

The validity of the entered date is first validated on the client-side, immediately when the input box loses focus. If the date format is invalid, the `v-datepicker-parseerror` style is set. Whether this causes a visible indication of a problem depends on the theme. The built-in reindeer theme does not show any indication by default, making server-side handling of the problem more convenient.

```
.mydate.v-datepicker-parseerror .v-textfield {  
    background: pink;  
}
```

The `setLenient(true)` setting enables relaxed interpretation of dates, so that invalid dates, such as February 30th or March 0th, are wrapped to the next or previous month, for example.

The server-side validation phase occurs when the date value is sent to the server. If the date field is set in immediate state, it occurs immediately after the field loses focus. Once this is done and if the status is still invalid, an error indicator is displayed beside the component. Hovering the mouse pointer over the indicator shows the error message.

You can handle the errors by overriding the handleUnparsableDateString() method. The method gets the user input as a string parameter and can provide a custom parsing mechanism, as shown in the following example.

```
// Create a date field with a custom parsing and a
// custom error message for invalid format
DateField date = new DateField("My Date") {
    @Override
    protected Result<LocalDate> handleUnparsableDateString(
        String dateString) {
        try {
            // try to parse with alternative format
            return Result.ok(LocalDate.of(dateString, myFormat));
        } catch (DateTimeParseException e) {
            return Result.error("Bad date");
        }
    }
}

// Display only year, month, and day in slash-delimited format
date.setDateFormat("yyyy/MM/dd");

// Don't be too tight about the validity of dates
// on the client-side
date.setLenient(true);
```

CSS Style Rules

```
.v-datepicker, v-datepicker-popupcalendar {}
.v-textfield, v-datepicker-textfield {}
.v-datepicker-button {}
```

The top-level element of **DateField** and **InlineDatePicker** have v-datepicker style. The **DateField** also has the v-datepicker-popupcalendar style.

In addition, the top-level element has a style that indicates the resolution, with v-datepicker- basename and an extension, which is one of full, day, month, or year. The -full style is enabled when the resolution is smaller than a day. These styles are used mainly for controlling the appearance of the popup calendar.

The text box has v-textfield and v-datepicker-textfield styles, and the calendar button v-datepicker-button.

Once opened, the calendar popup has the following styles at the top level:

```
.v-datepicker-popup {}  
.v-popupcontent {}  
.v-datepicker-calendarpanel {}
```

The top-level element of the floating popup calendar has .v-datepicker-popup style. Observe that the popup frame is outside the HTML structure of the component, hence it is not enclosed in the v-datepicker element and does not include any custom styles. The content in the v-datepicker-calendarpanel is the same as in **InlineDateField**, as described in Section 6.13.2, “**InlineDateField**”.

6.13.2. **InlineDateField**

The **InlineDateField** provides a date picker component with a month view. The user can navigate months and years by clicking the appropriate arrows. Unlike with the pop-up variant, the month view is always visible in the inline field.

```
// Create a DateField with the default style  
InlineDateField date = new InlineDateField();
```

```
// Set the date to present  
date.setValue(LocalDate.now());
```

The result is shown in Figure 6.30, “Example of the **InlineDateField**”.



Figure 6.30. Example of the InlineDateField

The user can also navigate the calendar using the cursor keys.

CSS Style Rules

```
.v-datepicker {}  
.v-datepicker-calendarpanel {}  
.v-datepicker-calendarpanel-header {}  
.v-datepicker-calendarpanel-prevyear {}  
.v-datepicker-calendarpanel-prevmonth {}  
.v-datepicker-calendarpanel-month {}  
.v-datepicker-calendarpanel-nextmonth {}  
.v-datepicker-calendarpanel-nextyear {}  
.v-datepicker-calendarpanel-body {}  
.v-datepicker-calendarpanel-weekdays {}  
.v-datepicker-calendarpanel-weeknumbers {}  
.v-first {}  
.v-last {}  
.v-datepicker-calendarpanel-weeknumber {}  
.v-datepicker-calendarpanel-day {}
```

The top-level element has the v-datepicker style. In addition, the top-level element has a style name that indicates the resolution of the calendar, with v-datepicker- basename and an extension, which is one of full, day, month, or year. The -full style is enabled when the resolution is smaller than a day.

The v-datepicker-calendarpanel-weeknumbers and v-datepicker-calendarpanel-weeknumber styles are enabled when the week

numbers are enabled. The former controls the appearance of the weekday header and the latter the actual week numbers.

The other style names should be self-explanatory. For weekdays, the v-first and v-last styles allow making rounded endings for the weekday bar.

6.13.3. Date Resolution

In addition to display a calendar with dates, **DatePicker** can also display just the month or year. The visibility of the input components is controlled by setDateResolution(), which you can set with setResolution(). The method takes as its parameter the highest resolution date element that should be visible. Please see the API Reference for the complete list of resolution parameters.

6.13.4. DatePicker Locale

The date is displayed according to the locale of the user, as reported by the browser. You can set a custom locale with the setLocale() method of **AbstractComponent**, as described in Section 6.3.5, "Locale". Only Gregorian calendar is supported.

6.13.5. Week Numbers

You can enable week numbers in a date field with setShowISOWeekNumbers(). The numbers are shown according to the ISO 8601 standard in a column on the left side of the field.

6.14. Button

The **Button** component is normally used for initiating some action, such as finalizing input in forms. When the user clicks a button, a **Button.ClickEvent** is fired, which can be handled by adding a *click listener* using either the onClick() or the addClickListener() method.



Do not press this button

Figure 6.31. A Button

You can handle button clicks most easily with an anonymous class or a lambda expression, as follows:

```
Button button = new Button("Do not press this button");  
  
button.addClickListener(clickEvent ->  
    Notification.show("Do not press this button again"));
```

The listener can also be given in the constructor, which is often perhaps simpler.

6.14.1. CSS Style Rules

```
.v-button {}  
.v-button-wrap {}  
.v-button-caption {}
```

A button has an overall v-button style. The caption has v-button-caption style. There is also an intermediate wrap element, which may help in styling in some cases.

The button component has many style variants in the Valo theme, as illustrated in Figure 6.32, "Button in different styles of the Valo theme". The styles are defined in the **ValoTheme** class.

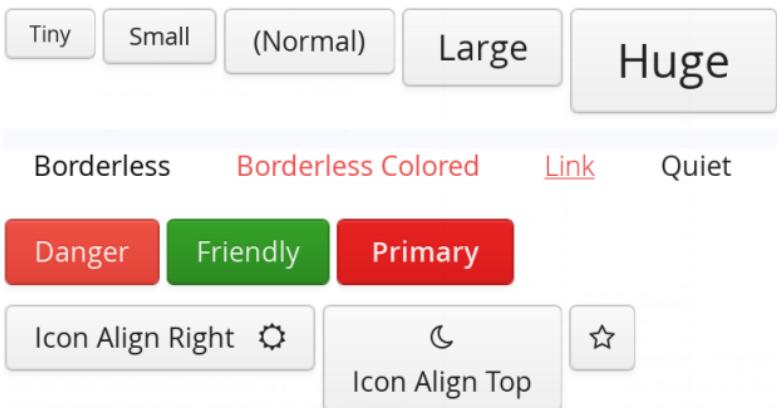


Figure 6.32. Button in different styles of the Valo theme

6.15. CheckBox

CheckBox is a two-state selection component that can be either checked or unchecked. The caption of the check box will be placed right of the actual check box. Vaadin provides two ways to create check boxes: individual check boxes with the **CheckBox** component described in this section and check box groups with the **CheckBoxGroup** component, as described in Section 6.19, “**CheckBoxGroup** and **RadioButtonGroup**”.

Clicking on a check box will change its state. The state is a **Boolean** property that you can set with the `setValue()` method and obtain with the `getValue()` method. Changing the value of a check box will cause a **ValueChangeEvent**, which can be handled by a **ValueChangeListener**.

```
CheckBox checkbox1 = new CheckBox("Box with no Check");
CheckBox checkbox2 = new CheckBox("Box with a Check");
```

```
checkbox2.setValue(true);
```

```
checkbox1.addValueChangeListener(event ->
    checkbox2.setValue(!checkbox1.getValue()));
```

The result is shown in Figure 6.33, “An Example of a Check Box”.

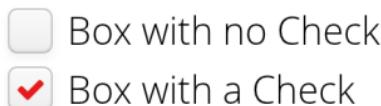


Figure 6.33. An Example of a Check Box

6.15.1. CSS Style Rules

```
.v-checkbox { }  
.v-checkbox > input { }  
.v-checkbox > label { }
```

The top-level element of a **CheckBox** has the v-checkbox style. It contains two sub-elements: the actual check box input element and the label element. If you want to have the label on the left, you can change the positions with "direction: rtl" for the top element.

6.16. ComboBox

ComboBox is a selection component allows selecting an item from a drop-down list. The component also has a text field area, which allows entering search text by which the items shown in the drop-down list are filtered. Common selection component features are described in Section 6.5. "Selection Components".

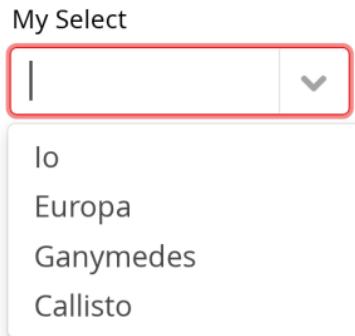


Figure 6.34. The ComboBox Component

6.16.1. Filtered Selection

ComboBox allows filtering the items available for selection in the drop-down list by the text entered in the input box. To apply custom filtering, the setItems(CaptionFilter, Collection) can be used. When using a **DataProvider**, the filtering is delegated to it.



Figure 6.35. Filtered Selection in ComboBox

Pressing **Enter** will complete the item in the input box. Pressing **Up** and **Down** arrow keys can be used for selecting an item from the drop-down list. The drop-down list is paged and clicking on the scroll buttons will change to the next or previous page. The list selection can also be done with the arrow keys on the keyboard. The shown items are loaded from the server as needed, so the number of items held in the component can be quite large. The number of matching items is displayed by the drop-down list.

6.16.2. Allowing Adding New Items

ComboBox allows the user to add new items, when the user types in a value and presses **Enter**. You need to enable this with setNewItemHandler().

Adding new items is not possible if the selection component is read-only. An attempt to do so may result in an exception.

Handling New Items

Adding new items is handled by a `NewItemHandler`, which gets the item caption string as parameter for the `accept(String)` method.

```
// List of planets
List<Planet> planets = new ArrayList<>();
planets.add(new Planet(1, "Mercury"));
planets.add(new Planet(2, "Venus"));
planets.add(new Planet(3, "Earth"));

ComboBox<Planet> select =
    new ComboBox<>("Select or Add a Planet");
select.setItems(planets);

// Use the name property for item captions
select.setItemCaptionGenerator(Planet::getName);

// Allow adding new items and add
// handling for new items
select.setNewItemHandler(inputString -> {

    Planet newPlanet = new Planet(planets.size(), inputString);
    planets.add(newPlanet);

    // Update combobox content
    select.setItems(planets);

    // Remember to set the selection to the new item
    select.select(newPlanet);
});
```

6.16.3. CSS Style Rules

```
.v-filterselect { }
.v-filterselect-input { }
.v-filterselect-button { }

// Under v-overlay-container
.v-filterselect-suggestpopup { }
.popupContent { }
.v-filterselect-prevpage,
.v-filterselect-prevpage-off { }
.v-filterselect-suggestmenu { }
.gwt-MenuItem { }
.v-filterselect-nextpage,
.v-filterselect-nextpage-off { }
.v-filterselect-status { }
```

In its default state, only the input field of the **ComboBox** component is visible. The entire component is enclosed in v-filterselect style (a legacy remnant), the input field has v-filterselect-input style and the button in the right end that opens and closes the drop-down result list has v-filterselect-button style.

The drop-down result list has an overall v-filterselect-suggest-popup style. It contains the list of suggestions with v-filterselect-suggestmenu style. When there are more items that fit in the menu, navigation buttons with v-filterselect-prevpage and v-filterselect-nextpage styles are shown. When they are not shown, the elements have -off suffix. The status bar in the bottom that shows the paging status has v-filterselect-status style.

6.17. ListSelect

The **ListSelect** component is list box that shows the selectable items in a vertical list. If the number of items exceeds the height of the component, a scrollbar is shown. The component allows selecting multiple values.

```
//Create the selection component
ListSelect<String> select = new ListSelect<>("The List");

//Add some items
select.setItems("Mercury", "Venus", "Earth", ...);

//Show 5 items and a scrollbar if there are more
select.setRows(5);

select.addValueChangeListener(event -> {
    Set<String> selected = event.getNewSelection();
    Notification.show(selected.size() + " items.");
});
```

The number of visible items is set with setRows().

The List

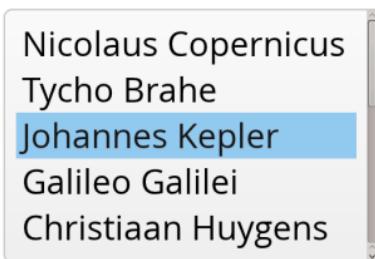


Figure 6.36. The ListSelect Component

Common selection component features are described in Section 6.5, "Selection Components".

6.17.1. CSS Style Rules

```
.v-select {}  
.v-select-select {}  
option {}
```

The component has an overall v-select style. The native <select> element has v-select-select style. The items are represented as <option> elements.

6.18. NativeSelect

NativeSelect is a selection component allows selecting an item from a drop-down list. It is implemented with the native selection input of web browsers, using the HTML <select> element.

```
// Create the selection component  
NativeSelect<String> select =  
    new NativeSelect<>("Native Selection");  
  
// Add some items  
select.setItems("Mercury", "Venus", ...);
```

The setColumns() allows setting the width of the list as "columns", which is a measure that depends on the browser.



Figure 6.37. The NativeSelect Component

Common selection component features are described in Section 6.5, “Selection Components”.

6.18.1. CSS Style Rules

```
.v-select {}  
.v-select-select {}
```

The component has a v-select overall style. The native select element has v-select-select style.

6.19. CheckBoxGroup and RadioButton-Group

RadioButtonGroup is a single selection component that allows selection from a group of radio buttons. **CheckBoxGroup** is a multiselection component where items are displayed as check boxes. The common selection component features are described in Section 6.5, “Selection Components”.

Single Selection Multiple Selection

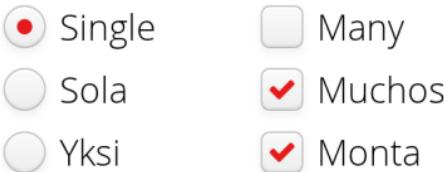


Figure 6.38. RadioButtonGroup and CheckBoxGroup

```
// A single-select radio button group
RadioButtonGroup<String> single =
    new RadioButtonGroup<>("Single Selection");
single.setItems("Single", "Sola", "Yksi");
```

```
// A multi-select check box group
CheckBoxGroup<String> multi =
    new CheckBoxGroup<>("Multiple Selection");
multi.setItems("Many", "Muchos", "Monta");
```

Figure 6.38, “RadioButtonGroup and CheckBoxGroup” shows the **RadioButtonGroup** and **CheckBoxGroup**.

You can also create check boxes individually using the **CheckBox** class, as described in Section 6.15, **“CheckBox”**. The advantages of the **CheckBoxGroup** component are that as it maintains the individual check box objects, you can get an array of the currently selected items easily, and that you can easily change the appearance of a single component and use it with a **Binder**.

6.19.1. Disabling Items

You can disable individual items in a **RadioButtonGroup** or a **CheckBoxGroup** with `setItemEnabledProvider()`. The user can not select or deselect disabled items in a **CheckBoxGroup**, but in a **RadioButtonGroup** the user can change the selection from a disabled to an enabled item. The selections can be changed programmatically regardless of whether an item is enabled or disabled.

```
// Have a radio button group with some items
RadioButtonGroup<String> group = new RadioButtonGroup<>("My Disabled Group");
group.setItems("One", "Two", "Three");
```

```
//Disable one item  
group.setItemEnabledProvider(item-> !"Two".equals(item));
```

My Disabled Group

- One
- Two
- Three

Figure 6.39. RadioButtonGroup with a Disabled Item

Setting an item as disabled turns on the v-disabled style for it.

6.19.2. CSS Style Rules

```
.v-select-optiongroup {}  
.v-select-option.v-checkbox {}  
.v-select-option.v-radiobutton {}
```

The v-select-optiongroup is the overall style for the component. Each check box will have the v-checkbox style, borrowed from the **CheckBox** component, and each radio button the v-radiobutton style. Both the radio buttons and check boxes will also have the v-select-option style that allows styling regardless of the option type. Disabled items have additionally the v-disabled style.

Horizontal Layout

The options are normally laid out vertically. You can switch to horizontal layout by using the style name *ValoTheme.OPTION-GROUP_HORIZONTAL* with addStyleName().

6.20. TwinColSelect

The **TwinColSelect** field provides a multiple selection component that shows two lists side by side, with the left column containing unselected items and the right column the selected items. The user can select items from the list on the left and click on the ">" button to move them to the list on the

right. Items can be deselected by selecting them in the right list and clicking on the "<" button.

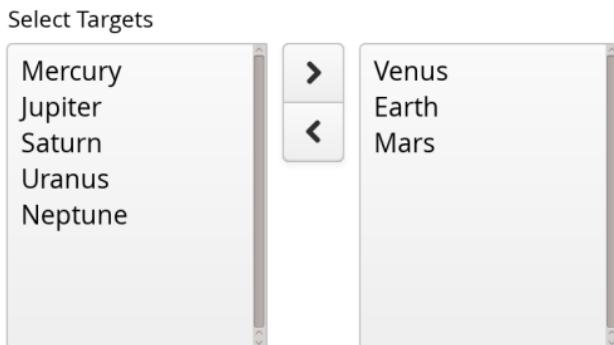


Figure 6.40. Twin Column Selection

TwinColSelect is always in multi-select mode, so its selection is always a collection of the selected items in the right column.

The selection columns can have their own captions, separate from the overall component caption, which is managed by the containing layout. You can set the column captions with `setLeftColumnCaption()` and `setRightColumnCaption()`.

```
TwinColSelect<String> select =  
    new TwinColSelect<>("Select Targets");  
  
    // Put some items in the select  
    select.setItems("Mercury", "Venus", "Earth", "Mars",  
        "Jupiter", "Saturn", "Uranus", "Neptune");  
  
    // Few items, so we can set rows to match item count  
    select.setRows(select.size());  
  
    // Preselect a few items  
    select.select("Venus", "Earth", "Mars");  
  
    // Handle value changes  
    select.addSelectionListener(event ->  
        layout.addComponent(  
            new Label("Selected: " + event.getNewSelection())));
```

The resulting component is shown in Figure 6.40, "Twin Column Selection".

The `setRows()` method sets the height of the component by the number of visible items in the selection boxes. Setting the height with `setHeight()` to a defined value overrides the `rows` setting.

Common selection component features are described in Section 6.5, “Selection Components”.

6.20.1. CSS Style Rules

```
.v-select-twincol {}  
.v-select-twincol-options-caption {}  
.v-select-twincol-selections-caption {}  
.v-select-twincol-options {}  
.v-select-twincol-buttons {}  
.v-button {}  
.v-button-wrap {}  
.v-button-caption {}  
.v-select-twincol-deco {}  
.v-select-twincol-selections {}
```

The **TwinColSelect** component has an overall `v-select-twincol` style. If set, the left and right column captions have `v-select-twincol-options-caption` and `v-select-twincol-selections-caption` style names, respectively. The left box, which displays the unselected items, has `v-select-twincol-options-caption` style and the right box, which displays the selected items, has `v-select-twincol-options-selections` style. Between them is the button area, which has overall `v-select-twincol-buttons` style; the actual buttons reuse the styles for the **Button** component. Between the buttons is a divider element with `v-select-twincol-deco` style.

6.21. Grid

6.21.1. Overview

Grid is for displaying and editing tabular data laid out in rows and columns. At the top, a *header* can be shown, and a *footer* at the bottom. In addition to plain text, the header and footer can contain HTML and components. Having components in the header allows implementing filtering easily. The grid data can be sorted by clicking on a column header; shift-clicking a column header enables secondary sorting criteria.

	Names		Years		
	First Name ^ 2	Last Name ^ 1	Born In	Died In	Lived For
<input type="checkbox"/>	Tycho	Brahe	1546 AD	1601 AD	55 years
<input type="checkbox"/>	Giordano	Bruno	1548 AD	1600 AD	52 years
<input type="checkbox"/>	Nicolaus	Copernicus	1473 AD	1543 AD	70 years
<input type="checkbox"/>	Galileo	Galilei	1564 AD	1642 AD	78 years
<input type="checkbox"/>	Christiaan	Huygens	1629 AD	1695 AD	66 years
			1555,17	1618,50	63,33

Figure 6.41. A Grid

The data area can be scrolled both vertically and horizontally. The leftmost columns can be frozen, so that they are never scrolled out of the view. The data is loaded lazily from the server, so that only the visible data is loaded. The smart lazy loading functionality gives excellent user experience even with low bandwidth, such as mobile devices.

The grid data can be edited with a row-based editor after double-clicking a row. The fields are set explicitly, and bound to data.

Grid is fully themeable with CSS and style names can be set for all grid elements. For data rows and cells, the styles can be generated with a row or cell style generator.

6.21.2. Binding to Data

Grid is normally used by binding it to a , described in Section 10.4, “Showing Many Items in a Listing”. By default, it is bound to List of items. You can set the items with the `setItems()` method.

For example, if you have a list of beans, you show them in a **Grid** as follows

```

// Have some data
List<Person> people = Lists.newArrayList(
    new Person("Nicolaus Copernicus", 1543),
    new Person("Galileo Galilei", 1564),
    new Person("Johannes Kepler", 1571));

// Create a grid bound to the list
Grid<Person> grid = new Grid<>();
grid.setItems(people);
grid.addColumn("Name", Person::getName);
grid.addColumn("Year of birth", Person::getBirthYear);
layout.addComponent(grid);

```

6.21.3. Handling Selection Changes

Selection in **Grid** is handled a bit differently from other selection components, as it is not a **HasValue**. Grid supports single, multiple, or no-selection, each defined by a specific selection model. Each selection model has a specific API depending on the type of the selection.

For basic usage, switching between the built-in selection models is possible by using the `setSelectionMode(SelectionMode)`. Possible options are `SINGLE` (default), `MULTI`, or `NONE`.

Listening to selection changes in any selection model is possible with a **SelectionListener**, which provides a generic **SelectionEvent** which for getting the selected value or values. Note that the listener is actually attached to the selection model and not the grid, and will stop getting any events if the selection mode is changed.

```

Grid<Person> grid = new Grid<>();

// switch to multiselect mode
grid.setSelectionMode(SelectionMode.MULTI);

grid.addSelectionListener(event -> {
    Set<Person> selected = event.getAllSelectedItems();
    Notification.show(selected.size() + " items selected");
})

```

Programmatically selecting the value is possible via `select(T)`. In multiselect mode, this will add the given item to the selection.

```
// in single-select, only one item is selected  
grid.select(defaultPerson);  
  
// switch to multi select, clears selection  
grid.setSelectionMode(SelectionMode.MULTI);  
// Select items 2-4  
people.subList(2,3).forEach(grid::select);
```

The current selection can be obtained from the **Grid** by `getSelectedItems()`, and the returned **Set** contains either only one item (in single-selection mode) or several items (in multi-selection mode).



Warning

If you change selection mode for a grid, it will clear the selection and fire a selection event. To keep the previous selection you must reset the selection afterwards using the `select()` method.



Warning

If you change the grid's items with `setItems()` or the used **DataProvider**, it will clear the selection and fire a selection event. To keep the previous selection you must reset the selection afterwards using the `select()` method.

Selection Models

For more control over the selection, you can access the used selection model with `getSelectionModel()`. The return type is **GridSelectionModel** which has generic selection model API, but you can cast that to the specific selection model type, typically either **SingleSelectionModel** or **MultiSelectionModel**.

The selection model is also returned by the `setSelectionMode(SelectionMode)` method.

```
// the default selection model  
SingleSelectionModel<Person> defaultModel =  
(SingleSelectionModel<Person>) grid.getSelectionModel();  
  
// Use multi-selection mode  
MultiSelectionModel<Person> selectionModel =  
(MultiSelectionModel<Person>) grid.setSelectionMode(SelectionMode.MULTI);
```

Single Selection Model

By obtaining a reference to the **SingleSelectionModel**, you can access more fine grained API for the single-select case.

The `addSingleSelect(SingleSelectionListener)` method provides access to **SingleSelectionEvent**, which has some extra API for more convenience.

In single-select mode, it is possible to control whether the empty (null) selection is allowed. By default it is enabled, but can be disabled with `setDeselectAllowed()`.

```
// preselect value  
grid.select(defaultItem);
```

```
SingleSelectionModel<Person> singleSelect =  
    (SingleSelectionModel<Person>) grid.getSelectionModel();  
// disallow empty selection  
singleSelect.setDeselectAllowed(false);
```

Multi-Selection Model

In the multi-selection mode, a user can select multiple items by clicking on the checkboxes in the leftmost column, or by using the **Space** to select/deselect the currently focused row. Space bar is the default key for toggling the selection, but it can be customized.

<input type="checkbox"/>	Name	City	Year
<input type="checkbox"/>	Charles Newton	Oxford	1 877
<input checked="" type="checkbox"/>	Ada Lovelace	Innsbruck	1 815
<input type="checkbox"/>	Charles Lovelace	London	1 878
<input checked="" type="checkbox"/>	Ada Darwin	Innsbruck	1 861
<input checked="" type="checkbox"/>	Charles Newton	Oxford	1 982
<input type="checkbox"/>	Charles Adams	Innsbruck	1 940
<input type="checkbox"/>	Ada Adams	Innsbruck	1 976

Delete Selected

Figure 6.42. Multiple Selection in Grid

By obtaining a reference to the **MultiSelectionModel**, you can access more fine grained API for the multi-select case.

The **MultiSelectionModel** provides `addMultiSelectionListener(MultiSelectionListener)` access to **MultiSelectionEvent**, which allows to easily access differences in the selection change.

```
// Grid in multi-selection mode
Grid<Person> grid = Grid<>()
grid.setItems(people);
MultiSelectionModel<Person> selectionModel
= (MultiSelectionModel<Person>) grid.setSelectionMode(SelectionMode.MULTI);

selectionModel.selectAll();

selectionModel.addMultiSelectionListener(event -> {
    Notification.show(selection.getAddedSelection().size()
        + " items added."
        + selection.getRemovedSelection().size()
        + " removed.");
}

// Allow deleting only if there's any selected
deleteSelected.setEnabled(
    event.getNewSelection().size() > 0);
});
```

Focus and Clicks

In addition to selecting rows, you can focus individual cells. The focus can be moved with arrow keys and, if editing is enabled, pressing **Enter** opens the editor. Normally, pressing **Tab** or **Shift+Tab** moves the focus to another component, as usual.

When editing or in unbuffered mode, **Tab** or **Shift+Tab** moves the focus to the next or previous cell. The focus moves from the last cell of a row forward to the beginning of the next row, and likewise, from the first cell backward to the end of the previous row. Note that you can extend **DefaultEditorEventHandler** to change this behavior.

With the mouse, you can focus a cell by clicking on it. The clicks can be handled with an `ItemClickListener`. The **ItemClickEvent** object contains various information, most importantly the ID of the clicked row and column.

```
grid.addCellClickListener(event ->
    Notification.show("Value: " + event.getItem());
```

The clicked grid cell is also automatically focused.

The focus indication is themed so that the focused cell has a visible focus indicator style by default, while the row does not. You can enable row focus, as well as disable cell focus, in a custom theme. See Section 6.21.11, "Styling with CSS".

6.21.4. Configuring Columns

Columns are normally defined in the container data source. The `addColumn()` method can be used to add columns to **Grid**.

Column configuration is defined in **Grid.Column** objects, which can be obtained from the grid with `getColumns()`.

```
Column<Date> bornColumn = grid.addColumn(Person::getBirthDate);
bornColumn.setHeaderCaption("Born date");
```

In the following, we describe the basic column configuration.

Column Order

You can set the order of columns with `setColumnOrder()` for the grid. Columns that are not given for the method are placed after the specified columns in their natural order.

```
grid.setColumnOrder(firstnameColumn, lastnameColumn,
                   bornColumn, birthplaceColumn,
                   diedColumn);
```

Note that the method can not be used to hide columns. You can hide columns with the `removeColumn()`, as described later.

Hiding and Removing Columns

Columns can be hidden by calling `setHidden()` in **Column**. Furthermore, you can set the columns user hideable using method `setHideable()`.

Columns can be removed with `removeColumn()` and `removeAllColumns()`. To restore a previously removed column, you can call `addColumn()`.

Column Captions

Column captions are displayed in the grid header. You can set the header caption explicitly through the column object with `setHeaderCaption()`.

```
Column<Date> bornColumn = grid.addColumn(Person::getBirthDate);  
bornColumn.setHeaderCaption("Born date");
```

This is equivalent to setting it with `setText()` for the header cell; the **HeaderCell** also allows setting the caption in HTML or as a component, as well as styling it, as described later in Section 6.21.7, "Header and Footer".

Column Widths

Columns have by default undefined width, which causes automatic sizing based on the widths of the displayed data. You can set column widths explicitly by pixel value with `setWidth()`, or relatively using expand ratios with `setExpandRatio()`.

When using expand ratios, the columns with a non-zero expand ratio use the extra space remaining from other columns, in proportion to the defined ratios.

You can specify minimum and maximum widths for the expanding columns with `setMinimumWidth()` and `setMaximumWidth()`, respectively.

The user can resize columns by dragging their separators with the mouse. When resized manually, all the columns widths are set to explicit pixel values, even if they had relative values before.

Frozen Columns

You can set the number of columns to be frozen with `setFrozenColumnCount()`, so that they are not scrolled off when scrolling horizontally.

```
grid.setFrozenColumnCount(2);
```

Setting the count to `0` disables frozen data columns; setting it to `-1` also disables the selection column in multi-selection mode.

6.21.5. Generating Columns

Columns with values computed from other columns can be simply added by using lambdas:

```
// Add generated full name column
Column<String> fullNameColumn = grid.addColumn(person ->
    person.getFirstName() + " " + person.getLastName());
fullNameColumn.setHeaderCaption("Full name");
```

6.21.6. Column Renderers

A *renderer* is a feature that draws the client-side representation of a data value. This allows having images, HTML, and buttons in grid cells.

Picture	Name	Born	Link	Button
	Nicolaus Copernicus	March 19, 1473	more info	Delete
	Galileo Galilei	March 15, 1564	more info	Delete
	Johannes Kepler	January 27, 1572	more info	Delete

Figure 6.43. Column renderers: image, date, HTML, and button

Renderers implement the Renderer interface. Renderers require a specific data type for the column. You set the column renderer in the **Grid.Column** object as follows:

```
// the type of birthYear is a number
Column<Integer> bornColumn = grid.addColumn(Person::getBirthYear,
    new NumberRenderer("born in %d AD"));
```

The following renderers are available, as defined in the server-side com.vaadin.ui.renderers package:

TextRenderer

The default renderer, displays plain text as is. Any HTML markup is quoted.

ButtonRenderer

Renders the data value as the caption of a button. A RendererClickListener can be given to handle the button clicks.

Typically, a button renderer is used to display buttons for operating on a data item, such as edit, view, delete, etc. It is not meaningful to store the button captions in the data source, rather you want to generate them, and they are usually all identical.

```
List<Person> people = new ArrayList<>();
```

```
people.add(new Person("Nicolaus Copernicus", 1473));  
people.add(new Person("Galileo Galilei", 1564));  
people.add(new Person("Johannes Kepler", 1571));
```

// Create a grid

```
Grid<Person> grid = new Grid(people);
```

// Render a button that deletes the data row (item)

```
grid.addColumn(person -> "Delete",  
    new ButtonRenderer(clickEvent -> {  
        people.remove(clickEvent.getValue());  
        grid.setItems(people);  
    }));
```

ImageRenderer

Renders the cell as an image. The column type must be a Resource, as described in Section 5.5, "Images and Other Resources"; only **ThemeResource** and **ExternalResource** are currently supported for images in **Grid**.

```
Column<ThemeResource> imageColumn = grid.addColumn("picture",  
    p -> new ThemeResource("img/" + p.getLastname() + ".jpg"));  
imageColumn.setRenderer(new ImageRenderer());
```

DateRenderer

Formats a column with a **Date** type using string formatter. The format string is same as for String.format() in Java API. The date is passed in the parameter index 1, which can be omitted if there is only one format specifier, such as "%tF".

```
Grid.Column<Date> bornColumn = grid.addColumn(person::getBirthDate,  
    new DateRenderer("%tB %tY",  
        Locale.ENGLISH));
```

Optionally, a locale can be given. Otherwise, the default locale (in the component tree) is used.

HTMLRenderer

Renders the cell as HTML. This allows formatting the cell content, as well as using HTML features such as hyperlinks.

Set the renderer in the **Grid.Column** object:

```
Column<String> htmlColumn grid.addColumn(person ->
    "<a href=\"" + person.getDetailsUrl() + " target='_top'>info</a>",
    new HtmlRenderer());
```

NumberRenderer

Formats column values with a numeric type extending **Number**: **Integer**, **Double**, etc. The format can be specified either by the subclasses of **java.text.NumberFormat**, namely **DecimalFormat** and **ChoiceFormat**, or by `String.format()`.

For example:

```
// Use String.format() formatting
Column<Double> ratingCol = grid.addColumn("rating",
    new NumberRenderer("%02.4f", Locale.ENGLISH));
```

ProgressBarRenderer

Renders a progress bar in a column with a **Double** type. The value must be between 0.0 and 1.0.

Custom Renderers

Renderers are component extensions that require a client-side counterpart. See Section 15.4.1, “Renderers” for information on implementing custom renderers.

6.21.7. Header and Footer

A grid by default has a header, which displays column names, and can have a footer. Both can have multiple rows and neighbouring header row cells can be joined to feature column groups.

Adding and Removing Header and Footer Rows

A new header row is added with `prependHeaderRow()`, which adds it at the top of the header, `appendHeaderRow()`, which adds it at the bottom of the header, or with `addHeaderRowAt()`, which inserts it at the specified 0-base index. All of the methods return a **HeaderRow** object, which you can use to work on the header further.

```
// Group headers by joining the cells
```

```
HeaderRow groupingHeader = grid.prependHeaderRow();
```

```
...
```

```
// Create a header row to hold column filters
```

```
HeaderRow filterRow = grid.appendHeaderRow();
```

```
...
```

Similarly, you can add footer rows with `appendFooterRow()`, `prependFooterRow()`, and `addFooterRowAt()`.

Joining Header and Footer Cells

You can join two or more header or footer cells with the `join()` method. For header cells, the intention is usually to create column grouping, while for footer cells, you typically calculate sums or averages.

```
// Group headers by joining the cells
```

```
HeaderRow groupingHeader = grid.prependHeaderRow();
```

```
HeaderCell namesCell = groupingHeader.join(
```

```
groupingHeader.getCell("firstname").
```

```
groupingHeader.getCell("lastname")).setText("Person");
```

```
HeaderCell yearsCell = groupingHeader.join(
```

```
groupingHeader.getCell("born").
```

```
groupingHeader.getCell("died").
```

```
groupingHeader.getCell("lived")).setText("Dates of Life");
```

Text and HTML Content

You can set the header caption with `setText()`, in which case any HTML formatting characters are quoted to ensure security.

```
HeaderRow mainHeader = grid.getDefaultHeaderRow();  
mainHeader.getCell("firstname").setText("First Name");  
mainHeader.getCell("lastname").setText("Last Name");  
mainHeader.getCell("born").setText("Born In");  
mainHeader.getCell("died").setText("Died In");  
mainHeader.getCell("lived").setText("Lived For");
```

To use raw HTML in the captions, you can use setHtml().

```
namesCell.setHtml("<b>Names</b>");  
yearsCell.setHtml("<b>Years</b>");
```

Components in Header or Footer

You can set a component in a header or footer cell with setComponent(). Often, this feature is used to allow filtering.

6.21.8. Sorting

A user can sort the data in a grid on a column by clicking the column header. Clicking another time on the current sort column reverses the sort direction. Clicking on other column headers while keeping the Shift key pressed adds a secondary or more sort criteria.

Name	▲ 2	City	▲ 1	Year	
Ada Adams		Innsbruck		1 843	
Ada Darwin		Innsbruck		1 880	
Ada Lovelace		Innsbruck		1 868	
Ada Newton		Innsbruck		1 001	

Figure 6.44. Sorting Grid on Multiple Columns

Defining sort criteria programmatically can be done with the various alternatives of the sort() method. You can sort on a specific column with sort(Column column), which defaults to ascending sorting order, or sort(Column column, SortDirection direction), which allows specifying the sort direction.

```
grid.sort(nameColumn, SortDirection.DESCENDING);
```

To sort on multiple columns, you need to use the fluid sort API with sort(Sort), which allows chaining sorting rules. Sorting rules are created with the static by() method, which defines the primary sort column, and then(), which can be used to specify any secondary sort columns. They default to ascending

sort order, but the sort direction can be given with an optional parameter.

```
// Sort first by city and then by name
grid.sort(Sort.by(cityColumn, SortDirection.ASCENDING)
    .then(nameColumn, SortDirection.DESCENDING));
```

6.21.9. Editing Items Inside Grid

Grid supports line-based editing, where double-clicking a row opens the row editor. In the editor, the input fields can be edited, as well as navigated with **Tab** and **Shift+Tab** keys. If validation fails, an error is displayed and the user can correct the inputs.

The **Editor** is accessible via `getEditor()`, and to enable editing, you need to call `[methodname]#setEnabled(true)` on it.

The editor is based on **Binder** which is used to bind the data to the editor. See Section 10.3.3, “Binding Beans to Forms” for more information on setting up field components and validation by using **Binder**. The `[classname]#Binder` needs to be set with `setBinder` in **Editor**.

```
List<Todo> items = Arrays.asList(new Todo("Done task", true),
    new Todo("Not done", false));

Grid<Todo> grid = new Grid<>();

TextField taskField = new TextField();
CheckBox doneField = new CheckBox();
Binder<Todo> binder = new Binder<>();

binder.bind(taskField, Todo::getTask, Todo::setTask);
binder.bind(doneField, Todo::isDone, Todo::setDone);

grid.getEditor().setBinder(binder);
grid.getEditor().setEnabled(true);

Column<Todo, String> column = grid
    .addColumn(todo -> String.valueOf(todo.isDone()));
column.setWidth(75);
column.setEditorComponent(doneField);

grid.addColumn(Todo::getTask).setEditorComponent(taskField);
```

It is possible to customize the used editor component for each column and row, by using `setEditorComponentGenerator(EditorComponentGenerator)` in **Column**.

Buffered / Unbuffered Mode

Grid supports two editor modes - buffered and unbuffered. The default mode is buffered. The mode can be changed with setBuffered(false).

In the buffered mode, editor has two buttons visible: a **Save** button that commits the modifications to the bean and closes the editor and a **Cancel** button discards the changes and exits the editor.

Editor in buffered mode is illustrated in Figure 6.45, "Editing a Grid Row".

Name	City	Year
Charles Lovelace	Innsbruck	1 965
Ada Lovelace	Turku	1 947
Charles Lovelace	Turku	1 968
Save Cancel		
Isaac Adams	Innsbruck	1 818
Isaac Newton	Innsbruck	1 804

Figure 6.45. Editing a Grid Row

In the unbuffered mode, the editor has no buttons and all changed data is committed directly to the data provider. If another row is clicked, the editor for the current row is closed and a row editor for the clicked row is opened.

Customizing Editor Buttons

In the buffered mode, the editor has two buttons: **Save** and **Cancel**. You can set their captions with setEditorSaveCaption() and setEditorCancelCaption(), respectively.

In the following example, we demonstrate one way to translate the captions:

```
// Localize the editor button captions  
grid.getEditor().setSaveCaption("Tallenna");  
grid.getEditor().setCancelCaption("Peruuta"));
```

Handling Validation Errors

The input fields are validated when the value is updated. The default error handler displays error indicators in the invalid fields, as well as the first error in the editor.

Name	City	Year
Douglas Darwin	Turku	1 873
<i>AdaNewton</i>	Innsbruck	1 906
! Name: Need first and last name		<input type="button" value="Save"/> <input type="button" value="Cancel"/>
Charles Darwin	Oxford	1 911

Figure 6.46. Editing a Grid Row

You can modify the error message by implementing a custom `EditorErrorGenerator` with for the `Editor`.

6.21.10. Generating Row or Cell Styles

You can style entire rows or individual cells with a `StyleGenerator`, typically used through Java lambdas.

Generating Row Styles

You set a `StyleGenerator` to a grid with `setStyleGenerator()`. The `getStyle()` method gets a date item, and should return a style name or `null` if no style is generated.

For example, to add a style names to rows having certain values in one property of an item, you can style them as follows:

```
grid.setStyleGenerator(person -> {  
    // Style based on alive status  
    person.isAlive() ? null : "dead";  
});
```

You could then style the rows with CSS as follows:

```
.v-grid-row.dead {  
    color: gray;  
}
```

Generating Cell Styles

You set a StyleGenerator to a grid with `setStyleGenerator()`. The `getStyle()` method gets a **CellReference**, which contains various information about the cell and a reference to the grid, and should return a style name or `null` if no style is generated.

For example, to add a style name to a specific column, you can match on the column as follows:

```
// Static style based on column  
bornColumn.setStyleGenerator(person -> "rightalign");
```

You could then style the cells with a CSS rule as follows:

```
.v-grid-cell.rightalign {  
    text-align: right;  
}
```

6.21.11. Styling with CSS

```
.v-grid {  
    .v-grid-scroller. v-grid-scroller-horizontal {}  
    .v-grid-tablewrapper {  
        .v-grid-header {  
            .v-grid-row {  
                .v-grid-cell, .frozen, .v-grid-cell-focused {}  
            }  
        }  
        .v-grid-body {  
            .v-grid-row,  
            .v-grid-row-stripe,  
            .v-grid-row-has-data {  
                .v-grid-cell, .frozen, .v-grid-cell-focused {}  
            }  
        }  
        .v-grid-footer {  
            .v-grid-row {  
                .v-grid-cell, .frozen, .v-grid-cell-focused {}  
            }  
        }  
    }  
    .v-grid-header-deco {}  
    .v-grid-footer-deco {}
```

```
.v-grid-horizontal-scrollbar-deco { }  
.v-grid-editor {  
  .v-grid-editor-cells { }  
  .v-grid-editor-footer {  
    .v-grid-editor-message { }  
  }  
  .v-grid-editor-buttons {  
    .v-grid-editor-save { }  
    .v-grid-editor-cancel { }  
  }  
}  
}
```

A **Grid** has an overall v-grid style. The actual grid has three parts: a header, a body, and a footer. The scrollbar is a custom element with v-grid-scroller style. In addition, there are some decoration elements.

Grid cells, whether they are in the header, body, or footer, have a basic v-grid-cell style. Cells in a frozen column additionally have a frozen style. Rows have v-grid-row style, and every other row has additionally a v-grid-row-stripe style.

The focused row has additionally v-grid-row-focused style and focused cell v-grid-cell-focused. By default, cell focus is visible, with the border stylable with `$v-grid-cell-focused-border` parameter in Sass. Row focus has no visible styling, but can be made visible with the `$v-grid-row-focused-background-color` parameter or with a custom style rule.

In editing mode, a v-grid-editor overlay is placed on the row under editing. In addition to the editor field cells, it has an error message element, as well as the buttons.

6.22. MenuBar

The **MenuBar** component allows creating horizontal drop-down menus, much like the main menu in desktop applications.

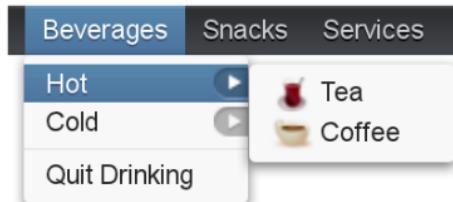


Figure 6.47. Menu Bar

The menu items open as the user navigates them by hovering or clicking with the mouse. Menus can have separators to divide items into sub-sections. Menu items can have an icon and styling. They can also be checkable, so that the user can click on them to toggle between checked and unchecked.

6.22.1. Creating a Menu

The actual menu bar component is first created as follows:

```
MenuBar barmenu = newMenuBar();
main.addComponent(barmenu);
```

You insert the top-level menu items to the **MenuBar** object with the `addItem()` method. It takes a string label, an icon resource, and a command as its parameters. The icon and command are not required and can be *null*. The `addItem()` method returns a **MenuBar.MenuItem** object, which you can use to add sub-menu items. The **MenuItem** has an identical `addItem()` method.

For example (the command is explained later):

```
//A top-level menu item that opens a submenu
MenuItem drinks = barmenu.addItem("Beverages", null, null);
```

```
//Submenu item with a sub-submenu
MenuItem hots = drinks.addItem("Hot", null, null);
hots.addItem("Tea",
    new ThemeResource("icons/tea-16px.png"), mycommand);
hots.addItem("Coffee",
    new ThemeResource("icons/coffee-16px.png"), mycommand);
```

```
//Another submenu item with a sub-submenu
MenuItem colds = drinks.addItem("Cold", null, null);
colds.addItem("Milk", null, mycommand);
colds.addItem("Weissbier", null, mycommand);
```

```
// Another top-level item
MenuItem snacks = barmenu.addItem("Snacks", null, null);
snacks.addItem("Weisswurst", null, mycommand);
snacks.addItem("Bratwurst", null, mycommand);
snacks.addItem("Currywurst", null, mycommand);

// Yet another top-level item
MenuItem servs = barmenu.addItem("Services", null, null);
servs.addItem("Car Service", null, mycommand);
```

6.22.2. Handling Menu Selection

Menu selection is handled by executing a *command* when the user selects an item from the menu. A command is a callback class that implements the **MenuBar.Command** interface.

```
// A feedback component
final Label selection = new Label("-");
main.addComponent(selection);

// Define a common menu command for all the menu items.
MenuBar.Command mycommand = newMenuBar.Command() {
    public void menuSelected(MenuItem selectedItem) {
        selection.setValue("Ordered a " +
            selectedItem.getText() +
            " from menu.");
    }
};
```

6.22.3. CSS Style Rules

```
.v-menubar { }
.v-menubar-submenu { }
.v-menubar-menuitem { }
.v-menubar-menuitem-caption { }
.v-menubar-menuitem-selected { }
.v-menubar-submenu-indicator { }
```

The menu bar has the overall style name `.v-menubar`. Each menu item has `.v-menubar-menuitem` style normally and additionally `.v-menubar-selected` when the item is selected, that is, when the mouse pointer hovers over it. The item caption is inside a `v-menubar-menuitem-caption`. In the top-level menu bar, the items are directly under the `component` element.

Submenus are floating `v-menubar-submenu` elements outside the menu bar element. Therefore, you should not try to match

on the component element for the submenu popups. In submenus, any further submenu levels are indicated with a v-menubar-submenu-indicator.

6.23. Upload

The **Upload** component allows a user to upload files to the server. It has two different modes controlled with `setImmediateMode(boolean)`, that affect the user workflow.

Immediate (default)

In the immediate mode, the upload displays a file name entry box and a button for selecting the file. The uploading is started immediately after the file has been selected.

Non-immediate

In the non-immediate mode, the upload displays a file name entry box, a button for selecting the file and a button for starting the upload. After the file is selected, the user starts the upload by clicking the submit button.

Uploading requires a receiver that implements `Upload.Receiver` to provide an output stream to which the upload is written by the server.

```
Upload upload = new Upload("Upload it here", receiver);
upload.setImmediateMode(false);
```

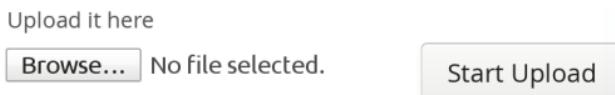


Figure 6.48. The Upload component in non-immediate mode

In the image above, the upload is in non-immediate mode. By default in the immediate mode, only the **Start Upload** button is visible.

You can set the text of the upload button with `setButtonCaption()`. Note that it is difficult to change the caption or look of the **Browse** button. This is a security feature of web browsers. The language of the **Browse** button is determined by the browser, so if you wish to have the language of the **Upload**

component consistent, you will have to use the same language in your application.

```
upload.setButtonCaption("Upload Now");
```

You can also hide the upload button with `.v-upload .v-button {display: none}` in theme, have custom logic for starting the upload, and call `startUpload()` to start it.

6.23.1. Receiving Upload Data

The uploaded files are typically stored as files in a file system, in a database, or as temporary objects in memory. The `upload` component writes the received data to an **java.io.OutputStream** so you have plenty of freedom in how you can process the upload content.

To use the **Upload** component, you need to implement the **Upload.Receiver** interface. The `receiveUpload()` method of the receiver is called when the user clicks the submit button. The method must return an **OutputStream**. To do this, it typically creates a file or a memory buffer to which the stream is written. The method gets the file name and MIME type of the file, as reported by the browser.

While uploading, the upload progress can be monitored with an `Upload.ProgressListener`. The `updateProgress()` method gets the number of read bytes and the content length as parameters. The content length is reported by the browser, but the reported value is not reliable, and can also be unknown, in which case the value is `-1`. It is therefore recommended to follow the upload progress and check the allowed size in a progress listener. Upload can be terminated by calling `interruptUpload()` on the upload component. You may want to use a **ProgressBar** to visualize the progress, and in indeterminate mode if the content length is not known.

When an upload is finished, successfully or unsuccessfully, the **Upload** component will emit the **Upload.FinishedEvent** event, which you can handle with an **Upload.FinishedListener** added to the upload component. The event object will include the file name, MIME type, and final length of the file. More specific **Upload.FailedEvent** and **Upload.SucceededEvent** events will

be called in the cases where the upload failed or succeeded, respectively.

The following example uploads images to /tmp/uploads directory in (UNIX) filesystem (the directory must exist or the upload fails). The component displays the uploaded image in an **Image** component.

```
// Show uploaded file in this placeholder
final Image image = new Image("Uploaded Image");

// Implement both receiver that saves upload in a file and
// listener for successful upload
class ImageUploader implements Receiver, SucceededListener {
    public File file;

    public OutputStream receiveUpload(String filename,
                                      String mimeType) {
        // Create and return a file output stream
        ...
    }

    public void uploadSucceeded(SucceededEvent event) {
        // Show the uploaded file in the image viewer
        image.setSource(new FileResource(file));
    }
}
ImageUploader receiver = new ImageUploader();

// Create the upload with a caption and set receiver later
Upload upload = new Upload("Upload Image Here", receiver);
upload.addSucceededListener(receiver);
```

6.23.2. CSS Style Rules

```
.v-upload { }
.gwt-FileUpload { }
.v-button { }
.v-button-wrap { }
.v-button-caption { }
```

The **Upload** component has an overall v-upload style. The upload button has the same structure and style as a regular **Button** component.

6.24. ProgressBar

The **ProgressBar** component allows visualizing progress of a task. The progress is specified as a floating-point value between 0.0 and 1.0.



Figure 6.49. The ProgressBar component

To display upload progress with the **Upload** component, you can update the progress bar in a ProgressListener.

When the position of a progress bar is done in a background thread, the change is not shown in the browser immediately. You need to use either polling or server push to update the browser. You can enable polling with `setPollInterval()` in the current UI instance. See Section 11.15, “Server Push” for instructions about using server push. Whichever method you use to update the UI, it is important to lock the user session by modifying the progress bar value inside `access()` call, as illustrated in the following example and described in Section 11.15.3, “Accessing UI from Another Thread”.

```
ProgressBar bar = new ProgressBar(0.0f);
layout.addComponent(bar);

layout.addComponent(new Button("Increase", click -> {
    float current = bar.getValue();
    if (current < 1.0f)
        bar.setValue(current + 0.1f);
}));
```

6.24.1. Indeterminate Mode

In the indeterminate mode, a non-progressive indicator is displayed continuously. The indeterminate indicator is a circular wheel in the built-in themes. The progress value has no meaning in the indeterminate mode.

```
ProgressBar bar = new ProgressBar();
bar.setIndeterminate(true);
```



Figure 6.50. Indeterminate progress bar

6.24.2. Doing Heavy Computation

The progress bar is typically used to display the progress of a heavy server-side computation task, often running in a background thread. The UI, including the progress bar, can be updated either with polling or by using server push. When doing so, you must ensure thread-safety, most easily by updating the UI inside a `UI.access()` call in a `Runnable`, as described in Section 11.15.3, “Accessing UI from Another Thread”.

6.24.3. CSS Style Rules

```
.v-progressbar, v-progressbar-indeterminate {}  
.v-progressbar-wrapper {}  
.v-progressbar-indicator {}
```

The progress bar has a `v-progressbar` base style. The progress is an element with `v-progressbar-indicator` style inside the wrapper, and therefore displayed on top of it. When the progress element grows, it covers more and more of the animated background.

The progress bar can be animated (some themes use that). Animation is done in the element with the `v-progressbar-wrapper` style, by having an animated GIF as the background image.

In the indeterminate mode, the top element also has the `v-progressbar-indeterminate` style. The built-in themes simply display the animated GIF in the top element and have the inner elements disabled.

6.25. Slider

The **Slider** is a vertical or horizontal bar that allows setting a numeric value within a defined range by dragging a bar handle with the mouse. The value is shown when dragging the handle.

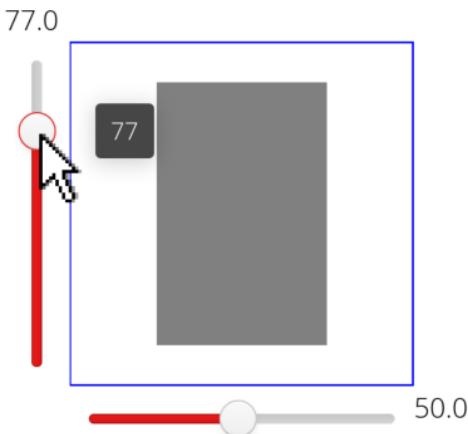


Figure 6.51. Vertical and horizontal Slider components

Slider has a number of different constructors that take a combination of the caption, *minimum* and *maximum* value, *resolution*, and the *orientation* of the slider.

// Create a vertical slider

```
Slider vertslider = new Slider(1, 100);  
vertslider.setOrientation(Orientation.VERTICAL);
```

min

Minimum value of the slider range. The default is 0.0.

max

Maximum value of the slider range. The default is 100.0.

resolution

The number of digits after the decimal point. The default is 0.

orientation

The orientation can be either horizontal (*Orientation.HORIZONTAL*) or vertical (*Orientation.VERTICAL*). The default is horizontal.

As the **Slider** is a field component, you can handle value changes with a **ValueChangeListener**. The value of the **Slider** field is a **Double** object.

```
// Shows the value of the vertical slider
final Label vertvalue = new Label();

// Handle changes in slider value.
vertslider.addValueChangeListener(event -> {
    float value = event.getValue().floatValue();
    box.setHeight(value, Unit.PERCENTAGE);
    vertvalue.setValue(String.valueOf(value));
});

});
```

You can set the value with the `setValue()` method defined in **Slider** that takes the value as a **Double**. If the value is outside the configured bounds, the setter throws a **ValueOutOfBoundsException**.

```
// Set the initial value. This has to be set after the
// listener is added if we want the listener to handle
// also this value change.
try {
    vertslider.setValue(50.0);
} catch (ValueOutOfBoundsException e) {
```

Figure 6.51, "Vertical and horizontal **Slider** components" shows both vertical (from the code examples) and horizontal sliders that control the size of a box. The slider values are displayed also in separate labels.

6.25.1. CSS Style Rules

```
.v-slider {}
.v-slider-base {}
.v-slider-handle {}
```

The enclosing style for the **Slider** is v-slider. The slider bar has style v-slider-base. Even though the handle is higher (for horizontal slider) or wider (for vertical slider) than the bar, the handle element is nevertheless contained within the slider bar element. The appearance of the handle comes from a background image defined in the *background* CSS property.

6.26. PopupView

The **PopupView** component allows opening a pop-up view either by clicking on a link or programmatically. The compon-

ent has two representations: a minimized textual representation and the popped-up content. The view can contain any components. The view closes automatically when the mouse pointer moves outside the view.

In the following, we have a popup view with a text field and a button that opens automatically when the user clicks on a "Open the popup" link:

// Content for the PopupView

```
VerticalLayout popupContent = new VerticalLayout();
popupContent.addComponent(new TextField("Textfield"));
popupContent.addComponent(new Button("Button"));
```

// The component itself

```
PopupView popup = new PopupView("Pop it up", popupContent);
layout.addComponent(popup);
```

If the textual minimized representation is not given (a null is given), the component is invisible in the minimized state. The pop-up can be opened programmatically by calling setPopupVisible(true). For example:

// A pop-up view without minimized representation

```
PopupView popup = new PopupView(null, myComponent);
```

// A component to open the view

```
Button button = new Button("Show details", click ->
    popup.setPopupVisible(true));
```

```
layout.addComponent(button, popup);
```

When the pop-up is opened or closed, a **PopupVisibilityEvent** is fired, which can be handled with a `PopupVisibilityListener` added with `setPopupVisibilityListener()`.

// Fill the pop-up content when it's popped up

```
popup.addPopupVisibilityListener(event -> {
    if (event.isPopupVisible()) {
        popupContent.removeAllComponents();
        popupContent.addComponent(new Table(null,
            TableExample.generateContent()));
    }
});
```

6.26.1. CSS Style Rules

```
.v-popupview {}  
.v-overlay-container {  
    .v-popupview-popup {  
        .popupContent {}  
    }  
}
```

In minimalized state, the component has v-popupview style. When popped up, the pop-up content is shown in a v-popupview-popup overlay element under the v-overlay-container, which contains all floating overlays outside the component hierarchy.

6.27. Composition with CustomComponent

The ease of making new user interface components is one of the core features of Vaadin. Typically, you simply combine existing built-in components to produce composite components. In many applications, such composite components make up the majority of the user interface.

As described earlier in Section 5.2.2, “Compositing Components”, you have two basic ways to create a composite - either by extending a layout component or the **CustomComponent**, which typically wraps around a layout component. The benefit of wrapping a layout composite in **CustomComponent** is mainly encapsulation - hiding the implementation details of the composition. Otherwise, a user of the composite could rely on implementation details, which would create an unwanted dependency.

To create a composite, you need to inherit the **CustomComponent** and set the *composition root* component in the constructor. The composition root is typically a layout component that contains other components.

For example:

```
class MyComposite extends CustomComponent {  
    public MyComposite(String message) {  
        // A layout structure used for composition  
        Panel panel = new Panel("My Custom Component");  
        VerticalLayout panelContent = new VerticalLayout();
```

```

panel.setContent(panelContent);

// Compose from multiple components
Label label = new Label(message);
panelContent.addComponent(label);
panelContent.addComponent(new Button("Ok"));

// Set the size as undefined at all levels
panelContent.setSizeUndefined();
panel.setSizeUndefined();
setSizeUndefined();

// The composition root MUST be set
setCompositionRoot(panel);
}
}

```

Take note of the sizing when trying to make a custom component that shrinks to fit the contained components. You have to set the size as undefined at all levels; the sizing of the composite component and the composition root are separate.

You can use the component as follows:

```
MyComposite mycomposite = new MyComposite("Hello");
```

The rendered component is shown in Figure 6.52, "A custom composite component".

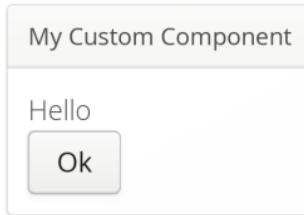


Figure 6.52. A custom composite component

You can also inherit any other components, such as layouts, to attain similar composition. Even further, you can create entirely new low-level components, by integrating pure client-side components or by extending the client-side functionality of built-in components. Development of new components is covered in Chapter 16, *Integrating with the Server-Side*.

6.28. Composite Fields with CustomField

The **CustomField** is a way to create composite components as with **CustomComponent**, except that it implements the Field interface and inherits **AbstractField**, described in Section 6.4, "Field Components". A field allows editing a property value in the data model, and can be bound to data with **Binder**, as described in Section 10.3, "Binding Data to Forms".

A composite field class must implement `initContent()` method. It should return the content composite of the field.

Methods overriding `setInternalValue()` should call the superclass method.

6.28.1. Basic Use

Let us consider a simple custom switch button component that allows you to click a button to switch it "on" and "off", as illustrated in Figure 6.53, "A custom switch button field".



Figure 6.53. A custom switch button field

The field has **Boolean** value type, which the `getType()` returns. In `initContent()`, we initialize the button and the layout. Notice how we handle user interaction with the button to change the field value. We customize the `setValue()` method to reflect the state back to the user.

```
public class BooleanField extends CustomField<Boolean> {
    private final Button button = new Button("Off");
    private boolean value;

    @Override
    protected Component initContent() {
        button.addClickListener(event -> {
            setValue(!getValue());
            button.setCaption(getValue() ? "On" : "Off");
        });
    }

    VerticalLayout layout = new VerticalLayout();
    layout.addComponent(new Label("Click the button"));
    layout.addComponent(button);
}
```

```

        return layout;
    }

    @Override
    public Boolean getValue() {
        return value;
    }

    @Override
    protected void doSetValue(Boolean value) {
        this.value = value;
        button.setCaption(value ? "On" : "Off");
    }
}

```

We can now use the field in all the normal ways for a field:

```

// Create it
BooleanField field = new BooleanField();

// It's a field so we can set its value
field.setValue(new Boolean(true));

// ...and read the value
Label value = new Label(field.getValue()?
    "Initially on" : "Initially off");

// ...and handle value changes
field.addValueChangeListener(event ->
    value.setValue(field.getValue()?
        "It's now on" : "It's now off"));

```

6.29. Embedded Resources

You can embed images in Vaadin UIs with the **Image** component, Adobe Flash graphics with **Flash**, and other web content with **BrowserFrame**. There is also a generic **Embedded** component for embedding other object types. The embedded content is referenced as *resources*, as described in Section 5.5, “Images and Other Resources”.

The following example displays an image as a class resource loaded with the class loader:

```

Image image = new Image("Yes, logo:",
    new ClassResource("vaadin-logo.png"));
main.addComponent(image);

```

The caption can be given as null to disable it. An empty string displays an empty caption which takes a bit space. The caption is managed by the containing layout.

You can set an alternative text for an embedded resource with `setAlternateText()`, which can be shown if images are disabled in the browser for some reason. The text can be used for accessibility purposes, such as for text-to-speech generation.

6.29.1. Embedded Image

The **Image** component allows embedding an image resource in a Vaadin UI.

```
// Serve the image from the theme  
Resource res = new ThemeResource("img/myimage.png");
```

```
// Display the image without caption  
Image image = new Image(null, res);  
layout.addComponent(image);
```

The **Image** component has by default undefined size in both directions, so it will automatically fit the size of the embedded image. If you want scrolling with scroll bars, you can put the image inside a **Panel** that has a defined size to enable scrolling, as described in Section 7.6.1, "Scrolling the Panel Content". You can also put it inside some other component container and set the `overflow: auto` CSS property for the container element in a theme to enable automatic scrollbars.

Generating and Reloading Images

You can also generate the image content dynamically using a **StreamResource**, as described in Section 5.5.5, "Stream Resources", or with a **RequestHandler**.

If the image changes, the browser needs to reload it. Simply updating the stream resource is not enough. Because of how caching is handled in some browsers, you can cause a reload easiest by renaming the filename of the resource with a unique name, such as one including a timestamp. You should set cache time to zero with `setCacheTime()` for the resource object when you create it.

```
// Create the stream resource with some initial filename
StreamResource imageResource =
    new StreamResource(imageSource, "initial-filename.png");

// Instruct browser not to cache the image
imageResource.setCacheTime(0);

// Display the image
Image image = new Image(null, imageResource);

When refreshing, you also need to call markAsDirty() for the
Image object.

// This needs to be done, but is not sufficient
image.markAsDirty();

// Generate a filename with a timestamp
SimpleDateFormat df = new SimpleDateFormat("yyyyMMddHHmmssSSS");
String filename = "myfilename-" + df.format(new Date()) + ".png";

// Replace the filename in the resource
imageResource.setFilename(makeImageFilename());
```

6.29.2. BrowserFrame

The **BrowserFrame** allows embedding web content inside an HTML `<iframe>` element. You can refer to an external URL with **ExternalResource**.

As the **BrowserFrame** has undefined size by default, it is critical that you define a meaningful size for it, either fixed or relative.

```
BrowserFrame browser = new BrowserFrame("Browser",
    new ExternalResource("http://demo.vaadin.com/sampler/"));
browser.setWidth("600px");
browser.setHeight("400px");
layout.addComponent(browser);
```

Notice that web pages can prevent embedding them in an `<iframe>`.

6.29.3. Generic Embedded Objects

The generic **Embedded** component allows embedding all sorts of objects, such as SVG graphics, Java applets, and PDF documents, in addition to the images, and browser frames which you can embed with the specialized components.

Display an SVG image:

```
// A resource reference to some object
Resource res = new ThemeResource("img/reindeer.svg");

// Display the object
Embedded object = new Embedded("My SVG", res);
object.setMimeType("image/svg+xml"); // Unnecessary
layout.addComponent(object);
```

The MIME type of the objects is usually detected automatically from the filename extension with the **FileTypeResolver** utility in Framework. If not, you can set it explicitly with `setMimeType()`, as was done in the example above (where it was actually unnecessary).

Some embeddable object types may require special support in the browser. You should make sure that there is a proper fallback mechanism if the browser does not support the embedded type.

Chapter 7

Managing Layout

7.1. Overview	216
7.2. UI, Window, and Panel Content	218
7.3. VerticalLayout and HorizontalLayout	219
7.4. GridLayout	227
7.5. FormLayout	231
7.6. Panel	233
7.7. Sub-Windows	236
7.8. HorizontalSplitPanel and VerticalSplitPanel	240
7.9. TabSheet	243
7.10. Accordion	247
7.11. AbsoluteLayout	249
7.12. CssLayout	252
7.13. Layout Formatting	256
7.14. Custom Layouts	264

Layout management of Vaadin is a direct successor of the web-based concept for separation of content and appearance and of the Java AWT solution for binding the layout and user interface components into objects in programs. Vaadin layout components allow you to position your UI components on the screen in a hierarchical fashion, much like in conventional Java UI toolkits such as AWT, Swing, or SWT. In addition, you can approach the layout from the direction of the web with the **CustomLayout** component, which you can use to write your layout as a template in HTML that provides locations of any contained components. The **AbsoluteLayout** allows the old-style pixel-position based layouting, but it also supports

percentual values, which makes it usable for scalable layouts. It is also useful as an area on which the user can position items with drag and drop.

7.1. Overview

The user interface components in Vaadin can roughly be divided in two groups: components that the user can interact with and layout components for placing the other components to specific places in the user interface. The layout components are identical in their purpose to layout managers in regular desktop frameworks for Java. You can use plain Java to accomplish sophisticated component layouts.

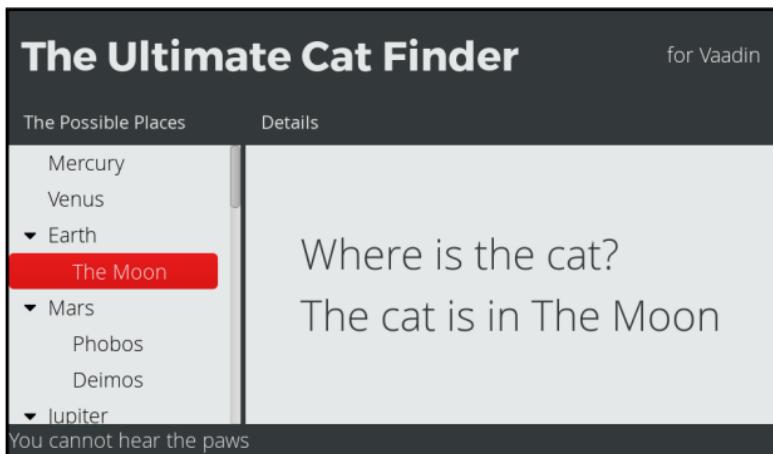


Figure 7.1. Layout example

You start by creating a content layout for the UI and then add other layout components hierarchically, and finally the interaction components as the leaves of the component tree.

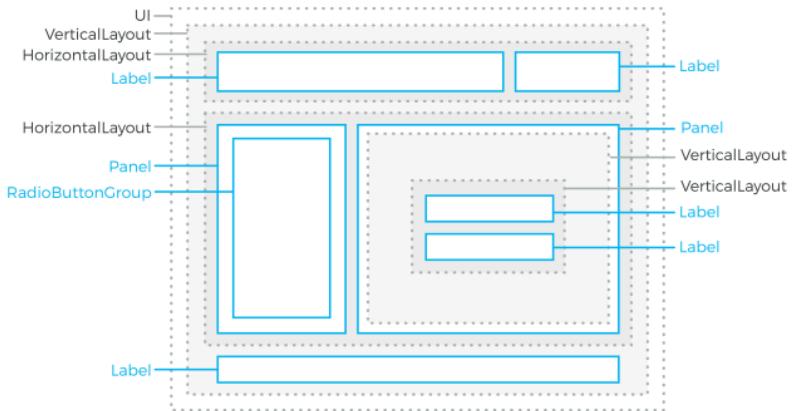


Figure 7.2. Layout schematic

Let us look at building a bit simplified version of the layout in Figure 7.1, "Layout example":

// Set the root layout for the UI

```
VerticalLayout content = new VerticalLayout();
setContent(content);
```

```
HorizontalLayout titleBar = new HorizontalLayout();
titleBar.setWidth("100%");
root.addComponent(titleBar);
```

```
Label title = new Label("The Ultimate Cat Finder");
titleBar.addComponent(title);
titleBar.setExpandRatio(title, 1.0f); // Expand
```

```
Label titleComment = new Label("for Vaadin");
titleComment.setSizeUndefined(); // Take minimum space
titleBar.addComponent(titleComment);
```

... build rest of the layout ...

Or in the declarative format (roughly):

```
<vaadin-vertical-layout>
  <vaadin-label>The Ultimate Cat Finder</vaadin-label>

  <vaadin-horizontal-layout>
    <vaadin-radio-button-group caption="Possible Places"/>
    <vaadin-panel/>
  ...
</vaadin-horizontal-layout>
</vaadin-vertical-layout>
```

You will usually need to tune the layout components a bit by setting sizes, expansion ratios, alignments, spacings, and so on. The general settings are described in Section 7.13, “Layout Formatting”.

Layouts are coupled with themes that specify various layout features, such as backgrounds, borders, text alignment, and so on. Definition and use of themes is described in Chapter 9, *Themes*.

You can see a finished version of the above example in Figure 7.1, “Layout example”.

7.2. UI, Window, and Panel Content

The **UI**, **Window**, and **Panel** all have a single content component, which you need to set with `setContent()`. The content is usually a layout component, although any component is allowed.

```
Panel panel = new Panel("This is a Panel");
VerticalLayout panelContent = new VerticalLayout();
panelContent.addComponent(new Label("Hello!"));
panel.setContent(panelContent);

// Set the panel as the content of the UI
setContent(panel);
```

The size of the content is the default size of the particular layout component, for example, a **VerticalLayout** has 100% width and undefined height by default (this coincides with the defaults for **Panel** and **Label**). If such a layout with undefined height grows higher than the browser window, it will flow out of the view and scrollbars will appear. In many applications, you want to use the full area of the browser view. Setting the components contained inside the content layout to full size is not enough, and would actually lead to an invalid state if the height of the content layout is undefined.

```
// First set the root content for the UI
VerticalLayout content = new VerticalLayout();
setContent(content);

// Set the content size to full width and height
content.setSizeFull();
```

```
// Add a title area on top of the screen. This takes  
// just the vertical space it needs.  
content.addComponent(new Label("My Application"));
```

```
// Add a menu-view area that takes rest of vertical space  
HorizontalLayout menuview = new HorizontalLayout();  
menuview.setSizeFull();  
content.addComponent(menuview);
```

See Section 7.13.1, “Layout Size” for more information about setting layout sizes.

7.3. VerticalLayout and HorizontalLayout

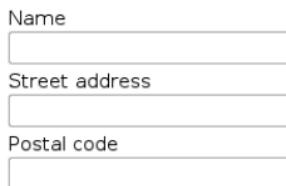
VerticalLayout and **HorizontalLayout** are ordered layouts for laying components out either vertically or horizontally, respectively. They both extend from **AbstractOrderedLayout**, together with the **FormLayout**. These are the two most important layout components in Vaadin, and typically you have a **VerticalLayout** as the root content of a UI.

VerticalLayout has 100% default width and undefined height, so it fills the containing layout (or UI) horizontally, and fits its content vertically. **HorizontalLayout** has undefined size in both dimensions.

Typical use of the layouts goes as follows:

```
VerticalLayout vertical = new VerticalLayout();  
vertical.addComponent(new TextField("Name"));  
vertical.addComponent(new TextField("Street address"));  
vertical.addComponent(new TextField("Postal code"));  
layout.addComponent(vertical);
```

In **VerticalLayout**, the captions of child components are placed above each component, so the layout would look as follows:



HorizontalLayout gives the following layout:

Name	Street address	Postal code
<input type="text"/>	<input type="text"/>	<input type="text"/>

7.3.1. Declarative Format

Ordered layouts have the following declarative elements:

Component	Element Name
VerticalLayout	v-vertical-layout
HorizontalLayout	v-horizontal-layout
FormLayout	v-form-layout

The have the following declarative attributes:

Table 7.1. Properties and Declarative Attributes

Property	Declarative Attribute
<i>componentAlignment</i>	Alignment of a child component is specified in the child with: :left (default), :center, :right, :top (default), :middle, :bottom
<i>spacing</i>	<i>spacing[=<boolean>]</i>
<i>margin</i>	<i>margin[=<boolean>]</i>
<i>expandRatio</i>	Expand ratio of a child component is specified in the child with: :expand[=<integer>] or .expand (implies ratio 1)

They can also have any attributes applicable to super classes.

For example:

```
<!-- Use margin and spacing -->
<vaadin-vertical-layout size-full margin spacing>
  <vaadin-label><b>Hello!</b> - How are you?</vaadin-label>

<!-- Use expand ratio -->
<vaadin-horizontal-layout size-full :expand>
  ...
<!-- Use expand ratio -->
<vaadin-grid _id="mygrid" caption="My Grid"
  size-full :expand/>
```

```
</vaadin-horizontal-layout>  
  
<vaadin-horizontal-layout width="full">  
  ...  
  
  <!-- Use alignment -->  
  <vaadin-button .right><b>OK</b></vaadin-button>  
</vaadin-horizontal-layout>  
</vaadin-vertical-layout>
```

7.3.2. Spacing in Ordered Layouts

The ordered layouts can have spacing between the horizontal or vertical cells. Spacing is enabled by default, and can be disabled with setSpacing(false) or declaratively with the spacing="false" attribute.

The spacing has a default height or width, which can be customized in CSS. You need to set the height or width for spacing elements with v-spacing style. You also need to specify an enclosing rule element in a CSS selector, such as v-vertical-layout for a **VerticalLayout** or v-horizontallayout for a **HorizontalLayout**. You can also use v-vertical and v-horizontal for all vertically or horizontally ordered layouts, such as **FormLayout**.

For example, the following sets the amount of spacing for all **VerticalLayouts** (as well as **FormLayouts**) in the UI:

```
.v-vertical > .v-spacing {  
  height: 30px;  
}
```

Or for **HorizontalLayout**:

```
.v-horizontal > .v-spacing {  
  width: 50px;  
}
```

7.3.3. Sizing Contained Components

The components contained within an ordered layout can be laid out in a number of different ways depending on how you specify their height or width in the primary direction of the layout component.

Undefined width layout:	<table border="1"> <tr> <td>Small</td><td>Medium-sized</td><td>Quite a big component</td></tr> </table>	Small	Medium-sized	Quite a big component
Small	Medium-sized	Quite a big component		
Fixed width layout:	<table border="1"> <tr> <td>Small</td><td style="background-color: yellow;">Medium-sized</td><td>Quite a big component</td></tr> </table>	Small	Medium-sized	Quite a big component
Small	Medium-sized	Quite a big component		
Full-sized components:	<table border="1"> <tr> <td>Small</td><td>Medium-sized</td><td>Quite a big component</td></tr> </table>	Small	Medium-sized	Quite a big component
Small	Medium-sized	Quite a big component		
Expanding component:	<table border="1"> <tr> <td>Small</td><td>Medium-sized</td><td>Expanding component</td></tr> </table>	Small	Medium-sized	Expanding component
Small	Medium-sized	Expanding component		

Figure 7.3. Component widths in HorizontalLayout

Figure 7.3, "Component widths in **HorizontalLayout**" gives a summary of the sizing options for a **HorizontalLayout** with spacing disabled. The figure is broken down in the following subsections.

Layout with Undefined Size

If a **VerticalLayout** has undefined height or **HorizontalLayout** undefined width, the layout will shrink to fit the contained components so that there is no extra space between them (apart from optional spacing).

```
HorizontalLayout fittingLayout = new HorizontalLayout();
fittingLayout.setWidth(Sizeable.SIZE_UNDEFINED, 0); // Default
fittingLayout.setSpacing(false); // Compact layout
fittingLayout.addComponent(new Button("Small"));
fittingLayout.addComponent(new Button("Medium-sized"));
fittingLayout.addComponent(new Button("Quite a big component"));
parentLayout.addComponent(fittingLayout);
```

The both layouts actually have undefined height by default and **HorizontalLayout** has also undefined width, while **VerticalLayout** has 100% relative width.

If such a vertical layout with undefined height continues below the bottom of a window (a **Window** object), the window will pop up a vertical scroll bar on the right side of the window area. This way, you get a "web page". The same applies to **Panel**.



A layout that contains components with percentual size must have a defined size!

If a layout has undefined size and a contained component has, say, 100% size, the component would fill the space given by the layout, while the layout would shrink to fit the space taken by the component, which would be a paradox. This requirement holds for height and width separately. The debug window allows detecting such invalid cases: see Section 11.3.5, “Inspecting Component Hierarchy”.

An exception to the above rule is a case where you have a layout with undefined size that contains a component with a fixed or undefined size together with one or more components with relative size. In this case, the contained component with fixed (or undefined) size in a sense defines the size of the containing layout, removing the paradox. That size is then used for the relatively sized components.

The technique can be used to define the width of a **VerticalLayout** or the height of a **HorizontalLayout**.

```
// Vertical layout would normally have 100% width
VerticalLayout vertical = new VerticalLayout();
```

```
// Shrink to fit the width of contained components
vertical.setWidth(Sizeable.SIZE_UNDEFINED, 0);
```

```
// Label has normally 100% width, but we set it as
// undefined so that it will take only the needed space
Label label =
    new Label("\u2190 The VerticalLayout shrinks to fit "+
        "the width of this Label \u2192");
label.setWidth(Sizeable.SIZE_UNDEFINED, 0);
vertical.addComponent(label);
```

```
// Button has undefined width by default
Button butt = new Button("\u2190 This Button takes 100% "+
    "of the width \u2192");
butt.setWidth("100%");
vertical.addComponent(butt);
```

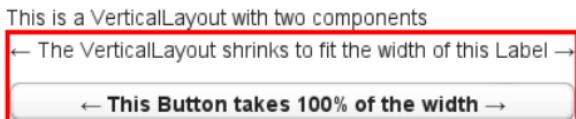


Figure 7.4. Defining the Size with a Component

Layout with Defined Size

If you set a **HorizontalLayout** to a defined size horizontally or a **VerticalLayout** vertically, and there is space left over from the contained components, the extra space is distributed equally between the component cells. The components are aligned within these cells according to their alignment setting, top left by default, as in the example below.

```
fixedLayout.setWidth("400px");
```

Expanding Components

Often, you want to have one component that takes all the available space left over from other components. You need to set its size as 100% and set it as *expanding* with `setExpandRatio()`. The second parameter for the method is an expansion ratio, which is relevant if there are more than one expanding component, but its value is irrelevant for a single expanding component.

```
HorizontalLayout layout = new HorizontalLayout();
layout.setWidth("400px");
```

```
// These buttons take the minimum size.
layout.addComponent(new Button("Small"));
layout.addComponent(new Button("Medium-sized"));
```

```
// This button will expand.
Button expandButton = new Button("Expanding component");
```

```
// Use 100% of the expansion cell's width.
expandButton.setWidth("100%");
```

```
// The component must be added to layout
// before setting the ratio
layout.addComponent(expandButton);
```

```
// Set the component's cell to expand.
layout.setExpandRatio(expandButton, 1.0f);
```

```
parentLayout.addComponent(layout);
```

In the declarative format, you need to specify the `:expand` attribute in the child components. The attribute defaults to expand ratio 1.

Notice that you can not call `setExpandRatio()` before you have added the component to the layout, because it can not operate on a component that it doesn't yet have.

Expand Ratios

If you specify an expand ratio for multiple components, they will all try to use the available space according to the ratio.

```
HorizontalLayout layout = new HorizontalLayout();
layout.setWidth("400px");
```

```
// Create three equally expanding components.
String[] captions = { "Small", "Medium-sized",
                     "Quite a big component" };
for (int i = 1; i <= 3; i++) {
    Button button = new Button(captions[i-1]);
    button.setWidth("100%");
    layout.addComponent(button);
}

// Have uniform 1:1:1 expand ratio.
layout.setExpandRatio(button, 1.0f);
}
```

As the example used the same ratio for all components, the ones with more content may have the content cut. Below, we use differing ratios:

```
// Expand ratios for the components are 1:2:3.
layout.setExpandRatio(button, i * 1.0f);
```

If the size of the expanding components is defined as a percentage (typically "100%"), the ratio is calculated from the *overall* space available for the relatively sized components. For example, if you have a 100 pixels wide layout with two cells with 1.0 and 4.0 respective expansion ratios, and both the components in the layout are set as `setWidth("100%)`, the cells will have respective widths of 20 and 80 pixels, regardless of the minimum size of the components.

However, if the size of the contained components is undefined or fixed, the expansion ratio is of the excess available space. In this case, it is the excess space that expands, not the components.

```
for (int i = 1; i <= 3; i++) {  
    // Button with undefined size.  
    Button button = new Button(captions[i - 1]);  
  
    layout4.addComponent(button);  
  
    // Expand ratios are 1:2:3.  
    layout4.setExpandRatio(button, i * 1.0f);  
}
```

It is not meaningful to combine expanding components with percentually defined size and components with fixed or undefined size. Such combination can lead to a very unexpected size for the percentually sized components.

Percentual Sizing

A percentual size of a component defines the size of the component *within its cell*. Usually, you use "100%", but a smaller percentage or a fixed size (smaller than the cell size) will leave an empty space in the cell and align the component within the cell according to its alignment setting, top left by default.

```
HorizontalLayout layout50 = new HorizontalLayout();  
layout50.setWidth("400px");  
  
String[] captions1 = { "Small 50%", "Medium 50%",  
                      "Quite a big 50%" };  
for (int i = 1; i <= 3; i++) {  
    Button button = new Button(captions1[i-1]);  
    button.setWidth("50%");  
    layout50.addComponent(button);  
  
    // Expand ratios for the components are 1:2:3.  
    layout50.setExpandRatio(button, i * 1.0f);  
}  
parentLayout.addComponent(layout50);
```

7.4. GridLayout

GridLayout container lays components out on a grid consisting of rows and columns. The columns and rows of the grid serve as coordinates that are used for laying out components on the grid. Each component can use multiple cells from the grid, defined as an area (x1,y1,x2,y2), although they typically take up only a single grid cell.

The grid layout maintains a cursor for adding components in left-to-right, top-to-bottom order. If the cursor goes past the bottom-right corner, it will automatically extend the grid downwards by adding a new row.

The following example demonstrates the use of **GridLayout**. The addComponent() method takes the component to be added and optional coordinates. The coordinates can be given for a single cell or for an area in x,y (column,row) order. The coordinate values have a base value of 0. If the coordinates are not given, the cursor will be used.

```
// Create a 4 by 4 grid layout.  
GridLayout grid = new GridLayout(4, 4);  
grid.setStyleName("example-gridlayout");  
  
// Fill out the first row using the cursor.  
grid.addComponent(new Button("R/C 1"));  
for (int i = 0; i < 3; i++) {  
    grid.addComponent(new Button("Col " +  
        (grid.getCursorX() + 1)));  
}  
  
// Fill out the first column using coordinates.  
for (int i = 1; i < 4; i++) {  
    grid.addComponent(new Button("Row " + i), 0, i);  
}  
  
// Add some components of various shapes.  
grid.addComponent(new Button("3x1 button"), 1, 1, 3, 1);  
grid.addComponent(new Label("1x2 cell"), 1, 2, 1, 3);  
InlineDateField date =  
    new InlineDateField("A 2x2 date field");  
date.setResolution(DateField.RESOLUTION_DAY);  
grid.addComponent(date, 2, 2, 3, 3);
```

The resulting layout is shown in Figure 7.5, “The **GridLayout** component”. The borders have been made visible to illustrate the layout cells.

R/C 1	Col 2	Col 3	Col 4
Row 2	3x1 button		
Row 3	1x2 cell	A 2x2 date field « < huhtikuuta 2015 > » ma ti ke to pe la su 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 29 30 1 2 3 4 5 6 7 8 9 10	
Row 4			

Figure 7.5. The GridLayout component

A component to be placed on the grid must not overlap with existing components. A conflict causes throwing a **GridLayout.OverlapsException**.

7.4.1. Sizing Grid Cells

You can define the size of both a grid layout and its components in either fixed or percentual units, or leave the size undefined altogether, as described in Section 6.3.9, “Sizing Components”. Section 7.13.1, “Layout Size” gives an introduction to sizing of layouts.

The size of the **GridLayout** component is undefined by default, so it will shrink to fit the size of the components placed inside it. In most cases, especially if you set a defined size for the layout but do not set the contained components to full size, there will be some unused space. The position of the non-full components within the grid cells will be determined by their *alignment*. See Section 7.13.3, “Layout Cell Alignment” for details on how to align the components inside the cells.

The components contained within a **GridLayout** layout can be laid out in a number of different ways depending on how you specify their height or width. The layout options are similar to **HorizontalLayout** and **VerticalLayout**, as described in Section 7.3, "VerticalLayout and HorizontalLayout".



A layout that contains components with percentual size must have a defined size!

If a layout has undefined size and a contained component has, say, 100% size, the component would fill the space given by the layout, while the layout would shrink to fit the space taken by the component, which is a paradox. This requirement holds for height and width separately. The debug mode allows detecting such invalid cases; see Section 11.3.1, "Enabling the Debug Mode".

Expanding Rows and Columns

Often, you want to have one or more rows or columns that take all the available space left over from non-expanding rows or columns. You need to set the rows or columns as *expanding* with `setRowExpandRatio()` and `setColumnExpandRatio()`. The first parameter for these methods is the index of the row or column to set as expanding. The second parameter for the methods is an expansion ratio, which is relevant if there are more than one expanding row or column, but its value is irrelevant if there is only one. With multiple expanding rows or columns, the ratio parameter sets the relative portion how much a specific row/column will take in relation with the other expanding rows/columns.

```
GridLayout grid = new GridLayout(3,2);
// Layout containing relatively sized components must have
// a defined size, here is fixed size.
grid.setWidth("600px");
grid.setHeight("200px");

// Add some content
String labels [] = {
    "Shrinking column<br/>Shrinking row".
```

```

"Expanding column (1:)<br/>Shrinking row".
"Expanding column (5:)<br/>Shrinking row".
"Shrinking column<br/>Expanding row".
"Expanding column (1:)<br/>Expanding row".
"Expanding column (5:)<br/>Expanding row"
};

for (int i=0; i<labels.length; i++) {
    Label label = new Label(labels[i], ContentMode.HTML);
    label.setWidth(null); // Set width as undefined
    grid.addComponent(label);
}

// Set different expansion ratios for the two columns
grid.setColumnExpandRatio(1, 1);
grid.setColumnExpandRatio(2, 5);

// Set the bottom row to expand
grid.setRowExpandRatio(1, 1);

// Align and size the labels.
for (int col=0; col<grid.getColumns(); col++) {
    for (int row=0; row<grid.getRows(); row++) {
        Component c = grid.getComponent(col, row);
        grid.setComponentAlignment(c, Alignment.TOP_CENTER);

        // Make the labels high to illustrate the empty
        // horizontal space.
        if (col != 0 || row != 0)
            c.setHeight("100%");
    }
}

```

Shrinking column Shrinking row	Expanding column (1:) Shrinking row	Expanding column (5:) Shrinking row
Shrinking column Expanding row	Expanding column (1:) Expanding row	Expanding column (5:) Expanding row

Figure 7.6. Expanding rows and columns in GridLayout

If the size of the contained components is undefined or fixed, the expansion ratio is of the excess space, as in Figure 7.6. “Expanding rows and columns in **GridLayout**” (excess horizontal space is shown in white). However, if the size of all the contained components in the expanding rows or columns is defined as a percentage, the ratio is calculated from the *overall* space available for the percentually sized components.

For example, if we had a 100 pixels wide grid layout with two columns with 1.0 and 4.0 respective expansion ratios, and all the components in the grid were set as `setWidth("100%")`, the columns would have respective widths of 20 and 80 pixels, regardless of the minimum size of their contained components.

7.4.2. CSS Style Rules

```
.v-gridlayout {}  
.v-gridlayout-margin {}
```

The root element of the **GridLayout** component has `v-gridlayout` style. The `v-gridlayout-margin` is a simple element inside it that allows setting a padding between the outer element and the cells.

For styling the individual grid cells, you should style the components inserted in the cells. Normally, if you want to have, for example, a different color for a certain cell, just make set the component inside it `setSizeFull()`, and add a style name for it. Sometimes, you may need to wrap a component inside a layout component just for styling the cell.

7.5. FormLayout

FormLayout lays the components and their captions out in two columns, with optional indicators for required fields and errors that can be shown for each field. The field captions can have an icon in addition to the text. **FormLayout** is an ordered layout and much like **VerticalLayout**. For description of margins, spacing, and other features in ordered layouts, see Section 7.3, “**VerticalLayout** and **HorizontalLayout**”.

The following example shows typical use of **FormLayout** in a form:

```
FormLayout form = new FormLayout();  
TextField tf1 = new TextField("Name");  
tf1.setIcon(FontAwesome.USER);  
tf1.setRequiredIndicatorVisible(true);  
form.addComponent(tf1);  
  
TextField tf2 = new TextField("Street address");
```

```

tf2.setIcon(FontAwesome.ROAD);
form.addComponent(tf2);

TextField tf3 = new TextField("Postal code");
tf3.setIcon(FontAwesome.ENVELOPE);
form.addComponent(tf3);
// normally comes from validation by Binder
tf3.setComponentError(new UserError("Doh!"));

```

The resulting layout will look as follows. The error message shows in a tooltip when you hover the mouse pointer over the error indicator.

The image shows a user interface with three input fields arranged vertically. The first field is labeled 'Name' with a person icon and a red asterisk indicating it is required. The second field is labeled 'Street address' with a location pin icon. The third field is labeled 'Postal code' with an envelope icon and a red exclamation mark indicating an error. A red-bordered box surrounds the 'Postal code' field. A tooltip window with a red border and white background appears over the error icon, containing the text 'Doh!'.

Figure 7.7. A FormLayout Layout for Forms

7.5.1. CSS Style Rules

```

.v-formlayout {}
.v-formlayout .v-caption {}

/* Columns in a field row. */
.v-formlayout-contentcell {} /* Field content. */
.v-formlayout-captioncell {} /* Field caption. */
.v-formlayout-errorcell {} /* Field error indicator. */

/* Overall style of field rows. */
.v-formlayout-row {}
.v-formlayout-firstrow {}
.v-formlayout-lastrow {}

/* Required field indicator. */
.v-formlayout .v-required-field-indicator {}
.v-formlayout-captioncell .v-caption
    .v-required-field-indicator {}

/* Error indicator. */

```

```
.v-formlayout-cell .v-errorindicator {}  
.v-formlayout-error-indicator .v-errorindicator {}
```

The top-level element of **FormLayout** has the v-formlayout style. The layout is tabular with three columns: the caption column, the error indicator column, and the field column. These can be styled with v-formlayout-captioncell, v-formlayout-errorcell, and v-formlayout-contentcell, respectively. While the error indicator is shown as a dedicated column, the indicator for required fields is currently shown as a part of the caption column.

For information on setting margins and spacing, see also Section 7.3.2, “Spacing in Ordered Layouts” and Section 7.13.5, “Layout Margins”.

7.6. Panel

Panel is a single-component container with a frame around the content. It has an optional caption and an icon which are handled by the panel itself, not its containing layout. The panel itself does not manage the caption of its contained component. You need to set the content with setContent().

Panel has 100% width and undefined height by default. This corresponds with the default sizing of **VerticalLayout**, which is perhaps most commonly used as the content of a **Panel**. If the width or height of a panel is undefined, the content must have a corresponding undefined or fixed size in the same direction to avoid a sizing paradox.

```
Panel panel = new Panel("Astronomer Panel");  
panel.addStyleName("mypanelexample");  
panel.setSizeUndefined(); // Shrink to fit content  
layout.addComponent(panel);
```

```
// Create the content  
FormLayout content = new FormLayout();  
content.addStyleName("mypanelcontent");  
content.addComponent(new TextField("Participant"));  
content.addComponent(new TextField("Organization"));  
content.setSizeUndefined(); // Shrink to fit  
content.setMargin(true);  
panel.setContent(content);
```

The resulting layout is shown in Figure 7.8, “A **Panel**”.

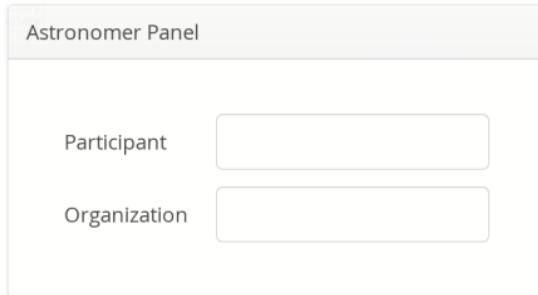


Figure 7.8. A Panel

7.6.1. Scrolling the Panel Content

Normally, if a panel has undefined size in a direction, as it has by default vertically, it will fit the size of the content and grow as the content grows. However, if it has a fixed or percentual size and its content becomes too big to fit in the content area, a scroll bar will appear for the particular direction. Scroll bars in a **Panel** are handled natively by the browser with the overflow: auto property in CSS.

In the following example, we have a 300 pixels wide and very high **Image** component as the panel content.

```
// Display an image stored in theme
Image image = new Image(null,
    new ThemeResource("img/Ripley_Scroll-300px.jpg"));

// To enable scrollbars, the size of the panel content
// must not be relative to the panel size
image.setSizeUndefined(); // Actually the default

// The panel will give it scrollbars.
Panel panel = new Panel("Scroll");
panel.setWidth("300px");
panel.setHeight("300px");
panel.setContent(image);

layout.addComponent(panel);
```

The result is shown in Figure 7.9, “Panel with Scroll Bars”. Notice that also the horizontal scrollbar has appeared even though

the panel has the same width as the content (300 pixels) - the 300px width for the panel includes the panel border and vertical scrollbar.

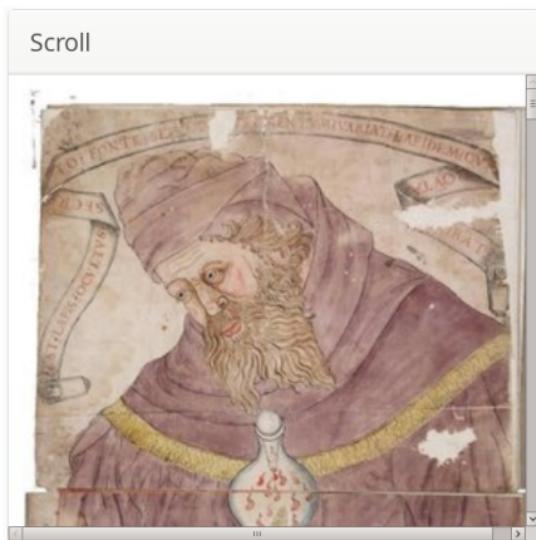


Figure 7.9. Panel with Scroll Bars

Programmatic Scrolling

Panel implements the Scrollable interface to allow programmatic scrolling. You can set the scroll position in pixels with `setScrollTop()` and `setScrollLeft()`. You can also get the scroll position set previously, but scrolling the panel in the browser does not update the scroll position to the server-side.

7.6.2. CSS Style Rules

```
.v-panel {}  
.v-panel-caption {}  
.v-panel-nocaption {}  
.v-panel-content {}  
.v-panel-deco {}
```

The entire panel has v-panel style. A panel consists of three parts: the caption, content, and bottom decorations (shadow). These can be styled with v-panel-caption, v-panel-content, and

v-panel-deco, respectively. If the panel has no caption, the caption element will have the style v-panel-nocaption.

The built-in borderless style in the Valo theme has no borders or border decorations for the **Panel**. You can use the `ValoTheme.PANEL_BORDERLESS` constant to add the style to a panel.

7.7. Sub-Windows

Sub-windows are floating panels within a native browser window. Unlike native browser windows, sub-windows are managed by the client-side runtime of Vaadin using HTML features. Vaadin allows opening, closing, resizing, maximizing and restoring sub-windows, as well as scrolling the window content.

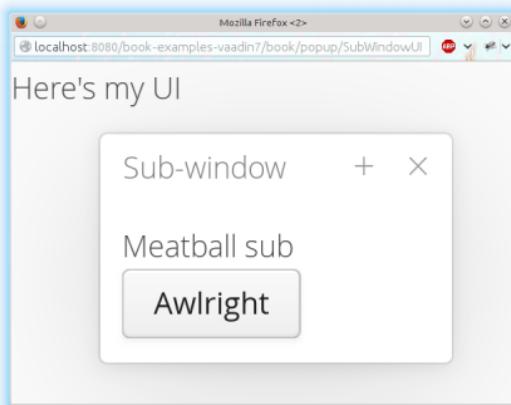


Figure 7.10. A Sub-Window

Sub-windows are typically used for *Dialog Windows* and *Multiple Document Interface* applications. Sub-windows are by default not modal; you can set them modal as described in Section 7.7.4, "Modal Sub-Windows".

7.7.1. Opening and Closing Sub-Windows

You can open a new sub-window by creating a new **Window** object and adding it to the UI with `addWindow()`, typically in

some event listener. A sub-window needs a content component, which is typically a layout.

In the following, we display a sub-window immediately when a UI opens:

```
public static class SubWindowUI extends UI {  
    @Override  
    protected void init(VaadinRequest request) {  
        //Some other UI content  
        setContent(new Label("Here's my UI"));  
  
        //Create a sub-window and set the content  
        Window subWindow = new Window("Sub-window");  
        VerticalLayout subContent = new VerticalLayout();  
        subWindow.setContent(subContent);  
  
        //Put some components in it  
        subContent.addComponent(new Label("Meatball sub"));  
        subContent.addComponent(new Button("Awright"));  
  
        //Center it in the browser window  
        subWindow.center();  
  
        //Open it in the UI  
        addWindow(subWindow);  
    }  
}
```

The result was shown in Figure 7.10, “A Sub-Window”. Sub-windows by default have undefined size in both dimensions, so they will shrink to fit the content.

The user can close a sub-window by clicking the close button in the upper-right corner of the window. The button is controlled by the *closable* property, so you can disable it with `setClosable(false)`. You can also use keyboard shortcuts for closing a sub-window. You can manage the shortcuts with the `addCloseShortcut()`, `removeCloseShortcut()`, `removeAllCloseShortcuts()`, `hasCloseShortcut()`, and `getCloseShortcuts()` methods.

You close a sub-window also programmatically by calling the `close()` for the sub-window, typically in a click listener for an **OK** or **Cancel** button. You can also call `removeWindow()` for the current **UI**.

Sub-Window Management

Usually, you would extend the **Window** class for your specific sub-window as follows:

```
// Define a sub-window by inheritance
class MySub extends Window {
    public MySub() {
        super("Subs on Sale"); // Set window caption
        center();

        // Disable the close button
        setClosable(false);

        setContent(new Button("Close me", event -> close()));
    }
}
```

You could open the window as follows:

```
// Some UI logic to open the sub-window
final Button open = new Button("Open Sub-Window");
open.addActionListener(event -> {
    MySub sub = new MySub();

    // Add it to the root component
    UI.getCurrent().addWindow(sub);
});
```

7.7.2. Window Positioning

When created, a sub-window will have an undefined default size and position. You can specify the size of a window with `setHeight()` and `setWidth()` methods. You can set the position of the window with `setPositionX()` and `setPositionY()` methods.

```
// Create a new sub-window
mywindow = new Window("My Dialog");

// Set window size.
mywindow.setHeight("200px");
mywindow.setWidth("400px");

// Set window position.
mywindow.setPositionX(200);
mywindow.setPositionY(50);

UI.getCurrent().addWindow(mywindow);
```

7.7.3. Scrolling Sub-Window Content

If a sub-window has a fixed or percentual size and its content becomes too big to fit in the content area, a scroll bar will appear for the particular direction. On the other hand, if the sub-window has undefined size in the direction, it will fit the size of the content and never get a scroll bar. Scroll bars in sub-windows are handled with regular HTML features, namely `overflow: auto` property in CSS.

As **Window** extends **Panel**, windows are also scrollable. Note that the interface defines *programmatic scrolling*, not scrolling by the user. Please see Section 7.6, "**Panel**".

7.7.4. Modal Sub-Windows

A modal window is a sub-window that prevents interaction with the other UI. Dialog windows, as illustrated in Figure 7.11, "Modal Sub-Window", are typical cases of modal windows. The advantage of modal windows is limiting the scope of user interaction to a sub-task, so changes in application state are more limited. The disadvantage of modal windows is that they can restrict workflow too much.

You can make a sub-window modal with `setModal(true)`.

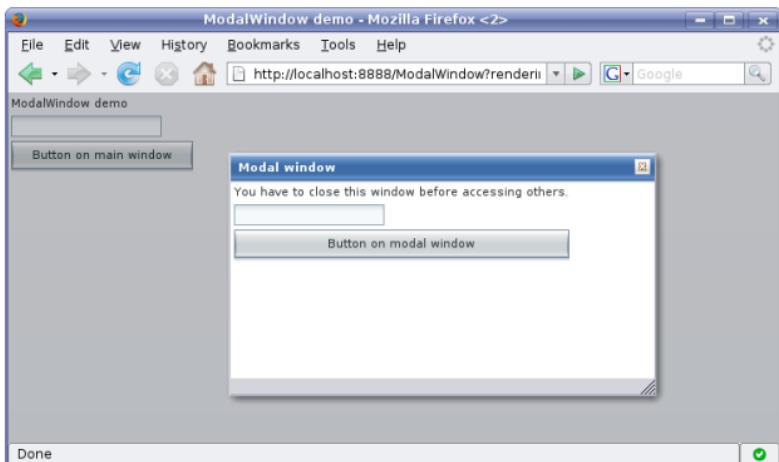


Figure 7.11. Modal Sub-Window

Depending on the theme, the parent window may be grayed when the modal window is open.



Security Warning

Modality of child windows is purely a client-side feature and can be circumvented with client-side attack code. You should not trust in the modality of child windows in security-critical situations such as login windows.

7.8. HorizontalSplitPanel and VerticalSplitPanel

HorizontalSplitPanel and **VerticalSplitPanel** are a two-component containers that divide the available space into two areas to accommodate the two components. **HorizontalSplitPanel** makes the split horizontally with a vertical splitter bar, and **VerticalSplitPanel** vertically with a horizontal splitter bar. The user can drag the bar to adjust its position.

You can set the two components with the `setFirstComponent()` and `setSecondComponent()` methods, or with the regular `addComponent()` method.

```
// Have a panel to put stuff in
Panel panel = new Panel("Split Panels Inside This Panel");
```

```
// Have a horizontal split panel as its content
HorizontalSplitPanel hsplit = new HorizontalSplitPanel();
panel.setContent(hsplit);
```

```
// Put a component in the left panel
Tree tree = new Tree("Menu", TreeExample.createTreeContent());
hsplit.setFirstComponent(tree);
```

```
// Put a vertical split panel in the right panel
VerticalSplitPanel vsplit = new VerticalSplitPanel();
hsplit.setSecondComponent(vsplit);
```

```
// Put other components in the right panel
vsplit.addComponent(new Label("Here's the upper panel"));
vsplit.addComponent(new Label("Here's the lower panel"));
```

The result is shown in Figure 7.12, "**HorizontalSplitPanel** and **VerticalSplitPanel**". Observe that the tree is cut horizontally

as it can not fit in the layout. If its height exceeds the height of the panel, a vertical scroll bar will appear automatically. If horizontal scroll bar is necessary, you could put the content in a **Panel**, which can have scroll bars in both directions.

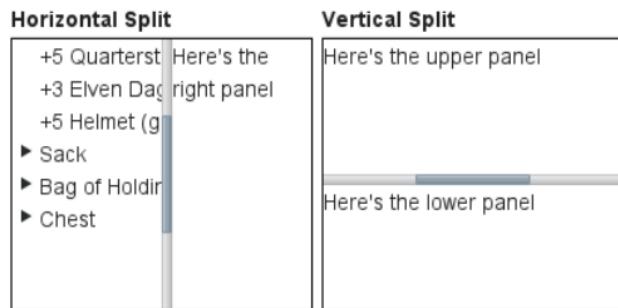


Figure 7.12. HorizontalSplitPanel and VerticalSplitPanel

You can set the split position with `setSplitPosition()`. It accepts any units defined in the **Sizeable** interface, with percentual size relative to the size of the component.

// Have a horizontal split panel

```
HorizontalSplitPanel hsplit = new HorizontalSplitPanel();
hsplit.setFirstComponent(new Label("75% wide panel"));
hsplit.setSecondComponent(new Label("25% wide panel"));
```

// Set the position of the splitter as percentage

```
hsplit.setSplitPosition(75, Sizeable.UNITS_PERCENTAGE);
```

Another version of the `setSplitPosition()` method allows leaving out the unit, using the same unit as previously. The method also has versions take a boolean parameter, `reverse`, which allows defining the size of the right or bottom panel instead of the left or top panel.

The split bar allows the user to adjust the split position by dragging the bar with mouse. To lock the split bar, use `setLocked(true)`. When locked, the move handle in the middle of the bar is disabled.

// Lock the splitter

```
hsplit.setLocked(true);
```

Setting the split position programmatically and locking the split bar is illustrated in Figure 7.13, "A Layout With Nested SplitPanels".

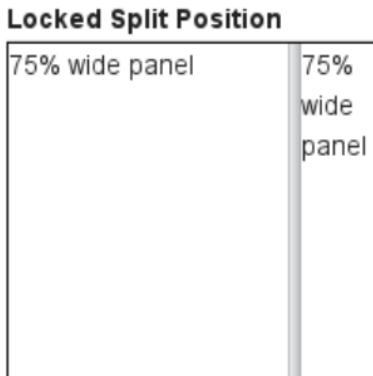


Figure 7.13. A Layout With Nested SplitPanels

Notice that the size of a split panel must not be undefined in the split direction.

7.8.1. CSS Style Rules

```
/* For a horizontal SplitPanel. */
.v-splitpanel-horizontal {}
.v-splitpanel-hsplitter {}
.v-splitpanel-hsplitter-locked {}

/* For a vertical SplitPanel. */
.v-splitpanel-vertical {}
.v-splitpanel-vsplitter {}
.v-splitpanel-vsplitter-locked {}

/* The two container panels. */
.v-splitpanel-first-container {} /* Top or left panel. */
.v-splitpanel-second-container {} /* Bottom or right panel. */
```

The entire split panel has the style v-splitpanel-horizontal or v-splitpanel-vertical, depending on the panel direction. The split bar or *splitter* between the two content panels has either the ...-splitter or ...-splitter-locked style, depending on whether its position is locked or not.

7.9. TabSheet

The **TabSheet** is a multicomponent container that allows switching between the components with "tabs". The tabs are organized as a tab bar at the top of the tab sheet. Clicking on a tab opens its contained component in the main display area of the layout. If there are more tabs than fit in the tab bar, navigation buttons will appear.

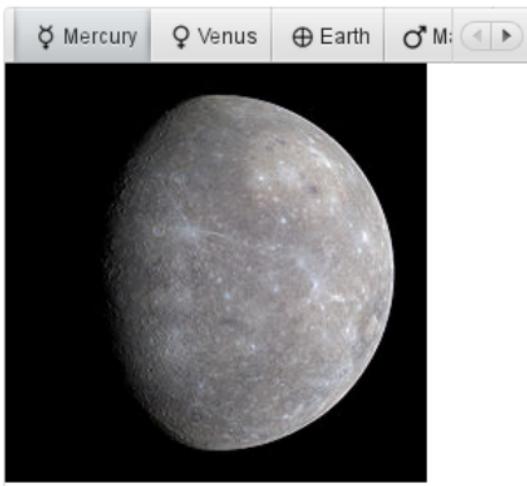


Figure 7.14. A Simple TabSheet Layout

7.9.1. Adding Tabs

You add new tabs to a tab sheet with the `addTab()` method. The simple version of the method takes as its parameter the root component of the tab. You can use the root component to retrieve its corresponding **Tab** object. Typically, you put a layout component as the root component.

You can also give the caption and the icon as parameters for the `addTab()` method. The following example demonstrates the creation of a simple tab sheet, where each tab shows a different **Label** component. The tabs have an icon, which are (in this example) loaded as Java class loader resources from the application.

```

TabSheet tabsheet = new TabSheet();
layout.addComponent(tabsheet);

// Create the first tab
VerticalLayout tab1 = new VerticalLayout();
tab1.addComponent(new Image(null,
    new ThemeResource("img/planets/Mercury.jpg")));
tabsheet.addTab(tab1, "Mercury",
    new ThemeResource("img/planets/Mercury_symbol.png"));

// This tab gets its caption from the component caption
VerticalLayout tab2 = new VerticalLayout();
tab2.addComponent(new Image(null,
    new ThemeResource("img/planets/Venus.jpg")));
tab2.setCaption("Venus");
tabsheet.addTab(tab2.setIcon(
    new ThemeResource("img/planets/Venus_symbol.png")));
...

```

7.9.2. Tab Objects

Each tab in a tab sheet is represented as a **Tab** object, which manages the tab caption, icon, and attributes such as hidden and visible. You can set the caption with `setCaption()` and the icon with `setIcon()`. If the component added with `addTab()` has a caption or icon, it is used as the default for the **Tab** object. However, changing the attributes of the root component later does not affect the tab, but you must make the setting through the **Tab** object. The `addTab()` returns the new **Tab** object, so you can easily set an attribute using the reference.

```

// Set an attribute using the returned reference
tabsheet.addTab(myTab).setCaption("My Tab");

```

Disabling and Hiding Tabs

A tab can be disabled by setting `setEnabled(false)` for the **Tab** object, thereby disallowing selecting it.

A tab can be made invisible by setting `setVisible(false)` for the **Tab** object. The `hideTabs()` method allows hiding the tab bar entirely. This can be useful in tabbed document interfaces (TDI) when there is only one tab.

7.9.3. Tab Change Events

Clicking on a tab selects it. This fires a **TabSheet.SelectedTabChangeEvent**, which you can handle by implementing

the **TabSheet.SelectedTabChangeListener** interface. You can access the tabsheet of the event with `getTabSheet()`, and find the new selected tab with `getSelectedTab()`.

You can programmatically select a tab with `setSelectedTab()`, which also fires the **SelectedTabChangeEvent** (beware of recursive events). Reselecting the currently selected tab does not fire the event.

Notice that when the first tab is added, it is selected and the change event is fired, so if you want to catch that, you need to add your listener before adding any tabs.

Creating Tab Content Dynamically

In the following example, we create the tabs as empty content layouts, and add the tab content dynamically when a tab is selected:

```
TabSheet tabsheet = new TabSheet();

// Create tab content dynamically when tab is selected
tabsheet.addSelectedTabChangeListener(
    new TabSheet.SelectedTabChangeListener() {
        public void selectedTabChange(SelectedTabChangeEvent event) {
            // Find the tabsheet
            TabSheet tabsheet = event.getTabSheet();

            // Find the tab (here we know it's a layout)
            Layout tab = (Layout) tabsheet.getSelectedTab();

            // Get the tab caption from the tab object
            String caption = tabsheet.getTab(tab).getCaption();

            // Fill the tab content
            tab.removeAllComponents();
            tab.addComponent(new Image(null,
                new ThemeResource("img/planets/"+caption+".jpg")));
        }
    });
}

// Have some tabs
String[] tabs = {"Mercury", "Venus", "Earth", "Mars"};
for (String caption: tabs)
    tabsheet.addTab(new VerticalLayout(), caption,
        new ThemeResource("img/planets/"+caption+"_symbol.png"));
```

7.9.4. Enabling and Handling Closing Tabs

You can enable a close button for individual tabs with the `closable` property in the **TabSheet.Tab** objects.

```
// Enable closing the tab  
tabsheet.getTab(tabComponent).setClosable(true);
```



Figure 7.15. TabSheet with Closable Tabs

Handling Tab Close Events

You can handle closing tabs by implementing a custom **TabSheet.CloseHandler**. The default implementation simply calls `removeTab()` for the tab to be closed, but you can prevent the close by not calling it. This allows, for example, opening a dialog window to confirm the close.

```
tabsheet.setCloseHandler(new CloseHandler() {  
    @Override  
    public void onTabClose(TabSheet tabsheet,  
                           Component tabContent) {  
        Tab tab = tabsheet.getTab(tabContent);  
        Notification.show("Closing " + tab.getCaption());  
  
        // We need to close it explicitly in the handler  
        tabsheet.removeTab(tab);  
    }  
});
```

7.9.5. CSS Style Rules

```
.v-tabsheets {}  
.v-tabsheets-tabs {}  
.v-tabsheets-content {}  
.v-tabsheets-deco {}  
.v-tabsheets-tabcontainer {}  
.v-tabsheets-tabsheetspanel {}  
.v-tabsheets-hidetabs {}  
  
.v-tabsheets-scroller {}  
.v-tabsheets-scrollerPrev {}  
.v-tabsheets-scrollerNext {}  
.v-tabsheets-scrollerPrev-disabled{}  
.v-tabsheets-scrollerNext-disabled{}
```

```
.v-tabsheet-tabitem {}
.v-tabsheet-tabitem-selected {}
.v-tabsheet-tabitemcell {}
.v-tabsheet-tabitemcell-first {}

.v-tabsheet-tabs td {}
.v-tabsheet-spacer td {}
```

The entire tabsheet has the v-tabsheet style. A tabsheet consists of three main parts: the tabs on the top, the main content pane, and decorations around the tabsheet.

The tabs area at the top can be styled with v-tabsheet-tabs, v-tabsheet-tabcontainer and v-tabsheet-tabitem*.

The style v-tabsheet-spacer is used for any empty space after the tabs. If the tabsheet has too little space to show all tabs, scroller buttons enable browsing the full tab list. These use the styles v-tabsheet-scroller*.

The content area where the tab contents are shown can be styled with v-tabsheet-content, and the surrounding decoration with v-tabsheet-deco.

7.10. Accordion

Accordion is a multicomponent container similar to **TabSheet**, except that the "tabs" are arranged vertically. Clicking on a tab opens its contained component in the space between the tab and the next one. You can use an **Accordion** identically to a **TabSheet**, which it actually inherits. See Section 7.9, "**TabSheet**" for more information.

The following example shows how you can create a simple accordion. As the **Accordion** is rather naked alone, we put it inside a Panel that acts as its caption and provides it a border.

```
// Create the accordion
Accordion accordion = new Accordion();

// Create the first tab, specify caption when adding
Layout tab1 = new VerticalLayout(); // Wrap in a layout
tab1.addComponent(new Image(null, // No component caption
    new ThemeResource("img/planets/Mercury.jpg")));
accordion.addTab(tab1, "Mercury");
new ThemeResource("img/planets/Mercury_symbol.png"));
```

```

// This tab gets its caption from the component caption
Component tab2 = new Image("Venus",
    new ThemeResource("img/planets/Venus.jpg"));
accordion.addTab(tab2.setIcon(
    new ThemeResource("img/planets/Venus_symbol.png"));

// And so forth the other tabs...
String[] tabs = {"Earth", "Mars", "Jupiter", "Saturn"};
for (String caption: tabs) {
    String basename = "img/planets/" + caption;
    VerticalLayout tab = new VerticalLayout();
    tab.addComponent(new Image(null,
        new ThemeResource(basename + ".jpg")));
    accordion.addTab(tab, caption,
        new ThemeResource(basename + "_symbol.png"));
}

```

Figure 7.16, “An Accordion” shows what the example would look like with the default theme.

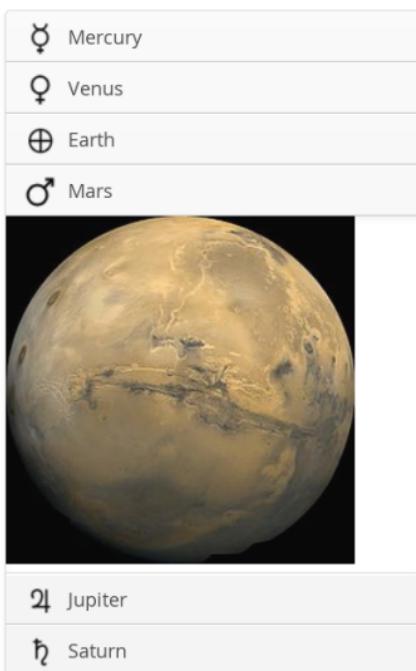


Figure 7.16. An Accordion

7.10.1. CSS Style Rules

```
.v-accordion {}  
.v-accordion-item.  
.v-accordion-item-open,  
.v-accordion-item-first {}  
.v-accordion-item-caption {}  
.v-caption {}  
.v-accordion-item-content {}  
.v-scrollable {}
```

The top-level element of **Accordion** has the v-accordion style. An **Accordion** consists of a sequence of item elements, each of which has a caption element (the tab) and a content area element.

The selected item (tab) has also the v-accordion-open style. The content area is not shown for the closed items.

7.11. AbsoluteLayout

AbsoluteLayout allows placing components in arbitrary positions in the layout area. The positions are specified in the addComponent() method with horizontal and vertical coordinates relative to an edge of the layout area. The positions can include a third depth dimension, the z-index, which specifies which components are displayed in front and which behind other components.

The positions are specified by a CSS absolute position string, using the left, right, top, bottom, and z-index properties known from CSS. In the following example, we have a 300 by 150 pixels large layout and position a text field 50 pixels from both the left and the top edge:

```
//A 400x250 pixels size layout  
AbsoluteLayout layout = new AbsoluteLayout();  
layout.setWidth("400px");  
layout.setHeight("250px");  
  
//A component with coordinates for its top-left corner  
TextField text = new TextField("Somewhere someplace");  
layout.addComponent(text, "left: 50px; top: 50px;");
```

The left and top specify the distance from the left and top edge, respectively. The right and bottom specify the distances from the right and bottom edge.

// At the top-left corner

```
Button button = new Button("left: 0px; top: 0px;");
layout.addComponent(button, "left: 0px; top: 0px;");
```

// At the bottom-right corner

```
Button buttCorner = new Button("right: 0px; bottom: 0px;");
layout.addComponent(buttCorner, "right: 0px; bottom: 0px;");
```

// Relative to the bottom-right corner

```
Button buttBrRelative = new Button("right: 50px; bottom: 50px;");
layout.addComponent(buttBrRelative, "right: 50px; bottom: 50px;");
```

// On the bottom, relative to the left side

```
Button buttBottom = new Button("left: 50px; bottom: 0px;");
layout.addComponent(buttBottom, "left: 50px; bottom: 0px;");
```

// On the right side, up from the bottom

```
Button buttRight = new Button("right: 0px; bottom: 100px;");
layout.addComponent(buttRight, "right: 0px; bottom: 100px;");
```

The result of the above code examples is shown in Figure 7.17, "Components Positioned Relative to Various Edges".

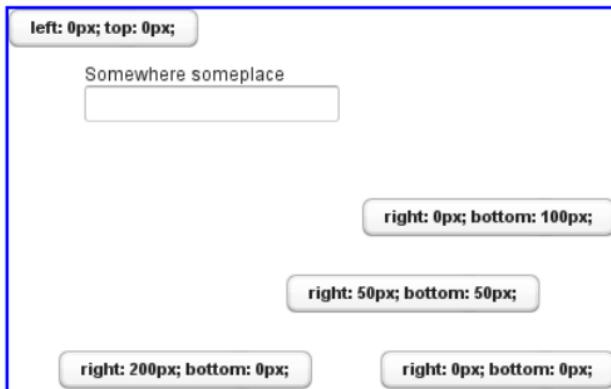


Figure 7.17. Components Positioned Relative to Various Edges

Drag and drop is very useful for moving the components contained in an **AbsoluteLayout**. Check out the example in Section 11.11.6, "Dropping on a Component".

7.11.1. Placing a Component in an Area

Earlier, we had components of undefined size and specified the positions of components by a single pair of coordinates. The other possibility is to specify an area and let the component fill the area by specifying a proportional size for the component, such as "100%". Normally, you use `setSizeFull()` to take the entire area given by the layout.

```
//Specify an area that a component should fill
Panel panel = new Panel("A Panel filling an area");
panel.setSizeFull(); //Fill the entire given area
layout.addComponent(panel, "left: 25px; right: 50px; "+
    "top: 100px; bottom: 50px;");
```

The result is shown in Figure 7.18, "Component Filling an Area Specified by Coordinates"

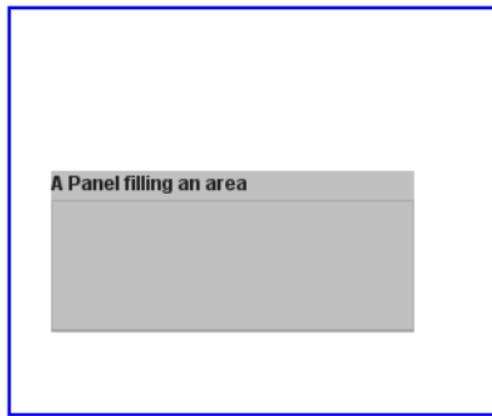


Figure 7.18. Component Filling an Area Specified by Coordinates

7.11.2. Proportional Coordinates

You can also use proportional coordinates to specify the placement of components:

```
//A panel that takes 30% to 90% horizontally and
//20% to 80% vertically
Panel panel = new Panel("A Panel");
panel.setSizeFull(); //Fill the specified area
```

```
layout.addComponent(panel, "left: 30%; right: 10%;" +  
    "top: 20%; bottom: 20%;");
```

The result is shown in Figure 7.19, “Specifying an Area by Proportional Coordinates”

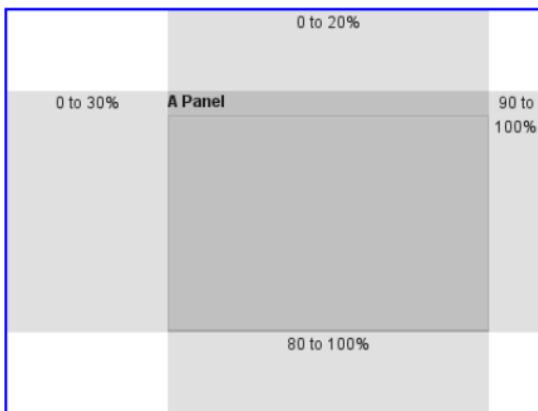


Figure 7.19. Specifying an Area by Proportional Coordinates

7.11.3. Styling with CSS

```
.v-absolutelayout {}  
.v-absolutelayout-wrapper {}
```

The **AbsoluteLayout** component has v-absolutelayout root style. Each component in the layout is contained within an element that has the v-absolutelayout-wrapper. The component captions are outside the wrapper elements, in a separate element with the usual v-caption style.

7.12. CssLayout

CssLayout allows strong control over styling of the components contained inside the layout. The components are contained in a simple DOM structure consisting of <div> elements. By default, the contained components are laid out horizontally and wrap naturally when they reach the width of the layout, but you can control this and most other behaviour with CSS. You can also inject custom CSS for each contained component. As **CssLayout** has a very simple DOM structure and no dynamic rendering logic, relying purely on the built-in render-

ing logic of the browsers, it is the fastest of the layout components.

The basic use of **CssLayout** is just like with any other layout component:

```
CssLayout layout = new CssLayout();
```

// Component with a layout-managed caption and icon

```
TextField tf = new TextField("A TextField");
tf.setIcon(new ThemeResource("icons/user.png"));
layout.addComponent(tf);
```

// Labels are 100% wide by default so must unset width

```
Label label = new Label("A Label");
label.setWidth(SizeMode.SIZE_UNDEFINED, 0);
layout.addComponent(label);
```

```
layout.addComponent(new Button("A Button"));
```

The result is shown in Figure 7.20, "Basic Use of **CssLayout**". Notice that the default spacing and alignment of the layout is quite crude and CSS styling is nearly always needed.



Figure 7.20. Basic Use of CssLayout

The display attribute of **CssLayout** is inline-block by default, so the components are laid out horizontally following another. **CssLayout** has 100% width by default. If the components reach the width of the layout, they are wrapped to the next "line" just as text would be. If you add a component with 100% width, it will take an entire line by wrapping before and after the component.

7.12.1. CSS Injection

Overriding the `getCss()` method allows injecting custom CSS for each component. The CSS returned by the method is inserted in the `style` attribute of the `<div>` element of the component, so it will override any style definitions made in CSS files.

```

CssLayout layout = new CssLayout() {
    @Override
    protected String getCss(Component c) {
        if (c instanceof Label) {
            //Color the boxes with random colors
            int rgb = (int) (Math.random()*1l<<24);
            return "background: #" + Integer.toHexString(rgb);
        }
        return null;
    }
};

layout.setWidth("400px"); //Causes to wrap the contents

//Add boxes of various sizes
for (int i=0; i<40; i++) {
    Label box = new Label("&nbsp;", ContentMode.HTML);
    box.addStyleName("flowbox");
    box.setWidth((float) Math.random()*50.
        Sizeable.UNITS_PIXELS);
    box.setHeight((float) Math.random()*50.
        Sizeable.UNITS_PIXELS);
    layout.addComponent(box);
}

```

The style name added to the components allows making common styling in a CSS file:

```

.v-label-flowbox {
    border: thin black solid;
}

```

Figure 7.21, "Use of getCss() and line wrap" shows the rendered result.

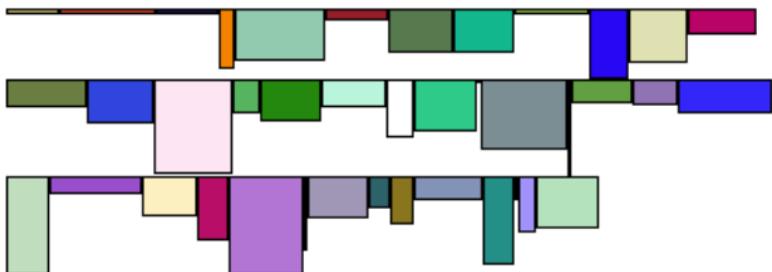


Figure 7.21. Use of getCss() and line wrap

7.12.2. Browser Compatibility

The strength of the **CssLayout** is also its weakness. Much of the logic behind the other layout components is there to give nice default behavior and to handle the differences in different browsers. With **CssLayout**, you may need to make use of the browser-specific style classes in the root element of the application to write browser specific CSS rules. Some features in the other layouts are not even solvable in pure CSS, at least in all browsers.

7.12.3. Styling with CSS

```
.v-csslayout {}
.v-csslayout-margin {}
.v-csslayout-container {}
```

The **CssLayout** component has v-csslayout root style. The margin element with v-csslayout-margin style is always enabled. The components are contained in an element with v-csslayout-container style.

For example, we could style the basic **CssLayout** example shown earlier as follows:

```
/* Have the caption right of the text box, bottom-aligned */
.csslayoutexample .mylayout .v-csslayout-container {
  direction: rtl;
  line-height: 24px;
  vertical-align: bottom;
}

/* Have some space before and after the caption */
.csslayoutexample .mylayout .v-csslayout-container .v-caption {
  padding-left: 3px;
  padding-right: 10px;
}
```

The example would now be rendered as shown in Figure 7.22, "Styling **CssLayout**".



Figure 7.22. Styling CssLayout

Captions and icons that are managed by the layout are contained in an element with `v-caption` style. These caption elements are contained flat at the same level as the actual component elements. This may cause problems with wrapping in inline-block mode, as wrapping can occur between the caption and its corresponding component element just as well as between components. Such use case is therefore not feasible.

7.13. Layout Formatting

While the formatting of layouts is mainly done with style sheets, just as with other components, style sheets are not ideal or even possible to use in some situations. For example, CSS does not allow defining the spacing of table cells, which is done with the `cellspacing` attribute in HTML.

Moreover, as many layout sizes are calculated dynamically in the Client-Side Engine of Vaadin, some CSS settings can fail altogether.

7.13.1. Layout Size

The size of a layout component can be specified with the `setWidth()` and `setHeight()` methods defined in the **Sizeable** interface, just like for any component. It can also be undefined, in which case the layout shrinks to fit the component(s) inside it. Section 6.3.9, “Sizing Components” gives details on the interface.

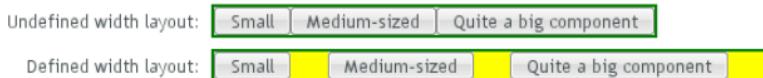


Figure 7.23. HorizontalLayout with Undefined vs Defined size

Many layout components take 100% width by default, while they have the height undefined.

The sizes of components inside a layout can also be defined as a percentage of the space available in the layout, for example with `setWidth("100%")`; or with the (most commonly used method) `setFullSize()` that sets 100% size in both directions. If you use a percentage in a **HorizontalLayout**, **VerticalLayout**,

or **GridLayout**, you will also have to set the component as *expanding*, as noted below.



Warning

A layout that contains components with percentage size must have a defined size!

If a layout has undefined size and a contained component has, say, 100% size, the component will try to fill the space given by the layout, while the layout will shrink to fit the space taken by the component, which is a paradox. This requirement holds for height and width separately. The debug mode allows detecting such invalid cases; see Section 11.3.5, “Inspecting Component Hierarchy”.

For example:

```
//This takes 100% width but has undefined height.  
VerticalLayout layout = new VerticalLayout();
```

```
//A button that takes all the space available in the layout.  
Button button = new Button("100%x100% button");  
button.setSizeFull();  
layout.addComponent(button);
```

```
//We must set the layout to a defined height vertically, in  
//this case 100% of its parent layout, which also must  
//not have undefined size.  
layout.setHeight("100%");
```

If you have a layout with undefined height, such as **VerticalLayout**, in a **UI**, **Window**, or **Panel**, and put enough content in it so that it extends outside the bottom of the view area, scrollbars will appear. If you want your application to use all the browser view, nothing more or less, you should use `setFullSize()` for the root layout.

```
//Create the UI content  
VerticalLayout content = new VerticalLayout();  
  
//Use entire view area  
content.setSizeFull();
```

```
setContent(content);
```

7.13.2. Expanding Components

If you set a **HorizontalLayout** to a defined size horizontally or a **VerticalLayout** vertically, and there is space left over from the contained components, the extra space is distributed equally between the component cells. The components are aligned within these cells, according to their alignment setting, top left by default, as in the example below.

Often, you don't want such empty space, but want one or more components to take all the leftover space. You need to set such a component to 100% size and use `setExpandRatio()`. If there is just one such expanding component in the layout, the ratio parameter is irrelevant.

If you set multiple components as expanding, the expand ratio dictates how large proportion of the available space (overall or excess depending on whether the components are sized as a percentage or not) each component takes. In the example below, the buttons have 1:2:3 ratio for the expansion.

GridLayout has corresponding method for both of its directions, `setRowExpandRatio()` and `setColumnExpandRatio()`.

Expansion is covered in detail in the documentation of the layout components that support it. See Section 7.3, “**VerticalLayout** and **HorizontalLayout**” and Section 7.4, “**GridLayout**” for details on components with relative sizes.

7.13.3. Layout Cell Alignment

When a component in a layout cell has size (width or height) smaller than the size of the cell, it will by default be aligned in the top-left corner of the cell. You can set the alignment with the `setComponentAlignment()` method. The method takes as its parameters the component contained in the cell to be formatted, and the horizontal and vertical alignment. The component must have been added to the layout before setting the alignment.

Figure 7.24, “Cell Alignments” illustrates the alignment of components within a **GridLayout**.

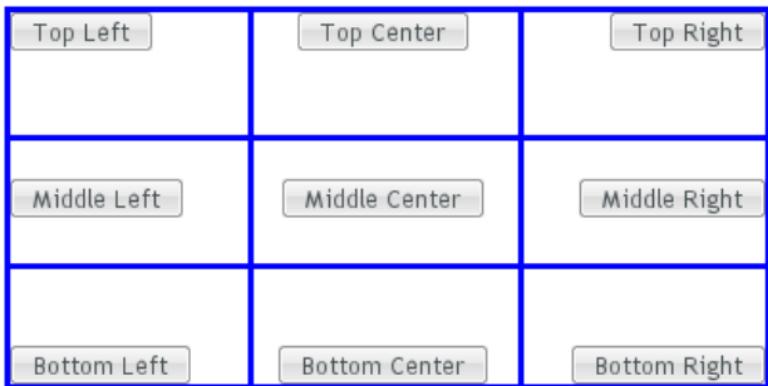


Figure 7.24. Cell Alignments

Note that a component with 100% relative size can not be aligned, as it will take the entire width or height of the cell, in which case alignment is meaningless. This should especially be noted with the **Label** component, which has 100% default width, and the text alignment *within* the component is separate, defined by the CSS text-align property.

The easiest way to set alignments is to use the constants defined in the **Alignment** class. Let us look how the buttons in the top row of the above **GridLayout** are aligned with constants:

```
//Create a grid layout
GridLayout grid = new GridLayout(3, 3);

grid.setWidth(400, Sizeable.UNITS_PIXELS);
grid.setHeight(200, Sizeable.UNITS_PIXELS);

Button topleft = new Button("Top Left");
grid.addComponent(topleft, 0, 0);
grid.setComponentAlignment(topleft, Alignment.TOP_LEFT);

Button topcenter = new Button("Top Center");
grid.addComponent(topcenter, 1, 0);
grid.setComponentAlignment(topcenter, Alignment.TOP_CENTER);

Button topright = new Button("Top Right");
grid.addComponent(topright, 2, 0);
```

```
grid.setComponentAlignment(topright, Alignment.TOP_RIGHT);
...
```

The following table lists all the **Alignment** constants by their respective locations:

Table 7.2. Alignment Constants

<i>TOP_LEFT</i>	<i>TOP_CENTER</i>	<i>TOP_RIGHT</i>
<i>MIDDLE_LEFT</i>	<i>MIDDLE_CENTER</i>	<i>MIDDLE_RIGHT</i>
<i>BOTTOM_LEFT</i>	<i>BOTTOM_CENTER</i>	<i>BOTTOM_RIGHT</i>

Another way to specify the alignments is to create an **Alignment** object and specify the horizontal and vertical alignment with separate constants. You can specify either of the directions, in which case the other alignment direction is not modified, or both with a bitmask operation between the two directions.

```
Button middleleft = new Button("Middle Left");
grid.addComponent(middleleft, 0, 1);
grid.setComponentAlignment(middleleft,
    new Alignment(Bits.ALIGNMENT_VERTICAL_CENTER |
        Bits.ALIGNMENT_LEFT));
```

```
Button middlecenter = new Button("Middle Center");
grid.addComponent(middlecenter, 1, 1);
grid.setComponentAlignment(middlecenter,
    new Alignment(Bits.ALIGNMENT_VERTICAL_CENTER |
        Bits.ALIGNMENT_HORIZONTAL_CENTER));
```

```
Button middleright = new Button("Middle Right");
grid.addComponent(middleright, 2, 1);
grid.setComponentAlignment(middleright,
    new Alignment(Bits.ALIGNMENT_VERTICAL_CENTER |
        Bits.ALIGNMENT_RIGHT));
```

Obviously, you may combine only one vertical bitmask with one horizontal bitmask, though you may leave either one out. The following table lists the available alignment bitmask constants:

Table 7.3. Alignment Bitmasks

Horizontal	<i>Bits.ALIGNMENT_LEFT</i>
------------	----------------------------

<code>Bits.ALIGNMENT_HORIZONTAL_CENTER</code>	<code>Bits.ALIGNMENT_RIGHT</code>
Vertical	<code>Bits.ALIGNMENT_TOP</code>
<code>Bits.ALIGNMENT_VERTICAL_CENTER</code>	<code>Bits.ALIGNMENT_BOTTOM</code>

You can determine the current alignment of a component with `getComponentAlignment()`, which returns an **Alignment** object. The class provides a number of getter methods for decoding the alignment, which you can also get as a bitmask value.

Size of Aligned Components

You can only align a component that is smaller than its containing cell in the direction of alignment. If a component has 100% width, as some components have by default, horizontal alignment does not have any effect. For example, **VerticalLayout** is 100% wide by default and can not therefore be horizontally aligned as such. The problem can sometimes be hard to notice, as the content inside a **VerticalLayout** is left-aligned by default.

You usually need to set either a fixed size, undefined size, or less than a 100% relative size for the component to be aligned - a size that is smaller than the containing layout has.

If you set the size as undefined and the component itself contains components, make sure that the contained components also have either undefined or fixed size.

7.13.4. Layout Cell Spacing

The **VerticalLayout**, **HorizontalLayout**, and **GridLayout** layouts offer a `setSpacing()` method to enable or disable spacing between the cells of the layout.

For example:

```
VerticalLayout layout = new VerticalLayout();
layout.setSpacing(false);
layout.addComponent(new Button("Component 1"));
```

```
layout.addComponent(new Button("Component 2"));  
layout.addComponent(new Button("Component 3"));
```

The effect of spacing in **VerticalLayout** and **HorizontalLayout** is illustrated in Figure 7.25, "Layout Spacings".

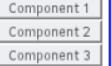
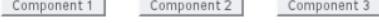
		No spacing:	Vertical spacing:
No spacing:			
Horizontal spacing:			

Figure 7.25. Layout Spacings

The exact amount of spacing is defined in CSS. If the default is not suitable, you can customize it in a custom theme.

In the Valo theme, you can specify the spacing with the `$v-layout-spacing-vertical` and `$v-layout-spacing-horizontal` parameters, as described in Section 9.7.2, "Common Settings". The spacing defaults to the `$v-unit-size` measure.

When adjusting spacing in other themes, you should note that it is implemented in a bit different ways in different layouts. In the ordered layouts, it is done with spacer elements, while in the **GridLayout** it has special handling. Please see the sections on the individual components for more details.

7.13.5. Layout Margins

Most layout components (with the exception of **VerticalLayout**) do not have any margin around them by default. The ordered layouts, as well as **GridLayout**, support enabling or disabling a margin with `setMargin()`. This enables CSS classes for each margin in the HTML element of the layout component.

In the Valo theme, the margin sizes default to `$v-unit-size`. You can customize them with `$v-layout-margin-top`, `right`, `bottom`, and `left`. See Section 9.7.2, "Common Settings" for a description of the parameters.

To customize the default margins in other themes, you can define each margin with the padding property in CSS. You may want to have a custom CSS class for the layout component to enable specific customization of the margins, as is done in the following with the `mymargins` class:

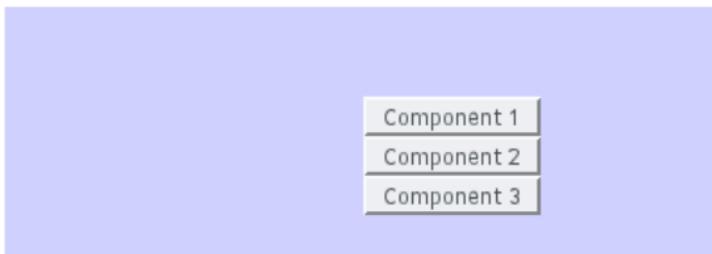
```
.mymargins.v-margin-left {padding-left: 10px;}  
.mymargins.v-margin-right {padding-right: 20px;}  
.mymargins.v-margin-top {padding-top: 40px;}  
.mymargins.v-margin-bottom {padding-bottom: 80px;}
```

You can enable only specific margins by passing a `MarginInfo` to the `setMargin()`. The margins are specified in clockwise order for top, right, bottom, and left margin. The following would enable the left and right margins:

```
layout.setMargin(new MarginInfo(false, true, false, true));
```

The resulting margins are shown in Figure 7.26, “Layout Margins” below. The two ways produce identical margins.

Regular layout margins:



Layout with a special margin element:

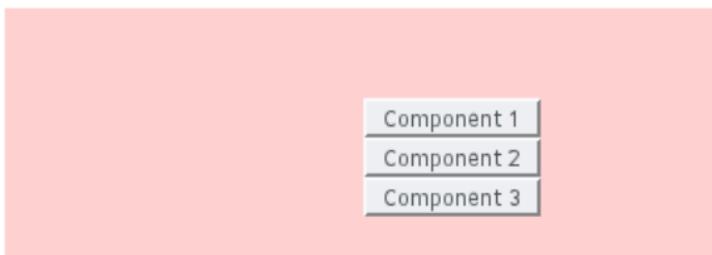


Figure 7.26. Layout Margins

7.14. Custom Layouts

While it is possible to create almost any typical layout with the standard layout components, it is sometimes best to separate the layout completely from code. With the **Custom-Layout** component, you can write your layout as a template in HTML that provides locations of any contained components. The layout template is included in a theme. This separation allows the layout to be designed separately from code, for example using WYSIWYG web designer tools such as Adobe Dreamweaver.

A template is a HTML file located under layouts folder under a theme folder under the /VAADIN/themes/ folder, for example, /VAADIN/themes/themename/layouts/mylayout.html. (Notice that the root path /VAADIN/themes/ for themes is fixed.) A template can also be provided dynamically from an **InputStream**, as explained below. A template includes `<div>` elements with a *location* attribute that defines the location identifier. All custom layout HTML-files must be saved using UTF-8 character encoding.

```
<table width="100%" height="100%">
<tr height="100%">
<td>
<table align="center">
<tr>
<td align="right">User &nbsp;name:</td>
<td><div location="username"></div></td>
</tr>
<tr>
<td align="right">Password:</td>
<td><div location="password"></div></td>
</tr>
</table>
</td>
</tr>
<tr>
<td align="right" colspan="2">
<div location="okbutton"></div>
</td>
</tr>
</table>
```

The client-side engine of Vaadin will replace contents of the location elements with the components. The components are bound to the location elements by the location identifier given to addComponent(), as shown in the example below.

```
Panel loginPanel = new Panel("Login");
CustomLayout content = new CustomLayout("layoutname");
content.setSizeUndefined();
loginPanel.setContent(content);
loginPanel.setSizeUndefined();

// No captions for fields is they are provided in the template
content.addComponent(new TextField(), "username");
content.addComponent(new TextField(), "password");
content.addComponent(new Button("Login"), "okbutton");
```

The resulting layout is shown below in Figure 7.27, "Example of a Custom Layout Component".



Figure 7.27. Example of a Custom Layout Component

You can use addComponent() also to replace an existing component in the location given in the second parameter.

In addition to a static template file, you can provide a template dynamically with the **CustomLayout** constructor that accepts an **InputStream** as the template source. For example:

```
new CustomLayout(new ByteArrayInputStream("<b>Template</b>".getBytes()));
```

or

```
new CustomLayout(new FileInputStream(file));
```


Chapter 8

Vaadin Designer

8.1. Overview	267
8.2. Installing in Eclipse	269
8.3. Installing in IntelliJ IDEA	270
8.4. Licensing	270
8.5. Getting Started	271
8.6. Designing	274
8.7. Theming and Styling	283
8.8. Wiring It Up	285
8.9. Templates	289
8.10. Limitations	292

This chapter describes how to create designs using the Vaadin Designer.

8.1. Overview

Vaadin Designer is a visual WYSIWYG tool for creating Vaadin UIs and views by using drag&drop and direct manipulation. With features such as live external preview and a strong connection between the clean declarative format and the Java code, it allows you to design and layout your UIs with speed and confidence.

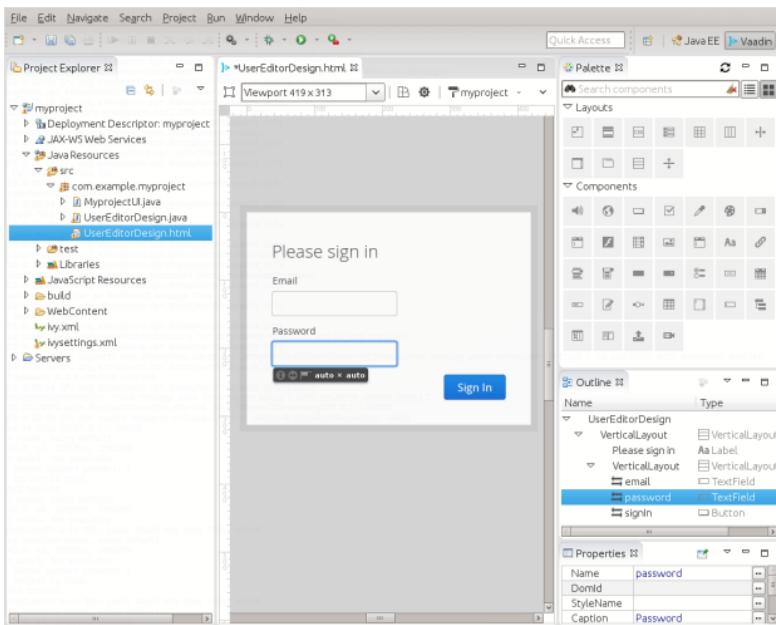


Figure 8.1. Vaadin Designer Views

Vaadin Designer is used to create two things:

1. A declarative file defining a UI (or part of a UI), also known as a *design* and
2. A *companion* Java file used to bind the UI components to Java logic.

The declarative format is a feature of the Vaadin Framework, and can be also used and edited without Vaadin Designer. See Section 5.3, “Designing UIs Declaratively” for a description of the format.

Vaadin Designer automatically creates and updates a Java file that exposes sub-components of the design as Java member variables, using variable names that you specify. This file provides the magic that creates a static binding between your design and your Java logic. It also enables Java syntax checking for using a design - if you remove from the design a component that your code needs, or change its variable name, you will get a compile-time error.

A design can be the whole UI or (more commonly) a smaller part of the UI, such as a view or its sub-component. A UI or view can contain many designs.

8.2. Installing in Eclipse

8.2.1. Installing Eclipse and Plug-Ins

You need to install the following to use Vaadin Designer:

1. Eclipse Luna SR2+ as described in Section 2.5, "Installing the Eclipse IDE and Plugin"
2. Vaadin Plug-in for Eclipse as described in Section 2.5.2, "Installing the Vaadin Eclipse Plugin"
3. Vaadin Designer from vaadin.com/eclipse

Vaadin Designer is compatible with Eclipse Luna (and later) available from www.eclipse.org/downloads. We recommend choosing *Eclipse IDE for Java EE Developers*.

If you're using an existing install of Eclipse Luna, please make sure it is up-to-date. Eclipse Luna versions prior to the SR2 version had a nasty bug that will cause problems for several plug-ins.

Vaadin Designer is installed together with the Vaadin Plug-in for Eclipse, from the same Eclipse update-site. In Eclipse, do **Help ▾ Install New Software**, press **Add...** next to the **Work with select**, enter Vaadin as name and <http://vaadin.com/eclipse> as location.

If you already have the Vaadin plug-in installed, just choose to Work with the Vaadin update site. Make sure the whole Vaadin category is selected (or at least Vaadin Designer), then click **Next** to review licensing information and finalize the install. Please restart Eclipse when prompted.

Once installed, Vaadin Designer can be kept up-to-date by periodically running **Help ▾ Check for Updates**.

8.2.2. Uninstalling

If you want to remove Vaadin Designer from your Eclipse installation, go to **Help > Installation Details**, select **Vaadin Designer** from the list, then click **Uninstall**.

8.3. Installing in IntelliJ IDEA

8.3.1. Installing IntelliJ IDEA and Vaadin Designer

Install IntelliJ IDEA as described in Section 2.7, “Installing and Configuring IntelliJ IDEA”.

Vaadin Designer is compatible with IntelliJ IDEA 14.1 (and later), both Community and Ultimate Editions.

To install Vaadin Designer in IntelliJ IDEA:

1. Open IntelliJ IDEA
2. Choose **IntelliJ IDEA > Preferences > Plugins** in macOS, **File > Settings > Plugins** in Windows and Linux.
3. Click **Browse Repositories...**
4. Search for **Vaadin Designer**
5. Select **Vaadin Designer** and click **Install**
6. Restart IntelliJ IDEA to make the plugin active

8.3.2. Uninstalling

If you want to remove Vaadin Designer from your IntelliJ IDEA installation, go to **IntelliJ IDEA > Preferences > Plugins** in macOS, **File > Settings > Plugins** in Windows and Linux, select **Vaadin Designer** from the list, then click **Uninstall**.

8.4. Licensing

The first time you start Vaadin Designer, it will ask for a license key. You can obtain a free trial-license, purchase a stand-alone perpetual license, or use the license included with your Pro Tools subscription. For instructions on how to install the li-

cense, see Section 17.3, "Installing Commercial Vaadin Add-on License".

Please note that a separate license key is required for each developer. If you choose not to enter a license, you will be unable to save your design.

If you for any reason need to remove or change a valid license, it is located in `~/vaadin.designer.developer.license` in UNIX systems and `C:\Users\<username>\vaadin.designer.developer.license` in Windows.

8.5. Getting Started

Vaadin Designer works projects using Vaadin 7.5 or later. In short, create a new project with **File □ New □ Vaadin 7 Project**, and choose 7.5 or later as Vaadin version, as described in Section 3.4, "Creating and Running a Project in Eclipse".

8.5.1. Creating a Design

With your project selected, select **File □ New □ Other** (or press **Ctrl+N**), choose **Vaadin Design** from the list, and click **Next**.

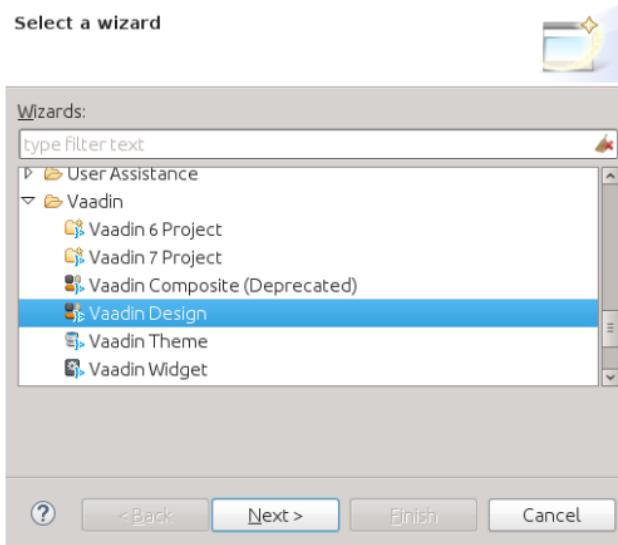


Figure 8.2. Creating a New Vaadin Design

In the design parameters step, make sure the locations are correct (if you are using Maven they might point to different folders, otherwise probably not). You can put the design(s) in a specific package if you wish.

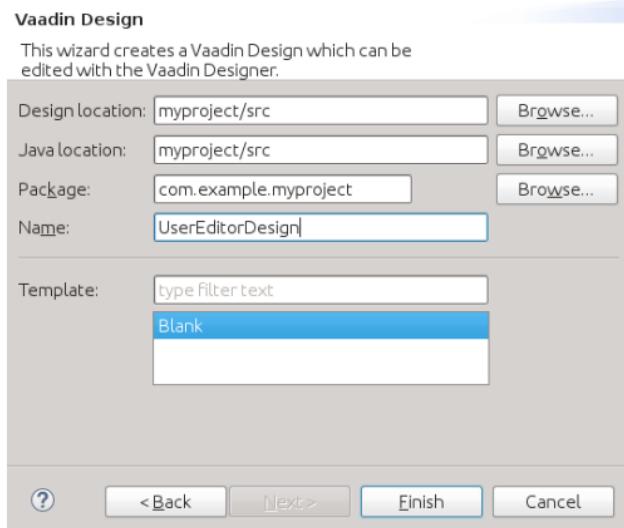


Figure 8.3. New Design Parameters

Give your design a descriptive name. Using a naming convention to separate the design's companion Java file from the classes using it will make things easier for you later.

For example, the name **UserEditorDesign** will result in UserEditorDesign.html and UserEditorDesign.java. You could then create a **UserEditor** component that extends the **UserEditorDesign**, and perhaps a **UserEditorView** to place the editor component in a bigger context.

In another case, you could make a **LoginDesign** that is used in a **LoginWindow** (but not extended).

Finally, you can choose a template as starting point, or start from scratch (Blank).

Choose **Finish** to create the design and open Vaadin Designer.

8.5.2. Vaadin Designer GUI Overview

Vaadin Designer is fully integrated with Eclipse and its views can therefore be freely moved and arranged as you wish. However, it is best to first try the Designer in its default setup by choosing **Show perspective** from the dialog that is presented when a new design is created.

To be able to successfully use the Designer, you will need the Outline, Palette and Properties views, in addition to the main editor. If you accidentally close a view, it can be opened from **Window ▾ Show view**.

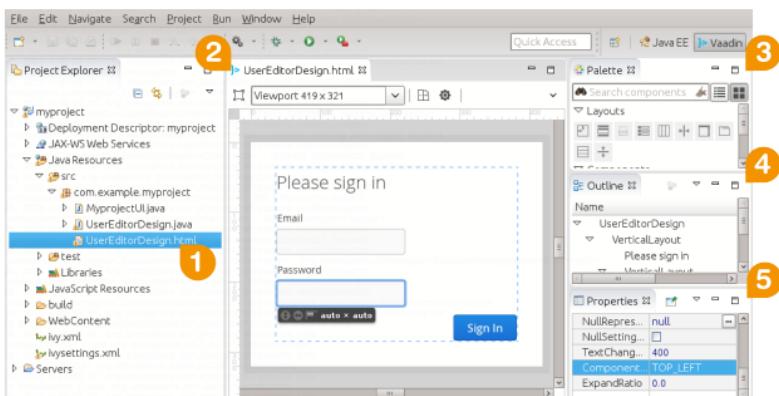


Figure 8.4. Panels in Vaadin Perspective

The elements of the perspective illustrated in Figure 8.4, "Panels in Vaadin Perspective" are as follows:

1. Design files
2. Editor (see below for close-up)
3. Component palette
4. Outline - component hierarchy
5. Properties for the selected component

In the editor view, illustrated in Figure 8.5, "Component Editor", you have a number of controls in the toolbar.

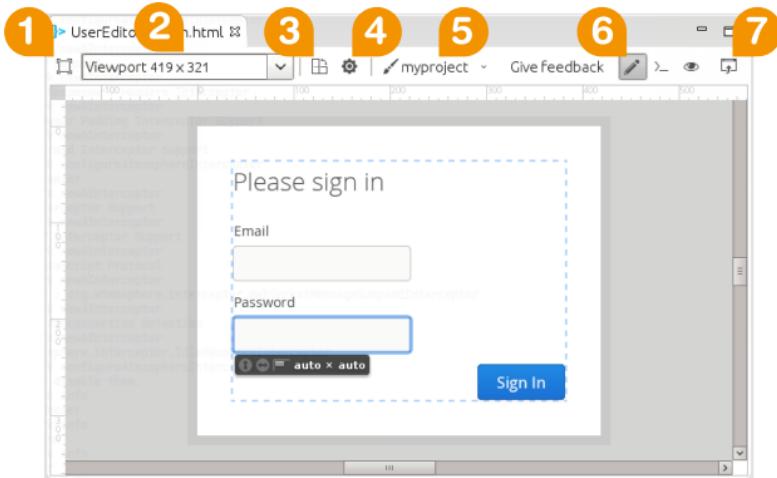


Figure 8.5. Component Editor

1. Center viewport
2. Viewport size and presets
3. Rotate viewport (portrait / landscape)
4. Configure canvas: rulers, grids, snapping
5. Theme selector
6. Design / Code / Quick preview -modes
7. External preview

8.6. Designing

To add a component to your design, drag it from the component **Palette** view and drop it in the desired location - either in the canvas area or in the hierarchical **Outline** view. Dropping in the desired location on the canvas is the most common approach, but in some situations (especially with complex, deeply nested component hierarchies) dropping on the **Outline** view gives more control.

8.6.1. About Layouts

Your designs should usually start with some sort of layout as the root component, or otherwise you are limited to a one-component design. You can also use a component that is not strictly a layout, but can still contain one (or several) components (or layouts) - this includes **TabSheet**, **Accordion**, **Panel**, etc.

There are three main types of layouts: ordered, absolute, and CSS.

Ordered layouts

Ordered layouts arrange the contained components in some ordered fashion, for instance vertically or horizontally with uniform spacing. This makes it easy to align components, and achieve a consistent look.

VerticalLayout, **HorizontalLayout**, and **FormLayout** fall into this category.

When you drop a component on a ordered layout, it will end up in a position determined by the layout, not exactly where you dropped it. Drop indicators help you estimate where the component will end up.

AbsoluteLayout

AbsoluteLayout allows free positioning of components, and supports anchoring freely in all directions. It is a powerful layout, but can be more challenging to use. You can use rulers, grids, guides, and snapping to aid your work.

AbsoluteLayout allows you to position components freely - a component is placed where you drop it. However, if you anchor the component elsewhere than to top/left, or use relative positioning, it might move when you change the size of the layout.

CssLayout

As the name indicates, **CssLayout** uses CSS to position components. It is very flexible, and with appropriate CSS, it can be used to achieve responsive layouts and a consistent look and feel. However, it requires CSS -

either pre-made and copied to your theme, or hand-crafted by you.

8.6.2. Starting from Blank

When you add the first layout to your blank canvas, it will be sized 100% x 100%, filling the whole viewport. Whether or not this is a good idea depends on your design. For many UIs having a **VerticalLayout** as root, it makes sense to have the layout 100% wide, but *auto* high. This will make the layout grow vertically as you add components, instead of splitting the available vertical space evenly between components.

Most UIs will not look good without margin and spacing. You can enable them for ordered layouts in the **Properties** view. Figure 8.6, “Effects of Margin and Spacing” illustrates the same layout without margin or spacing, with margin, and with spacing.

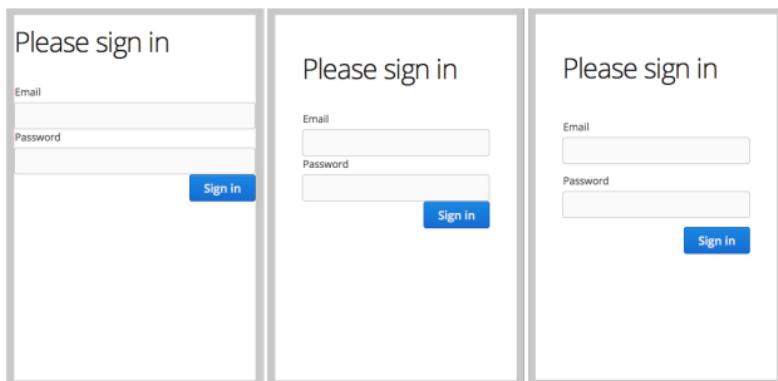


Figure 8.6. Effects of Margin and Spacing

The *info bar*, illustrated in Figure 8.7, “Info Bar for Quick Adjustments”, lets you quickly toggle between *auto* sizing and 100%. You can try out the effect of these changes by grabbing just outside the viewport (canvas) corner and resizing it (add a few components to your layout first).



Figure 8.7. Info Bar for Quick Adjustments

8.6.3. Using Templates

Templates provide a starting-point for your design - add, remove, and modify the created design as you see fit. You can pick the template for your design when you create the design using the New Design wizard of your IDE. See Section 8.5.1, "Creating a Design".

To learn how to create templates of your own or how to import templates others have made, see Section 8.9, "Templates".

8.6.4. Adding Components

Components can be added by dragging from the **Palette** view, either to the canvas or to the **Outline** view.

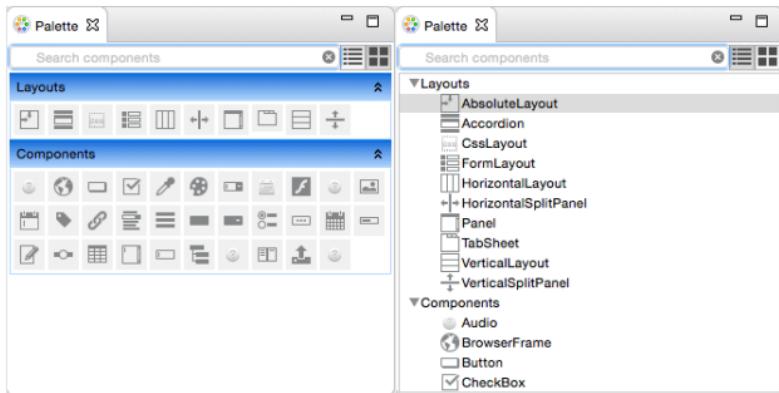


Figure 8.8. Component Palette (Alternate Layouts)

The component **Palette** view has a search field, and also two modes: list and tile. In the list mode, you can see the component name next to the icon, which is convenient at first. The tile mode lets you see all components at once, which will speed up your work quite a bit. It requires a little patience, but is really worth your while in the long run. The component name can also be seen when hovering on the icon.

The component you add will be selected in the editor view, and you can immediately edit its properties, such as the caption.

8.6.5. Editing Properties

You can edit component properties in the **Properties** view. It is a good idea to give components at least a **name** if they are to be used from Java code to add logic (such as click listeners for buttons). Generally, this is needed for most controls, but not for most layouts.

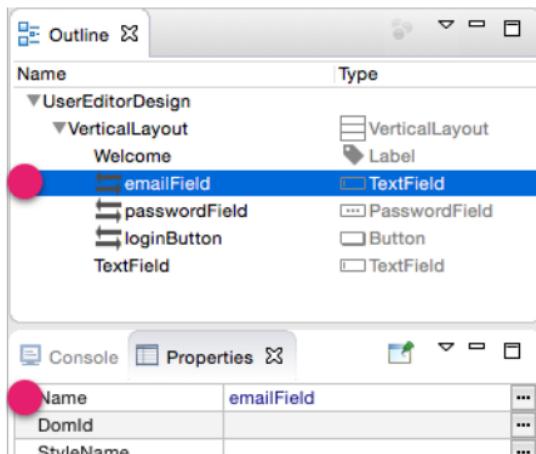


Figure 8.9. Property View

In addition to exporting the named components to Java, you will end up with things like `saveButton` and `emailField` in your **Outline** view, which will help you keep track of your components.

Note the ellipsis (...) button next to most properties - in many cases a more helpful editor is presented when you click it.

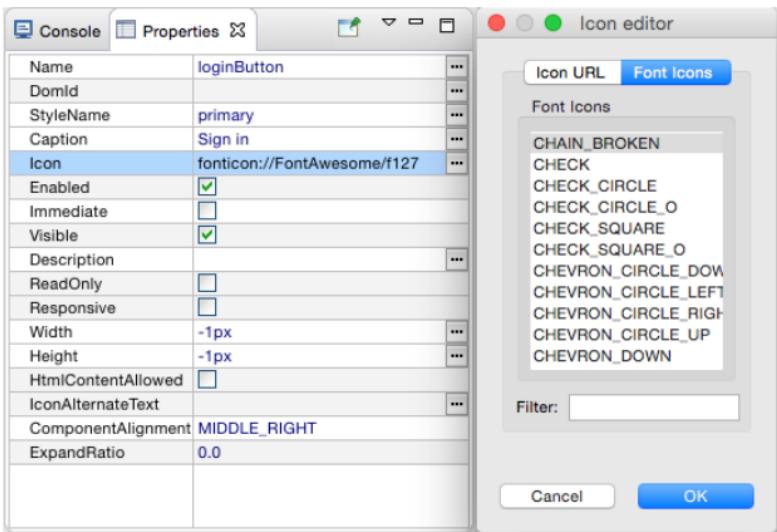


Figure 8.10. A Property Editor

Using Resources

Some properties refer to a resource object that should be loaded. This includes the *icon* property, and the *source* property for some components. In Vaadin there are multiple types of resources, as described in Section 5.5, “Images and Other Resources”. They are represented by a URI in the Designer UI and in the declarative format used by Designer.

Table 8.1. Different Resource types

Resource Name	URI Format	Description
ExternalResource	http[s]:// ftp[s]://	Browser loads the resource from the given address.
ThemeResource	theme://	The resource is loaded from the application theme folder.
FileResource	file://	The resource is loaded from the server filesystem.

Resource Name	URI Format	Description
FontIcon	fonticon://	Font icon path. Only for the Icon component.
ClassResource	N/A	Not supported by the declarative format.
StreamResource	N/A	Not supported by the declarative format.

Components that have a source property include **Audio**, **Flash**, **Image**, **Link** and **Video**.

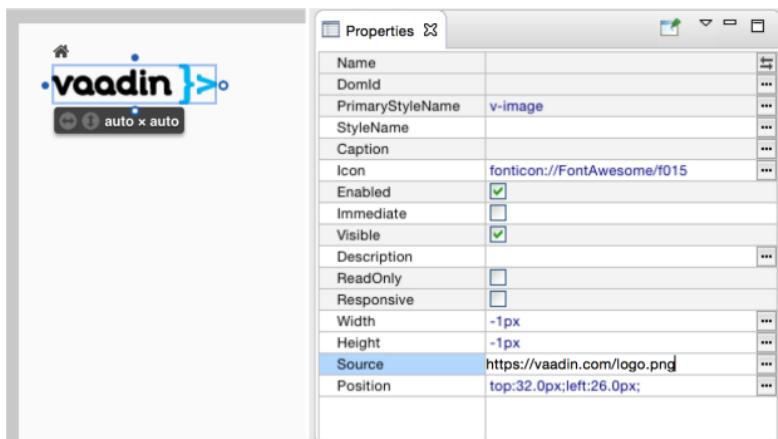


Figure 8.11. An Image component with a Font Icon

Wrapping a Component

Once in a while, you may need to wrap a component with a layout, in order to achieve the desired result (quite often injecting a **HorizontalLayout** into a **VerticalLayout**, or vice versa). You can achieve this by right-clicking the component in the **Outline** view, and choosing Wrap with in the context menu.

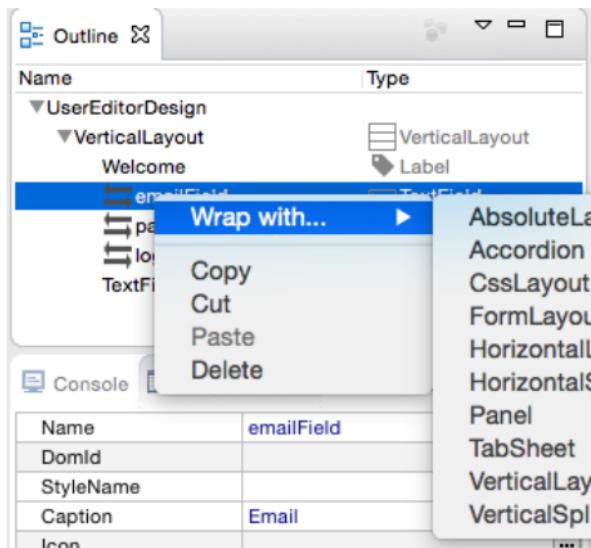


Figure 8.12. Wrapping a Component

8.6.6. Previewing

While creating a design, it is convenient to preview how the UI will behave in different sizes and on different devices. There are a number of features geared for this.

Resizing Viewport and Presets

The WYSIWYG canvas area also doubles as viewport. By resizing it, you can preview how your design will behave in different sizes, just as if it was displayed in a browser window that is being resized, or dropped in a **Panel** of a specific size.

You can manually resize the viewport by grabbing just outside of an edge or corner of the viewport, and dragging to the desired size. When you resize the viewport, you can see that the viewport control on the toolbar changes to indicate the current size.

By typing in the viewport control, you can also input a specific size (such as "200 x 200"), or open it up to reveal size presets. Choose a preset, such as **Phone** to instantly preview the design on that size.

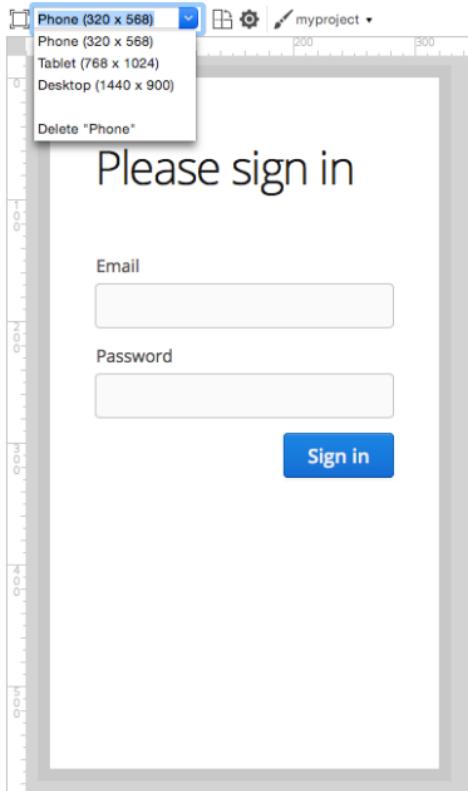


Figure 8.13. Viewport Preset Sizes

You can also add your own presets - for instance known portlet or dashboard tile sizes, or other specific sizes you want to target.

To preview the design in the other orientation (portrait vs. landscape), press the icon right of the viewport size control.

Quick preview

The **Quick preview** is one of the edit-modes available to the right in the toolbar (the other modes being **Design** and **Code**). In this mode, all designing tools and indicators are removed from the UI, and you can interact with components - type text, open dropdowns, check boxes, tab between fields, and so on. It allows you to quickly get a feel for (for instance) how a form will work when filling it in. Logic is still not run (hence

"quick"), so no real data is shown and, for example, buttons do nothing.

External Preview

The external preview popup shows a QR code and its associated URL. By browsing to the URL with and browser or device that can access your computer (that is, on the same LAN), you can instantly see the design and interact with it. This view has no extra designer-specific controls or viewports added, instead it just shows the design as-is; the browser is the viewport.

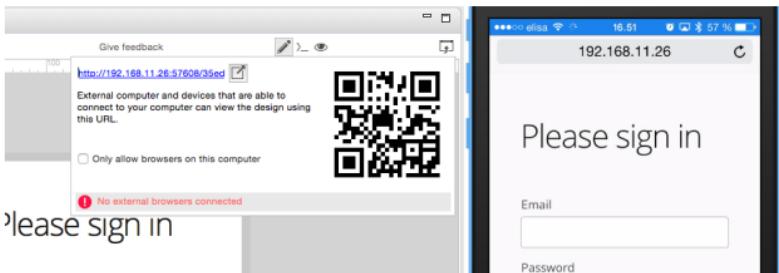


Figure 8.14. External Preview

External preview allows multiple browsers and devices to be connected at once, and they are all updated live as you change the design in Eclipse. There is an indicator in the toolbar when the design is viewed externally.

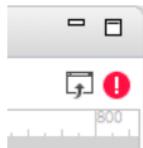


Figure 8.15. Indicator for External Preview

This is an awesome way to instantly preview results on multiple devices and browsers, or to show off a design and collaborate on it - for instance in a meeting setting.

8.7. Theming and Styling

By default, Vaadin Designer shows your design using your application theme, so usually what you see is really what you

get. You can also use the theme dropdown to apply a specific theme.

8.7.1. Theme Based on Valo

If your theme is based on the Valo theme (the default), you can make use of the built-in Valo features. For example, if you can apply the built-in component styles by adding the appropriate stylename. You will see the result instantly.

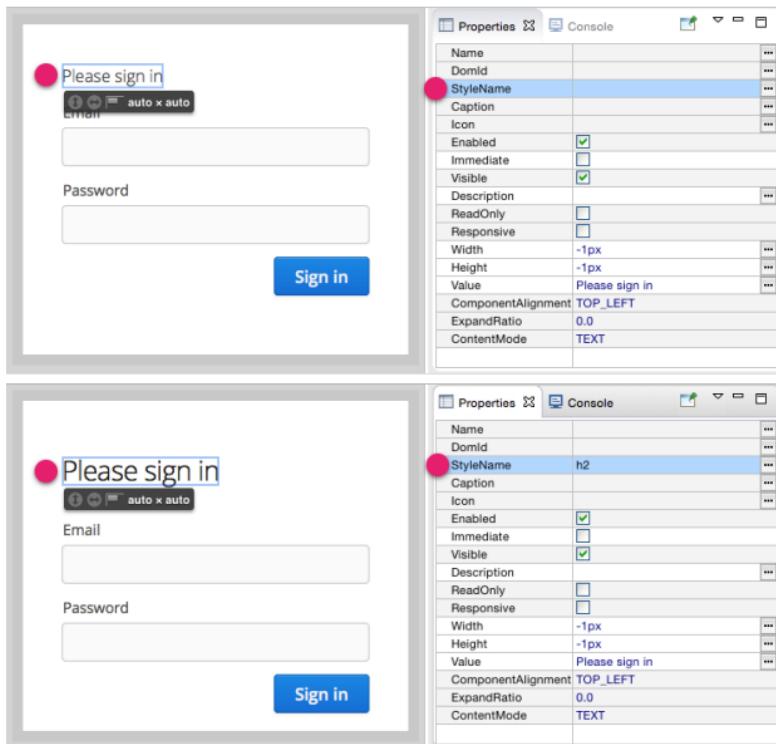


Figure 8.16. Adding Style Names

You can also modify the Valo settings by changing the parameters in your theme file (see below for more information about the theme file). See Section 9.7, "Valo Theme" for information about Valo theme.

8.7.2. Theme File

In a regular Vaadin application, your theme will be located in the VAADIN/themes/<projectname> folder, in the <projectname>.scss file. This is where you can modify Valo settings, and add your own styles.

When you make changes to this file (and save it), Vaadin Designer will notice and update the design, which is very convenient when styling your design, or generally when learning to make an application theme.

You can apply global styles (such as to style all buttons), or scoped as you wish. You can "scope" styles by specifying one or more space-separated style names in the `StyleName` property, then matching to that in CSS/Sass.

```
/* Applies to all buttons */  
.v-button { ... }  
  
/* Applies to components having the stylename "mybutton" */  
.mybutton { ... }  
  
/* Applies to all "mybutton" components within a "mydialog" layout */  
.mydialog .mybutton { ... }
```

If you use the same style names in multiple designs, the same styles will be applied, allowing you to create a consistent look.

If you do not want some styles to apply to other designs, you should give your root layout a unique style name (for instance matching the design name), and prefix all styles with that.

```
.usereditordesign .mybutton { ... }
```

8.8. Wiring It Up

Connecting Java logic to your design is made easy by splitting the UI definition and code into several layers, laying the foundation for a good separation of concerns.

From a coding perspective, a design will have three separate parts:

1. A declarative definition of the UI

2. A "companion" class exposing select components as Java fields
3. Custom Java code connected to the components exposed to Java

The declarative file (1) is normally created by Vaadin Designer, but can be created and edited by hand as well, and changes you make will be reflected in the Designer.

The companion class (2) is auto-generated based on the declarative file by Vaadin Designer, and you should not edit this file - it will be overwritten and any changes you make will not be reflected in the design.

Finally, the custom Java code (3) is completely created and maintained by you (or some other programmer). As long as the companion class (2) is used to connect logic to components, you will notice if, for example, some component goes missing. In effect, you can safely edit the design with Vaadin Designer, because you will notice if you break the logic.

8.8.1. Declarative Code

The declarative format is based on HTML/WebComponents files, and is supported directly by the Vaadin Framework. The design files have the .html suffix.

Note that the format does still not support arbitrary HTML at the moment. See Section 5.3, "Designing UIs Declaratively" for more information regarding the declarative format.

You can edit the declarative file with any text or HTML editor, but the Vaadin Designer is needed to automatically create and update the connection between declarative and Java.

Any changes you make to the declarative file are also reflected in the Designer.

Split View

In fact, you can keep the Designer open next to an HTML code editor, and see the changes you make visually reflected. This can be a powerful mode of operation.

You can open a code editor by right-clicking on a HTML design file and selecting **Open With ▾ HTML Editor**. You can then drag the editor tab to under the Designer view to create a split view, as illustrated in Figure 8.17, “Split View with Designer”.

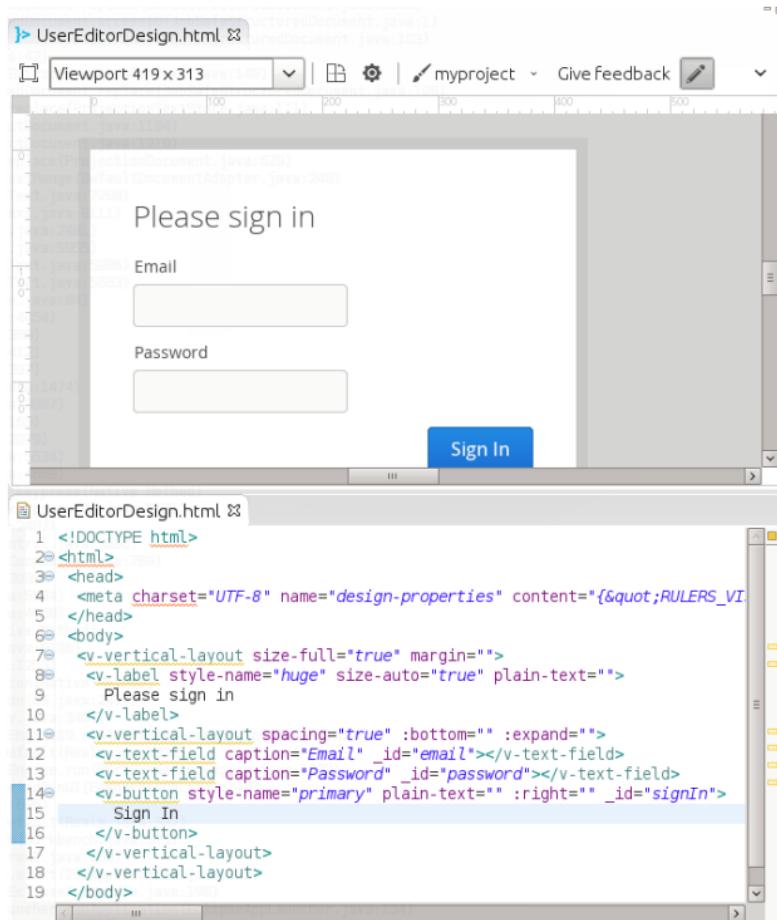


Figure 8.17. Split View with Designer

In a similar way, you can open your theme file (Sass or CSS) in a split view. When you save the file, Designer runs on-the-fly compilation for it and shows the changes to the visual appearance immediately.

8.8.2. Java Code

Vaadin Designer automatically creates a "companion" Java class, with all the components you choose to export from your design exposed as Java fields, all wired up and laid out according to your design.

The file will be overwritten by the Designer, and should not be edited.

This provides the compile-time connection between the design and Java code, as long as you are using Vaadin Designer to edit your UI. For instance, if you remove a component from the design that your code is using, you will immediately notice the error in Eclipse.

Exporting Components

Components are "exported" to Java by setting the "name" property in Vaadin Designer. The name is represented as a "`_id`" attribute in the declarative format (where it can also be manually set) and the corresponding field will be added to the Java companion class.

Note that the name is used as Java field name, so Java naming conventions are recommended.

If you change the name, the declarative file and the companion Java class will be updated, but custom code referencing the field will currently not.

Extending or Referencing

The companion Java class is overwritten and should not be edited. This is intentional, to create a clear and predictable separation of concerns. The declarative format configures the components, the companion class exposes the components to Java, and the logic goes in a separate file - either just referencing the companion class (in a composition) or by extending it.

In many cases, it is best to encapsulate the logic pertaining to a design by extending the companion class, and only ex-

posing the API and events as needed. It might even make sense to place the designs in package(s) of their own.

8.9. Templates

Templates allow you to quickly get up to speed with developing your application. Instead of starting with an empty page, you can get a head start by basing your new design on a template that defines, for example, a login screen or the basic structure of an application.

Vaadin Designer comes with a set of built-in templates. You can use those, and you can also add to the selection by installing templates created by you or others.

This chapter explains what templates are, how to use them, and how to create and install new templates.

8.9.1. What Is a Template?

A design template is simply a file that follows the Section 5.3.1, “Declarative Syntax”, with some additional information. Each template defines a design in terms of components and layouts but it can also contain style information that further defines the design’s look-and-feel.

Here is an example of a simple template file called `StyledTemplate.html` that defines a design with a **VerticalLayout** and a **Button** whose color matches the Valo theme’s focus color:

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8" name="design-properties">
  <style>
    .root-layout.styled-template {
      .focus-colored-text {
        color: $v-focus-color;
      }
    }
  </style>
</head>
<body>
  <vaadin-vertical-layout style-name="root-layout styled-template" size-full>
    <vaadin-button style-name="focus-colored-text">Focus Button</vaadin-button>
  </vaadin-vertical-layout>
</body>
</html>
```

8.9.2. Using Templates

When you create a new design, the New Design Wizard allows you to pick a template. You can choose a built-in template or a template that you have installed yourself.

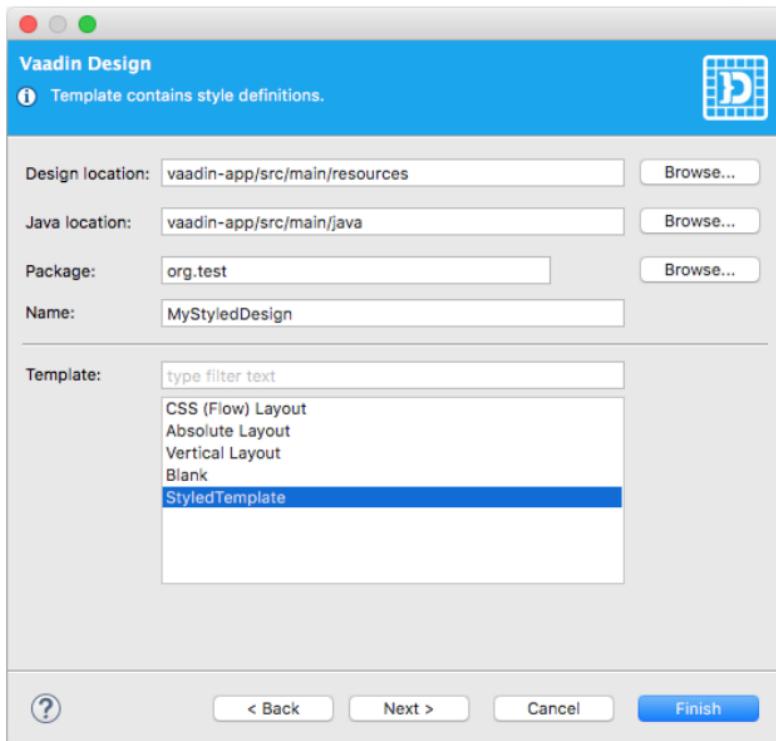


Figure 8.18. Creating a Design Based on a Template

If the template contains style information, those styles should be imported into the used theme so that they are available to the created design. In Eclipse previewing the style definitions and choosing to which theme they will be imported to can be done in the New Design wizard. In IntelliJ IDEA the styles are automatically imported to the first user theme found in the project, which - if you created the project using a maven Vaadin application archetype - is called **mytheme**.

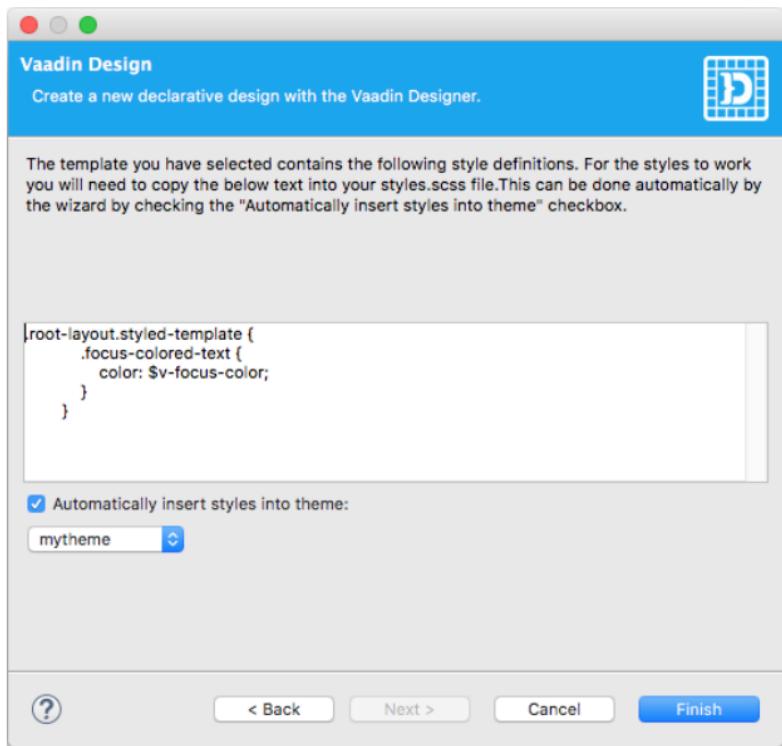


Figure 8.19. Importing Styles from a Template

The style definitions will be copied to a file called designs.scss and an @import clause will be inserted into styles.scss.

8.9.3. Installing a Template

Vaadin Designer looks for templates in a special directory located in the user's home directory. The location of the template directory varies between operating systems:

- on macOS and Linux `~/vaadin/designer/templates`
- on Windows `%HOMEPATH%\vaadin\designer\templates`

Installing a template is done by copying the template file manually to the template directory. Notice that the directory might not exist, in which case you have to create it first (using `mkdir` from a command line if using Windows OS).

8.9.4. Creating Your Own Template

You can also create templates of your own; just design a suitable starting point, then place the resulting HTML in the template directory, and it will show up in the New Design wizard as a template (without the .html extension). You don't have to modify the file in any way.

If you want to share theme style definitions with your template, you can do so by storing the SASS code inside the template file. Insert a `<style>` tag in the `<head>` tag and copy-paste the style definitions there.

8.10. Limitations

Limitations in Vaadin Designer 1.1:

- Custom client-side components are not supported when designing.
- Custom (project) components and nested designs are rendered as place-holders in Designer.
- Limited support for GridLayout: no column- or row-spanning.
- The declarative format does not support **ClassResource** or **StreamResource** for loading resources such as icons or images.

BOOK OF VAADIN

VAADIN FRAMEWORK 8

Volume 2

Vaadin Team

2017
Vaadin Ltd

BOOK OF VAADIN

Vaadin Team

Vaadin Ltd

Revision 1

Published: 2017-01-10

Vaadin Framework 8.0

This book can be downloaded for free at

<https://vaadin.com/book>

Published by

Vaadin Ltd

Ruukinkatu 2-4

20540 Turku

Finland

Abstract

Vaadin is a web application development framework that enables developers to build high-quality user interfaces with Java. It provides a set of ready-to-use user interface components and allows creating your own components. The focus is on ease-of-use, re-usability, extensibility, and meeting the requirements of large enterprise applications.

Copyright © 2000-2017 Vaadin Ltd

All rights reserved. This work is licensed under the Creative Commons CC-BY-ND License Version 2.0.

Printed by Bookwell Oy in Juva, Finland, 2017.

Chapter 9. Themes	321
9.1. Overview	321
9.2. Introduction to Cascading Style Sheets	323
9.3. Syntactically Awesome Stylesheets (Sass)	333
9.4. Compiling Sass Themes	336
9.5. Creating and Using Themes	340
9.6. Creating a Theme in Eclipse	344
9.7. Valo Theme	347
9.8. Font Icons	356
9.9. Custom Fonts	360
9.10. Responsive Themes	362
Chapter 10. Binding Components to Data	365
10.1. Overview	365
10.2. Editing Values in Fields	366
10.3. Binding Data to Forms	368
10.4. Showing Many Items in a Listing	381
10.5. Selecting Items	390
Chapter 11. Advanced Web Application Topics	395
11.1. Handling Browser Windows	396
11.2. Embedding UIs in Web Pages	400
11.3. Debug Mode and Window	403
11.4. Request Handlers	410
11.5. Shortcut Keys	412
11.6. Printing	418
11.7. Common Security Issues	421
11.8. Navigating in an Application	422
11.9. Advanced Application Architectures	428
11.10. Managing URI Fragments	434
11.11. Drag and Drop	437
11.12. Logging	449
11.13. JavaScript Interaction	451
11.14. Accessing Session-Global Data	453
11.15. Server Push	458
11.16. Vaadin CDI Add-on	465
11.17. Vaadin Spring Add-on	474
Chapter 12. Portal Integration	485
12.1. Overview	485
12.2. Creating a Generic Portlet in Eclipse	486
12.3. Developing Vaadin Portlets for Liferay	488
12.4. Portlet UI	497

12.5. Deploying to a Portal	500
12.6. Vaadin IPC for Liferay	507
Chapter 13. Client-Side Vaadin Development	517
13.1. Overview	517
13.2. Installing the Client-Side Development Environment	518
13.3. Client-Side Module Descriptor	518
13.4. Compiling a Client-Side Module	520
13.5. Creating a Custom Widget	522
13.6. Debugging Client-Side Code	524
Chapter 14. Client-Side Applications	527
14.1. Overview	527
14.2. Client-Side Module Entry-Point	530
14.3. Compiling and Running a Client-Side Application	531
14.4. Loading a Client-Side Application	532
Chapter 15. Client-Side Widgets	533
15.1. Overview	533
15.2. GWT Widgets	534
15.3. Vaadin Widgets	534
15.4. Grid	535
Chapter 16. Integrating with the Server-Side	537
16.1. Overview	538
16.2. Starting It Simple With Eclipse	541
16.3. Creating a Server-Side Component	545
16.4. Integrating the Two Sides with a Connector	546
16.5. Shared State	547
16.6. RPC Calls Between Client- and Server-Side	553
16.7. Component and UI Extensions	555
16.8. Styling a Widget	558
16.9. Component Containers	559
16.10. Advanced Client-Side Topics	559
16.11. Creating Add-ons	561
16.12. Migrating from Vaadin 6	564
16.13. Integrating JavaScript Components and Extensions	565
Chapter 17. Using Vaadin Add-ons	571
17.1. Overview	571
17.2. Using Add-ons in a Maven Project	573

17.3. Installing Commercial Vaadin Add-on License	582
17.4. Troubleshooting	585
Chapter 18. Vaadin Charts	587
18.1. Overview	587
18.2. Installing Vaadin Charts	590
18.3. Basic Use	593
18.4. Chart Types	603
18.5. Chart Configuration	634
18.6. Chart Data	645
18.7. Advanced Uses	652
18.8. Timeline	654
Chapter 19. Vaadin Spreadsheet	659
19.1. Overview	659
19.2. Installing Vaadin Spreadsheet	664
19.3. Basic Use	665
19.4. Spreadsheet Configuration	667
19.5. Cell Content and Formatting	669
19.6. Context Menus	671
19.7. Tables Within Spreadsheets	672
Chapter 20. Vaadin TestBench	675
20.1. Overview	675
20.2. Quickstart	681
20.3. Installing Vaadin TestBench	684
20.4. Developing JUnit Tests	688
20.5. Creating a Test Case	694
20.6. Querying Elements	697
20.7. Element Selectors	700
20.8. Special Testing Topics	702
20.9. Creating Maintainable Tests	708
20.10. Taking and Comparing Screenshots	713
20.11. Running Tests	719
20.12. Running Tests in a Distributed Environment	722
20.13. Parallel Execution of Tests	730
20.14. Headless Testing	732
20.15. Behaviour-Driven Development	734
20.16. Integration Testing with Maven	736
20.17. Known Issues	740
20.18. TestBench 4 to TestBench 5 Migration Guide	742

Chapter 9

Themes

9.1. Overview	321
9.2. Introduction to Cascading Style Sheets	323
9.3. Syntactically Awesome Stylesheets (Sass)	333
9.4. Compiling Sass Themes	336
9.5. Creating and Using Themes	340
9.6. Creating a Theme in Eclipse	344
9.7. Valo Theme	347
9.8. Font Icons	356
9.9. Custom Fonts	360
9.10. Responsive Themes	362

This chapter provides details about using and creating *themes* that control the visual look of web applications. Themes are created using Sass, which is an extension of CSS (Cascading Style Sheets), or with plain CSS. We provide an introduction to CSS, especially concerning the styling of HTML by element classes.

9.1. Overview

Vaadin separates the appearance of the user interface from its logic using *themes*. Themes can include Sass or CSS style sheets, custom HTML layouts, and any necessary graphics. Theme resources can also be accessed from application code as **ThemeResource** objects.

Custom themes are placed under the VAADIN/themes/ folder of the web application (under WebContent in Eclipse or src/main/webapp in Maven projects). This location is fixed – the VAADIN folder contains static resources that are served by the Vaadin servlet. The servlet augments the files stored in the folder by resources found from corresponding VAADIN folders contained in JARs in the class path. For example, the built-in themes are stored in the vaadin-themes.jar.

Figure 9.1. "Contents of a Theme" illustrates the contents of a theme.

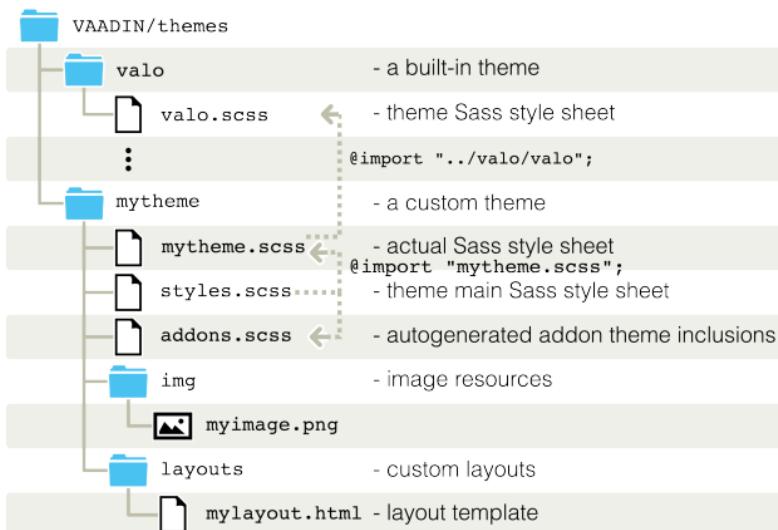


Figure 9.1. Contents of a Theme

The name of a theme folder defines the name of the theme. The name is used in the @Theme annotation that sets the theme. A theme must contain either a styles.scss for Sass themes, or styles.css stylesheet for plain CSS themes, but other contents have free naming. We recommend that you have the actual theme content in a SCSS file named after the theme, such as mytheme.scss, to make the names more unique.

We also suggest a convention for naming the folders as img for images, layouts for custom layouts, and css for additional stylesheets.

Custom themes need to extend a base theme, as described in Section 9.5, “Creating and Using Themes”. Copying and modifying an existing theme is also possible, but it is not recommended, as it may need more work to maintain if the modifications are small.

You use a theme by specifying it with the `@Theme` annotation for the UI class of the application as follows:

```
@Theme("mytheme")
public class MyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        ...
    }
}
```

A theme can contain alternate styles for user interface components, which can be changed as needed.

In addition to style sheets, a theme can contain HTML templates for custom layouts used with `CustomLayout`. See Section 7.14, “Custom Layouts” for details.

Resources provided in a theme can also be accessed using the `ThemeResource` class, as described in Section 5.5.4, “Theme Resources”. This allows displaying theme resources in component icons, in the `Image` component, and other such uses.

9.2. Introduction to Cascading Style Sheets

Cascading Style Sheets or CSS is the basic technique to separate the appearance of a web page from the content represented in HTML. In this section, we give an introduction to CSS and look how they are relevant to software development with Vaadin.

As we can only give a short instruction in this book, we encourage you to refer to the rich literature on CSS and the many resources available in the web. You can find the authoritative specifications of CSS standards from the W3C website .

9.2.1. Applying CSS to HTML

Let us consider the following HTML document that contains various markup elements for formatting text. Vaadin UIs work in essentially similar documents, even though they use somewhat different elements to draw the user interface.

```
<html>
  <head>
    <title>My Page</title>
    <link rel="stylesheet" type="text/css"
          href="mystylesheet.css"/>
  </head>
  <body>
    <p>This is a paragraph</p>
    <p>This is another paragraph</p>
    <table>
      <tr>
        <td>This is a table cell</td>
        <td>This is another table cell</td>
      </tr>
    </table>
  </body>
</html>
```

The HTML elements that will be styled later by matching CSS rules are emphasized above.

The link element in the HTML header defines the CSS stylesheet. The definition is automatically generated by Vaadin in the HTML page that loads the UI of the application. A stylesheet can also be embedded in the HTML document itself, as is done when optimizing their loading in Vaadin TouchKit, for example.

9.2.2. Basic CSS Rules

A stylesheet contains a set of *rules* that can match the HTML elements in the page. Each rule consists of one or more *selectors*, separated with commas, and a *declaration block* enclosed in curly braces. A declaration block contains a list of *property* statements. Each property has a label and a value, separated with a colon. A property statement ends with a semicolon.

Let us look at an example that matches certain elements in the simple HTML document given in the previous section:

```
p, td {  
    color: blue;  
}  
  
td {  
    background: yellow;  
    font-weight: bold;  
}
```

The p and td are element type selectors that match with p and td elements in HTML, respectively. The first rule matches with both elements, while the second matches only with td elements. Let us assume that you have saved the above style sheet with the name mystylesheet.css and consider the following HTML file located in the same folder.

This is a paragraph.

This is another paragraph.

This is a table cell	This is another table cell
----------------------	----------------------------

Figure 9.2. Simple Styling by Element Type

Style Inheritance in CSS

CSS has *inheritance* where contained elements inherit the properties of their parent elements. For example, let us change the above example and define it instead as follows:

```
table {  
    color: blue;  
    background: yellow;  
}
```

All elements contained in the table element would have the same properties. For example, the text in the contained td elements would be in blue color.

HTML Element Types

HTML has a number of element types, each of which accepts a specific set of properties. The `div` elements are generic elements that can be used to create almost any layout and formatting that can be created with a specific HTML element type. Vaadin uses `div` elements extensively to draw the UI, especially in layout components.

Matching elements by their type as shown above is, however, rarely if ever used in style sheets for Vaadin applications. We used it above, because it is the normal way in regular HTML documents that use the various HTML elements for formatting text, but it is not applicable in Vaadin UIs that consist mostly of `div` elements. Instead, you need to match by element class, as described next.

9.2.3. Matching by Element Class

Matching HTML elements by the `class` attribute is the most common form of matching in Vaadin stylesheets. It is also possible to match with the `identifier` of a unique HTML element.

The class of an HTML element is defined with the `class` attribute as follows:

```
<html>
<body>
  <p class="normal">This is the first paragraph</p>

  <p class="another">This is the second paragraph</p>

  <table>
    <tr>
      <td class="normal">This is a table cell</td>
      <td class="another">This is another table cell</td>
    </tr>
  </table>
</body>
</html>
```

The class attributes of HTML elements can be matched in CSS rules with a selector notation where the class name is written after a period following the element name. This gives us full control of matching elements by their type and class.

```
p.normal {color: red;}  
p.another {color: blue;}  
td.normal {background: pink;}  
td.another {background: yellow;}
```

The page would look as shown below:

This is a paragraph with "normal" class

This is a paragraph with "another" class

This is a table cell with "normal" class	This is a table cell with "another" class
--	---

Figure 9.3. Matching HTML Element Type and Class

We can also match solely by the class by using the universal selector * for the element name, for example *.normal. The universal selector can also be left out altogether so that we use just the class name following the period, for example .normal.

```
.normal {  
    color: red;  
}  
  
.another {  
    background: yellow;  
}
```

In this case, the rule will match with all elements of the same class regardless of the element type. The result is shown in Figure 9.4, "Matching Only HTML Element Class". This example illustrates a technique to make style sheets compatible regardless of the exact HTML element used in drawing a component.

This is a paragraph with "normal" class

This is a paragraph with "another" class

This is a table cell with "normal" class	This is a table cell with "another" class
--	---

Figure 9.4. Matching Only HTML Element Class

To ensure future compatibility, we recommend that you use only matching based on the classes and *do not* match for specific HTML element types in CSS rules, because Vaadin may change the exact HTML implementation how compon-

ents are drawn in the future. For example, Vaadin earlier used `div` element to draw **Button** components, but later it was changed to use the special-purpose button element in HTML. Because of using the `v-button` style class in the CSS rules for the button, styling it has changed only very little.

9.2.4. Matching by Descendant Relationship

CSS allows matching HTML by their containment relationship. For example, consider the following HTML fragment:

```
<body>
<p class="mytext">Here is some text inside a
    paragraph element</p>
<table class="mytable">
<tr>
    <td class="mytext">Here is text inside
        a table and inside a td element.</td>
</tr>
</table>
</body>
```

Matching by the class name `.mytext` alone would match both the `p` and `td` elements. If we want to match only the table cell, we could use the following selector:

```
.mytable .mytext {color: blue;}
```

To match, a class listed in a rule does not have to be an immediate descendant of the previous class, but just a descendant. For example, the selector "`.v-panel.v-button`" would match all elements with class `.v-button` somewhere inside an element with class `.v-panel`.

9.2.5. Importance of Cascading

CSS or Cascading Stylesheets are, as the name implies, about *cascading stylesheets*, which means applying the stylesheet rules according to their origin, importance, scope, specificity, and order.

For exact rules for cascading in CSS, see the section Cascading in the CSS specification.

Importance

Declarations in CSS rules can be made override declarations with otherwise higher priority by annotating them as !important. For example, an inline style setting made in the style attribute of an HTML element has a higher specificity than any rule in a CSS stylesheet.

```
<div class="v-button" style="height: 20px;">...
```

You can override the higher specificity with the !important annotation as follows:

```
.v-button {height: 30px !important;}
```

Specificity

A rule that specifies an element with selectors more closely overrides ones that specify it less specifically. With respect to the element class selectors most commonly used in Vaadin themes, the specificity is determined by the number of class selectors in the selector.

```
.v-button {}  
.v-verticallayout .v-button {}  
.v-app .v-verticallayout .v-button {}
```

In the above example, the last rule would have the highest specificity and would match.

As noted earlier, style declarations given in the style attribute of a HTML element have higher specificity than declarations in a CSS rule, except if the !important annotation is given.

See the CSS3 selectors module specification for details regarding how the specificity is computed.

Order

CSS rules given later have higher priority than ones given earlier. For example, in the following, the latter rule overrides the former and the color will be black:

```
.v-button {color: white}  
.v-button {color: black}
```

As specificity has a higher cascading priority than order, you could make the first rule have higher priority by adding specificity as follows:

```
.v-app .v-button {color: white}
.v-button {color: black}
```

The order is important to notice in certain cases, because Vaadin does not guarantee the order in which CSS stylesheets are loaded in the browser, which can in fact be random and result in very unexpected behavior. This is not relevant for Sass stylesheets, which are compiled to a single stylesheet. For plain CSS stylesheets, such as add-on or TouchKit stylesheets, the order can be relevant.

9.2.6. Style Class Hierarchy of a Vaadin UI

Let us give a real case in a Vaadin UI by considering a simple Vaadin UI with a label and a button inside a vertical layout:

```
// UI has v-ui style class
@Theme("mytheme")
public class HelloWorld extends UI {
    @Override
    protected void init(VaadinRequest request) {
        // VerticalLayout has v-verticallayout style
        VerticalLayout content = new VerticalLayout();
        setContent(content);

        // Label has v-label style
        content.addComponent(new Label("Hello World!"));

        // Button has v-button style
        content.addComponent(new Button("Push Me!").
            new Button.ClickListener() {
                @Override
                public void buttonClick(ClickEvent event) {
                    Notification.show("Pushed!");
                }
            });
    }
}
```

The UI will look by default as shown in Figure 9.5, “An Un-themed Vaadin UI”. By using a HTML inspector such as Firebug, you can view the HTML tree and the element classes and applied styles for each element.

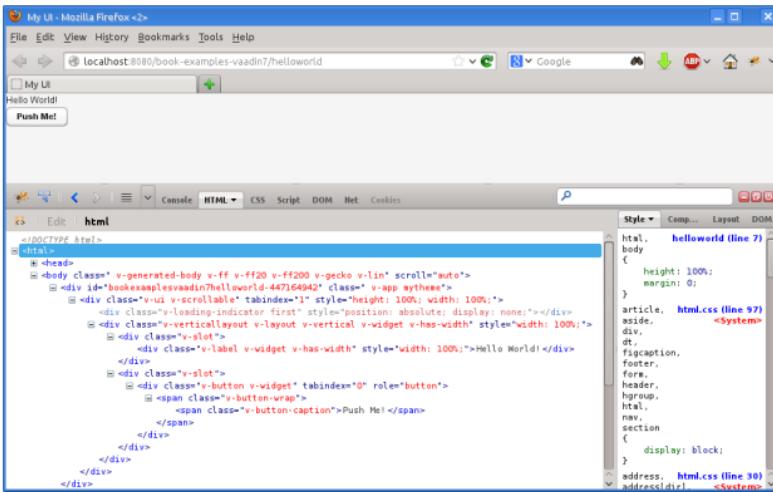


Figure 9.5. An Unthemed Vaadin UI

Now, let us look at the HTML element class structure of the UI, as we can see it in the HTML inspector:

```
<body class='v-generated-body v-ff v-ff20 v-ff200 v-gecko v-lin'
      scroll='auto'>
<div id='bookexamplesvaadin7helloworld-447164942'
      class='v-app mytheme'>
<div class='v-ui v-scrollable'
      tabindex='1' style='height: 100%; width: 100%;'>
<div class='v-loading-indicator first'
      style='position: absolute; display: none;'></div>
<div class='v-verticallayout v-layout v-vertical v-widget v-has-width'
      style='width: 100%;'>
<div class='v-slot'>
<div class='v-label v-widget v-has-width' style='width: 100%;'>Hello World!</div>
</div>
</div>
</div>
</div>
```

Now, consider the following theme where we set the colors and margins of various elements. The theme is actually a Sass theme.

```

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

@mixin mytheme {
  @include valo;

  /* White background for the entire UI */
  .v-ui {
    background: white;
  }

  /* All labels have white text on black background */
  .v-label {
    background: black;
    color: white;
    font-size: 24pt;
    line-height: 24pt;
    padding: 5px;
  }

  /* All buttons have blue caption and some margin */
  .v-button {
    margin: 10px;

    /* A nested selector to increase specificity */
    .v-button-caption {
      color: blue;
    }
  }
}

```

The look has changed as shown in Figure 9.6, “Themed Vaadin UI”.

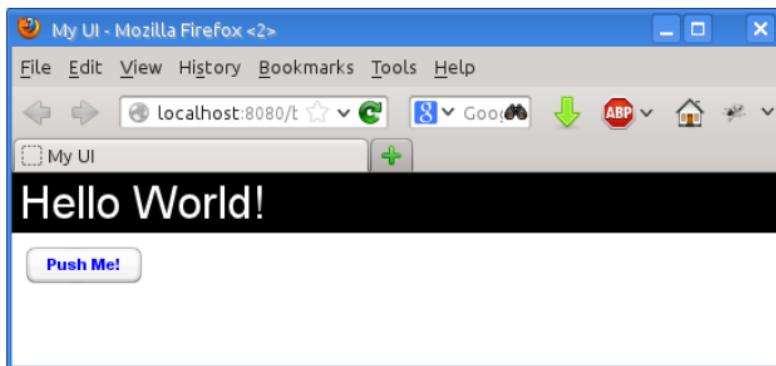


Figure 9.6. Themed Vaadin UI

An element can have multiple classes separated with a space. With multiple classes, a CSS rule matches an element if any of the classes match. This feature is used in many Vaadin components to allow matching based on the state of the component. For example, when the mouse is over a [Link](#) component, over class is added to the component. Most of such styling is a feature of Google Web Toolkit.

9.2.7. Notes on Compatibility

CSS is a standard continuously under development. It was first proposed in 1994. The specification of CSS is maintained by the CSS Working Group of World Wide Web Consortium (W3C). Versioned with backward-compatible "levels", CSS Level 1 was published in 1996, Level 2 in 1998, and the ongoing development of CSS Level 3 started in 1998. CSS3 is divided into a number of separate modules, each developed and progressing separately, and many of the modules are already Level 4.

While the support for CSS has been universal in all graphical web browsers since at least 1995, the support has been very incomplete at times and there still exists an unfortunate number of incompatibilities between browsers. While we have tried to take these incompatibilities into account in the built-in themes in Vaadin, you need to consider them while developing your own themes. Compatibility issues are detailed in various CSS handbooks.

9.3. Syntactically Awesome Stylesheets (Sass)

Vaadin uses Sass for stylesheets. Sass is an extension of CSS3 that adds nested rules, variables, mixins, selector inheritance, and other features to CSS. Sass supports two formats for stylesheet: Vaadin themes are written in SCSS (`.scss`), which is a superset of CSS3, but Sass also allows a more concise indented format (`.sass`).

Sass can be used in two basic ways in Vaadin applications, either by compiling SCSS files to CSS or by doing the compilation on the fly. The latter way is possible if the development mode is enabled for the Vaadin servlet, as described in Section 5.9.6, "Other Servlet Configuration Parameters".

9.3.1. Sass Overview

Variables

Sass allows defining variables that can be used in the rules.

```
$textcolor: blue;
```

```
.v-button-caption {  
  color: $textcolor;  
}
```

The above rule would be compiled to CSS as:

```
.v-button-caption {  
  color: blue;  
}
```

Also mixins can have variables as parameters, as explained later.

Nesting

Sass supports nested rules, which are compiled into inside-selectors. For example:

```
.v-app {  
  background: yellow;  
  
.mybutton {  
  font-style: italic;  
  
.v-button-caption {  
  color: blue;  
}  
}  
}
```

is compiled as:

```
.v-app {  
  background: yellow;  
}  
  
.v-app .mybutton {  
  font-style: italic;  
}
```

```
.v-app .mybutton .v-button-caption {  
    color: blue;  
}
```

Mixins

Mixins are rules that can be included in other rules. You define a mixin rule by prefixing it with the @mixin keyword and the name of the mixin. You can then use @include to apply it to another rule. You can also pass parameters to it, which are handled as local variables in the mixin.

For example:

```
@mixin mymixin {  
    background: yellow;  
}  
  
@mixin other mixin($param) {  
    margin: $param;  
}  
  
.v-button-caption {  
    @include mymixin;  
    @include other mixin(10px);  
}
```

The above SCSS would be translated to the following CSS:

```
.v-button-caption {  
    background: yellow;  
    margin: 10px;  
}
```

You can also have nested rules in a mixin, which makes them especially powerful. Mixing in rules is used when extending Vaadin themes, as described in Section 9.5.1, "Sass Themes".

Vaadin themes are defined as mixins to allow for certain uses, such as different themes for different portlets in a portal.

9.3.2. Sass Basics with Vaadin

We are not going to give in-depth documentation of Sass and refer you to its excellent documentation at [http://sass-](http://sass)

lang.com/. In the following, we give just basic introduction to using it with Vaadin.

You can create a new Sass-based theme with the Eclipse plugin, as described in Section 9.6, “Creating a Theme in Eclipse”.

9.4. Compiling Sass Themes

Sass themes must be compiled to CSS understood by browsers. Compilation can be done with the Vaadin Sass Compiler, which you can run in Eclipse, Maven, or it can be run on-the-fly when the application is loaded in the browser. You can also use any other Sass compiler.

9.4.1. Compiling On the Fly

The easiest way to develop Sass themes is to compile them at runtime, when the page is loaded. The Vaadin servlet does this automatically if a compiled theme CSS file is not found from the theme folder. You need to have the SCSS source files placed in the theme folder. The theme is compiled when the styles.css is first requested from the server. If you edit the Sass theme, it is recompiled the next time you reload the page.

The on-the-fly compilation takes a bit time, so it is only available when the Vaadin servlet is in the development mode, as described in Section 5.9.6, “Other Servlet Configuration Parameters”. Also, it requires the theme compiler and all its dependencies to be in the class path of the servlet. At least for production, you must compile the theme to CSS, as described next.

A blue circular icon containing a white lowercase letter 'i', representing an information or note.

Note

If the on-the-fly compilation does not seem to work, ensure that your build script has not generated a pre-compiled CSS file. If styles.css file is found the servlet, the compilation is skipped. Delete such an existing styles.css file and disable theme compilation temporarily.

9.4.2. Compiling in Eclipse

If using Eclipse and the Vaadin Plugin for Eclipse, its project wizard creates a Sass theme. It includes Compile Theme command in the toolbar to compile the project theme to CSS. Another command compiles also the widget set.

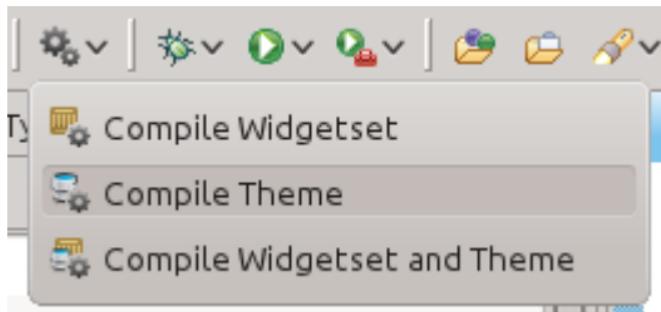


Figure 9.7. Compiling Sass Theme

The WebContent/VAADIN/mytheme/styles.scss and any Sass sources included by it are compiled to styles.css.

9.4.3. Compiling with Maven

To compile the themes with Maven, you need to include the built-in themes as a dependency:

```
...
<dependencies>
  ...
    <dependency>
      <groupId>com.vaadin</groupId>
      <artifactId>vaadin-themes</artifactId>
      <version>${vaadin.version}</version>
    </dependency>
</dependencies>
...
```

This is automatically included at least in the vaadin-archetype-application archetype for Vaadin applications. The actual theme compilation is most conveniently done by the Vaadin Maven Plugin with update-theme and compile-theme goals.

```
...
<plugin>
  <groupId>com.vaadin</groupId>
  <artifactId>vaadin-maven-plugin</artifactId>
  ...
  <executions>
    <execution>
      ...
      <goals>
        <goal>clean</goal>
        <goal>resources</goal>
        <goal>update-theme</goal>
        <goal>update-widgetset</goal>
        <goal>compile-theme</goal>
        <goal>compile</goal>
      </goals>
    </execution>
  </executions>
```

Once these are in place, the theme is compiled as part of relevant lifecycle phases, such as package.

mvn package

You can also compile just the theme with the compile-theme goal:

mvn vaadin:compile-theme

9.4.4. Compiling with Ant

With Apache Ant, you can easily resolve the dependencies with Ivy and compile the theme with the Theme Compiler included in Vaadin as follows. This build step can be conveniently included in a WAR build script.

Start with the following configuration:

```
<project xmlns:ivy="antlib:org.apache.ivy.ant"
  name="My Project" basedir=".."
  default="package-war">

  <target name="configure">
    <!-- Where project source files are located -->
    <property name="src-location" value="src" />
```

... other project build definitions ...

```
<!-- Name of the theme -->
<property name="theme" value="mytheme"/>

<!-- Compilation result directory -->
<property name="result" value="build/result"/>
</target>

<!-- Initialize build -->
<target name="init" depends="configure">
    <!-- Construct and check classpath -->
    <path id="compile.classpath">
        <!-- Source code to be compiled -->
        <pathelement path="\${src-location}" />

        <!-- Vaadin libraries and dependencies -->
        <fileset dir="\${result}/lib">
            <include name="*.jar"/>
        </fileset>
    </path>

    <mkdir dir="\${result}"/>
</target>
```

You should first resolve all Vaadin libraries to a single directory, which you can use for deployment, but also for theme compilation.

```
<target name="resolve" depends="init">
    <ivy:retrieve
        pattern="\${result}/lib/[module]-[type]-[artifact]-[revision].[ext]"/>
</target>
```

Then, you can compile the theme as follows:

```
<!-- Compile theme -->
<target name="compile-theme"
    depends="init, resolve">
    <delete dir="\${result}/VAADIN/themes/\${theme}"/>
    <mkdir dir="\${result}/VAADIN/themes/\${theme}"/>

    <java classname="com.vaadin.sass.SassCompiler"
        fork="true">
        <classpath>
            <path refid="compile.classpath"/>
        </classpath>
        <arg value="WebContent/VAADIN/themes/\${theme}/styles.scss"/>
        <arg value="\${result}/VAADIN/themes/\${theme}/styles.css"/>
    </java>
```

```
<!-- Copy theme resources -->
<copy todir="${result}/VAADIN/themes/${theme}">
  <fileset dir="WebContent/VAADIN/themes/${theme}">
    <exclude name="**/*.scss"/>
  </fileset>
</copy>
</target>
</project>
```

9.5. Creating and Using Themes

This section has not yet been updated for Vaadin Framework 8.

Custom themes are placed in the VAADIN/themes folder of the web application, in an Eclipse project under the WebContent folder or src/main/webapp in Maven projects, as was illustrated in Figure 9.1, “Contents of a Theme”. This location is fixed. You need to have a theme folder for each theme you use in your application, although applications rarely need more than a single theme.

9.5.1. Sass Themes

You can use Sass themes in Vaadin in two ways, either by compiling them to CSS by yourself or by letting the Vaadin servlet compile them for you on-the-fly when the theme CSS is requested by the browser, as described in Section 9.4, “Compiling Sass Themes”.

To define a Sass theme with the name mytheme, you must place a file with name styles.scss in the theme folder VAADIN/themes/mytheme. If no styles.css exists in the folder, the Sass file is compiled on-the-fly when the theme is requested by a browser.

We recommend that you organize the theme in at least two SCSS files so that you import the actual theme from a Sass file that is named more uniquely than the styles.scss, to make it distinguishable in the editor. This organization is how the Vaadin Plugin for Eclipse creates a new theme.

If you use Vaadin add-ons that contain themes, Vaadin Plugin for Eclipse and Maven automatically add them to the addons.scss file.

Theme SCSS

We recommend that the rules in a theme should be prefixed with a selector for the theme name. You can do the prefixing in Sass by enclosing the rules in a nested rule with a selector for the theme name.

Themes are defined as Sass mixins, so after you import the mixin definitions, you can @include them in the theme rule as follows:

```
@import "addons.scss";  
@import "mytheme.scss";  
  
.mytheme {  
  @include addons;  
  @include mytheme;  
}
```

However, this is mainly necessary if you use the UI in portlets, each of which can have its own theme, or in the special circumstance that the theme has rules that use empty parent selector & to refer to the theme name.

Otherwise, you can safely leave the nested theme selector out as follows:

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

The actual theme should be defined as follows, as a mixin that includes the base theme.

```
@import "./valo/valo.scss";  
  
@mixin mytheme {  
  @include valo;  
  
  /* An actual theme rule */  
  .v-button {  
    color: blue;  
  }  
}
```

Add-on Themes

Some Vaadin add-ons include Sass styles that need to be compiled into the theme. These are managed in the addons.scss file in a theme, included from the styles.scss. The addons.scss file is automatically generated by the Vaadin Plugin for Eclipse or Maven.

```
/* This file is automatically managed and will be
   overwritten from time to time. */
/Do not manually edit this file. */

/* Provided by vaadin-spreadsheet-1.0.0.beta.jar */ @import "../VAADIN/addons/spreadsheet/spreadsheet.scss";
/Import and include this mixin into your project
theme to include the addon themes*/
@Mixin addons {
@include spreadsheet;
}
```

9.5.2. Plain Old CSS Themes

In addition to Sass themes, you can create plain old CSS themes. CSS theme are more restricted than Sass styles - you can't parameterize CSS themes in any way, unlike you can Valo, for example. Further, an application can only have one CSS theme while you can have multiple Sass themes.

A CSS theme is defined in a styles.css file in the VAADIN/themes/mytheme folder. You need to import the legacy-styles.css of the built-in theme as follows:

```
@import "../reindeer/legacy-styles.css";

.v-app {
  background: yellow;
}
```

9.5.3. Styling Standard Components

Each user interface component in Vaadin has a CSS style class that you can use to control the appearance of the component. Many components have additional sub-elements that also allow styling. You can add context-specific stylenames with addStyleName(). Notice that getStyleName() returns only the custom stylenames, not the built-in stylenames for the component.

Please see the section on each component for a description of its styles. Most of the style names are determined in the client-side widget of each component. The easiest way to find

out the styles of the elements is to use a HTML inspector such as FireBug.

Some client-side components or component styles can be shared by different server-side components. For example, v-textfield style is used for all text input boxes in components, in addition to **TextField**.

9.5.4. Built-in Themes

Vaadin currently includes the following built-in themes:

- valo, the primary theme since Vaadin 7.3
- reindeer, the primary theme in Vaadin 6 and 7
- chameleon, an easily customizable theme
- runo, the default theme in IT Mill Toolkit 5

In addition, there is the base theme, which should not be used directly, but is extended by the other built-in themes, except valo.

The built-in themes are provided in the respective VAADIN/themes/<theme>/styles.scss stylesheets in the vaadin-themes JAR. Also the precompiled CSS files are included, in case you want to use the themes directly.

Various constants related to the built-in themes are defined in the theme classes in com.vaadin.ui.themes package. These are mostly special style names for specific components.

```
@Theme("runo")
public class MyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        ...
        Panel panel = new Panel("Regular Panel in the Runo Theme");
        panel.addComponent(new Button("Regular Runo Button"));

        // A button with the "small" style
        Button smallButton = new Button("Small Runo Button");
        smallButton.addStyleName(Runo.BUTTON_SMALL);

        Panel lightPanel = new Panel("Light Panel");
        lightPanel.addStyleName(Runo.PANEL_LIGHT);
        lightPanel.addComponent(
```

```
new Label("With addStyleName(\"light\"));
```

```
...
```

The example with the Runo theme is shown in Figure 9.8. "Runo Theme".

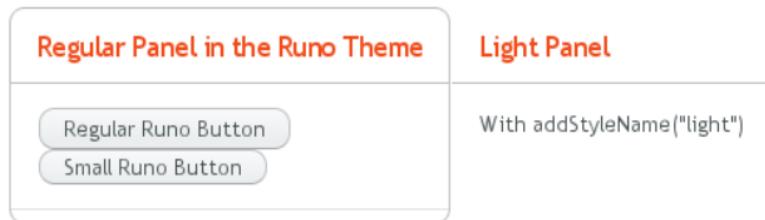


Figure 9.8. Runo Theme

The built-in themes come with a custom icon font, FontAwesome, which is used for icons in the theme, and which you can use as font icons, as described in Section 9.8, "Font Icons".

Creation of a default theme for custom GWT widgets is described in Section 16.8, "Styling a Widget".

9.5.5. Add-on Themes

You can find more themes as add-ons from the Vaadin Directory. In addition, many component add-ons contain a theme for the components they provide.

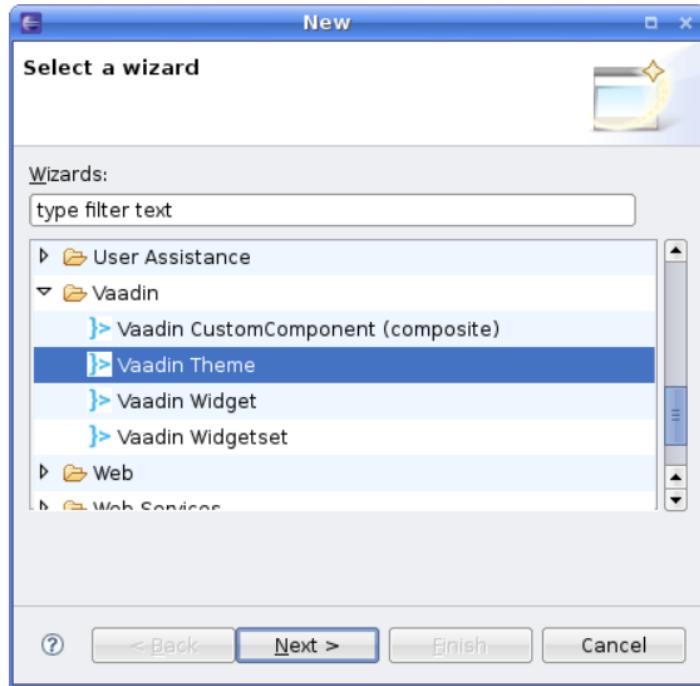
The add-on themes need to be included in the project theme. Vaadin Plugin for Eclipse and Maven automatically include them in the addons.scss file in the project theme folder. It should be included by the project theme.

9.6. Creating a Theme in Eclipse

The Eclipse plugin automatically creates a theme stub for new Vaadin projects. It also includes a wizard for creating new custom themes. Do the following steps to create a new theme.

1. Select **File □ New □ Other...** in the main menu or right-click the **Project Explorer** and select **New □ Other...**. A window will open.

2. In the **Select a wizard** step, select the **Vaadin ▾ Vaadin Theme** wizard.



Click **Next** to proceed to the next step.

3. In the **Create a new Vaadin theme** step, you have the following settings:

Project(mandatory)

The project in which the theme should be created.

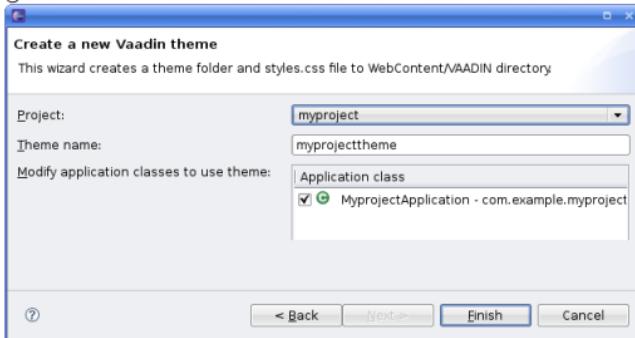
Theme name(mandatory)

The theme name is used as the name of the theme folder and in a CSS tag (prefixed with "v-theme-"), so it must be a proper identifier. Only latin alphanumerics, underscore, and minus sign are allowed.

Modify application classes to use theme(optional)

The setting allows the wizard to write a code statement that enables the theme in the constructor or the selected application (UI) class(es). If you

need to control the theme with dynamic logic, you can leave the setting unchecked or change the generated line later.



Click **Finish** to create the theme.

The wizard creates the theme folder under the WebContent/VAADIN/themes folder and the actual style sheet as mytheme.scss and styles.scss files, as illustrated in Figure 9.9, "Newly Created Theme".

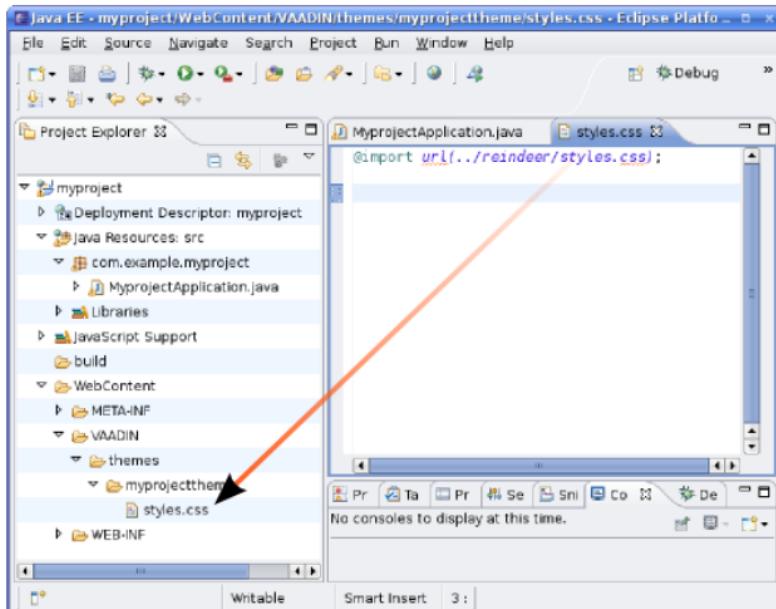


Figure 9.9. Newly Created Theme

The created theme extends a built-in base theme with an @import statement. See the explanation of theme inheritance in Section 9.5, “Creating and Using Themes”. Notice that the reindeer theme is not located in the widgetsets folder, but in the Vaadin JAR. See Section 9.5.4, “Built-in Themes” for information for serving the built-in themes.

If you selected a UI class or classes in the **Modify application classes to use theme** in the theme wizard, the wizard will add the @Theme annotation to the UI class.

If you later rename the theme in Eclipse, notice that changing the name of the folder will not automatically change the @Theme annotation. You need to change such references to theme names in the calls manually.

9.7. Valo Theme

Valo is the word for light in Finnish. The Valo theme incorporates the use of light in its logic, in how it handles shades and highlights. It creates lines, borders, highlights, and shadows adaptively according to a background color, always with contrasts pleasant to human visual perception. Auxiliary colors are computed using an algorithmic color theory to blend gently with the background. The static art is complemented with responsive animations.

The true power of Valo lies in its configurability with parameters, functions, and Sass mixins. You can use the built-in definitions in your own themes or override the defaults. Detailed documentation of the available mixins, functions, and variables can be found in the Valo API documentation available at <http://vaadin.com/valo>.

9.7.1. Basic Use

Valo is used just like other themes. Its optional parameters must be given before the @import statement.

Your project theme file, such as mytheme.scss, included from the styles.scss file, could be as follows:

```
// Modify the base color of the theme  
$v-background-color: hsl(200, 50%, 50%);
```

```
// Import valo after setting the parameters
@import "./valo/valo";

.mythemename {
  @include valo;

  // Your theme's rules go here
}
```

If you need to override mixins or function definitions in the valo theme, you must do that after the import statement, but before including the valo mixin. Also, with some configuration parameters, you can use variables defined in the theme. In this case, they need to be overridden after the import statement.

9.7.2. Common Settings

In the following, we describe the optional parameters that control the visual appearance of the Valo theme. In addition to the ones given here, component styles have their own parameters, listed in the sections describing the components in the other chapters.

General Settings

\$v-background-color(default:[literal]hsl(210, 0%, 98%))

The background color is the main control parameter for the Valo theme and it is used for computing all other colors in the theme. If the color is dark (has low luminance), light foreground colors that give high contrast with the background are automatically used.

You can specify the color in any way allowed in CSS: hexadecimal RGB color code, RGB/A value specified with `rgb()` or `rgba()`, HSL/A value specified with `hsl()` or `hsla()`. You can also use color names, but it should be avoided, as not all CSS color names are currently supported.

\$v-app-background-color(default:\$v-background-color)

Background color of the UI's root element. You can specify the color in any way allowed in CSS.

`$v-app-loading-text(default:[literal] "")`

A static text that is shown under the loading spinned while the client-side engine is being loaded and started. The text must be given in quotes. The text can not be localized currently.

`$v-app-loading-text: "Loading Resources...";`

`$v-line-height(default:[literal]7.55)`

Base line height for all widgets. It must be given a unitless number.

`$v-line-height: 1.6;`

Font Settings

`$v-font-size(default:[literal]16px)`

Base font size. It should be specified in pixels.

`$v-font-size: 18px;`

`$v-font-weight(default:[literal]300)`

Font weight for normal fonts. The size should be given as a numeric value, not symbolic.

`$v-font-weight: 400;`

`$v-font-color(default: computed)`

Foreground text color, specified as any CSS color value. The default is computed from the background color so that it gives a high contrast with the background.

`$v-font-family(default:[literal] "Open Sans", sans-serif)`

Font family and fallback fonts as a comma-separated list. Font names containing spaces must be quoted. The default font Open Sans is a web font included in the Valo theme. Other used Valo fonts must be specified in the list to enable them. See Section 9.7.4, “Valo Fonts”.

`$v-font-family: "Source Sans Pro", sans-serif;`

`$v-caption-font-size(default:[literal]round($v-font-size * 0.9))`

Font size for component captions. The value should be a pixel value.

`$v-caption-font-weight(default:[literal]max(400, $v-font-weight))`

Font weight for captions. It should be defined with a numeric value instead of symbolic.

Layout Settings

`$v-unit-size (default: round(2.3 * $v-font-size))`

This is the base size for various layout measures. It is directly used in some measures, such as button height and layout margins, while other measures are derived from it. The value must be specified in pixels, with a suitable range of 18-50.

`$v-unit-size: 40px;`

`$v-layout-margin-top, $v-layout-margin-right, $v-layout-margin-bottom, $v-layout-margin-left (default: $v-unit-size)`

Layout margin sizes for all built-in layout components, when the margin is enabled with `setMargin()`, as described in Section 7.13.5, "Layout Margins".

`$v-layout-spacing-vertical and $v-layout-spacing-horizontal (default: round($v-unit-size/3))`

Amount of vertical or horizontal space when spacing is enabled for a layout with `setSpacing()`, as described in Section 7.13.4, "Layout Cell Spacing".

Component Features

The following settings apply to various graphical features of some components.

`$v-border(default:[literal]1px solid (v-shade 0.7))`

Border specification for the components that have a border. The thickness measure must be specified in pixels. For the border color, you can specify any CSS color or one of the `v-tint`, `v-shade`, and `v-tone` keywords described later in this section.

`$v-border-radius(default:[literal]4px)`

Corner radius for components that have a border. The measure must be specified in pixels.

`$v-border-radius: 8px;`

`$v-gradient(default:[literal]v-linear 8%)`

Color gradient style for components that have a gradient. The gradient style may use the following keywords: v-linear and v-linear-reverse. The opacity must be given as percentage between 0% and 100%.

`$v-gradient: v-linear 20%;`

`$v-bevel(default:[literal]inset 0 1px 0 v-tint, inset 0 -1px 0 v-shade)`

Inset shadow style to define how some components are "raised" from the background. The value follows the syntax of CSS box-shadow, and should be a list of insets. For the bevel color, you can specify any CSS color or one of the v-tint, v-shade, and v-tone keywords described later in this section.

`$v-bevel-depth(default:[literal]30%)`

Specifies the "depth" of the bevel shadow, as applied to one of the color keywords for the bevel style. The actual amount of tint, shade, or tone is computed from the depth.

`$v-shadow(default:[literal]0 2px 3px v-shade)`

Default shadow style for all components. As with \$v-bevel, the value follows the syntax of CSS box-shadow, but without the inset. For the shadow color, you can specify any CSS color or one of the v-tint or v-shade keywords described later in this section.

`$v-shadow-opacity(default:[literal]5%)`

Specifies the opacity of the shadow, as applied to one of the color keywords for the shadow style. The actual amount of tint or shade is computed from the depth.

`$v-focus-style(default:[literal]0 0 0 2px rgba($v-focus-color, .5))`

Box-shadow specification for the field focus indicator. The space-separated values are the horizontal shadow position in pixels, vertical shadow position in pixels, blur distance in pixels, spread distance in pixels, and the color. The color can be any CSS color. You can only specify the color, in which case defaults for the position

are used. `rgba()` or `hsla()` can be used to enable transparency.

For example, the following creates a 2 pixels wide orange outline around the field:

`$v-focus-style: 0 0 0 2px orange;`

`$v-focus-color(default:[literal]valo-focus-color())`

Color for the field focus indicator. The `valo-focus-color()` function computes a high-contrast color from the context, which is usually the background color. The color can be any CSS color.

`$v/animations-enabled(default:[literal]true)`

Specifies whether various CSS animations are used.

`$v-hover-styles-enabled(default:[literal]true)`

Specifies whether various `:hover` styles are used for indicating that mouse pointer hovers over an element.

`$v-disabled-opacity(default:[literal]0.5)`

Opacity of disabled components, as described in Section 6.3.3, “Enabled”.

`$v-selection-color(default:[literal]$v-focus-color)`

Color for indicating selection in selection components.

`$v-default-field-width(default:[literal]$v-unit-size * 5)`

Default width of certain field components, unless overridden with `setWidth()`.

`$v-error-indicator-color(default:[literal]#ed473b)`

Color of the component error indicator, as described in Section 5.6.1, “Error Indicator and Message”.

`$v-required-field-indicator-color(default:[literal]$v-error-indicator-color)`

Color of the required indicator in field components.

Color specifications for `$v-border`, `$v-bevel`, and `$v-shadow` may use, in addition to CSS colors, the following keywords:

`v-tint`

Lighter than the background color.

v-shade

Darker than the background color.

v-tone

Adaptive color specification: darker on light background and lighter on dark background. Not usable in \$v-shadow.

For example:

```
$v-border: 1px solid v-shade;
```

You can fine-tune the contrast by giving a weight parameter in parentheses:

```
$v-border: 1px solid (v-tint 2);
```

```
$v-border: 1px solid (v-tone 0.5);
```

Theme Compilation and Optimization

\$v-relative-paths(default:[literal]true)

This flag specifies whether relative URL paths are relative to the currently parsed SCSS file or to the compilation root file, so that paths are correct for different resources. Vaadin theme compiler parses URL paths differently from the regular Sass compiler (Vaadin modifies relative URL paths). Use false for Ruby compiler and true for Vaadin Sass compiler.

\$v-included-components(default: component list)

Theme optimization parameter to specify the included component themes, as described in Section 9.7.6, “Theme Optimization”.

\$v-included-additional-styles(default:[literal]\$v-included-components)

Theme optimization parameter that lists the components for which the additional component stylenames should be included. See Section 9.7.5, “Component Styles” for more details.

9.7.3. Valo Mixins and Functions

Valo uses Sass mixins and functions heavily to compute various theme features, such as colors and shades. Also, all component styles are mixins. You can use the built-in mixins or override them. For detailed documentation of the mixins and functions, please refer to the Valo API documentation available at <http://vaadin.com/valo/api>.

9.7.4. Valo Fonts

Valo includes the following custom fonts:

- Open Sans
- Source Sans Pro
- Roboto
- Lato
- Lora

The used fonts must be specified with the \$v-font-family parameter for Valo, in a fallback order. A font family is used in decreasing order of preference, in case a font with higher preference is not available in the browser. You can specify any font families and generic families that browsers may support. In addition to the primary font family, you can use also others in your application. To enable using the fonts included in Valo, you need to list them in the variable.

```
$v-font-family: 'Open Sans', sans-serif, 'Source Sans Pro';
```

Above, we specify Open Sans as the preferred primary font, with any sans-serif font that the browser supports as a fallback. In addition, we include the Source Sans Pro as an auxiliary font that we can use in custom rules as follows:

```
.v-label pre {  
    font-family: 'Source Sans Pro', monospace;  
}
```

This would specify using the font in any **Label** component with the PREFORMATTED content mode.

9.7.5. Component Styles

Many components have component-specific styles to make them smaller, bigger, and so forth. You can specify the component styles with `addStyleName()` using the constants defined in the **ValoTheme** enum.

```
table.addStyleName(ValoTheme.TABLE_COMPACT);
```

For a complete up-to-date list of component-specific styles, please refer to Vaadin API documentation on the **ValoTheme** enum. Some are also described in the component-specific styling sections.

Disabling Component Styles

Component styles are optional, but all are enabled by default. They can be enabled on per-component basis with the `$v-included-additional-styles` parameter. It defaults to `$v-included-components` and can be customized in the same way, as described in Section 9.7.6, "Theme Optimization".

Configuration Parameters

The following variables control some common component styles:

`$v-scaling-factor—tiny(default:[literal]0.75)`
A scaling multiplier for TINY component styles.

`$v-scaling-factor—small(default:[literal]0.85)`
A scaling multiplier for SMALL component styles.

`$v-scaling-factor—large(default:[literal]1.2)`
A scaling multiplier for LARGE component styles.

`$v-scaling-factor—huge(default:[literal]1.6)`
A scaling multiplier for HUGE component styles.

9.7.6. Theme Optimization

Valo theme allows optimizing the size of the compiled theme CSS by including the rules for only the components actually used in the application. The included component styles can

be specified in the `$v-included-components` variable, which by default includes all components. The variable should include a comma-separated list of component names in lower-case letters. Likewise, you can specify which additional component styles, as described in Section 9.7.5, “Component Styles”, should be included using the `$v-included-additional-styles` parameter and the same format. The list of additional styles defaults to `$v-included-components`.

For example, if your UI contains just **VerticalLayout**, **TextField**, and **Button** components, you could define the variable as follows:

```
$v-included-components:  
    verticallylayout,  
    textfield,  
    button;
```

You can use the `remove()` function reversely to remove just some component themes from the standard selection.

For example, with the following you can remove the theme definitions for the **Calendar** component:

```
$v-included-components: remove($v-included-components, calendar);
```

Note that in this case, you need to give the statement *after* the `@import` statement for the Valo theme, because it overrides a variable by using its value that is defined in the theme.

9.8. Font Icons

This section has not yet been updated for Vaadin Framework 8.

Font icons are icons included in a font. Fonts have many advantages over bitmap images. Browsers are usually faster in rendering fonts than loading image files. Web fonts are vector graphics, so they are scalable. As font icons are text characters, you can define their color in CSS by the regular foreground color property.

9.8.1. Loading Icon Fonts

Vaadin currently comes with one custom icon font: FontAwesome. It is automatically enabled in the Valo theme. For other themes, you need to include it with the following line in your project theme, after importing the base theme:

```
@include fonticons;
```

If you use other icon fonts, as described in Section 9.8.5, "Custom Font Icons", and the font is not loaded by a base theme, you need to load it with a font mixin in Sass, as described in Section 9.9.1, "Loading Local Fonts".

9.8.2. Basic Use

Font icons are resources of type **FontIcon**, which implements the Resource interface. You can use these special resources for component icons and such, but not as embedded images, for example.

Each icon has a Unicode codepoint, by which you can use it. Vaadin includes an awesome icon font, FontAwesome, which comes with an enumeration of all the icons included in the font.

Most typically, you set a component icon as follows:

```
TextField name = new TextField("Name");
name.setIcon(FontAwesome.USER);
layout.addComponent(name);

// Button allows specifying icon resource in constructor
Button ok = new Button("OK", FontAwesome.CHECK);
layout.addComponent(ok);
```

The result is illustrated in Figure 9.10, "Basic Use of Font Icons", with the color styling described next.

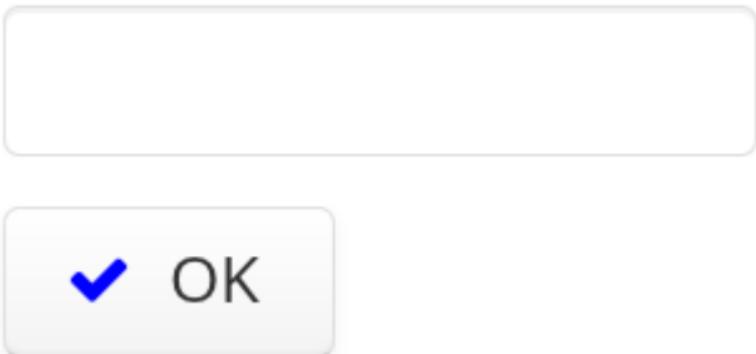


Figure 9.10. Basic Use of Font Icons

Styling the Icons

As font icons are regular text, you can specify their color with the color attribute in CSS to specify the foreground text color. All HTML elements that display icons in Vaadin have the v-icon style name.

```
.v-icon {  
    color: blue;  
}
```

If you use the font icon resources in other ways, such as in an **Image** component, the style name will be different.

9.8.3. Using Font icons in HTML

You can use font icons in HTML code, such as in a **Label**, by generating the HTML to display the icon with the getHtml() method.

```
Label label = new Label("I " +  
    FontAwesome.HEART.getHtml() + " Vaadin".  
    ContentMode.HTML);  
label.addStyleName("redicon");  
layout.addComponent(label);
```

The HTML code has the v-icon style, which you can modify in CSS:

```
.redicon .v-icon {  
    color: red;  
}
```

The result is illustrated in Figure 9.11, “Using Font Icons in Label”, with the color styling described next.



Figure 9.11. Using Font Icons in Label

You could have set the font color in the label's HTML code as well, or for all icons in the UI.

You can easily use font icons in HTML code in other ways as well. You just need to use the correct font family and then use the hex-formatted Unicode codepoint for the icon. See for example the implementation of the getHtml() method in **FontAwesome**:

```
@Override  
public String getHtml() {  
    return "<span class=\"v-icon\" style=\"font-family: " +  
        getFontFamily() + ";\\>&#x" +  
        Integer.toHexString(codepoint) + "</span>";  
}
```

9.8.4. Using Font Icons in Other Text

You can include a font icon in any text by its Unicode codepoint, which you can get with the getCodePoint() method. In such case, however, you need to use the same font for other text in the same string as well. The FontAwesome provided in Vaadin includes a basic character set.

```
TextField amount = new TextField("Amount (in " +  
    new String(Character.toChars(  
        FontAwesome.BTC.getCodepoint()) +  
    ")");
```

```
amount.addStyleName("awesomcaption");
layout.addComponent(amount);
```

You need to set the font family in CSS.

```
.v-caption-awesomcaption .v-captiontext {
    font-family: FontAwesome;
}
```

9.8.5. Custom Font Icons

You can easily use glyphs in existing fonts as icons, or create your own.

Creating New Icon Fonts With IcoMoon

You are free to use any of the many ways to create icons and embed them into fonts. Here, we give basic instructions for using the IcoMoon service, where you can pick icons from a large library of well-designed icons.

Font Awesome is included in IcoMoon's selection of icon libraries. Note that the codepoints of the icons are not fixed, so the **FontAwesome** enum is not compatible with such custom icon fonts.

After you have selected the icons that you want in your font, you can download them in a ZIP package. The package contains the icons in multiple formats, including WOFF, TTF, EOT, and SVG. Not all browsers support any one of them, so all are needed to support all the common browsers. Extract the fonts folder from the package to under your theme.

See Section 9.9.1, "Loading Local Fonts" for instructions for loading a custom font.

9.9. Custom Fonts

In addition to using the built-in fonts of the browser and the web fonts included in the Vaadin themes, you can use custom web fonts.

9.9.1. Loading Local Fonts

You can load locally served web fonts with the font mixin as follows:

```
@include font(MyFontFamily,  
  './././mytheme/fonts/myfontfamily');
```

The statement must be given in the styles.scss file *outside* the .mytheme {} block.

The first parameter is the name of the font family, which is used to identify the font. If the font family name contains spaces, you need to use single or double quotes around the name.

The second parameter is the base name of the font files without an extension, including a relative path. Notice that the path is relative to the base theme, where the mixin is defined, not the used theme. We recommend placing custom font files under a fonts folder in a theme.

Not all browsers support any single font file format, so the base name is appended with .ttf, .eot, .woff, or .svg suffix for the font file suitable for a user's browser.

9.9.2. Loading Web Fonts

You can load externally served web fonts such as Google Fonts simply by specifying the loading stylesheet for the UI with the **@StyleSheet** annotation.

For example, to load the "Cabin Sketch" font from Google Fonts:

```
@StyleSheet(["http://fonts.googleapis.com/css?family=Cabin+Sketch"])  
public class MyUI extends UI {  
  ...
```

9.9.3. Using Custom Fonts

After loaded, you can use a custom font, or actually font family, by its name in CSS and otherwise.

```
.mystyle {  
    font-family: MyFontFamily;  
}
```

Again, if the font family name contains spaces, you need to use single or double quotes around the name.

9.10. Responsive Themes

Vaadin includes support for responsive design which enables size range conditions in CSS selectors, allowing conditional CSS rules that respond to size changes in the browser window on the client-side.

You can use the **Responsive** extension to extend either a component, typically a layout, or the entire UI. You specify the component by the static `makeResponsive()` method.

```
// Have some component with an appropriate style name  
Label c = new Label("Here be text");  
c.addStyleName("myresponsive");  
content.addComponent(c);
```

```
// Enable Responsive CSS selectors for the component  
Responsive.makeResponsive(c);
```

You can now use width-range and height-range conditions in CSS selectors as follows:

```
/* Basic settings for all sizes */  
.myresponsive {  
    padding: 5px;  
    line-height: 36pt;  
}  
  
/* Small size */  
.myresponsive[width-range~="0-300px"] {  
    background: orange;  
    font-size: 16pt;  
}  
  
/* Medium size */  
.myresponsive[width-range~="301px-600px"] {  
    background: azure;  
    font-size: 24pt;  
}
```

```
/* Anything bigger */  
.myresponsive[width-range~="601px-"] {  
    background: palegreen;  
    font-size: 36pt;  
}
```

You can have overlapping size ranges, in which case all the selectors matching the current size are enabled.

Chapter 10

Binding Components to Data

10.1. Overview	365
10.2. Editing Values in Fields	366
10.3. Binding Data to Forms	368
10.4. Showing Many Items in a Listing	381
10.5. Selecting Items	390

This chapter describes the Vaadin Data Model and shows how you can use it to bind components directly to data sources, such as database queries.

10.1. Overview

The Vaadin Data Model is one of the core concepts of the library. There is a standard data interface that all UI components use to access and modify the application's data.

The most basic UI component for handling data is a field component that lets the user define a single value, for instance a text field for writing the name of a product or a dropdown menu for selecting which department an employee belongs

to. See ??? to learn how these components can be used on their own.

In most applications, there are business classes that represent real-world objects like a single employee or a product in an inventory. The user interface is structured as a form that lets the user edit all the different properties of a single business object instance. Vaadin Framework makes it easy to create forms for editing these sorts. You have full control over how you configure and lay out the individual input fields making up a form, and then you can use Binder to hook up those fields to a business object instance. ??? shows how to bind data to fields.

There are UI components in the framework that lists multiple similar objects and lets the user view, select and in some cases even edit those objects. A listing component can get its data from an in-memory collection or lazily fetch it from some backend. In either case, there are options available for defining how the data is sorted and filtered before being displayed to the user. Read more about how to provide lists of data to these components in ??? . Using a listing component as an input field to select one or many of the listed items is described in ???.

Vaadin Data Model topic references

- ???
- ???
- ???
- ???

10.2. Editing Values in Fields

Input field components (implementing the HasValue interface) are in a very central role for handling data in an application since different types of fields are the main user interface controls used for displaying and editing data.

While each field implementation has its own functionality, all fields also have some common core functionality. By using

these common building blocks, the data binding part of the framework can help simplify the code we need to write for many common data entry cases.

At the very core, each field has a value that the user can see and edit through the user interface. The value can also be read and set through code.

```
TextField nameField = new TextField("Enter your name");

Button sayHelloButton = new Button("Say hello", clickEvent -> {
    String name = nameField.getValue();
    Notification.show("Hello " + name);
});
```

Each field implementation has its own specific value type – for instance, the type of a TextField is String, the type of a Slider is Double, the type of a DateField is LocalDate, and so on.

10.2.1. Reacting to Value Changes

When the value of a field changes, it fires a value change event. By listening to the event, we can find out the new value of the field and whether the value was changed by the user through the user interface or by code through the setValue method.

```
TextField nameField = new TextField("Enter your name");
nameField.addValueChangeListener(event -> {
    String origin = event.isUserOriginated()
        ? "by the user"
        : "from code";
    String message = "Name is " + event.getValue()
        + " as set " + origin;
    Notification.show(message);
});

Button setButton = new Button("Set name", event -> {
    // Will show "Name is Zaphod as set from code"
    nameField.setValue("Zaphod");
});
```

Fields can also be set in read-only mode, which means that the user is not able to directly edit the value through the user interface, but the value can still be changed through code. This is useful for showing the user that the data is there, even though the user is currently not allowed to edit it.

When editing multiple values from the same business object, you can use Binder to simplify how the values of all input fields in a form are handled.

??? describes how this is done.

10.3. Binding Data to Forms

A typical application lets the user fill out structured data and maybe also browse previously entered data. The data that is being entered is typically represented in code as an instance of a business object (bean), for instance a **Person** in an HR application.

Vaadin Framework provides a **Binder** class that the developer can use to define how the values in a business object should be bound to the fields shown in the user interface. **Binder** takes care of reading values from the business object and converting the user's data between the format expected by the business object and the format expected by the field. The input entered by the user can also be validated, and the current validation status can be presented to the user in different ways.

The first step to binding fields for a form is to create a **Binder** and bind some input fields. There is only one **Binder** instance for each form and it is used for all fields in that form.

```
Binder<Person> binder = new Binder<>();

TextField titleField = new TextField();

// Start by defining the Field instance to use
binder.forField(titleField)
// Finalize by doing the actual binding to the Person class
.bind()
// Callback that loads the title from a person instance
Person::getTitle.
// Callback that saves the title in a person instance
Person::setTitle);

TextField nameField = new TextField();

// Shorthand for cases without extra configuration
binder.bind(nameField, Person::getName, Person::setName);
```

When we have bound field components using our binder, we can use the binder to load values from a person into the field, let the user edit the values and finally save the values back into a person instance.

```
// The person to edit
// Would be loaded from the backend in a real application
Person person = new Person("John Doe", 1957);

// Updates the value in each bound field component
binder.readBean(person);

Button saveButton = new Button("Save".
event -> {
    try {
        binder.writeBean(person);
        // A real application would also save the updated person
        // using the application's backend
    } catch (BindingException e) {
        Notification.show("Person could not be saved." +
            "please check error messages for each field.");
    }
});

// Updates the fields again with the previously saved values
Button resetButton = new Button("Reset".
event -> binder.readBean(person));
```

With these basic steps, we have defined everything that is needed for loading, editing and saving values for a form.

The above example uses Java 8 method references for defining how field values are loaded and saved. It is also possible to use a lambda expression or an explicit instance of the callback interface instead of a method reference.

```
// With lambda expressions
binder.bind(titleField,
person -> person.getTitle(),
(person, title) -> person.setTitle(title));

// With explicit callback interface instances
binder.bind(nameField,
new ValueProvider<Person, String>() {
    @Override
    public String apply(Person person) {
        return person.getName();
    }
},
new Setter<Person, String>() {
```

```
@Override  
public void accept(Person person, String name) {  
    person.setName(name);  
}  
});
```

10.3.1. Validating and Converting User Input

Binder supports checking the validity of the user's input and converting the values between the type used in business objects and the bound UI components. These two concepts go hand in hand since validation can be based on a converted value, and being able to convert a value is a kind of validation.

Validation

An application typically has some restrictions on exactly what kinds of values the user is allowed to enter into different fields. **Binder** lets us define validators for each field that we are binding. The validator is by default run whenever the user changes the value of a field, and the validation status is also checked again when saving.

Validators for a field are defined between the `forField` and `bind` steps when a binding is created. A validator can be defined using a **Validator** instance or inline using a lambda expression.

```
binder.forField(emailField)  
    // Explicit validator instance  
    .withValidator(new EmailValidator(  
        "This doesn't look like a valid email address"))  
    .bind(Person::getEmail, Person::setEmail);  
  
binder.forField(nameField)  
    // Validator defined based on a lambda and an error message  
    .withValidator(  
        name -> name.length() >= 3,  
        "Full name must contain at least three characters")  
    .bind(Person::getName, Person::setName);  
  
binder.forField(titleField)  
    // Shorthand for requiring the field to be non-empty  
    .asRequired("Every employee must have a title")  
    .bind(Person::getTitle, Person::setTitle);
```



Note

Binder.forField works like a builder where forField starts the process, is followed by various configuration calls for the field and bind acts as the finalizing method which applies the configuration.

The validation state of each field is updated whenever the user modifies the value of that field. The validation state is by default shown using **AbstractComponent**.setComponentError which is used by the layout that the field is shown in. Whenever an error is set, the component will also get a v-<component>-error class name, e.g. v-textfield-error. This error class will by default add a red border on the component. The component will also get a tooltip that shows the error message text.

We can also customize the way a binder displays error messages to get more flexibility than what setComponentError provides. The easiest way of customizing this is to configure each binding to use its own **Label** that is used to show the status for each field.



Note

The status label is not only used for validation errors but also for showing confirmation and helper messages.

```
Label emailStatus = new Label();
```

```
binder.forField(emailField)
    .withValidator(new EmailValidator(
        "This doesn't look like a valid email address"))
    // Shorthand that updates the label based on the status
    .withStatusLabel(emailStatus)
    .bind(Person::getEmail, Person::setEmail);
```

```
Label nameStatus = new Label();
```

```
binder.forField(nameField)
    // Define the validator
    .withValidator(
```

```

name -> name.length() >= 3,
"Full name must contain at least three characters")
// Define how the validation status is displayed
.withValidationStatusHandler(status -> {
    nameStatus.setValue(status.getMessage().orElse(""));
    nameStatus.setVisible(status.isError());
})
// Finalize the binding
.bind(Person::getName, Person::setName);

```

It is possible to add multiple validators for the same binding. The following example will first validate that the entered text looks like an email address, and only for seemingly valid email addresses it will continue checking that the email address is for the expected domain.

```

binder.forField(emailField)
    .withValidator(new EmailValidator(
        "This doesn't look like a valid email address"))
    .withValidator(
        email -> email.endsWith("@acme.com"),
        "Only acme.com email addresses are allowed")
    .bind(Person::getEmail, Person:: setEmail);

```

In some cases, the validation of one field depends on the value of some other field. We can save the binding to a local variable and trigger a revalidation when another field fires a value change event.

```

DateField departing = new DateField("Departing");
DateField returning = new DateField("Returning");

// Store return date binding so we can revalidate it later
Binding<Trip, LocalDate> returnBinding = binder.forField(returning)
    .withValidator(returnDate -> !returnDate.isBefore(departing.getValue()),
        "Cannot return before departing");
returnBinding.bind(Trip::getReturnDate, Trip::setReturnDate);

// Revalidate return date when departure date changes
departing.addValueChangeListener(event -> returnBinding.validate());

```

Conversion

You can also bind application data to a UI field component even though the types do not match. In some cases, there might be types specific for the application, such as custom type that encapsulates a postal code that the user enters through a **TextField**. Another quite typical case is for entering integer numbers using a **TextField** or a **Slider**. Similarly to validators, we can define a converter using a **Converter** in-

stance or inline using lambda expressions. We can optionally specify also an error message.

```
TextField yearOfBirthField = new TextField("Year of birth");
```

```
binder.forField(yearOfBirthField)
    .withConverter(
        new StringToIntegerConverter("Must enter a number"))
    .bind(Person::getYearOfBirth, Person::setYearOfBirth);
```

// Slider for integers between 1 and 10

```
Slider salaryLevelField = new Slider("Salary level", 1, 10);
```

```
binder.forField(salaryLevelField)
    .withConverter(Double::intValue, Integer::doubleValue)
    .bind(Person::getSalaryLevel, Person::setSalaryLevel);
```

Multiple validators and converters can be used for building one binding. Each validator or converter is used in the order they were defined for a value provided by the user. The value is passed along until a final converted value is stored in the business object, or until the first validation error or impossible conversion is encountered. When updating the UI components, values from the business object are passed through each converter in the reverse order without doing any validation.



Note

A converter can be used as a validator but for code clarity and to avoid boilerplate code, you should use a validator when checking the contents and a converter when modifying the value.

```
binder.forField(yearOfBirthField)
// Validator will be run with the String value of the field
    .withValidator(text -> text.length() == 4,
                  "Doesn't look like a year")
// Converter will only be run for strings with 4 characters
    .withConverter(
        new StringToIntegerConverter("Must enter a number"))
// Validator will be run with the converted value
    .withValidator(year -> year >= 1900 && year < 2000,
                  "Person must be born in the 20th century")
    .bind(Person::getYearOfBirth, Person::setYearOfBirth);
```

You can define your own conversion either by using callbacks, typically lambda expressions or method references, or by implementing the Converter interface.

When using callbacks, there is one for converting in each direction. If the callback used for converting the user-provided value throws an unchecked exception, then the field will be marked as invalid and the message of the exception will be used as the validation error message. Messages in Java runtime exceptions are typically written with developers in mind and might not be suitable to show to end users. We can provide a custom error message that is used whenever the conversion throws an unchecked exception.

```
binder.forField(yearOfBirthField)
    .withConverter(
        Integer::valueOf,
        String::valueOf,
        //Text to use instead of the NumberFormatException message
        "Please enter a number")
    .bind(Person::getYearOfBirth, Person::setYearOfBirth);
```

There are two separate methods to implement in the Converter interface. convertToModel receives a value that originates from the user. The method should return a Result that either contains a converted value or a conversion error message. convertToPresentation receives a value that originates from the business object. Since it is assumed that the business object only contains valid values, this method directly returns the converted value.

```
class MyConverter implements Converter<String, Integer> {
    @Override
    public Result<Integer> convertToModel(String fieldValue, ValueContext context) {
        // Produces a converted value or an error
        try {
            //ok is a static helper method that creates a Result
            return Result.ok(Integer.valueOf(fieldValue));
        } catch (NumberFormatException e) {
            // error is a static helper method that creates a Result
            return Result.error("Please enter a number");
        }
    }

    @Override
    public String convertToPresentation(Integer integer, ValueContext context) {
        // Converting to the field type should always succeed.
        // so there is no support for returning an error Result.
        return String.valueOf(integer);
    }

    // Using the converter
    binder.forField(yearOfBirthField)
        .withConverter(new MyConverter())
        .bind(Person::getYearOfBirth, Person::setYearOfBirth);
```

The provided ValueContext can be used for finding Locale to be used for the conversion.

10.3.2. Loading from and Saving to Business Objects

Once all bindings have been set up, you are ready to actually fill the bound UI components with data from your business object. Changes can be written to the business object automatically or manually.

Writing the changes automatically when the user makes any change through the UI is often the most convenient option, but it might have undesirable side effects – the user may see unsaved changes if some other part of the application uses the same business object instance. To prevent that, you either need to use a copy of the edited object or use manual writing to only update the object when the user wants to save.

Manual Reading and Writing

The `readBean` method reads values from a business object instance into the UI components.

```
Person person = new Person("John Doe", 1957);
```

```
binder.readBean(person);
```

Assuming binder has already been configured as in previous examples with a `TextField` bound to the `name` property, this example would show the value "John Doe" in that field.

To avoid showing lots of errors to the user, validation errors are not shown until the user edits each field after the form has been bound or loaded.

Even if the user has not edited a field, all validation errors will be shown if we explicitly validate the form or try to save the values to a business object.

```
// This will make all current validation errors visible
BinderValidationStatus<Person> status = binder.validate();

if (status.hasErrors()) {
    Notification.show("Validation error count: "
        + status.getValidationErrors().size());
}
```

Trying to write the field values to a business object will fail if any of the bound fields has an invalid value. There are different methods that let us choose how to structure the code for dealing with invalid values.

Handling a checked exception

```
try {
    binder.writeBean(person);
    MyBackend.updatePersonInDatabase(person);
} catch (ValidationException e) {
    Notification.show("Validation error count: "
        + e.getValidationErrors().size());
}
```

Checking a return value

```
boolean saved = binder.writeBeanIfValid(person);
if (saved) {
    MyBackend.updatePersonInDatabase(person);
} else {
    Notification.show("Validation error count: "
        + binder.getValidationErrors().size());
}
```

Binder keeps track of which bindings have been updated by the user and which bindings are in an invalid state. It also fires an event when this status changes. We can use that event to make the save and reset buttons of our forms become enabled or disabled depending on the current status of the form.

```
binder.addStatusChangeListener(event -> {
    boolean isValid = !event.hasValidationErrors();
    boolean hasChanges = binder.hasChanges();

    saveButton.setEnabled(hasChanges && isValid);
    resetButton.setEnable(hasChanges);
});
```

Automatic Saving

Instead of manually saving field values to a business object instance, we can also bind the values directly to an instance. In this way, Binder takes care of automatically saving values from the fields.

```

Binder<Person> binder = new Binder<>():

// Field binding configuration omitted, it should be done here

Person person = new Person("John Doe", 1957);

// Loads the values from the person instance
// Sets person to be updated when any bound field is updated
binder.setBean(person);

Button saveButton = new Button("Save", event -> {
    if (binder.validate().isOk()) {
        // person is always up-to-date as long as there are no
        // validation errors
    }
});

MyBackend.updatePersonInDatabase(person);
}:

```



Warning

When using the setBean method, the business object instance will be updated whenever the user changes the value in any bound field. If some other part of the application is also using the same instance, then that part might show changes before the user has clicked the save button.

10.3.3. Binding Beans to Forms

The business objects used in an application are in most cases implemented as Java beans or POJOs. There is special support for that kind of business object in **BeanBinder**. It can use reflection based on bean property names to bind values. This reduces the amount of code you have to write when binding to fields in the bean.

```

BeanBinder<Person> binder = new BeanBinder<>(Person.class);

// Bind based on property name
binder.bind(nameField, "name");
// Bind based on sub property path
binder.bind(streetAddressField, "address.street");
// Bind using forField for additional configuration
binder.forField(yearOfBirthField)
    .withConverter(
        new StringToIntegerConverter("Please enter a number"))
    .bind("yearOfBirth");

```



Note

BeanBinder uses strings to identify the properties so it is not refactor safe.

BeanBinder will automatically use JSR 303 Bean Validation annotations from the bean class if a Bean Validation implementation is available. Constraints defined for properties in the bean will work in the same way as if configured when the binding is created.

```
public class Person {  
    @Min(2000)  
    private int yearOfBirth;  
  
    //Non-standard constraint provided by Hibernate Validator  
    @NotEmpty  
    private String name;  
  
    // + other fields, constructors, setters, and getters  
}
```

Constraint annotations can also be defined on the bean level instead of being defined for any specific property.



Note

Bean level validation can only be performed once the bean has been updated. This means that this functionality can only be used together with `setBean`. You need to trigger validation manually if using `readBean` and `writeBean`.

Validation errors caused by that bean level validation might not be directly associated with any field component shown in the user interface, so **BeanBinder** cannot know where such messages should be displayed.

Similarly to how the `withStatusLabel` method can be used for defining where messages for a specific binding should be showed, we can also define a **Label** that is used for showing status messages that are not related to any specific field.

```
Label formStatusLabel = new Label();
BeanBinder<Person> binder = new BeanBinder<>(Person.class);
binder.setStatusLabel(formStatusLabel);
// Continue by binding fields
```

We can also define our own status handler to provide a custom way of handling statuses.

```
// We will first set the status label's content mode to HTML
// in order to display generated error messages separated by a <br> tag
formStatusLabel.setContentMode(ContentMode.HTML);

BinderValidationStatusHandler defaultHandler = binder.getValidationStatusHandler();

binder.setValidationStatusHandler(status -> {
    // create an error message on failed bean level validations
    List<Result<?>> errors = status.getBeanValidationErrors();

    // collect all bean level error messages into a single string,
    // separating each message with a <br> tag
    String errorMessage = errors.stream().map(Result::getMessage)
        .map(o -> o.get())
        // sanitize the individual error strings to avoid code injection
        // since we are displaying the resulting string as HTML
        .map(errorString -> Jsoup.clean(errorString, Whitelist.simpleText()))
        .collect(Collectors.joining("<br>"));

    // finally, display all bean level validation errors in a single label
    formStatusLabel.setValue(errorMessage);
    formStatusLabel.setVisible(errorMessage.isEmpty());

    // Let the default handler show messages for each field
    defaultHandler.accept(status);
});
```

10.3.4. Using Binder with Declarative Layouts

We can use **Binder** to connect data to a form that is defined in the declarative format.

This is the design HTML file that we can create using Vaadin Designer:

```
<vaadin-form-layout size-full>
    <vaadin-text-field _id="name"
        caption="Name"></vaadin-text-field>
    <vaadin-text-field _id="yearOfBirth"
        caption="Year of birth"></vaadin-text-field>
    <vaadin-button _id="save">
        Save
    </vaadin-button>
</vaadin-form-layout>
```

This is the companion Java file that Vaadin Designer creates for us based on the design.

```
@DesignRoot  
@AutoGenerated  
public class PersonFormDesign extends FormLayout {  
    protected TextField name;  
    protected TextField yearOfBirth;  
    protected Button save;  
  
    public MyFormDesign() {  
        Design.read(this);  
    }  
}
```

Based on those files, we can create a subclass of the design that uses a **BeanBinder** to automatically connect bean properties to field instances. This will look at all instance fields that are of a Field type in the class and try to find a bean property with the same name.

```
public class PersonForm extends PersonFormDesign {  
    private BeanBinder<Person> binder  
    = new BeanBinder<>(Person.class);  
  
    public PersonForm(Person person) {  
        binder.bindInstanceFields(this);  
  
        binder.readBean(person);  
  
        save.addClickListener(event -> {  
            if (binder.writeBeanIfValid(person)) {  
                MyBackend.updatePersonInDatabase(person);  
            }  
        });  
    }  
}
```

We can also bind some of the fields before calling bindInstanceFields. In this way, fields that require special configuration can still be configured manually while regular fields can be configured automatically.

```
binder.forField(yearOfBirth)  
.withConverter(  
    new StringToIntegerConverter("Please enter a number"))
```

```
.bind(Person::getYearOfBirth, Person::setYearOfBirth));
```

```
binder.bindInstanceFields(this);
```

10.4. Showing Many Items in a Listing

A common pattern in applications is that the user is first presented with a list of items, from which she selects one or several items to continue working with. These items could be inventory records to survey, messages to respond to or blog drafts to edit or publish.

A Listing is a component that displays one or several properties from a list of item, allowing the user to inspect the data, mark items as selected and in some cases even edit the item directly through the component. While each listing component has its own API for configuring exactly how the data is represented and how it can be manipulated, they all share the same mechanisms for receiving data to show.

The items are generally either loaded directly from memory or lazy loaded from some kind of backend. Regardless of how the items are loaded, the component is configured with one or several callbacks that define how the item should be displayed.

In the following example, a **ComboBox** that lists status items is configured to use the `Status.getCaption()` method to represent each status. There is also a **Grid**, which is configured with one column from the person's name and another showing the year of birth.

```
ComboBox<Status> comboBox = new ComboBox<>();  
comboBox.setItemCaptionGenerator(Status::getCaption);
```

```
Grid<Person> grid = new Grid<>();  
grid.addColumn(Person::getName).setCaption("Name");  
grid.addColumn(Person::getYearOfBirth)  
.setCaption("Year of birth");
```



Note

In this example, it would not even be necessary to define any item caption provider for the

combo box if **Status**.`toString()` would be implemented to return a suitable text. **ComboBox** is by default configured to use `toString()` for finding a caption to show.



Note

The Year of birth column will use **Grid**'s default **TextRenderer** which shows any values as a String. We could use a **NumberRenderer** instead, and then the renderer would take care of converting the the number according to its configuration with a formatting setting of our choice.

After we have told the component how the data should be shown, we only need to give it some data to actually show. The easiest way of doing that is to directly pass the values to show to `setItems`.

```
// Sets items as a collection
comboBox.setItems(EnumSet.allOf(Status.class));

// Sets items using varargs
grid.setItems(
    new Person("George Washington", 1732),
    new Person("John Adams", 1735),
    new Person("Thomas Jefferson", 1743),
    new Person("James Madison", 1751)
);
```

Listing components that allow the user to control the display order of the items are automatically able to sort data by any property as long as the property type implements **Comparable**.

We can also define a custom **Comparator** if we want to customize the way a specific column is sorted. The comparator can either be based on the item instances or on the values of the property that is being shown.

```
grid.addColumn("Name", Person::getName)
// Override default natural sorting
```

```
.setComparator(Comparator.comparing(  
    person -> person.getName().toLowerCase()));
```



Note

This kind of sorting is only supported for in-memory data. Sorting with data that is lazy loaded from a backend is described later in this chapter.

With listing components that let the user filter items, we can in the same way define our own `CaptionFilter` that is used to decide whether a specific item should be shown when the user has entered a specific text into the text field. The filter is defined as an additional parameter to `setItems`.

```
comboBox.setItems(  
    (itemCaption, filterText) ->  
        itemCaption.startsWith(filterText),  
    itemsToShow);
```



Note

This kind of filtering is only supported for in-memory data. Filtering with data that is lazy loaded from a backend is described later in this chapter.

Instead of directly assigning the item collection as the items that a component should be using, we can instead create a **ListDataProvider** that contains the items. One list data provider instance can be shared between different components to make them show the same data.

We can apply different sorting options for each component using the `sortingBy` method. The method creates a new data provider using the same data, but different settings. This means that we can apply different sorting options for different components.

```
ListDataProvider<Person> dataProvider =  
    new ListDataProvider<>(persons);  
  
ComboBox<Person> comboBox = new ComboBox<>();
```

```
// The combo box shows the person sorted by name
comboBox.setDataProvider(
    dataProvider.sortingBy(Person::getName));

Grid<Person> grid = new Grid<>();
// The grid shows the same persons sorted by year of birth
grid.setDataProvider(
    dataProvider.sortingBy(Person::getYearOfBirth));
```

The **Listing** component cannot automatically know about changes to the list of items or to any individual item. We must notify the data provider when items are changed, added or removed so that components using the data will show the new values.

```
ListDataProvider<Person> dataProvider =
    new ListDataProvider<>(persons);

Button addPersonButton = new Button("Add person",
    clickEvent -> {
        persons.add(new Person("James Monroe", 1758));

        dataProvider.refreshAll();
});

Button modifyPersonButton = new Button("Modify person",
    clickEvent -> {
        Person personToChange = persons.get(0);

        personToChange.setName("Changed person");

        dataProvider.refreshAll();
});
```

10.4.1. Lazy Loading Data to a Listing

All the previous examples have shown cases with a limited amount of data that can be loaded as item instances in memory. There are also situations where it is more efficient to only load the items that will currently be displayed. This includes situations where all available data would use lots of memory or when it would take a long time to load all the items.



Note

Regardless of how we make the items available to the listing component on the server, components like **Grid** will always take care of only sending the currently needed items to the browser.

For example, if we have the following existing backend service that fetches items from a database or a REST service .

```
public interface PersonService {  
    List<Person> fetchPersons(int offset, int limit);  
    int getPersonCount();  
}
```

To use this service with a listing component, we need to define one callback for loading specific items and one callback for finding how many items are currently available. Information about which items to fetch as well as some additional details are made available in a Query object that is passed to both callbacks.

```
DataProvider<Person, Void> dataProvider = new BackendDataProvider<>(  
    // First callback fetches items based on a query  
    query -> {  
        // The index of the first item to load  
        int offset = query.getOffset();  
  
        // The number of items to load  
        int limit = query.getLimit();  
  
        List<Person> persons = getPersonService().fetchPersons(offset, limit);  
  
        return persons.stream();  
    },  
    // Second callback fetches the number of items for a query  
    query -> getPersonService().getPersonCount()  
);  
  
Grid<Person> grid = new Grid<>();  
grid.setDataProvider(dataProvider);  
  
// Columns are configured in the same way as before  
...
```



Note

The results of the first and second callback must be symmetric so that fetching all available

items using the first callback returns the number of items indicated by the second callback. Thus if you impose any restrictions on e.g. a database query in the first callback, you must also add the same restrictions for the second callback.



Note

The second type parameter of DataProvider defines how the provider can be filtered. In this case the filter type is Void, meaning that it doesn't support filtering. Backend filtering will be covered later in this chapter.

Sorting

It is not practical to order items based on a Comparator when the items are loaded on demand, since it would require all items to be loaded and inspected.

Each backend has its own way of defining how the fetched items should be ordered, but they are in general based on a list of property names and information on whether ordering should be ascending or descending.

As an example, there could be a service interface which looks like the following.

```
public interface PersonService {  
    List<Person> fetchPersons(  
        int offset,  
        int limit,  
        List<PersonSort> sortOrders);  
  
    int getPersonCount();  
  
    PersonSort createSort(  
        String propertyName,  
        boolean descending);  
}
```

With the above service interface, our data source can be enhanced to convert the provided sorting options into a format

expected by the service. The sorting options set through the component will be available through `Query.getSortOrders()`.

```
DataProvider<Person, Void> dataProvider = new BackEndDataProvider<>()
query -> {
    List<PersonSort> sortOrders = new ArrayList<>();
    for(SortOrder<String> queryOrder : query.getSortOrders()) {
        PersonSort sort = getPersonService().createSort(
            // The name of the sorted property
            queryOrder.getSorted());
        // The sort direction for this property
        queryOrder.getDirection() == SortDirection.DESCENDING);
        sortOrders.add(sort);
    }

    return getPersonService().fetchPersons(
        query.getOffset(),
        query.getLimit(),
        sortOrders
    ).stream();
}

// The number of persons is the same regardless of ordering
query -> getPersonService().getPersonCount()
};
```

We also need to configure our grid so that it can know what property name should be included in the query when the user wants to sort by a specific column. When a data source that does lazy loading is used, **Grid** and other similar components will only let the user sort by columns for which a sort property name is provided.

```
Grid<Person> grid = new Grid<>();

grid.setDataProvider(dataProvider);

// Will be sortable by the user
// When sorting by this column, the query will have a SortOrder
// where getSorted() returns "name"
grid.addColumn(Person::getName)
.setCaption("Name")
.setSortProperty("name");

// Will not be sortable since no sorting info is given
grid.addColumn(Person::getYearOfBirth)
.setCaption("Year of birth");
```

There might also be cases where a single property name is not enough for sorting. This might be the case if the backend needs to sort by multiple properties for one column in the user interface or if the backend sort order should be inverted compared to the sort order defined by the user. In such cases,

we can define a callback that generates suitable **SortOrder** values for the given column.

```
grid.addColumn("Name",  
    person -> person.getFirstName() + " " + person.getLastName())  
.setSortOrderProvider(  
    // Sort according to last name, then first name  
    direction -> Stream.of(  
        new SortOrder("lastName", direction),  
        new SortOrder("firstName", direction)  
    ));
```

Filtering

Different types of backends support filtering in different ways. Some backends support no filtering at all, some support filtering by a single value of some specific type and some have a complex structure of supported filtering options.

A DataProvider<Person, String> supports filtering by string values, but it's up to the implementation to actually define how the filter is actually used. It might, for instance, look for all persons with a name beginning with the provided string.

You can use the withFilter method on a data provider to create a new provider that uses the same data, but applies the given filtering to all queries. The original provider instance is not changed.

```
DataProvider<Person, String> allPersons = getPersonProvider();
```

```
Grid<Person> grid = new Grid<>();  
grid.setDataProvider(allPersons);
```

```
DataProvider<Person, Void> johnPersons = allPersons.withFilter("John");
```

```
NativeSelect<Person> johns = new NativeSelect<>();  
johns.setDataProvider(johnPersons);
```

Note that the filter type of the johnPersons instance is Void, which means that the data provider doesn't support any further filtering.

ListDataProvider is filtered by callbacks that you can define as lambda expressions, method references or implementations of SerializablePredicate.

```
ListDataProvider<Person> allPersons =  
    new ListDataProvider<>(persons);
```

```
Grid<Person> grid = new Grid<>();
grid.setDataProvider(allPersons.withFilter(
    person -> person.getName().startsWith("John")
));
```



Tip

ListDataProvider lets you combine multiple filters since the return value of withFilter is itself also filterable by SerializablePredicate.

A listing component that lets the user control how the displayed data is filtered has some specific filter type that it uses. For ComboBox, the filter is the String that the user has typed into the search field. This means that ComboBox can only be used with a data provider whose filtering type is String.

To use a data provider that filters by some other type, you need to use the convertFilter. This method creates a new data provider that uses the same data but a different filter type: converting the filter value before passing it to the original data provider instance.

```
DataProvider<Person, String> allPersons = getPersonProvider();
ListDataProvider<Person> listProvider = new ListDataProvider<>(persons);

ComboBox<Person> comboBox = new ComboBox();

// Can use DataProvider<Person, String> directly
comboBox.setDataProvider(allPersons);

// Must define how to convert from a string to a predicate
comboBox.setDataProvider(listProvider.convertFilter(
    filterText -> {
        // Create a predicate that filters persons by the given text
        return person -> person.getName().contains(filterText);
    }
));
```

To create a data provider that supports filtering, you only need to look for a filter in the provided query and use that filter when fetching and counting items. withFilter and convertFilter are automatically implemented for you.

As an example, our service interface with support for filtering could look like this. Ordering support has been omitted in this example to keep focus on filtering.

```

public interface PersonService {
    List<Person> fetchPersons(
        int offset,
        int limit,
        String namePrefix);
    int getPersonCount(String namePrefix);
}

```

A data provider using this service could use String as its filtering type. It would then look for a string to filter by in the query and pass it to the service method.

```

DataProvider<Person, String> dataProvider = new BackEndDataProvider<>{
    query -> {
        //getFilter returns Optional<String>
        String filter = query.getFilter().orElse(null);
        return getPersonService().fetchPersons(
            query.getOffset(),
            query.getLimit(),
            filter
        ).stream();
    },
    query -> {
        String filter = query.getFilter().orElse(null);
        return getPersonService().getPersonCount(filter);
    }
};

```

10.5. Selecting Items

Listing components displaying data also support allowing the user to select items. Depending on component, it the user can select either one or several items at a time.

How selection is handled in listings is split into three categories:

- Single selection: components that only allow a single item to be selected at a time, for example RadioButtonGroup belongs in this category. More generally, all components that implement the SingleSelect interface belong in this category.
- Multi selection: components that allow for selecting any number of the displayed items, for example CheckBoxGroup. All components that implement the MultiSelect interface belong in this category.

- Listing components whose selection can be configured through the usage of the SelectionModel interface. The Grid component is an example of this type of listing and it currently has built in implementations for both the single selection and multi selection cases, as well as disabling selection altogether.

10.5.1. Single and Multi Selection

Single and multi selection components implement the HasValue interface, where the current selection represents the value that is currently held by the component. In practice this means that it is possible to get, set and listen to selection changes the same way you would with value changes in Vaadin field components. In the case of single select components HasValue is further extended with SingleSelect, and correspondingly with MultiSelect in the case of multi select components, giving further control over the current selection.

An example of basic single selection with the ComboBox component:

```
ComboBox<Availability> comboBox = new ComboBox<>();
// Populate the combo box with items
comboBox.setItems(EnumSet.allOf(Availability.class));

// Set the current selection
comboBox.setValue(Availability.DISCONTINUED);
// Get the current selection
Availability availability = comboBox.getValue();

// Add a value change listener, a ValueChangeEvent<Availability> will be fired
// any time a change to the selection is made.
comboBox.addValueChangeListener(event -> Notification.show(event.getValue()));
```

A similar example for the multi select listing CheckBoxGroup follows. A difference to note in this example is the parameter type of setValue and the return type of getValue being Set<Category>, the members of which represent the selection contents.

```
CheckBoxGroup<Category> checkBoxGroup = new CheckBoxGroup<>();
checkBoxGroup.setItems(EnumSet.allOf(Category.class));

checkBoxGroup.setValue(EnumSet.allOf(Category.class));
Set<Category> categories = checkBoxGroup.getValue();

checkBoxGroup.addValueChangeListener(event -> [
    Notification.show("Number of selected items: " + event.getValue().size());
]);
```

Additionally, MultiSelect provides numerous utility functions for simpler programmatic handling of selections, such as:

```
checkBoxGroup.select(Category.DVD, Category.BOOK);
checkBoxGroup.isSelected(Category.BOOK); // true
checkBoxGroup.deselectAll();
checkBoxGroup.getSelectedItems(); // now returns an empty set of Categories
```

10.5.2. Selection Models

Grid component can hold either multi- or single- selection. Since grid can not be both SingleSelect<SomePojo> and MultiSelect<SomePojo> in the same time, grid itself is not a select component, but it delegates the selection to a subclass of SelectionModel class. By default, Grid is in single selection mode, and we can obtain selection object using asSingleSelect method.

```
Grid<Person> grid = new Grid<>();
SingleSelect<Person> selection = grid.asSingleSelect();
//...
Notification.show(selection.getValue().getName() + " was selected");
```

If selection of multiple rows is required, then Grid needs to be switched into multiselection mode, and multiple item selection object can be obtained using asMultiSelect method.

```
Grid<Person> grid = new Grid<>();
grid.setSelectionMode(Grid.SelectionMode.MULTI);
MultiSelect<Person> selection = grid.asMultiSelect();
//...
Notification.show(
    selection.getValue().stream().map(Person::getName).collect(Collectors.joining(", "))
    + " were selected");
```

Selected Items

Selection models (subclasses of SelectionModel) allow retrieving a HasValue object corresponding to the selection with the asSingleSelect and asMultiSelect methods, and thus can be used bound to data using a Binder. This way, conversions and validation can be used for selections.

```
public static class Company {
    private Person boss;
    private Set<Person> managers;

    public Person getBoss() {
        return boss;
    }

    public void setBoss(Person boss) {
        this.boss = boss;
    }

    public Set<Person> getManagers() {
        return managers;
    }
}
```

```

public void setManagers(Set<Person> managers) {
    this.managers = managers;
}

Binder<Company> companyBinder = new Binder<>();

//Setup single selection binding
Grid<Person> bossGrid = new Grid<>();
SingleSelect<Person> bossSelection = bossGrid.asSingleSelect();
companyBinder.forField(bossSelection).bind(Company::getBoss, Company::setBoss);

//Setup multi selection binding
Grid<Person> managersGrid = new Grid<>();
managersGrid.setSelectionMode(Grid.SelectionMode.MULTI);
MultiSelect<Person> managersSelection = managersGrid.asMultiSelect();
companyBinder.forField(managersSelection).bind(Company::getManagers, Company::setManagers);

```

Selection Events

SelectionModel implementations allow retrieving a HasValue object corresponding to the selection with the asSingleSelect and asMultiSelect methods. The HasValue implementations returned by those methods support the standard addValueChangeListener method and all added listeners are notified about any selection change. In addition, selections support their own, selection-specific listeners, SelectionListener, SingleSelectionListener, and MultiSelectionListener. To add those listeners, we need to explicitly cast a selection to SingleSelectionModel, or MultiSelectionModel respectively.

Chapter 11

Advanced Web Application Topics

11.1. Handling Browser Windows	396
11.2. Embedding UIs in Web Pages	400
11.3. Debug Mode and Window	403
11.4. Request Handlers	410
11.5. Shortcut Keys	412
11.6. Printing	418
11.7. Common Security Issues	421
11.8. Navigating in an Application	422
11.9. Advanced Application Architectures	428
11.10. Managing URI Fragments	434
11.11. Drag and Drop	437
11.12. Logging	449
11.13. JavaScript Interaction	451
11.14. Accessing Session-Global Data	453
11.15. Server Push	458
11.16. Vaadin CDI Add-on	465
11.17. Vaadin Spring Add-on	474

This chapter covers various features and topics often needed in applications.

11.1. Handling Browser Windows

The UI of a Vaadin application runs in a web page displayed in a browser window or tab. An application can be used from multiple UIs in different windows or tabs, either opened by the user using an URL or by the Vaadin application.

In addition to native browser windows, Vaadin has a **Window** component, which is a floating panel or *sub-window* inside a page, as described in Section 7.7, "Sub-Windows".

- *Native popup windows.* An application can open popup windows for sub-tasks.
- *Page-based browsing.* The application can allow the user to open certain content to different windows. For example, in a messaging application, it can be useful to open different messages to different windows so that the user can browse through them while writing a new message.
- *Bookmarking.* Bookmarks in the web browser can provide an entry-point to some content provided by an application.
- *Embedding UIs.* UIs can be embedded in web pages, thus making it possible to provide different views to an application from different pages or even from the same page, while keeping the same session. See Section 11.2, "Embedding UIs in Web Pages".

Use of multiple windows in an application may require defining and providing different UIs for the different windows. The UIs of an application share the same user session, that is, the **VaadinSession** object, as described in Section 5.8.3, "User Session". Each UI is identified by a URL that is used to access it, which makes it possible to bookmark application UIs. UI instances can even be created dynamically based on the URLs or other request parameters, such as browser information, as described in Section 5.8.4, "Loading a UI".

Because of the special nature of AJAX applications, use of multiple windows uses require some caveats.

11.1.1. Opening Pop-up Windows

Pop-up windows are native browser windows or tabs opened by user interaction with an existing window. Due to browser security reasons, it is made inconvenient for a web page to open popup windows using JavaScript commands. At the least, the browser will ask for a permission to open the popup, if it is possible at all. This limitation can be circumvented by letting the browser open the new window or tab directly by its URL when the user clicks some target. This is realized in Vaadin with the **BrowserWindowOpener** component extension, which causes the browser to open a window or tab when the component is clicked.

The Pop-up Window UI

A popup Window displays an **UI**. The UI of a popup window is defined just like a main UI in a Vaadin application, and it can have a theme, title, and so forth.

For example:

```
@Theme("mytheme")
public static class MyPopupUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        getPage().setTitle("Popup Window");

        // Have some content for it
        VerticalLayout content = new VerticalLayout();
        Label label =
            new Label("I just popped up to say hi!");
        label.setSizeUndefined();
        content.addComponent(label);
        content.setComponentAlignment(label,
            Alignment.MIDDLE_CENTER);
        content.setSizeFull();
        setContent(content);
    }
}
```

Popping It Up

A popup window is opened using the **BrowserWindowOpener** extension, which you can attach to any component. The

constructor of the extension takes the class object of the UI class to be opened as a parameter.

You can configure the features of the popup window with `setFeatures()`. It takes as its parameter a comma-separated list of window features, as defined in the HTML specification.

`status=0/1`

Whether the status bar at the bottom of the window should be enabled.

`[parameter]##, scrollbars`

Enables scrollbars in the window if the document area is bigger than the view area of the window.

`resizable`

Allows the user to resize the browser window (no effect for tabs).

`menubar`

Enables the browser menu bar.

`location`

Enables the location bar.

`toolbar`

Enables the browser toolbar.

`height=value`

Specifies the height of the window in pixels.

`width=value`

Specifies the width of the window in pixels.

For example:

// Create an opener extension

```
BrowserWindowOpener opener =  
    new BrowserWindowOpener(MyPopupUI.class);  
opener.setFeatures("height=200,width=300,resizable");
```

// Attach it to a button

```
Button button = new Button("Pop It Up");  
opener.extend(button);
```

The resulting popup window, which appears when the button is clicked, is shown in Figure 11.1, "A Popup Window".

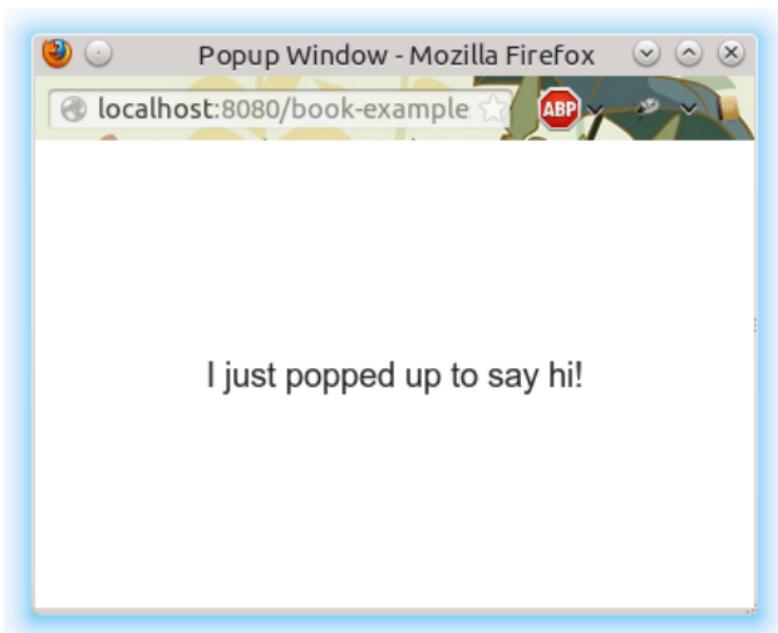


Figure 11.1. A Popup Window

Popup Window Name (Target)

The target name is one of the default HTML target names (`_new`, `_blank`, `_top`, etc.) or a custom target name. How the window is exactly opened depends on the browser. Browsers that support tabbed browsing can open the window in another tab, depending on the browser settings.

URL and Session

The URL path for a popup window UI is by default determined from the UI class name, by prefixing it with "popup/". For example, for the UI given earlier, the URL would be `/example/base/popup/MyPopupUI`.

11.1.2. Closing Popup Windows

Besides closing popup windows from a native window close button, you can close them programmatically by calling the JavaScript `close()` method as follows.

```
public class MyPopup extends UI {  
    @Override  
    protected void init(VaadinRequest request) {  
        setContent(new Button("Close Window", event -> {  
            // Java 8  
            // Close the popup  
            JavaScript.eval("close()");  
  
            // Detach the UI from the session  
            getUI().close();  
        }));  
    }  
}
```

11.2. Embedding UIs in Web Pages

Many web sites are not all Vaadin, but Vaadin UIs are used only for specific functionalities. In practice, many web applications are a mixture of dynamic web pages, such as JSP, and Vaadin UIs embedded in such pages.

Embedding Vaadin UIs in web pages is easy and there are several different ways to embed them. One is to have a `<div>` placeholder for the UI and load the Vaadin Client-Side Engine with some simple JavaScript code. Another method is even easier, which is to simply use the `<iframe>` element. Both of these methods have advantages and disadvantages. One disadvantage of the `<iframe>` method is that the size of the `<iframe>` element is not flexible according to the content while the `<div>` method allows such flexibility. The following sections look closer into these two embedding methods.

11.2.1. Embedding Inside a `div` Element

You can embed one or more Vaadin UIs inside a web page with a method that is equivalent to loading the initial page content from the Vaadin servlet in a non-embedded UI. Normally, the **VaadinServlet** generates an initial page that contains the correct parameters for the specific UI. You can easily configure it to load multiple Vaadin UIs in the same page. They can have different widget sets and different themes.

Embedding an UI requires the following basic tasks:

- Set up the page header
- Call the vaadinBootstrap.js file
- Define the <div> element for the UI
- Configure and initialize the UI

Notice that you can view the loader page for the UI easily by opening the UI in a web browser and viewing the HTML source code of the page. You could just copy and paste the embedding code from the page, but some modifications and additional settings are required, mainly related to the URLs that have to be made relative to the page instead of the servlet URL.

11.2.2. Embedding Inside an iframe Element

Embedding a Vaadin UI inside an <iframe> element is even easier than the method described above, as it does not require definition of any Vaadin specific definitions.

You can embed an UI with an element such as the following:

```
<iframe src="/myapp/myui"></iframe>
```

The <iframe> elements have several downsides for embedding. One is that their size of is not flexible depending on the content of the frame, but the content must be flexible to accommodate in the frame. You can set the size of an <iframe> element with height and width attributes. Other issues arise from themeing and communication with the frame content and the rest of the page.

Below is a complete example of using the <iframe> to embed two applications in a web page.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Embedding in IFrame</title>
  </head>

  <body style="background: #d0ffd0;">
```

```
<h1>This is a HTML page</h1>
<p>Below are two Vaadin applications embedded inside
a table:</p>

<table align="center" border="3">
<tr>
<th>The Calculator</th>
<th>The Color Picker</th>
</tr>
<tr valign="top">
<td>
<iframe src="/vaadin-examples/Calc" height="200"
width="150" frameborder="0"></iframe>
</td>
<td>
<iframe src="/vaadin-examples/colorpicker"
height="330" width="400"
frameborder="0"></iframe>
</td>
</tr>
</table>
</body>
</html>
```

The page will look as shown in Figure 11.2, "Vaadin Applications Embedded Inside IFrames" below.

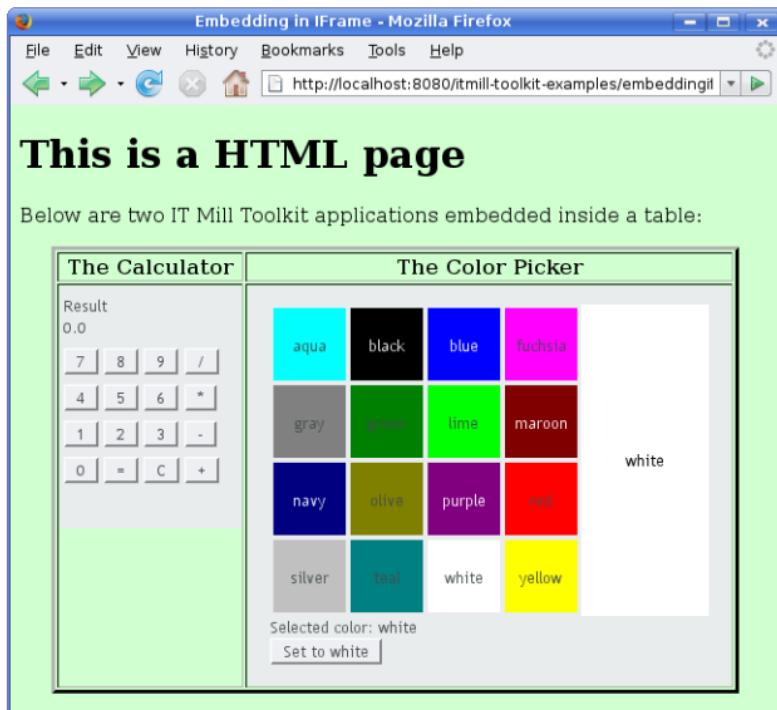


Figure 11.2. Vaadin Applications Embedded Inside Iframes

You can embed almost anything in an iframe, which essentially acts as a browser window. However, this creates various problems. The iframe must have a fixed size, inheritance of CSS from the embedding page is not possible, and neither is interaction with JavaScript, which makes mashups impossible, and so on. Even bookmarking with URI fragments will not work.

Note also that websites can forbid iframe embedding by specifying an X-Frame-Options: SAMEORIGIN header in the HTTP response.

11.3. Debug Mode and Window

Vaadin applications can be run in two modes: *debug mode* and *production mode*. The debug mode, which is on by default, enables a number of built-in debug features for Vaadin developers:

- Debug Window
- Display debug information in the Debug Window and server console
- On-the-fly compilation of Sass themes
- Timings of server calls for Vaadin TestBench

It is recommended to always deploy production applications in production mode for security reasons.

11.3.1. Enabling the Debug Mode

The debug mode is enabled and production mode disabled by default in the UI templates created with the Eclipse plugin or the Maven archetypes. Some archetypes have a separate module and profile for producing a production mode application. The debug mode can be enabled by giving a *productionMode=false* parameter to the Vaadin servlet configuration:

```
@VaadinServletConfiguration(  
    productionMode = false,  
    ui = MyprojectUI.class)
```

Or with a context parameter in the web.xml deployment descriptor:

```
<context-param>  
    <description>Vaadin production mode</description>  
    <param-name>productionMode</param-name>  
    <param-value>false</param-value>  
</context-param>
```

Enabling the production mode disables the debug features, thereby preventing users from easily inspecting the inner workings of the application from the browser.

11.3.2. Opening the Debug Window

Running an application in the debug mode enables the client-side Debug Window in the browser. You can open the Debug Window by adding " ?debug" parameter to the URL of the UI, for example, <http://localhost:8080/myapp/?debug>.

The Debug Window has buttons for controlling the debugging features and a scrollable log of debug messages.

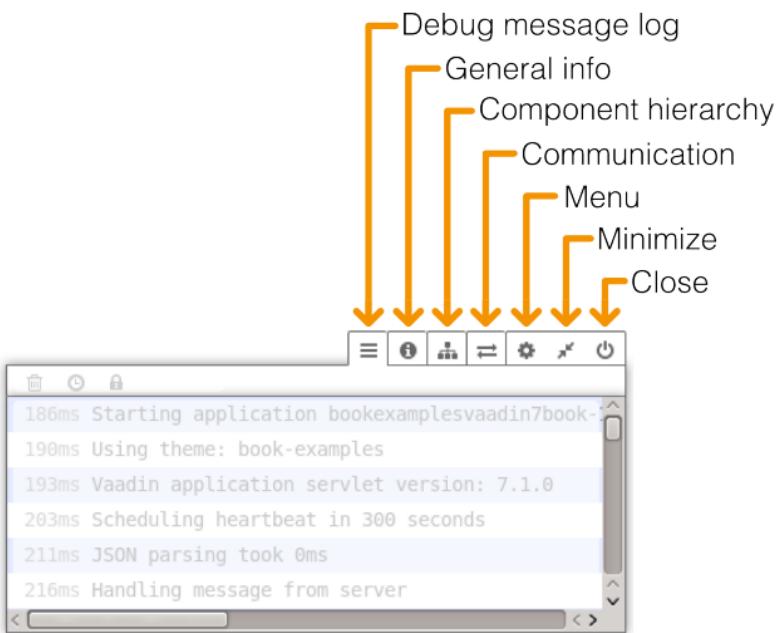


Figure 11.3. Debug Window

The functionalities are described in detail in the subsequent sections. You can move the window by dragging it from the title bar and resize it from the corners. The **Minimize** button minimizes the debug window in the corner of the browser window, and the **Close** button closes it.

If you use the Firebug plugin for Firefox or the Developer Tools console in Chrome, the log messages will also be printed to the inspector console. In such a case, you may want to enable client-side debugging without showing the Debug Window with "?debug=quiet" in the URL. In the quiet debug mode, log messages will only be printed to the console of the browser debugger.

11.3.3. Debug Message Log

The debug message log displays client-side debug messages, with time counter in milliseconds. The control buttons allow you to clear the log, reset the timer, and lock scrolling.

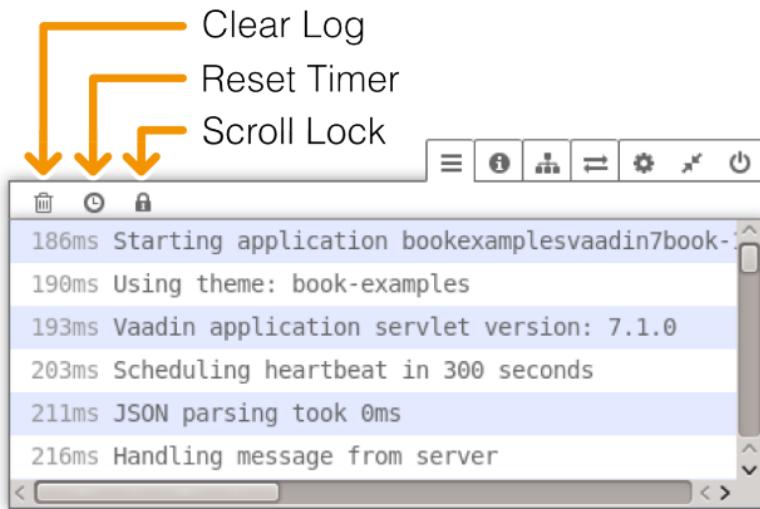


Figure 11.4. Debug Message Log

Logging to Debug Window

You can take advantage of the debug mode when developing client-side components, by using the standard Java **Logger** to write messages to the log. The messages will be written to the debug window and Firebug console. No messages are written if the debug window is not open or if the application is running in production mode.

11.3.4. General Information

The **General information about the application(s)** tab displays various information about the UI, such as version numbers of the client and servlet engine, and the theme. If they do not match, you may need to compile the widget set or theme.

General information about the application(s)	
Client engine version	7.1.0
Server engine version	7.1.0
Theme version	7.1.0
Widget set	com.vaadin.book.widgetset.BookExamplesWidgetSet
Theme	book-examples
Communication method	XHR
Heartbeat	300s

Figure 11.5. General Information

11.3.5. Inspecting Component Hierarchy

The **Component Hierarchy** tab has several sub-modes that allow debugging the component tree in various ways.

Connector Hierarchy Tree

The **Show the connector hierarchy tree** button displays the client-side connector hierarchy. As explained in Chapter 16, *Integrating with the Server-Side*, client-side widgets are managed by connectors that handle communication with the server-side component counterparts. The connector hierarchy therefore corresponds with the server-side component tree, but the client-side widget tree and HTML DOM tree have more complexity.

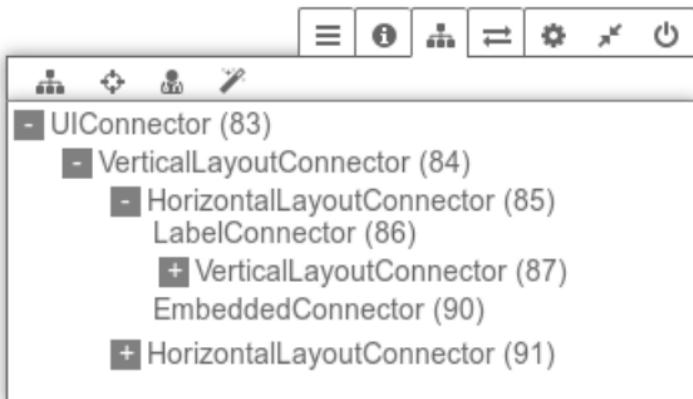


Figure 11.6. Connector Hierarchy Tree

Clicking on a connector highlights the widget in the UI.

Inspecting a Component

The **Select a component in the page to inspect it** button lets you select a component in the UI by clicking it and display its client-side properties.

To view the HTML structure and CSS styles in more detail, you can use Inspector in Firefox or the Developer Tools in Chrome.

Analyzing Layout Problems

The **Check layouts for potential problems** button analyzes the currently visible UI and makes a report of possible layout related problems. All detected layout problems are displayed in the log and also printed to the console.



Figure 11.7. Debug window showing the result of layout analysis.

Clicking on a reported problem highlights the component with the problem in the UI.

The most common layout problem is caused by placing a component that has a relative size inside a container (layout) that has undefined size in the particular direction (height or width). For example, adding a **Button** with 100% width inside a **VerticalLayout** with undefined width. In such a case, the error would look as shown in Figure 11.7, "Debug window showing the result of layout analysis.".

CustomLayout components can not be analyzed in the same way as other layouts. For custom layouts, the button analyzes all contained relative-sized components and checks if any relative dimension is calculated to zero so that the component will be invisible. The error log will display a warning for each of these invisible components. It would not be meaningful to emphasize the component itself as it is not visible, so when you select such an error, the parent layout of the component is emphasized if possible.

Displaying Used Connectors

The last button, **Show used connectors and how to optimize widget set**, displays a list of all currently visible connectors. It

also generates a connector bundle loader factory, which you can use to optimize the widget set so that it only contains the widgets actually used in the UI. Note, however, that it only lists the connectors visible in the current UI state, and you usually have more connectors than that.

11.3.6. Communication Log

The **Communication** tab displays all server requests. You can unfold the requests to view details, such as the connectors involved. Clicking on a connector highlights the corresponding element in the UI.

You can use Firebug or Developer Tools in Firefox or Chrome, respectively, to get more detailed information about the requests and responses.

11.3.7. Debug Modes

The **Menu** tab in the window opens a sub-menu to select various settings. Here you can also launch the GWT SuperDev-Mode, as described in Section 13.6, “Debugging Client-Side Code”.

11.4. Request Handlers

Request handlers are useful for catching request parameters or generating dynamic content, such as HTML, images, PDF, or other content. You can provide HTTP content also with stream resources, as described in Section 5.5.5, “Stream Resources”. The stream resources, however, are only usable from within a Vaadin application, such as in an **Image** component. Request handlers allow responding to HTTP requests made with the application URL, including GET or POST parameters. You could also use a separate servlet to generate dynamic content, but a request handler is associated with the user session and it can easily access data associated with the session and the user.

To handle requests, you need to implement the `RequestHandler` interface. The `handleRequest()` method gets the session, request, and response objects as parameters.

If the handler writes a response, it must return true. This stops running other possible request handlers. Otherwise, it should return false so that another handler could return a response. Eventually, if no other handler writes a response, a UI will be created and initialized.

In the following example, we catch requests for a sub-path in the URL for the servlet and write a plain text response. The servlet path consists of the context path and the servlet (sub-)path. Any additional path is passed to the request handler in the *pathInfo* of the request. For example, if the full path is /myapp/myui/rexample, the path info will be /rexample. Also, request parameters are available.

```
// A request handler for generating some content
VaadinSession.getCurrent().addRequestHandler(
    new RequestHandler() {
        @Override
        public boolean handleRequest(VaadinSession session,
                                     VaadinRequest request,
                                     VaadinResponse response)
            throws IOException {
            if ("/rexample".equals(request.getPathInfo())) {
                // Generate a plain text document
                response.setContentType("text/plain");
                response.getWriter().append(
                    "Here's some dynamically generated content.\n");
                response.getWriter().format(Locale.ENGLISH,
                    "Time: %T\n", new Date());

                // Use shared session data
                response.getWriter().format("Session data: %s\n",
                    session.getAttribute("mydata"));

                return true; // We wrote a response
            } else
                return false; // No response was written
        }
    });
});
```

A request handler can be used by embedding it in a page or opening a new page with a link or a button. In the following example, we pass some data to the handler through a session attribute.

```
// Input some shared data in the session
TextField dataInput = new TextField("Some data");
dataInput.addValueChangeListener(event ->
    VaadinSession.getCurrent().setAttribute("mydata",
        event.getProperty().getValue()));
```

```
dataInput.setValue("Here's some");

// Determine the base path for the servlet
String servletPath = VaadinServlet.getCurrent()
    .getServletContext().getContextPath()
    + "/book"; // Servlet

// Display the page in a pop-up window
Link open = new Link("Click to Show the Page",
    new ExternalResource(servletPath + "/rhexample"),
    "_blank", 500, 350, BorderStyle.DEFAULT);

layout.addComponents(dataInput, open);
```

11.5. Shortcut Keys

Vaadin provides simple ways to define shortcut keys for field components, as well as to a default button, and a lower-level generic shortcut API based on actions.

A *shortcut* is an action that is executed when a key or key combination is pressed within a specific scope in the UI. The scope can be the entire **UI** or a **Window** inside it.

11.5.1. Shortcut Keys for Default Buttons

You can add a *click shortcut* to a button to set it as "default" button; pressing the defined key, typically **Enter**, in any component in the scope (sub-window or UI) causes a click event for the button to be fired.

You can define a click shortcut with the `setClickShortcut()` shorthand method:

```
// Have an OK button and set it as the default button
Button ok = new Button("OK");
ok.setClickShortcut(KeyCode.ENTER);
ok.addStyleName(ValoTheme.BUTTON_PRIMARY);
```

The `setClickShortcut()` is a shorthand method to create, add, and manage a **ClickShortcut**, rather than to add it with `addShortcutListener()`.

Themes offer special button styles to show that a button is special. In the Valo theme, you can use the `BUTTON_PRIMARY`

style name. The result can be seen in Figure 11.8, "Default Button with Click Shortcut".

Name of the Field Agent

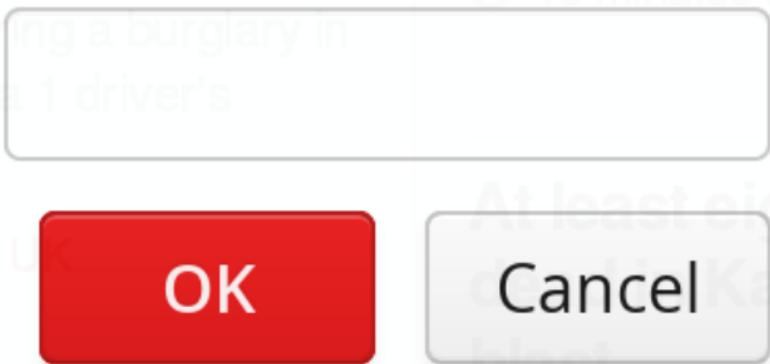


Figure 11.8. Default Button with Click Shortcut

11.5.2. Field Focus Shortcuts

You can define a shortcut key that sets the focus to a field component (any component that inherits **AbstractField**) by adding a **FocusShortcut** as a shortcut listener to the field.

The constructor of the **FocusShortcut** takes the field component as its first parameter, followed by the key code, and an optional list of modifier keys, as listed in Section 11.5.4, "Supported Key Codes and Modifier Keys".

```
// A field with Alt+N bound to it
TextField name = new TextField("Name (Alt+N)");
name.addShortcutListener(
    new AbstractField.FocusShortcut(name, KeyCode.N,
        ModifierKey.ALT));
layout.addComponent(name);
```

You can also specify the shortcut by a shorthand notation, where the shortcut key is indicated with an ampersand (&).

```
// A field with Alt+A bound to it, using shorthand notation
TextField address = new TextField("Address (Alt+A)");
address.addShortcutListener(
    new AbstractField.FocusShortcut(address, "&Address"));
```

This is especially useful for internationalization, so that you can determine the shortcut key from the localized string.

11.5.3. Generic Shortcut Actions

Shortcut keys can be defined as *actions* using the **ShortcutAction** class. It extends the generic **Action** class that is used for example in **Tree** and **Table** for context menus. Currently, the only classes that accept **ShortcutActions** are **Window** and **Panel**.

To handle key presses, you need to define an action handler by implementing the **Handler** interface. The interface has two methods that you need to implement: `getActions()` and `handleAction()`.

The `getActions()` method must return an array of **Action** objects for the component, specified with the second parameter for the method, the *sender* of an action. For a keyboard shortcut, you use a **ShortcutAction**. The implementation of the method could be following:

```
// Have the unmodified Enter key cause an event
Action action_ok = new ShortcutAction("Default key",
    ShortcutAction.KeyCode.ENTER, null);

// Have the C key modified with Alt cause an event
Action action_cancel = new ShortcutAction("Alt+C",
    ShortcutAction.KeyCode.C,
    new int[] { ShortcutAction.ModifierKey.ALT });

Action[] actions = new Action[] {action_cancel, action_ok};

public Action[] getActions(Object target, Object sender) {
    if (sender == myPanel)
        return actions;
    return null;
}
```

The returned **Action** array may be static or you can create it dynamically for different senders according to your needs.

The constructor of **ShortcutAction** takes a symbolic caption for the action; this is largely irrelevant for shortcut actions in their current implementation, but might be used later if implementors use them both in menus and as shortcut actions.

The second parameter is the key code and the third a list of modifier keys, which are listed in Section 11.5.4, "Supported Key Codes and Modifier Keys".

The following example demonstrates the definition of a default button for a user interface, as well as a normal shortcut key, **Alt+C** for clicking the **Cancel** button.

```
public class DefaultButtonExample extends CustomComponent
    implements Handler {
    // Define and create user interface components
    Panel panel = new Panel("Login");
    FormLayout formlayout = new FormLayout();
    TextField username = new TextField("Username");
    TextField password = new TextField("Password");
    HorizontalLayout buttons = new HorizontalLayout();

    // Create buttons and define their listener methods.
    Button ok = new Button("OK", event -> okHandler());
    Button cancel = new Button("Cancel", event -> cancelHandler());

    // Have the unmodified Enter key cause an event
    Action action_ok = new ShortcutAction("Default key",
        ShortcutAction.KeyCode.ENTER, null);

    // Have the C key modified with Alt cause an event
    Action action_cancel = new ShortcutAction("Alt+C",
        ShortcutAction.KeyCode.C,
        new int[] { ShortcutAction.ModifierKey.ALT });

    public DefaultButtonExample() {
        // Set up the user interface
        setCompositionRoot(panel);
        panel.addComponent(formlayout);
        formlayout.addComponent(username);
        formlayout.addComponent(password);
        formlayout.addComponent(buttons);
        buttons.addComponent(ok);
        buttons.addComponent(cancel);

        // Set focus to username
        username.setFocus();

        // Set this object as the action handler
        panel.addActionHandler(this);
    }

    /**
     * Retrieve actions for a specific component. This method
     * will be called for each object that has a handler; in
     * this example just for login panel. The returned action
     * list might as well be static list.
     */
    public Action[] getActions(Object target, Object sender) {
        System.out.println("getActions()");
        return new Action[] { action_ok, action_cancel };
    }
}
```

```


    /**
     * Handle actions received from keyboard. This simply directs
     * the actions to the same listener methods that are called
     * with ButtonClick events.
    */
    public void handleAction(Action action, Object sender,
        Object target) {
        if (action == action_ok) {
            okHandler();
        }
        if (action == action_cancel) {
            cancelHandler();
        }
    }

    public void okHandler() {
        // Do something: report the click
        formLayout.addComponent(new Label("OK clicked. "
            + "User=" + username.getValue() + ". password="
            + password.getValue()));
    }

    public void cancelHandler() {
        // Do something: report the click
        formLayout.addComponent(new Label("Cancel clicked. User="
            + username.getValue() + ". password="
            + password.getValue()));
    }
}


```

Notice that the keyboard actions can currently be attached only to **Panels** and **Windows**. This can cause problems if you have components that require a certain key. For example, multi-line **TextField** requires the **Enter** key. There is currently no way to filter the shortcut actions out while the focus is inside some specific component, so you need to avoid such conflicts.

11.5.4. Supported Key Codes and Modifier Keys

The shortcut key definitions require a key code to identify the pressed key and modifier keys, such as **Shift**, **Alt**, or **Ctrl**, to specify a key combination.

The key codes are defined in the **ShortcutAction.KeyCode** interface and are:

Keys A to Z

Normal letter keys

F1 to F12

Function keys

BACKSPACE, DELETE, ENTER, ESCAPE, INSERT, TAB
Control keys

NUM0 to NUM9
Number pad keys

ARROW_DOWN, ARROW_UP, ARROW_LEFT, ARROW_RIGHT
Arrow keys

HOME, END, PAGE_UP, PAGE_DOWN
Other movement keys

Modifier keys are defined in **ShortcutAction.ModifierKey** and are:

ModifierKey.ALT
Alt key

ModifierKey.CTRL
Ctrl key

ModifierKey.SHIFT
Shift key

All constructors and methods accepting modifier keys take them as a variable argument list following the key code, separated with commas. For example, the following defines a **Ctrl+Shift+N** key combination for a shortcut.

```
TextField name = new TextField("Name (Ctrl+Shift+N)");
name.addShortcutListener(
    new AbstractField.FocusShortcut(name, KeyCode.N,
        ModifierKey.CTRL,
        ModifierKey.SHIFT));
```

Supported Key Combinations

The actual possible key combinations vary greatly between browsers, as most browsers have a number of built-in shortcut keys, which can not be used in web applications. For example, Mozilla Firefox allows binding almost any key combination, while Opera does not even allow binding **Alt** shortcuts. Other browsers are generally in between these two. Also, the operating system can reserve some key combinations and some computer manufacturers define their own system key combinations.

11.6. Printing

Vaadin does not have any special support for printing. There are two basic ways to print - in a printer controlled by the application server or by the user from the web browser. Printing in the application server is largely independent of the UI, you just have to take care that printing commands do not block server requests, possibly by running the print commands in another thread.

For client-side printing, most browsers support printing the web page. You can either print the current or a special print page that you open. The page can be styled for printing with special CSS rules, and you can hide unwanted elements. You can also print other than Vaadin UI content, such as HTML or PDF.

11.6.1. Printing the Browser Window

Vaadin does not have special support for launching the printing in browser, but you can easily use the JavaScript print() method that opens the print window of the browser.

```
Button print = new Button("Print This Page");
print.addClickListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        //Print the current page
        JavaScript.getCurrent().execute("print()");
    }
});
```

The button in the above example would print the current page, including the button itself. You can hide such elements in CSS, as well as otherwise style the page for printing. Style definitions for printing are defined inside a @media print {} block in CSS.

11.6.2. Opening a Print Window

You can open a browser window with a special UI for print content and automatically launch printing the content.

```
public static class PrintUI extends UI {
    @Override
```

```

protected void init(VaadinRequest request) {
    // Have some content to print
    setContent(new Label(
        "<h1>Here's some dynamic content</h1>" + 
        "<p>This is to be printed.</p>",
        ContentMode.HTML));

    // Print automatically when the window opens
    JavaScript.getCurrent().execute(
        "setTimeout(function() {" +
            " print(); self.close(); }, 0);");
}

}

...
// Create an opener extension
BrowserWindowOpener opener =
    new BrowserWindowOpener(PrintUI.class);
opener.setFeatures("height=200,width=400,resizable");

// A button to open the printer-friendly page.
Button print = new Button("Click to Print");
opener.extend(print);

```

How the browser opens the window, as an actual (popup) window or just a tab, depends on the browser. After printing, we automatically close the window with JavaScript close() call.

11.6.3. Printing PDF

To print content as PDF, you need to provide the downloadable content as a static or a dynamic resource, such as a **StreamResource**.

You can let the user open the resource using a **Link** component, or some other component with a **PopupWindowOpener** extension. When such a link or opener is clicked, the browser opens the PDF in the browser, in an external viewer (such as Adobe Reader), or lets the user save the document.

It is crucial to notice that clicking a **Link** or a **PopupWindowOpener** is a client-side operation. If you get the content of the dynamic PDF from the same UI state, you can not have the link or opener enabled, as then clicking it would not get the current UI content. Instead, you have to create the resource object before the link or opener are clicked. This usually

requires a two-step operation, or having the print operation available in another view.

```
// A user interface for a (trivial) data model from which
// the PDF is generated.
final TextField name = new TextField("Name");
name.setValue("Slartibartfast");

// This has to be clicked first to create the stream resource
final Button ok = new Button("OK");

// This actually opens the stream resource
final Button print = new Button("Open PDF");
print.setEnabled(false);

ok.addClickListener(new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        // Create the PDF source and pass the data model to it
        StreamSource source =
            new MyPdfSource((String) name.getValue());

        // Create the stream resource and give it a file name
        String filename = "pdf_printing_example.pdf";
        StreamResource resource =
            new StreamResource(source, filename);

        // These settings are not usually necessary. MIME type
        // is detected automatically from the file name, but
        // setting it explicitly may be necessary if the file
        // suffix is not ".pdf".
        resource.setMIMEType("application/pdf");
        resource.getStream().setParameter(
            "Content-Disposition",
            "attachment; filename="+filename);

        // Extend the print button with an opener
        // for the PDF resource
        BrowserWindowOpener opener =
            new BrowserWindowOpener(resource);
        opener.extend(print);

        name.setEnabled(false);
        ok.setEnabled(false);
        print.setEnabled(true);
    }
});

layout.addComponent(name);
layout.addComponent(ok);
layout.addComponent(print);
```

11.7. Common Security Issues

11.7.1. Sanitizing User Input to Prevent Cross-Site Scripting

You can put raw HTML content in many components, such as the **Label** and **CustomLayout**, as well as in tooltips and notifications. In such cases, you should make sure that if the content has any possibility to come from user input, you must make sure that the content is safe before displaying it. Otherwise, a malicious user can easily make a cross-site scripting attack by injecting offensive JavaScript code in such components. See other sources for more information about cross-site scripting.

Offensive code can easily be injected with `<script>` markup or in tag attributes as events, such as `onLoad`.

Cross-site scripting vulnerabilities are browser dependent, depending on the situations in which different browsers execute scripting markup.

Therefore, if the content created by one user is shown to other users, the content must be sanitized. There is no generic way to sanitize user input, as different applications can allow different kinds of input. Pruning (X)HTML tags out is somewhat simple, but some applications may need to allow (X)HTML content. It is therefore the responsibility of the application to sanitize the input.

Character encoding can make sanitization more difficult, as offensive tags can be encoded so that they are not recognized by a sanitizer. This can be done, for example, with HTML character entities and with variable-width encodings such as UTF-8 or various CJK encodings, by abusing multiple representations of a character. Most trivially, you could input `<` and `>` with `<` and `>`, respectively. The input could also be malformed and the sanitizer must be able to interpret it exactly as the browser would, and different browsers can interpret malformed HTML and variable-width character encodings differently.

Notice that the problem applies also to user input from a **RichTextArea** is transmitted as HTML from the browser to server-side and is not sanitized. As the entire purpose of the **RichTextArea** component is to allow input of formatted text, you can not just remove all HTML tags. Also many attributes, such as *style*, should pass through the sanitization.

11.8. Navigating in an Application

Plain Vaadin applications do not have normal web page navigation as they usually run on a single page, as all Ajax applications do. Quite commonly, however, applications have different views between which the user should be able to navigate. The **Navigator** in Vaadin can be used for most cases of navigation. Views managed by the navigator automatically get a distinct URI fragment, which can be used to be able to bookmark the views and their states and to go back and forward in the browser history.

11.8.1. Setting Up for Navigation

The **Navigator** class manages a collection of *views* that implement the *View* interface. The views can be either registered beforehand or acquired from a *view provider*. When registering, the views must have a name identifier and be added to a navigator with `addView()`. You can register new views at any point. Once registered, you can navigate to them with `navigateTo()`.

Navigator manages navigation in a component container, which can be either a `ComponentContainer` (most layouts) or a `SingleComponentContainer` (**UI**, **Panel**, or **Window**). The component container is managed through a `ViewDisplay`. Two view displays are defined: `ComponentContainerViewDisplay` and `SingleComponentContainerViewDisplay`, for the respective component container types. Normally, you can let the navigator create the view display internally, as we do in the example below, but you can also create it yourself to customize it.

Let us consider the following UI with two views: `start` and `main`. Here, we define their names with enums to be typesafe. We

manage the navigation with the UI class itself, which is a SingleComponentContainer.

```
public class NavigatorUI extends UI {  
    Navigator navigator;  
    protected static final String MAINVIEW = "main";  
  
    @Override  
    protected void init(VaadinRequest request) {  
        getPage().setTitle("Navigation Example");  
  
        // Create a navigator to control the views  
        navigator = new Navigator(this, this);  
  
        // Create and register the views  
        navigator.addView("", new StartView());  
        navigator.addView(MAINVIEW, new MainView());  
    }  
}
```

The **Navigator** automatically sets the URI fragment of the application URL. It also registers a UriFragmentChangedListener in the page

to show the view identified by the URI fragment if entered or navigated to in the browser. This also enables browser navigation history in the application.

View Providers

You can create new views dynamically using a *view provider* that implements the ViewProvider interface. A provider is registered in **Navigator** with addProvider().

The ClassBasedViewProvider is a view provider that can dynamically create new instances of a specified view class based on the view name.

The StaticViewProvider returns an existing view instance based on the view name. The addView() in **Navigator** is actually just a shorthand for creating a static view provider for each registered view.

View Change Listeners

You can handle view changes also by implementing a ViewChangeListener and adding it to a **Navigator**. When a view change occurs, a listener receives a **ViewChangeEvent** object, which has references to the old and the activated view, the name of the activated view, as well as the fragment parameters.

11.8.2. Implementing a View

Views can be any objects that implement the View interface. When the `navigateTo()` is called for the navigator, or the application is opened with the URI fragment associated with the view, the navigator switches to the view and calls its `enter()` method.

To continue with the example, consider the following simple start view that just lets the user to navigate to the main view. It only pops up a notification when the user navigates to it and displays the navigation button.

```
/** A start view for navigating to the main view */
public class StartView extends VerticalLayout implements View {
    public StartView() {
        setSizeFull();

        Button button = new Button("Go to Main View",
            new Button.ClickListener() {
                @Override
                public void buttonClick(ClickEvent event) {
                    navigator.navigateTo(MAINVIEW);
                }
            });
        addComponent(button);
        setComponentAlignment(button, Alignment.MIDDLE_CENTER);
    }

    @Override
    public void enter(ViewChangeEvent event) {
        Notification.show("Welcome to the Animal Farm");
    }
}
```

You can initialize the view content in the constructor, as was done in the example above, or in the `enter()` method. The advantage with the latter method is that the view is attached to the view container as well as to the UI at that time, which is not the case in the constructor.

11.8.3. Handling URI Fragment Path

URI fragment part of a URL is the part after a hash # character. Is used for within-UI URLs, because it is the only part of the URL that can be changed with JavaScript from within a page without reloading the page. The URLs with URI fragments can be used for hyperlinking and bookmarking, as well as browser history, just like any other URLs. In addition, an exclamation mark #! after the hash marks that the page is a stateful AJAX page, which can be crawled by search engines. Crawling requires that the application also responds to special URLs to get the searchable content. URI fragments are managed by **Page**, which provides a low-level API.

URI fragments can be used with **Navigator** in two ways: for navigating to a view and to a state within a view. The URI fragment accepted by `navigateTo()` can have the view name at the root, followed by fragment parameters after a slash (" /"). These parameters are passed to the `enter()` method in the View.

In the following example, we implement within-view navigation. Here we use the following declarative design for the view:

```
<vaadin-vertical-layout size-full>
<vaadin-horizontal-layout size-full :expand>
  <vaadin-panel caption="List of Equals" height-full width-auto>
    <vaadin-vertical-layout _id="menuContent" width-auto margin/>
  </vaadin-panel>

  <vaadin-panel _id="equalPanel" caption="An Equal" size-full :expand/>
</vaadin-horizontal-layout>

<vaadin-button _id="logout">Logout</vaadin-button>
</vaadin-vertical-layout>
```

The view's logic code would be as follows:

```
/** Main view with a menu (with declarative layout design) */
@DesignRoot
public class MainView extends VerticalLayout implements View {
    // Menu navigation button listener
    class ButtonListener implements Button.ClickListener {
        String menuitem;
        public ButtonListener(String menuitem) {
            this.menuitem = menuitem;
        }
        @Override
        public void buttonClick(ClickEvent event) {
            // Navigate to a specific state
```

```

        navigator.navigateTo(MAINVIEW + "/" + menuitem);
    }
}

VerticalLayout menuContent;
Panel equalPanel;
Button logout;

public MainView() {
    Design.read(this);

    menuContent.addComponent(new Button("Pig".
        new ButtonListener("pig")));
    menuContent.addComponent(new Button("Cat".
        new ButtonListener("cat")));
    menuContent.addComponent(new Button("Dog".
        new ButtonListener("dog")));
    menuContent.addComponent(new Button("Reindeer".
        new ButtonListener("reindeer")));
    menuContent.addComponent(new Button("Penguin".
        new ButtonListener("penguin")));
    menuContent.addComponent(new Button("Sheep".
        new ButtonListener("sheep")));

    //Allow going back to the start
    logout.addClickListener(event -> // Java 8
        navigator.navigateTo(""));
}

@DesignRoot
class AnimalViewer extends VerticalLayout {
    Label watching;
    Embedded pic;
    Label back;

    public AnimalViewer(String animal) {
        Design.read(this);

        watching.setValue("You are currently watching a " +
            animal);
        pic.setSource(new ThemeResource(
            "img/" + animal + "-128px.png"));
        back.setValue("and " + animal +
            " is watching you back");
    }
}

@Override
public void enterViewChangeEvent event) {
    if (event.getParameters() == null
        || event.getParameters().isEmpty()) {
        equalPanel.setContent(
            new Label("Nothing to see here. " +
                "just pass along."));
    return;
} else
    equalPanel.setContent(new AnimalViewer(

```

```
        event.getParameters().));  
    }  
}
```

The animal sub-view would have the following declarative design:

```
<vaadin-vertical-layout size-full>  
  <vaadin-label _id="watching" size-auto :middle :center/>  
  <vaadin-embedded _id="pic" :middle :center :expand/>  
  <vaadin-label _id="back" size-auto :middle :center/>  
</vaadin-vertical-layout>
```

The main view is shown in Figure 11.9, "Navigator Main View". At this point, the URL would be <http://localhost:8080/myapp#!main/reindeer>.



Figure 11.9. Navigator Main View

11.9. Advanced Application Architectures

In this section, we continue from the basic application architectures described in Section 5.2, "Building the UI" and discuss some of the more advanced patterns that are often used in Vaadin applications.

11.9.1. Layered Architectures

Layered architectures, where each layer has a clearly distinct responsibility, are probably the most common architectures. Typically, applications follow at least a three-layer architecture:

- User interface (or presentation) layer
- Domain layer
- Data store layer

Such an architecture starts from a *domain model*, which defines the data model and the "business logic" of the application, typically as beans or POJOs. A user interface is built on top of the domain model, in our context with the Vaadin Framework. The Vaadin user interface could be bound directly to the data model through the Vaadin Data Model, described in Chapter 10, *Binding Components to Data*. Beneath the domain model lies a data store, such as a relational database. The dependencies between the layers are restricted so that a higher layer may depend on a lower one, but never the other way around.

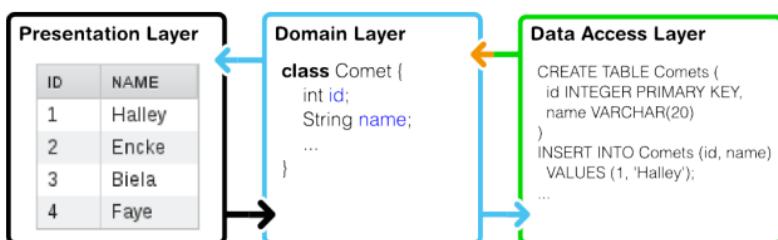


Figure 11.10. Three-layer architecture

An *application layer* (or *service layer*) is often distinguished from the domain layer, offering the domain logic as a service.

which can be used by the user interface layer, as well as for other uses. In Java EE development, Enterprise JavaBeans (EJBs) are typically used for building this layer.

11.9.2. Model-View-Presenter Pattern

The Model-View-Presenter (MVP) pattern is one of the most common patterns in developing large applications with Vaadin. It is similar to the older Model-View-Controller (MVC) pattern, which is not as meaningful in Vaadin development. Instead of an implementation-aware controller, there is an implementation-agnostic presenter that operates the view through an interface. The view does not interact directly with the model. This isolates the view implementation better than in MVC and allows easier unit testing of the presenter and model.

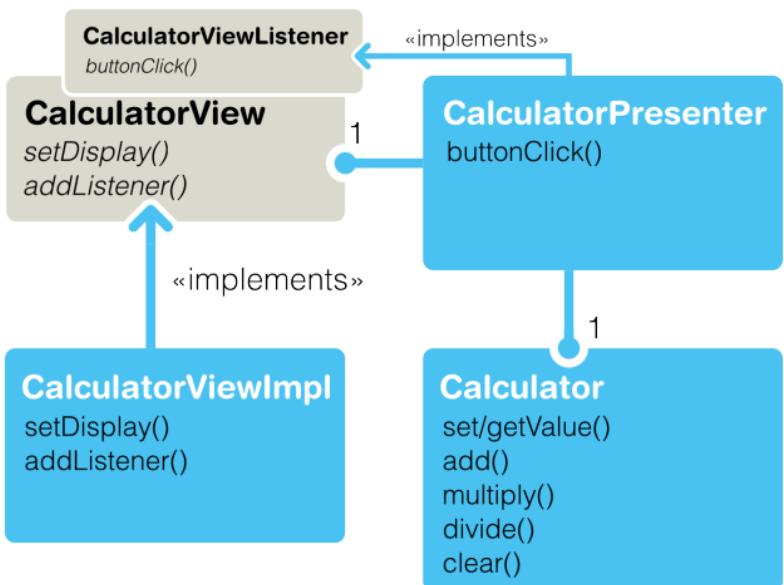


Figure 11.11. Model-View-Presenter pattern

Figure 11.11, “Model-View-Presenter pattern” illustrates the MVP pattern with a simple calculator. The domain model is realized in the **Calculator** class, which includes a data model and some model logic operations. The **CalculatorViewImpl** is a Vaadin implementation of the view, defined in the Calculator-

View interface. The **CalculatorPresenter** handles the user interface logic. User interaction events received in the view are translated into implementation-independent events for the presenter to handle (the view implementation could also just call the presenter).

Let us first look how the model and view are bound together by the presenter in the following example:

```
// Create the model and the Vaadin view implementation
CalculatorModel model = new CalculatorModel();
CalculatorViewImpl view = new CalculatorViewImpl();

// The presenter binds the model and view together
new CalculatorPresenter(model, view);

// The view implementation is a Vaadin component
layout.addComponent(view);
```

You could add the view anywhere in a Vaadin application, as it is a composite component.

The Model

Our business model is quite simple, with one value and a number of operations for manipulating it.

```
/** The model */
class CalculatorModel {
    private double value = 0.0;

    public void clear() {
        value = 0.0;
    }

    public void add(double arg) {
        value += arg;
    }

    public void multiply(double arg) {
        value *= arg;
    }

    public void divide(double arg) {
        if (arg != 0.0)
            value /= arg;
    }
}
```

```

public double getValue() {
    return value;
}

public void setValue(double value) {
    this.value = value;
}
}

```

The View

The purpose of the view in MVP is to display data and receive user interaction. It relays the user interaction to the presenter in a fashion that is independent of the view implementation, that is, no Vaadin events. It is defined as a UI framework interface that can have multiple implementations.

```

interface CalculatorView {
    public void setDisplay(double value);

    interface CalculatorViewListener {
        void buttonClick(char operation);
    }
    public void addListener(CalculatorViewListener listener);
}

```

There are design alternatives for the view. It could receive the listener in its constructor, or it could just know the presenter. Here, we forward button clicks as an implementation-independent event.

As we are using Vaadin, we make a Vaadin implementation of the interface as follows:

```

class CalculatorViewImpl extends CustomComponent
    implements CalculatorView, ClickListener {
    private Label display = new Label("0.0");

    public CalculatorViewImpl() {
        GridLayout layout = new GridLayout(4, 5);

        // Create a result label that spans over all
        // the 4 columns in the first row
        layout.addComponent(display, 0, 0, 3, 0);

        // The operations for the calculator in the order
        // they appear on the screen (left to right, top
        // to bottom)
        String[] operations = new String[] {

```

```

"7", "8", "9", "/", "4", "5", "6",
"\"", "1", "2", "3", "-", "0", "=", "C", "+"];

// Add buttons and have them send click events
// to this class
for (String caption: operations)
    layout.addComponent(new Button(caption, this));

setCompositionRoot(layout);
}

public void setDisplay(double value) {
    display.setValue(Double.toString(value));
}


List<CalculatorViewListener> listeners =
    new ArrayList<CalculatorViewListener>();

public void addListener(CalculatorViewListener listener) {
    listeners.add(listener);
}

/* Relay button clicks to the presenter with an
* implementation-independent event */
@Override
public void buttonClick(ClickEvent event) {
    for (CalculatorViewListener listener: listeners)
        listener.buttonClick(event.getButton()
            .getCaption().charAt(0));
}
}

```

The Presenter

The presenter in MVP is a middle-man that handles all user interaction logic, but in an implementation-independent way, so that it doesn't actually know anything about Vaadin. It shows data in the view and receives user interaction back from it.

```

class CalculatorPresenter
    implements CalculatorView.CalculatorViewListener {
    CalculatorModel model;
    CalculatorView view;

    private double current = 0.0;
    private char lastOperationRequested = 'C';

    public CalculatorPresenter(CalculatorModel model,
                                CalculatorView view) {
        this.model = model;
        this.view = view;
    }
}

```

```

        view.setDisplay(current);
        view.addListener(this);
    }

@Override
public void buttonClick(char operation) {
    // Handle digit input
    if ('0' <= operation && operation <= '9') {
        current = current * 10
            + Double.parseDouble("") + operation);
        view.setDisplay(current);
        return;
    }

    // Execute the previously input operation
    switch (lastOperationRequested) {
        case '+':
            model.add(current);
            break;
        case '-':
            model.add(-current);
            break;
        case '/':
            model.divide(current);
            break;
        case '*':
            model.multiply(current);
            break;
        case 'C':
            model.setValue(current);
            break;
    } // '=' is implicit

    lastOperationRequested = operation;

    current = 0.0;
    if (operation == 'C')
        model.clear();
    view.setDisplay(model.getValue());
}
}

```

In the above example, we held some state information in the presenter. Alternatively, we could have had an intermediate controller between the presenter and the model to handle the low-level button logic.

11.10. Managing URI Fragments

A major issue in AJAX applications is that as they run in a single web page, bookmarking the application URL (or more generally the *URI*) can only bookmark the application, not an application state. This is a problem for many applications, such as product catalogs and discussion forums, in which it would be good to provide links to specific products or messages. Consequently, as browsers remember the browsing history by URI, the history and the **Back** button do not normally work. The solution is to use the *fragment identifier* part of the URI, which is separated from the primary part (address + path + optional query parameters) of the URI with the hash (#) character. For example:

`http://example.com/path#myfragment`

The exact syntax of the fragment identifier part is defined in RFC 3986 (Internet standard STD 66) that defines the URI syntax. A fragment may only contain the regular URI *path characters* (see the standard) and additionally the slash and the question mark.

Vaadin offers two ways to enable the use of URI fragments: the high-level **Navigator** utility described in Section 11.8, "Navigating in an Application" and the low-level API described here.

11.10.1. Setting the URI Fragment

You can set the current fragment identifier with the `setUriFragment()` method in the **Page** object.

```
Page.getCurrent().setUriFragment("mars");
```

Setting the URI fragment causes an `UriFragmentChangeEvent`, which is processed in the same server request. As with UI rendering, the URI fragment is changed in the browser after the currently processed server request returns the response.

Prefixing the fragment identifier with an exclamation mark enables the web crawler support described in Section 11.10.4, "Supporting Web Crawling".

11.10.2. Reading the URI Fragment

The current URI fragment can be acquired with the `getUriFragment()` method from the current **Page** object. The fragment is known when the `init()` method of the UI is called.

```
// Read initial URI fragment to create UI content
String fragment = getPage().getUriFragment();
enter(fragment);
```

To enable reusing the same code when the URI fragment is changed, as described next, it is usually best to build the relevant part of the UI in a separate method. In the above example, we called an `enter()` method, in a way that is similar to handling view changes with **Navigator**.

11.10.3. Listening for URI Fragment Changes

After the UI has been initialized, changes in the URI fragment can be handled with a `UriFragmentChangeListener`. The listeners are called when the URI fragment changes, but not when the UI is initialized, where the current fragment is available from the `page` object as described earlier.

For example, we could define the listener as follows in the `init()` method of a UI class:

```
public class MyUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        getPage().addUriFragmentChangedListener(
            new UriFragmentChangedListener() {
                public void uriFragmentChanged(
                    UriFragmentChangedEvent source) {
                    enter(source.getUriFragment());
                }
            });
    }

    // Read the initial URI fragment
    enter(getPage().getUriFragment());
}

void enter(String fragment) {
    ... initialize the UI ...
```

```
}
```

Figure 11.12, “Application State Management with URI Fragment Utility” shows an application that allows specifying the menu selection with a URI fragment and correspondingly sets the fragment when the user selects a menu item.



Figure 11.12. Application State Management with URI Fragment Utility

11.10.4. Supporting Web Crawling

Stateful AJAX applications can not normally be crawled by a search engine, as they run in a single page and a crawler can not navigate the states even if URI fragments are enabled. The Google search engine and crawler support a convention where the fragment identifiers are prefixed with exclamation mark, such as `#!myfragment`. The servlet needs to have a separate searchable content page accessible with the same URL, but with a `_escaped_fragment_` parameter. For example, for `/myapp/myui#!myfragment` it would be `/myapp/myui?_escaped_fragment_=myfragment`.

You can provide the crawl content by overriding the `service()` method in a custom servlet class. For regular requests, you should call the super implementation in the **VaadinServlet** class.

```
public class MyCustomServlet extends VaadinServlet
    @Override
    protected void service(HttpServletRequest request,
        HttpServletResponse response)
```

```

throws ServletException, IOException {
String fragment = request
    .getParameter("_escaped_fragment_");
if (fragment != null) {
    response.setContentType("text/html");
    Writer writer = response.getWriter();
    writer.append("<html><body>" +
        "<p>Here is some crawlable +" +
        "content about " + fragment + "</p>");
    // A list of all crawlable pages
    String items[] = {"mercury", "venus",
        "earth", "mars"};
    writer.append("<p>Index of all content:</p><ul>");

    for (String item: items) {
        String url = request.getContextPath() +
            request.getServletPath() +
            request.getPathInfo() + "#" + item;
        writer.append("<li><a href=\"" + url + "\">" +
            item + "</a></li>");
    }
    writer.append("</ul></body>");
} else
    super.service(request, response);
}
}

```

The crawlable content does not need to be human readable. It can provide an index of links to other application states, as we did in the example above. The links should use the "#!" notation, but can not be relative to avoid having the _escaped_fragment_ parameter.

You need to use the custom servlet class in the web.xml deployment descriptor instead of the normal **VaadinServlet** class, as described in Section 5.9.4, “Using a web.xml Deployment Descriptor”.

11.11. Drag and Drop

Dragging an object from one location to another by grabbing it with mouse, holding the mouse button pressed, and then releasing the button to "drop" it to the other location is a common way to move, copy, or associate objects. For example, most operating systems allow dragging and dropping files between folders or dragging a document on a program to open it. In Vaadin, it is possible to drag and drop components and parts of certain components.

Dragged objects, or *transferables*, are essentially data objects. You can drag and drop rows in **Table** and nodes in **Tree** components, either within or between the components. You can also drag entire components by wrapping them inside **DragAndDropWrapper**.

Dragging starts from a *drag source*, which defines the transferable. Transferables implement the **Transferable** interfaces. For trees and tables, which are bound to **Container** data sources, a node or row transferable is a reference to an **Item** in the Vaadin Data Model. Dragged components are referenced with a **WrapperTransferable**. Starting dragging does not require any client-server communication, you only need to enable dragging. All drag and drop logic occurs in two operations: determining (*accepting*) where dropping is allowed and actually dropping. Drops can be done on a *drop target*, which implements the **DropTarget** interface. Three components implement the interface: **Tree**, **Table**, and **DragAndDropWrapper**. These accept and drop operations need to be provided in a *drop handler*. Essentially all you need to do to enable drag and drop is to enable dragging in the drag source and implement the `getAcceptCriterion()` and `drop()` methods in the **DropHandler** interface.

The client-server architecture of Vaadin causes special requirements for the drag and drop functionality. The logic for determining where a dragged object can be dropped, that is, *accepting* a drop, should normally be done on the client-side, in the browser. Server communications are too slow to have much of such logic on the server-side. The drag and drop feature therefore offers a number of ways to avoid the server communications to ensure a good user experience.

11.11.1. Handling Drops

Most of the user-defined drag and drop logic occurs in a *drop handler*, which is provided by implementing the `drop()` method in the **DropHandler** interface. A closely related definition is the drop accept criterion, which is defined in the `getAcceptCriterion()` method in the same interface. It is described in Section 11.11.4, “Accepting Drops” later.

The drop() method gets a **DragAndDropEvent** as its parameters. The event object provides references to two important objects: **Transferable** and **TargetDetails**.

A **Transferable** contains a reference to the object (component or data item) that is being dragged. A tree or table item is represented as a **TreeTransferable** or **TableTransferable** object, which carries the item identifier of the dragged tree or table item. These special transferables, which are bound to some data in a container, are **DataBoundTransferable**. Dragged components are represented as **WrapperTransferable** objects, as the components are wrapped in a **DragAndDropWrapper**.

The **TargetDetails** object provides information about the exact location where the transferable object is being dropped. The exact class of the details object depends on the drop target and you need to cast it to the proper subclass to get more detailed information. If the target is selection component, essentially a tree or a table, the **AbstractSelectTargetDetails** object tells the item on which the drop is being made. For trees, the **TreeTargetDetails** gives some more details. For wrapped components, the information is provided in a **WrapperDropDetails** object. In addition to the target item or component, the details objects provide a *drop location*. For selection components, the location can be obtained with the `getDropLocation()` and for wrapped components with `verticalDropLocation()` and `horizontalDropLocation()`. The locations are specified as either **VerticalDropLocation** or **HorizontalDropLocation** objects. The drop location objects specify whether the transferable is being dropped above, below, or directly on (at the middle of) a component or item.

Dropping on a **Tree**, **Table**, and a wrapped component is explained further in the following sections.

11.11.2. Dropping Items On a Tree

You can drag items from, to, or within a **Tree**. Making tree a drag source requires simply setting the drag mode with `setDragMode()`. **Tree** currently supports only one drag mode, `TreeDragMode.NODE`, which allows dragging single tree nodes. While dragging, the dragged node is referenced with a

TreeTransferable object, which is a **DataBoundTransferable**. The tree node is identified by the item ID of the container item.

When a transferable is dropped on a tree, the drop location is stored in a **TreeTargetDetails** object, which identifies the target location by item ID of the tree node on which the drop is made. You can get the item ID with `getitemIdOver()` method in **AbstractSelectTargetDetails**, which the **TreeTargetDetails** inherits. A drop can occur directly on or above or below a node; the exact location is a **VerticalDropLocation**, which you can get with the `getDropLocation()` method.

In the example below, we have a **Tree** and we allow reordering the tree items by drag and drop.

```
final Tree tree = new Tree("Inventory");
tree.setContainerDataSource(TreeExample.createTreeContent());
layout.addComponent(tree);

// Expand all items
for (Iterator<?> it = tree.rootItemIds().iterator(); it.hasNext();)
    tree.expandItemsRecursively(it.next());

// Set the tree in drag source mode
tree.setDragMode(TreeDragMode.NODE);

// Allow the tree to receive drag drops and handle them
tree.setDropHandler(new DropHandler() {
    public AcceptCriterion getAcceptCriterion() {
        return AcceptAll.get();
    }

    public void drop(DragAndDropEvent event) {
        // Wrapper for the object that is dragged
        Transferable t = event.getTransferable();

        // Make sure the drag source is the same tree
if (t.getSourceComponent() != tree)
    return;

        TreeTargetDetails target = (TreeTargetDetails)
            event.getTargetDetails();

        // Get ids of the dragged item and the target item
Object sourceItemId = t.getData("itemid");
Object targetItemId = target.getItemIdOver();

        // On which side of the target the item was dropped
VerticalDropLocation location = target.getDropLocation();

        HierarchicalContainer container = (HierarchicalContainer)
            tree.getContainerDataSource();

        // Drop right on an item -> make it a child
    }
})
```

```

if (!location == VerticalDropLocation.MIDDLE)
    tree.setParent(sourceItemId, targetItemId);

// Drop at the top of a subtree -> make it previous
else if (location == VerticalDropLocation.TOP) {
    Object parentId = container.getParent(targetItemId);
    container.setParent(sourceItemId, parentId);
    container.moveAfterSibling(sourceItemId, targetItemId);
    container.moveAfterSibling(targetItemId, sourceItemId);
}

// Drop below another item -> make it next
else if (location == VerticalDropLocation.BOTTOM) {
    Object parentId = container.getParent(targetItemId);
    container.setParent(sourceItemId, parentId);
    container.moveAfterSibling(sourceItemId, targetItemId);
}
}
});

```

Accept Criteria for Trees

Tree defines some specialized accept criteria for trees.

TargetInSubtree(client-side)

Accepts if the target item is in the specified sub-tree. The sub-tree is specified by the item ID of the root of the sub-tree in the constructor. The second constructor includes a depth parameter, which specifies how deep from the given root node are drops accepted. Value -1 means infinite, that is, the entire sub-tree, and is therefore the same as the simpler constructor.

TargetItemAllowsChildren(client-side)

Accepts a drop if the tree has `setChildrenAllowed()` enabled for the target item. The criterion does not require parameters, so the class is a singleton and can be acquired with `Tree.TargetItemAllowsChildren.get()`. For example, the following composite criterion accepts drops only on nodes that allow children, but between all nodes:

```
return new Or(Tree.TargetItemAllowsChildren.get(), new Not(VerticalLocationIs.MIDDLE));
```

TreeDropCriterion(server-side)

Accepts drops on only some items, which as specified by a set of item IDs. You must extend the abstract class and implement the `getAllowedItemIds()` to return the set. While the criterion is server-side, it is lazy-loading, so that the list of accepted target nodes is loaded only

once from the server for each drag operation. See Section 11.11.4, “Accepting Drops” for an example.

In addition, the accept criteria defined in **AbstractSelect** are available for a **Tree**, as listed in Section 11.11.4, “Accepting Drops”.

11.11.3. Dropping Items On a Table

You can drag items from, to, or within a **Table**. Making table a drag source requires simply setting the drag mode with `setDragMode()`. **Table** supports dragging both single rows, with `TableDragMode.ROW`, and multiple rows, with `TableDragMode.MULTIROW`. While dragging, the dragged node or nodes are referenced with a **TreeTransferable** object, which is a **DataBoundTransferable**. Tree nodes are identified by the item IDs of the container items.

When a transferable is dropped on a table, the drop location is stored in a **AbstractSelectTargetDetails** object, which identifies the target row by its item ID. You can get the item ID with `getItemIdOver()` method. A drop can occur directly on or above or below a row; the exact location is a **VerticalDropLocation**, which you can get with the `getDropLocation()` method from the details object.

Accept Criteria for Tables

Table defines one specialized accept criterion for tables.

TableDropCriterion(server-side)

Accepts drops only on (or above or below) items that are specified by a set of item IDs. You must extend the abstract class and implement the `getAllowedItemIds()` to return the set. While the criterion is server-side, it is lazy-loading, so that the list of accepted target items is loaded only once from the server for each drag operation.

11.11.4. Accepting Drops

You can not drop the objects you are dragging around just anywhere. Before a drop is possible, the specific drop location on which the mouse hovers must be *accepted*. Hovering a dragged object over an accepted location displays an *accept*

indicator, which allows the user to position the drop properly. As such checks have to be done all the time when the mouse pointer moves around the drop targets, it is not feasible to send the accept requests to the server-side, so drops on a target are normally accepted by a client-side *accept criterion*.

A drop handler must define the criterion on the objects which it accepts to be dropped on the target. The criterion needs to be provided in the **getAcceptCriterion()** method of the **DropHandler** interface. A criterion is represented in an **AcceptCriterion** object, which can be a composite of multiple criteria that are evaluated using logical operations. There are two basic types of criteria: *client-side* and *server-side criteria*. The various built-in criteria allow accepting drops based on the identity of the source and target components, and on the *data flavor* of the dragged objects.

To allow dropping any transferable objects, you can return a universal accept criterion, which you can get with `AcceptAll.get()`.

```
tree.setDropHandler(new DropHandler() {
    public AcceptCriterion getAcceptCriterion() {
        return AcceptAll.get();
    }
    ...
}
```

Client-Side Criteria

The *client-side criteria*, which inherit the **ClientSideCriterion**, are verified on the client-side, so server requests are not needed for verifying whether each component on which the mouse pointer hovers would accept a certain object.

The following client-side criteria are define in `com.vaadin.event.dd.acceptcriterion`:

AcceptAll

Accepts all transferables and targets.

And

Performs the logical AND operation on two or more client-side criteria; accepts the transferable if all the given sub-criteria accept it.

ContainsDataFlavour

The transferable must contain the defined data flavour.

Not

Performs the logical NOT operation on a client-side criterion; accepts the transferable if and only if the sub-criterion does not accept it.

Or

Performs the logical OR operation on two or more client-side criteria; accepts the transferable if any of the given sub-criteria accepts it.

Sources

Accepts all transferables from any of the given source components

SourcesTarget

Accepts the transferable only if the source component is the same as the target. This criterion is useful for ensuring that items are dragged only within a tree or a table, and not from outside it.

TargetDetails

Accepts any transferable if the target detail, such as the item of a tree node or table row, is of the given data flavor and has the given value.

In addition, target components such as **Tree** and **Table** define some component-specific client-side accept criteria. See Section 11.11.2, “Dropping Items On a **Tree**” for more details.

AbstractSelect defines the following criteria for all selection components, including **Tree** and **Table**.

AcceptItem

Accepts only specific items from a specific selection component. The selection component, which must inherit **AbstractSelect**, is given as the first parameter for the constructor. It is followed by a list of allowed item identifiers in the drag source.

AcceptItem.ALL

Accepts all transferables as long as they are items.

TargetItems

Accepts all drops on the specified target items. The constructor requires the target component (**AbstractSelect**) followed by a list of allowed item identifiers.

VerticalLocationIs.MIDDLE, TOP, and BOTTOM

The three static criteria accepts drops on, above, or below an item. For example, you could accept drops only in between items with the following:

```
public AcceptCriterion getAcceptCriterion() {  
    return new Not(VerticalLocationIs.MIDDLE);  
}
```

Server-Side Criteria

The *server-side criteria* are verified on the server-side with the `accept()` method of the **ServerSideCriterion** class. This allows fully programmable logic for accepting drops, but the negative side is that it causes a very large amount of server requests. A request is made for every target position on which the pointer hovers. This problem is eased in many cases by the component-specific lazy loading criteria **TableDropCriterion** and **TreeDropCriterion**. They do the server visit once for each drag and drop operation and return all accepted rows or nodes for current **Transferable** at once.

The `accept()` method gets the drag event as a parameter so it can perform its logic much like in `drop()`.

```
public AcceptCriterion getAcceptCriterion() {  
    // Server-side accept criterion that allows drops on any other  
    // location except on nodes that may not have children  
    ServerSideCriterion criterion = new ServerSideCriterion() {  
        public boolean accept(DragAndDropEvent dragEvent) {  
            TreeTargetDetails target = (TreeTargetDetails)  
                dragEvent.getTargetDetails();  
  
            // The tree item on which the load hovers  
            Object targetItemId = target.getItemIdOver();  
  
            // On which side of the target the item is hovered  
            VerticalDropLocation location = target.getDropLocation();  
            if (location == VerticalDropLocation.MIDDLE)  
                if (!tree.areChildrenAllowed(targetItemId))  
                    return false; // Not accepted  
  
            return true; // Accept everything else  
        }  
    };
```

```
    return criterion;
}
```

The server-side criteria base class **ServerSideCriterion** provides a generic `accept()` method. The more specific **TableDropCriterion** and **TreeDropCriterion** are convenience extensions that allow defining allowed drop targets as a set of items. They also provide some optimization by lazy loading, which reduces server communications significantly.

```
public AcceptCriterion getAcceptCriterion() {
    // Server-side accept criterion that allows drops on any
    // other tree node except on node that may not have children
    TreeDropCriterion criterion = new TreeDropCriterion() {
        @Override
        protected Set<Object> getAllowedItemIds(
            DragAndDropEvent dragEvent, Tree tree) {
            HashSet<Object> allowed = new HashSet<Object>();
            for (Iterator<Object> i =
                tree.getItemIdIds().iterator(); i.hasNext(); ) {
                Object itemId = i.next();
                if (tree.hasChildren(itemId))
                    allowed.add(itemId);
            }
            return allowed;
        }
    };
    return criterion;
}
```

Accept Indicators

When a dragged object hovers on a drop target, an *accept indicator* is displayed to show whether or not the location is accepted. For *MIDDLE* location, the indicator is a box around the target (tree node, table row, or component). For vertical drop locations, the accepted locations are shown as horizontal lines, and for horizontal drop locations as vertical lines.

For **DragAndDropWrapper** drop targets, you can disable the accept indicators or *drag hints* with the *no-vertical-drag-hints*, *no-horizontal-drag-hints*, and *no-box-drag-hints* styles. You need to add the styles to the *layout that contains* the wrapper, not to the wrapper itself.

```
// Have a wrapper
DragAndDropWrapper wrapper = new DragAndDropWrapper(c);
layout.addComponent(wrapper);
```

```
// Disable the hints
```

```
layout.addStyleName("no-vertical-drag-hints");
layout.addStyleName("no-horizontal-drag-hints");
layout.addStyleName("no-box-drag-hints");
```

11.11.5. Dragging Components

Dragging a component requires wrapping the source component within a **DragAndDropWrapper**. You can then allow dragging by putting the wrapper (and the component) in drag mode with `setDragStartMode()`. The method supports two drag modes: `DragStartMode.WRAPPER` and `DragStartMode.COMPONENT`, which defines whether the entire wrapper is shown as the drag image while dragging or just the wrapped component.

```
// Have a component to drag
final Button button = new Button("An Absolute Button");

// Put the component in a D&D wrapper and allow dragging it
final DragAndDropWrapper buttonWrap = new DragAndDropWrapper(button);
buttonWrap.setDragStartMode(DragStartMode.COMPONENT);

// Set the wrapper to wrap tightly around the component
buttonWrap.setSizeUndefined();

// Add the wrapper, not the component, to the layout
layout.addComponent(buttonWrap, "left: 50px; top: 50px;");
```

The default height of **DragAndDropWrapper** is undefined, but the default width is 100%. If you want to ensure that the wrapper fits tightly around the wrapped component, you should call `setSizeUndefined()` for the wrapper. Doing so, you should make sure that the wrapped component does not have a relative size, which would cause a paradox.

Dragged components are referenced in the **WrapperTransferable**. You can get the reference to the dragged component with `getDraggedComponent()`. The method will return null if the transferable is not a component. Also HTML 5 drags (see later) are held in wrapper transferables.

11.11.6. Dropping on a Component

Drops on a component are enabled by wrapping the component in a **DragAndDropWrapper**. The wrapper is an ordinary component; the constructor takes the wrapped component as a parameter. You just need to define the **DropHandler** for the wrapper with `setDropHandler()`.

In the following example, we allow moving components in an absolute layout. Details on the drop handler are given later.

```
// A layout that allows moving its contained components
// by dragging and dropping them
final AbsoluteLayout absLayout = new AbsoluteLayout();
absLayout.setWidth("100%");
absLayout.setHeight("400px");

... put some (wrapped) components in the layout ...

// Wrap the layout to allow handling drops
DragAndDropWrapper layoutWrapper =
    new DragAndDropWrapper(absLayout);

// Handle moving components within the AbsoluteLayout
layoutWrapper.setDropHandler(new DropHandler() {
    public AcceptCriterion getAcceptCriterion() {
        return AcceptAll.get();
    }

    public void drop(DragAndDropEvent event) {
        ...
    }
});
```

Target Details for Wrapped Components

The drop handler receives the drop target details in a **WrapperTargetDetails** object, which implements the **TargetDetails** interface.

```
public void drop(DragAndDropEvent event) {
    WrapperTransferable t =
        (WrapperTransferable) event.getTransferable();
    WrapperTargetDetails details =
        (WrapperTargetDetails) event.getTargetDetails();
```

The wrapper target details include a **MouseEventDetails** object, which you can get with `getMouseEvent()`. You can use it to get the mouse coordinates for the position where the mouse button was released and the drag ended. Similarly, you can find out the drag start position from the transferable object (if it is a **WrapperTransferable**) with `getMouseDownEvent()`.

```
// Calculate the drag coordinate difference
int xChange = details.getMouseEvent().getClientX()
```

```
- t.getMouseEvent().getClientX();
int yChange = details.getMouseEvent().getClientY()
    - t.getMouseEvent().getClientY();

// Move the component in the absolute layout
ComponentPosition pos =
    absLayout.getPosition(t.getSourceComponent());
pos.setLeftValue(pos.getLeftValue() + xChange);
pos.setTopValue(pos.getTopValue() + yChange);
```

You can get the absolute x and y coordinates of the target wrapper with `getAbsoluteLeft()` and `getAbsoluteTop()`, which allows you to translate the absolute mouse coordinates to coordinates relative to the wrapper. Notice that the coordinates are really the position of the wrapper, not the wrapped component; the wrapper reserves some space for the accept indicators.

The `verticalDropLocation()` and `horizontalDropLocation()` return the more detailed drop location in the target.

11.11.7. Dragging Files from Outside the Browser

The **DragAndDropWrapper** allows dragging files from outside the browser and dropping them on a component wrapped in the wrapper. Dropped files are automatically uploaded to the application and can be acquired from the wrapper with `getFiles()`. The files are represented as **Html5File** objects as defined in the inner class. You can define an upload **Receiver** to receive the content of a file to an **OutputStream**.

Dragging and dropping files to browser is supported in HTML 5 and requires a compatible browser, such as Mozilla Firefox 3.6 or newer.

11.12. Logging

You can do logging in Vaadin application using the standard `java.util.logging` facilities. Configuring logging is as easy as putting a file named `logging.properties` in the default package of your Vaadin application (`src` in an Eclipse project or `src/main/java` or `src/main/resources` in a Maven project). This file is read by the **Logger** class when a new instance of it is initialize.

11.12.1. Logging in Apache Tomcat

For logging Vaadin applications deployed in Apache Tomcat, you do not need to do anything special to log to the same place as Tomcat itself. If you need to write the Vaadin application related messages elsewhere, just add a custom logging.properties file to the default package of your Vaadin application.

If you would like to pipe the log messages through another logging solution, see Section 11.12.3, “Piping to Log4j using SLF4J” below.

11.12.2. Logging in Liferay

Liferay mutes logging through java.util.logging by default. In order to enable logging, you need to add a logging.properties file of your own to the default package of your Vaadin application. This file should define at least one destination where to save the log messages.

You can also log through SLF4J, which is used in and bundled with Liferay. Follow the instructions in Section 11.12.3, “Piping to Log4j using SLF4J”.

11.12.3. Piping to Log4j using SLF4J

Piping output from java.util.logging to Log4j is easy with SLF4J (<http://slf4j.org/>). The basic way to go about this is to add the SLF4J JAR file as well as the jul-to-slf4j.jar file, which implements the bridge from java.util.logging, to SLF4J. You will also need to add a third logging implementation JAR file, that is, slf4j-log4j12-x.x.x.jar, to log the actual messages using Log4j. For more info on this, please visit the SLF4J site.

In order to get the java.util.logging to SLF4J bridge installed, you need to add the following snippet of code to your **UI** class at the very top://TODO: Sure it's UI class and not the servlet?

```
static {
    SLF4JBridgeHandler.install();
}
```

This will make sure that the bridge handler is installed and working before Vaadin starts to process any logging calls.



Please note!

This can seriously impact on the cost of disabled logging statements (60-fold increase) and a measurable impact on enabled log statements (20% overall increase). However, Vaadin doesn't log very much, so the effect on performance will be negligible.

11.12.4. Using Logger

You can do logging with a simple pattern where you register a static logger instance in each class that needs logging, and use this logger wherever logging is needed in the class. For example:

```
public class MyClass {  
    private final static Logger logger =  
        Logger.getLogger(MyClass.class.getName());  
  
    public void myMethod() {  
        try {  
            //do something that might fail  
        } catch (Exception e) {  
            logger.log(Level.SEVERE, "FAILED CATASTROPHICALLY!", e);  
        }  
    }  
}
```

11.13. JavaScript Interaction

Vaadin supports two-direction JavaScript calls from and to the server-side. This allows interfacing with JavaScript code without writing client-side integration code.

11.13.1. Calling JavaScript

You can make JavaScript calls from the server-side with the `execute()` method in the **JavaScript** class. You can get a **JavaScript** instance from the current **Page** object with `getJavaScript()`.

```
// Execute JavaScript in the currently processed page  
Page.getCurrent().getJavaScript().execute("alert('Hello')");
```

The **JavaScript** class itself has a static shorthand method `getCurrent()` to get the instance for the currently processed page.

```
// Shorthand  
JavaScript.getCurrent().execute("alert('Hello')");
```

The JavaScript is executed after the server request that is currently processed returns. If multiple JavaScript calls are made during the processing of the request, they are all executed sequentially after the request is done. Hence, the JavaScript execution does not pause the execution of the server-side application and you can not return values from the JavaScript.

11.13.2. Handling JavaScript Function Callbacks

You can make calls with JavaScript from the client-side to the server-side. This requires that you register JavaScript callback methods from the server-side. You need to implement and register a **JavaScriptFunction** with `addFunction()` in the current **JavaScript** object. A function requires a name, with an optional package path, which are given to the `addFunction()`. You only need to implement the `call()` method to handle calls from the client-side JavaScript.

```
JavaScript.getCurrent().addFunction("com.example.foo.myfunc",  
    new JavaScriptFunction() {  
        @Override  
        public void call(JsonArray arguments) {  
            Notification.show("Received call");  
        }  
    });  
  
Link link = new Link("Send Message", new ExternalResource(  
    "javascript:com.example.foo.myfunc()"));
```

Parameters passed to the JavaScript method on the client-side are provided in a **JSONArray** passed to the `call()` method. The parameter values can be acquired with the `get()` method by the index of the parameter, or any of the type-casting getters. The getter must match the type of the passed parameter, or an exception is thrown.

```

JavaScript.getCurrent().addFunction("com.example.foo.myfunc".
    new JavaScriptFunction() {
        @Override
        public void call(JsonArray arguments) {
            try {
                String message = arguments.getString(0);
                int value = arguments.getInt(1);
                Notification.show("Message: " + message +
                    ". value: " + value);
            } catch (Exception e) {
                Notification.show("Error: " + e.getMessage());
            }
        }
    });
}

```

```

Link link = new Link("Send Message", new ExternalResource(
    "javascript:com.example.foo.myfunc(prompt('Message'), 42)"));

```

The function callback mechanism is the same as the RPC mechanism used with JavaScript component integration, as described in Section 16.13.4, “RPC from JavaScript to Server-Side”.

11.14. Accessing Session-Global Data

This section is mostly up-to-date with Vaadin 7, but has some information which still needs to be updated.

Applications typically need to access some objects from practically all user interface code, such as a user object, a business data model, or a database connection. This data is typically initialized and managed in the UI class of the application, or in the session or servlet.

For example, you could hold it in the UI class as follows:

```

class MyUI extends UI {
    UserData userData;

    public void init() {
        userData = new UserData();
    }

    public UserData getUserData() {
        return userData;
    }
}

```

Vaadin offers two ways to access the UI object: with getUI() method from any component and the global UI.getCurrent() method.

The getUI() works as follows:

```
data = ((MyUI)component.getUI()).getUserData();
```

This does not, however work in many cases, because it requires that the components are attached to the UI. That is not the case most of the time when the UI is still being built, such as in constructors.

```
class MyComponent extends CustomComponent {  
    public MyComponent() {  
        // This fails with NullPointerException  
        Label label = new Label("Country: " +  
            getUI().getLocale().getCountry());  
  
        setCompositionRoot(label);  
    }  
}
```

The global access methods for the currently served servlet, session, and UI allow an easy way to access the data:

```
data = ((MyUI) UI.getCurrent()).getUserData();
```

11.14.1. The Problem

The basic problem in accessing session-global data is that the getUI() method works only after the component has been attached to the application. Before that, it returns *null*. This is the case in constructors of components, such as a **Custom-Component**:

Using a static variable or a singleton implemented with such to give a global access to user session data is not possible, because static variables are global in the entire web application, not just the user session. This can be handy for communicating data between the concurrent sessions, but creates a problem within a session.

The data would be shared by all users and be reinitialized every time a new user opens the application.

11.14.2. Overview of Solutions

To get the application object or any other global data, you have the following solutions:

- Pass a reference to the global data as a parameter
- Initialize components in attach() method
- Initialize components in the enter() method of the navigation view (if using navigation)
- Store a reference to global data using the *ThreadLocal Pattern*

Each solution is described in the following sections.

11.14.3. Passing References Around

You can pass references to objects as parameters. This is the normal way in object-oriented programming.

```
class MyApplication extends Application {  
    UserData userData;  
  
    public void init() {  
        Window mainWindow = new Window("My Window");  
        setMainWindow(mainWindow);  
  
        userData = new UserData();  
  
        mainWindow.addComponent(new MyComponent(this));  
    }  
  
    public UserData getUserData() {  
        return userData;  
    }  
}  
  
class MyComponent extends CustomComponent {  
    public MyComponent(MyApplication app) {  
        Label label = new Label("Name: " +  
            app.getUserData().getName());  
  
        setCompositionRoot(label);  
    }  
}
```

If you need the reference in other methods, you either have to pass it again as a parameter or store it in a member variable.

The problem with this solution is that practically all constructors in the application need to get a reference to the application object, and passing it further around in the classes is another hard task.

11.14.4. Overriding attach()

The attach() method is called when the component is attached to the UI through containment hierarchy. The getUI() method always works.

```
class MyComponent extends CustomComponent {  
    public MyComponent() {  
        // Must set a dummy root in constructor  
        setCompositionRoot(new Label(""));  
    }  
  
    @Override  
    public void attach() {  
        Label label = new Label("Name: " +  
            ((MyUI)component.getUI())  
            .getUserData().getName());  
  
        setCompositionRoot(label);  
    }  
}
```

While this solution works, it is slightly messy. You may need to do some initialization in the constructor, but any construction requiring the global data must be done in the attach() method. Especially, **CustomComponent** requires that the setCompositionRoot() method is called in the constructor. If you can't create the actual composition root component in the constructor, you need to use a temporary dummy root, as is done in the example above.

Using getUI() also needs casting if you want to use methods defined in your UI class.

11.14.5. ThreadLocal Pattern

Vaadin uses the ThreadLocal pattern for allowing global access to the **UI**, and **Page** objects of the currently processed server request with a static `getCurrent()` method in all the respective classes. This section explains why the pattern is used in Vaadin and how it works. You may also need to reimplement the pattern for some purpose.

The ThreadLocal pattern gives a solution to the global access problem by solving two sub-problems of static variables.

As the first problem, assume that the servlet container processes requests for many users (sessions) sequentially. If a static variable is set in a request belonging one user, it could be read or re-set by the next incoming request belonging to another user. This can be solved by setting the global reference at the beginning of each HTTP request to point to data of the current user, as illustrated in Figure Figure 11.13, "Switching a static (or ThreadLocal) reference during sequential processing of requests".

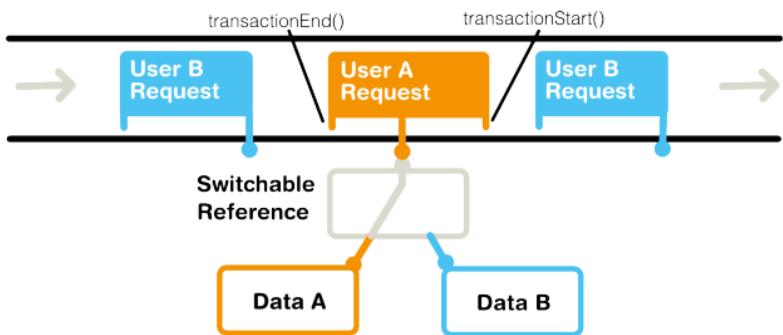


Figure 11.13. Switching a static (or ThreadLocal) reference during sequential processing of requests

The second problem is that servlet containers typically do thread pooling with multiple worker threads that process requests. Therefore, setting a static reference would change it in all threads running concurrently, possibly just when another thread is processing a request for another user. The solution is to store the reference in a thread-local variable instead of

a static. You can do so by using the **ThreadLocal** class in Java for the switch reference.

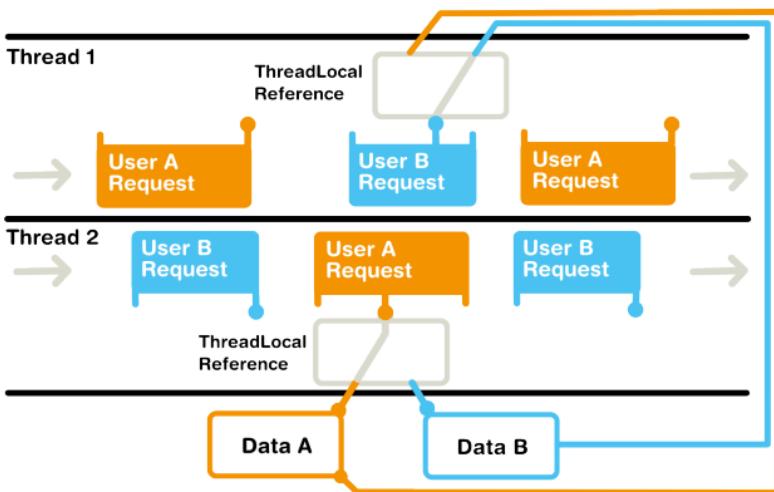


Figure 11.14. Switching `ThreadLocal` references during concurrent processing of requests

11.15. Server Push

When you need to update a UI from another UI, possibly of another user, or from a background thread running in the server, you usually want to have the update show immediately, not when the browser happens to make the next server request. For this purpose, you can use *server push* that sends the data to the browser immediately. Push is based on a client-server connection, usually a WebSocket connection, that the client establishes and the server can then use to send updates to the client.

The server-client communication is done by default with a WebSocket connection if the browser and the server support it. If not, Vaadin will fall back to a method supported by the browser. Vaadin Push uses a custom build of the Atmosphere framework for client-server communication.

11.15.1. Installing the Push Support

The server push support in Vaadin requires the separate Vaadin Push library. It is included in the installation package as vaadin-push.jar.

Retrieving with Ivy

With Ivy, you can get it with the following declaration in the ivy.xml:

```
<dependency org="com.vaadin" name="vaadin-push"
    rev="&vaadin.version;" conf="default->default"/>
```

In some servers, you may need to exclude a sl4j dependency as follows:

```
<dependency org="com.vaadin" name="vaadin-push"
    rev="&vaadin.version;" conf="default->default">
    <exclude org="org.slf4j" name="slf4j-api"/>
</dependency>
```

Pay note that the Atmosphere library is a bundle, so if you retrieve the libraries with Ant, for example, you need to retrieve type="jar.bundle".

Retrieving with Maven

In Maven, you can get the push library with the following dependency in the POM:

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-push</artifactId>
    <version>${vaadin.version}</version>
</dependency>
```

11.15.2. Enabling Push for a UI

To enable server push, you need to define the push mode either in the deployment descriptor or with the **@Push** annotation for the UI.

Push Modes and Transports

You can use server push in two modes: automatic and manual. The automatic mode pushes changes to the browser auto-

matically after `access()` finishes. With the manual mode, you can do the push explicitly with `push()`, which allows more flexibility.

Server push can use several transports: WebSockets, long polling, or combined WebSockets+XHR. WebSockets is the default transport.

The `@Push` annotation

You can enable server push for a UI with the `@Push` annotation as follows. It defaults to automatic mode (`PushMode.AUTO-MATIC`).

```
@Push  
public class PushyUI extends UI {
```

To enable manual mode, you need to give the `PushMode.MANUAL` parameter as follows:

```
@Push(PushMode.MANUAL)  
public class PushyUI extends UI {
```

To use the long polling transport, you need to set the transport parameter as `Transport.LONG_POLLING` as follows:

```
@Push(transport=Transport.LONG_POLLING)  
public class PushyUI extends UI {
```

Servlet Configuration

You can enable the server push and define the push mode also in the servlet configuration with the `pushmode` parameter for the servlet in the `web.xml` deployment descriptor. If you use a Servlet 3.0 compatible server, you also want to enable asynchronous processing with the `async-supported` parameter. Note the use of Servlet 3.0 schema in the deployment descriptor.

```
<xml version="1.0" encoding="UTF-8">  
<web-app  
id="WebApp_ID" version="3.0"  
xmlns="http://java.sun.com/xml/ns/javaee"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">  
<serverlet>  
<serverlet-name>Pushy UI</serverlet-name>  
<serverlet-class>  
com.vaadin.server.VaadinServlet</serverlet-class>  
  
<init-param>  
<param-name>UI</param-name>  
<param-value>com.example.my.PushyUI</param-value>  
</init-param>
```

```
<!-- Enable server push -->
<init-param>
    <param-name>pushmode</param-name>
    <param-value>automatic</param-value>
</init-param>
<async-supported>true</async-supported>
</servlet>
</web-app>
```

11.15.3. Accessing UI from Another Thread

Making changes to a **UI** object from another thread and pushing them to the browser requires locking the user session when accessing the UI. Otherwise, the UI update done from another thread could conflict with a regular event-driven update and cause either data corruption or deadlocks. Because of this, you may only access an UI using the `access()` method, which locks the session to prevent conflicts. It takes a `Runnable` which it executes as its parameter.

For example:

```
ui.access(new Runnable() {
    @Override
    public void run() {
        series.add(new DataSeriesItem(x, y));
    }
});
```

In Java 8, where a parameterless lambda expression creates a runnable, you could simply write:

```
ui.access(() ->
    series.add(new DataSeriesItem(x, y)));
```

If the push mode is manual, you need to push the pending UI changes to the browser explicitly with the `push()` method.

```
ui.access(new Runnable() {
    @Override
    public void run() {
        series.add(new DataSeriesItem(x, y));
        ui.push();
    }
});
```

Below is a complete example of a case where we make UI changes from another thread.

```
public class PushyUI extends UI {
    Chart chart = new Chart(ChartType.AREASLINE);
```

```

    DataSeries series = new DataSeries();

    @Override
    protected void init(VaadinRequest request) {
        chart.setSizeFull();
        setContent(chart);

        // Prepare the data display
        Configuration conf = chart.getConfiguration();
        conf.setTitle("Hot New Data");
        conf.setSeries(series);

        // Start the data feed thread
        new FeederThread().start();
    }

    class FeederThread extends Thread {
        int count = 0;

        @Override
        public void run() {
            try {
                // Update the data for a while
                while (count < 100) {
                    Thread.sleep(1000);

                    access(new Runnable() {
                        @Override
                        public void run() {
                            double y = Math.random();
                            series.add(
                                new DataSeriesItem(count++, y),
                                true, count > 10);
                        }
                    });
                }
            };
        }

        // Inform that we have stopped running
        access(new Runnable() {
            @Override
            public void run() {
                setContent(new Label("Done!"));
            }
        });
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

When sharing data between UIs or user sessions, you need to consider the message-passing mechanism more carefully, as explained next.

11.15.4. Broadcasting to Other Users

Broadcasting messages to be pushed to UIs in other user sessions requires having some sort of message-passing mechanism that sends the messages to all UIs that register as recipients. As processing server requests for different UIs is done concurrently in different threads of the application server, locking the threads properly is very important to avoid deadlock situations.

The Broadcaster

The standard pattern for sending messages to other users is to use a *broadcaster* singleton that registers the UIs and broadcasts messages to them safely. To avoid deadlocks, it is recommended that the messages should be sent through a message queue in a separate thread. Using a Java **ExecutorService** running in a single thread is usually the easiest and safest way.

```
public class Broadcaster implements Serializable {
    static ExecutorService executorService =
        Executors.newSingleThreadExecutor();

    public interface BroadcastListener {
        void receiveBroadcast(String message);
    }

    private static LinkedList<BroadcastListener> listeners =
        new LinkedList<BroadcastListener>();

    public static synchronized void register(
        BroadcastListener listener) {
        listeners.add(listener);
    }

    public static synchronized void unregister(
        BroadcastListener listener) {
        listeners.remove(listener);
    }

    public static synchronized void broadcast(
        final String message) {
        for (final BroadcastListener listener: listeners)
            executorService.execute(new Runnable() {
                @Override
                public void run() {
                    listener.receiveBroadcast(message);
                }
            });
    }
}
```

```
    });
}
}
```

In Java 8, you could use lambda expressions for the listeners instead of the interface, and a parameterless expression to create the runnable:

```
for (final Consumer<String> listener: listeners)
    executorService.execute(() ->
        listener.accept(message));
```

Receiving Broadcasts

The receivers need to implement the receiver interface and register to the broadcaster to receive the broadcasts. A listener should be unregistered when the UI expires. When updating the UI in a receiver, it should be done safely as described earlier, by executing the update through the access() method of the UI.

```
@Push
public class PushAroundUI extends UI
    implements Broadcaster.BroadcastListener {

    VerticalLayout messages = new VerticalLayout();

    @Override
    protected void init(VaadinRequest request) {
        ... build the UI ...

        // Register to receive broadcasts
        Broadcaster.register(this);
    }

    // Must also unregister when the UI expires
    @Override
    public void detach() {
        Broadcaster.unregister(this);
        super.detach();
    }

    @Override
    public void receiveBroadcast(final String message) {
        // Must lock the session to execute logic safely
        access(new Runnable() {
            @Override
            public void run() {
                // Show it somehow
                messages.addComponent(new Label(message));
            }
        });
    }
}
```

```
        }
    }):
}
}
```

Sending Broadcasts

To send broadcasts with a broadcaster singleton, such as the one described above, you would only need to call the broadcast() method as follows.

```
final TextField input = new TextField();
sendBar.addComponent(input);

Button send = new Button("Send");
send.addClickListener(new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        // Broadcast the message
        Broadcaster.broadcast(input.getValue());

        input.setValue("");
    }
});
```

11.16. Vaadin CDI Add-on

Vaadin CDI add-on makes it easier to use contexts and dependency injection (CDI) in Vaadin applications. CDI is a Java EE feature especially targeted for web applications, which have well-defined contextual scopes, such as sessions, views, requests, and so forth. The lifecycle of objects, such as beans, can be managed by binding their lifecycles to such contexts. Vaadin CDI enables these features with two additional kinds of Vaadin-specific contextual scopes: UIs and navigation views.

To learn more about Vaadin CDI, the link:[Vaadin CDI Tutorial] gives a hands-on introduction. The source code of the CDI Tutorial demo is available for browsing or cloning at <https://github.com/vaadin/cdi-tutorial>.

11.16.1. CDI Overview

Contexts and dependency injection, defined in the JSR-299 standard, is a Java EE feature that, through a set of services,

helps in improving application architecture by decoupling the management of service object lifecycles from client objects using them. The lifecycle of objects stored in a CDI container is defined by a context. The managed objects or beans are accessed using dependency injection.

CDI builds on the Java concept of beans, but with somewhat different definition and requirements.

Regarding general CDI topics, such as use of qualifiers, interceptors, decorators, event notifications, and other CDI features, we refer you to CDI documentation.

Dependency Injection

Dependency injection is a way to pass dependencies (service objects) to dependent objects (clients) by injecting them in member variables or initializer parameters, instead of managing the lifecycle in the clients or passing them explicitly as parameters. In CDI, injection of a service object to a client is specified by the **@Inject** annotation.

For example, if we have a UI view that depends on user data, we could inject the data in the client as follows:

```
public class MainView extends CustomComponent implements View {  
    @Inject  
    User user;  
  
    ...  
    @Override  
    public void enter(ViewChangeEvent event) {  
        greeting.setValue("Hello, " + user.getName());  
    }  
}
```

In addition to injecting managed beans with the annotation, you can query for them from the bean manager.

Contexts and Scopes

Contexts in CDI are services that manage the lifecycle of objects and handle their injection. Generally speaking, a context is a situation in which an instance is used with a unique identity. Such objects are essentially "singletons" in the context. While conventional singletons are application-wide, objects managed by a CDI container can be "singletons" in a more

narrow scope: a user session, a particular UI instance associated with the session, a view within the UI, or even just a single request. Such a context defines the lifecycle of the object: its creation, use, and finally its destruction.

As a very typical example in a web application, you would have a user data object associated with a user session.

```
@SessionScoped  
public class User {  
    private String name;  
  
    public void setName(String name) {this.name = name;}  
    public String getName() {return name;}  
}
```

Now, when you need to refer to the user, you can use CDI injection to inject the session-scoped "singleton" to a member variable or a constructor parameter.

```
public class MainView extends CustomComponent implements View {  
    @Inject  
    User user;  
  
    ...  
  
    @Override  
    public void enter(ViewChangeEvent event) {  
        greeting.setValue("Hello, " + user.getName());  
    }  
}
```

11.16.2. Installing Vaadin CDI Add-on

Vaadin CDI requires a Java EE 7 compatible servlet container, such as Glassfish or Apache TomEE Web Profile, as mentioned for the reference toolchain in Section 2.2, “A Reference Tool-chain”.

To install the Vaadin CDI add-on, either define it as an Ivy or Maven dependency or download it from the Vaadin Directory add-on page at <vaadin.com/directory#addon/vaadin-cdi>. See Chapter 17, *Using Vaadin Add-ons* for general instructions for installing and using Vaadin add-ons.

The Ivy dependency is as follows:

```
<dependency org="com.vaadin" name="vaadin-cdi"  
rev="latest.release"/>
```

The Maven dependency is as follows:

```
<dependency>  
  <groupId>com.vaadin</groupId>  
  <artifactId>vaadin-cdi</artifactId>  
  <version>[replaceable]LATEST</version>  
</dependency>  
<dependency>  
  <groupId>javax.enterprise</groupId>  
  <artifactId>cdi-api</artifactId>  
  <version>[replaceable]1.2</version>  
</dependency>
```

11.16.3. Preparing Application for CDI

A Vaadin application that uses CDI must have a file named beans.xml in the WEB-INF directory. The file can be completely empty (it has content only in certain limited situations), but it must be present.

The application should not have a servlet extending **Vaadin-Servlet**, as Vaadin servlet has its own **VaadinCDIServlet** that is deployed automatically. If you need multiple servlets or need to customize the Vaadin CDI servlet, see Section 11.16.6, "Deploying CDI UIs and Servlets".

11.16.4. Injecting a UI with @CDIUI

Vaadin CDI offers an easier way to instantiate UIs and to define the URL mapping for them than the usual ways described in Section 5.9, "Deploying an Application". To define a UI class that should be instantiated for a given URL, you simply need to annotate the class with **@CDIUI**. It takes an optional URL path as parameter.

```
@CDIUI("myniceui")  
@Theme("valo")  
public class MyNiceUI extends UI {  
  ...
```

Giving empty UI path maps the UI to the root of the application context.

```
@CDIUI(88)
```

If the optional UI path is not given, the path is determined automatically from the class name by removing a possible "-UI" suffix in the class name, making it lower-case, and for capitalized letters, a hyphen is added. For example, a UI with class name **MyNiceUI** would have path my-nice. The URL consists of the server address, application context, and the UI path. For example, when running a Vaadin application in a development workstation, you would have URL such as <http://localhost:8080/myproject/my-nice>.

UI path mappings are reported in the server log during deployment.

See Section 11.16.6, "Deploying CDI UIs and Servlets" for how to handle servlet URL mapping of CDI UIs when working with multiple servlets in the same web application.

11.16.5. Scopes

As in programming languages, where a variable name refers to a unique object within the scope of the variable, a CDI scope is a context in which an object has unique identity. In CDI, objects to be injected are identified by their type and any qualifiers they may have. The scope can be defined as an annotation to the service class as follows:

```
@SessionScoped  
public class User {  
    ...}
```

CDI defines a number of scopes. Note that the standard CDI scopes are defined under the `javax.enterprise.context` package and Vaadin CDI scopes under `com.vaadin.cdi`, while JSF scopes are defined in `javax.faces.bean`.

UI Scope

UI-scoped beans are uniquely identified within a UI instance, that is, a browser window or tab.

Vaadin CDI provides two annotations for the UI scope, differing in how they enable proxies, as explained later.

@UIScoped(com.vaadin.cdi)

Injection with this annotation will create a direct reference to the bean rather than a proxy. There are some limitations when not using proxies. Circular references (injecting A to B and B to A) will not work, and neither do CDI interceptors and decorators.

@NormalUIScoped(com.vaadin.cdi)

As **@UIScoped**, but injecting a managed bean having this annotation injects a proxy for the bean instead of a direct reference. This is the normal behaviour with CDI, as many CDI features utilize the proxy.

Defining a CDI view (annotated with **@CDIView** as described later) as **@UIScoped** makes the view retain the same instance when the user navigates away and back to the view.

View Scopes

The lifecycle of a view-scoped bean starts when the user navigates to a view referring to the object and ends when the user navigates out of the view (or when the UI is closed or expires).

Vaadin CDI provides two annotations for the view scope, differing in how they enable proxies, as explained later.

@ViewScoped(com.vaadin.cdi)

Injection with this annotation will create a direct reference to the bean rather than a proxy. There are some limitations when not using proxies. Circular references (injecting A to B and B to A) will not work, and neither do CDI interceptors and decorators.

@NormalViewScoped(com.vaadin.cdi)

As **@NormalScoped**, except that injecting with this annotation will create a proxy for the contextual instance rather than provide the contextual instance itself. See the explanation of proxies below.

Standard CDI Scopes

@ApplicationScoped

Application-scoped beans are shared by all servlets in the web application, and are essentially equal to singletons.//TODO This is just a guess - is it true? Note that referencing application-scoped beans is not thread-safe and access must be synchronized.

@SessionScoped

The lifecycle and visibility of session-scoped beans is bound to a HTTP or user session, which in Vaadin applications is associated with the **VaadinSession** (see Section 5.8.3, "User Session"). This is a very typical scope to store user data, as is done in many examples in this section, or database connections. The lifecycle of session-scoped beans starts when a user opens the page for a UI in the browser, and ends when the session expires after the last UI in the session is closed.

Proxies vs Direct References

CDI uses proxy objects to enable many of the CDI features, by hooking into message-passing from client to service beans. Under the hood, a proxy is an instance of an automatically generated class that extends the proxied bean type, so communicating through a proxy occurs transparently, as it has the same polymorphic type as the actual bean. Whether proxying is enabled or not is defined in the scope: CDI scopes are either *normal scopes*, which can be proxied, or *pseudo-scopes*, which use direct references to injected beans.

The proxying mechanism creates some requirements for injecting objects in normal scope:

- The objects may not be primitive types or arrays
- The bean class must not be final
- The bean class must not have final methods

Beans annotated with **@UIScoped** or **@ViewScoped** use a pseudoscope, and are therefore injected with direct references to the bean instances, while **@NormalUIScoped** and **@NormalViewScoped** beans will use a proxy for communicating with the beans.

When using proxies, be aware that it is not guaranteed that the hashCode() or equals() will match when comparing a proxy to its underlying instance. It is imperative to be aware of this when, for example, adding proxies to a Collection.

You should avoid using normal scopes with Vaadin components, as proxies may not work correctly within the Vaadin framework. If Vaadin CDI plugin detects such use, it displays a warning such as the following:

INFO: The following Vaadin components are injected into normal scoped contexts:

 @NormalUIScoped org.example.User

This approach uses proxy objects and has not been extensively tested with the framework. Please report any unexpected behavior. Switching to a pseudo-scoped context may also resolve potential issues.

11.16.6. Deploying CDI UIs and Servlets

Vaadin CDI hooks into Vaadin framework by using a special **VaadinCDIServlet**. As described earlier, you do not need to map an URL path to a UI, as it is handled by Vaadin CDI. However, in the following, we go through some cases where you need to customize the servlet or use CDI with non-CDI servlets and UIs in a web application.

Defining Servlet Root with @URLMapping

CDI UIs are managed by a CDI servlet (**VaadinCDIServlet**), which is by default mapped to the root of the application context. For example, if the name of a CDI UI is "my-cdi" and application context is /myproject, the UI would by default have URL "/myproject/my-cdi". If you do not want to have the servlet mapped to context root, you can use the **@URLMapping** annotation to map all CDI UIs to a sub-path. The annotation must be given to only one CDI UI, usually the one with the default ("") path.

For example, if we have a root UI and another:

```
@CDIUI("") //At CDI servlet root  
@URLMapping("mycdiuis") //Define CDI Servlet root  
public class MyCDIRootUI extends UI {...}  
  
@CDIUI("another")  
public class AnotherUI extends UI {...}
```

These two UIs would have URLs /myproject/mycdiuis and /myproject/mycdiuis/another, respectively.

You can also map the CDI servlet to another URL in servlet definition in web.xml, as described the following.

Mixing With Other Servlets

The **VaadinCDIServlet** is normally used as the default servlet, but if you have other servlets in the application, such as for non-CDI UIs, you need to define the CDI servlet explicitly in the web.xml. You can map the servlet to any URL path, but perhaps typically, you define it as the default servlet as follows, and map the other servlets to other URL paths:

```
<web-app>  
...  
  
<servlet>  
  <servlet-name>Default</servlet-name>  
  <servlet-class>  
    com.vaadin.cdi.internal.VaadinCDIServlet  
  </servlet-class>  
</servlet>  
  
<servlet-mapping>  
  <servlet-name>Default</servlet-name>  
  <url-pattern>[replaceable]/mycdiuis</url-pattern>  
</servlet-mapping>  
  
<servlet-mapping>  
  <servlet-name>Default</servlet-name>  
  <url-pattern>/VAADIN</url-pattern>  
</servlet-mapping>  
</web-app>
```

With such a setting, paths to CDI UIs would have base path /myapp/mycdiuis, to which the (optional) UI path would be

appended. The `/VAADIN/*` only needs to be mapped to the servlet if there are no other Vaadin servlets.

Custom Servlets

When customizing the Vaadin servlet, as outlined in Section 5.8.2, “Vaadin Servlet, Portlet, and Service”, you simply need to extend `com.vaadin.cdi.internal.VaadinCDIServlet` instead of `com.vaadin.servlet.VaadinServlet`.

The custom servlet must not have `@WebServlet` annotation or `@VaadinServletConfiguration`, as you would normally with a Vaadin servlet, as described in Section 5.9, “Deploying an Application”.

11.17. Vaadin Spring Add-on

Vaadin Spring and Vaadin Spring Boot add-ons make it easier to use Spring in Vaadin applications. Vaadin Spring enables Spring dependency injection with custom UI and view providers, and provides three custom scopes: `UIScope`, `ViewScope`, and `VaadinSessionScope`.

API documentation for add-ons is available at:

- Vaadin Spring API at vaadin.com/api/vaadin-spring
- Vaadin Spring Boot API at vaadin.com/api/vaadin-spring-boot

To learn more about Vaadin Spring, the Vaadin Spring Tutorial gives a hands-on introduction. The source code of the Spring Tutorial demo is available for browsing or cloning at github.com/Vaadin/spring-tutorial.

11.17.1. Spring Overview

Spring Framework is a Java application framework that provides many useful services for building applications, such as authentication, authorization, data access, messaging, testing, and so forth. In the Spring core, one of the central features is dependency injection, which accomplishes inversion of control for dependency management in managed beans. Other Spring features rely on it extensively. As such,

Spring offers capabilities similar to CDI, but with integration with other Spring services. Spring is well-suited for applications where Vaadin provides the UI layer and Spring is used for other aspects of the application logic.

Spring Boot is a Spring application development tool that allows creating Spring applications easily. *Vaadin Spring Boot* builds on Spring Boot to allow creating Vaadin Spring applications easily. It starts up a servlet, handles the configuration of the application context, registers a UI provider, and so forth.

Regarding general Spring topics, we recommend the following Spring documentation:

Dependency Injection

Dependency injection is a way to pass dependencies (service objects) to dependent objects (clients) by injecting them in member variables or initializer parameters, instead of managing the lifecycle in the clients or passing them explicitly as parameters. In Spring, injection of a service object to a client is configured with the **@Autowired** annotation.

For a very typical example in a web application, you could have a user data object associated with a user session:

```
@SpringComponent  
@VaadinSessionScope  
public class User implements Serializable {  
    private String name;  
  
    public void setName(String name) {this.name = name;}  
    public String getName() {return name;}  
}
```

The **@SpringComponent** annotation allows for automatic detection of managed beans by Spring. (The annotation is exactly the same as the regular Spring **@Component**, but has been given an alias, because Vaadin has a Component interface, which can cause trouble.)

Now, if we have a UI view that depends on user data, we could inject the data in the client as follows:

```
public class MainView extends CustomComponent implements View {  
    User user;
```

```
Label greeting = new Label();

@Autowired
public MainView(User user) {
    this.user = user;
    ...
}

@Override
public void enter(ViewChangeEvent event) {
    // Then you can use the injected data
    // for some purpose
    greeting.setValue("Hello, " + user.getName());
}
}
```

Here, we injected the user object in the constructor. The user object would be created automatically when the view is created, with all such references referring to the same shared instance in the scope of the Vaadin user session.

Contexts and Scopes

Contexts in Spring are services that manage the lifecycle of objects and handle their injection. Generally speaking, a context is a situation in which an instance is used with a unique identity. Such objects are essentially "singletons" in the context. While conventional singletons are application-wide, objects managed by a Spring container can be "singletons" in a more narrow *scope*: a user session, a particular UI instance associated with the session, a view within the UI, or even just a single request. Such a context defines the lifecycle of the object: its creation, use, and finally its destruction.

Earlier, we introduced a user class defined as session-scoped:

```
@SpringComponent
@VaadinSessionScope
public class User {
```

Now, when you need to refer to the user, you can use Spring injection to inject the session-scoped "singleton" to a member variable or a constructor parameter; the former in the following:

```
public class MainView extends CustomComponent implements View {
    @Autowired
    User user;
```

```
Label greeting = new Label();
...
@Override
public void enter(ViewChangeEvent event) {
    greeting.setValue("Hello, " + user.getName());
}
}
```

11.17.2. Quick Start with Vaadin Spring Boot

The Vaadin Spring Boot is an add-on that allows for easily creating a project that uses Vaadin Spring. It is meant to be used together with Spring Boot, which enables general Spring functionalities in a web application.

You can use the Spring Initializr at start.spring.io website to generate a project, which you can then download as a package and import in your IDE. Pick Vaadin as a dependency on the site to include all the required Vaadin Spring dependencies and auto-configuration. The generated project is a Spring application stub; you need to add at least a UI class to the generated project.

See the Vaadin Spring Tutorial for detailed instructions for using Spring Boot.

11.17.3. Installing Vaadin Spring Add-on

Vaadin Spring requires a Java EE 7 compatible servlet container, such as Glassfish or Apache TomEE Web Profile, as mentioned for the reference toolchain in Section 2.2, “A Reference Toolchain”.

To install the Vaadin Spring and/or Vaadin Spring Boot add-ons, either define them as an Ivy or Maven dependency or download from the Vaadin Directory add-on page at vaadin.com/directory#addon/vaadin-spring or vaadin.com/directory#addon/vaadin-spring-boot. See Chapter 17, *Using Vaadin Add-ons* for general instructions for installing and using Vaadin add-ons.

The Ivy dependency is as follows:

```
<dependency org="com.vaadin" name="vaadin-spring"
    rev="latest.release"/>
```

The Maven dependency is as follows:

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-spring</artifactId>
    <version>LATEST</version>
</dependency>
```

For Vaadin Spring Boot, depending on vaadin-spring-boot-starter will include all the required Vaadin dependencies.

11.17.4. Preparing Application for Spring

A Vaadin application that uses Spring must have a file named applicationContext.xml in the WEB-INF directory. Using Spring Initializr automatically generates a suitable file, but if you configure Vaadin Spring manually, you can follow the model below.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns: xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns: context="http://www.springframework.org/schema/context"
    xsi: schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.1.xsd">

    <!-- Configuration object -->
    <bean class="com.example.myapp.MySpringUI.MyConfiguration" />

    <!-- Location for automatically scanned beans -->
    <context:component-scan
        base-package="com.example.myapp.domain" />
</beans>
```

The application should not have a servlet extending **Vaadin-Servlet**, as Vaadin servlet has its own **SpringVaadinServlet** that is deployed automatically. If you need multiple servlets or need to customize the Vaadin Spring servlet, see Section 11.17.8, “Deploying Spring UIs and Servlets”.

You can configure managed beans explicitly in the file, or configure them to be scanned using the annotations, which is the preferred way described in this section.

11.17.5. Injecting a UI with `@SpringUI`

Vaadin Spring offers an easier way to instantiate UIs and to define the URL mapping for them than the usual ways described in Section 5.9, “Deploying an Application”. It is also needed for enabling Spring features in the UI. To define a UI class that should be instantiated for a given URL, you simply need to annotate the class with `@SpringUI`. It takes an optional path as parameter.

```
@SpringUI(path="/myniceui")
@Theme("valo")
public class MyNiceUI extends UI {
    ...
}
```

The path in the URL for accessing the UI consists of the context path of the application and the UI path, for example, /myapp/myniceui. Giving empty UI path maps the UI to the root of the application context, for example, /myapp.

`@SpringUI`

See Section 11.17.8, “Deploying Spring UIs and Servlets” for how to handle servlet URL mapping of Spring UIs when working with multiple servlets in the same web application.

11.17.6. Scopes

As in programming languages, where a variable name refers to a unique object within the scope of the variable, an object has unique identity within a scope in Spring. However, instead of identifying the objects by variable names, they are identified by their type (object class) and any qualifiers they may have.

The scope of an object can be defined with an annotation to the class as follows:

```
@VaadinSessionScope
public class User {
    ...
}
```

Defining a scope in Spring is normally done with the `@Scope` annotation. For example, `@Scope("prototype")` creates a new instance every time one is requested/auto-wired. Such standard scopes can be used with some limitations. For ex-

ample, Spring session and request scopes do not work in background threads and with certain push transport modes.

Vaadin Spring provides three scopes useful in Vaadin applications: a session scope, a UI scope, a view scope, all defined in the com.vaadin.spring.annotation package.

@VaadinSessionScope

The session scope is the broadest of the custom scopes defined in Vaadin Spring. Objects in the Vaadin session scope are unique in a user session, and shared between all UIs open in the session. This is the most basic scope in Vaadin applications, useful for accessing data for the user associated with the session. It is also useful when updating UIs from a background thread, as in those cases the UI access is locked on the session and also data should be in that scope.

@UIScope

UI-scoped beans are uniquely identified within a UI instance, that is, a browser window or tab. The lifecycle of UI-scoped beans is bound between the initialization and closing of a UI. Whenever you inject a bean, as long as you are within the same UI, you will get the same instance.

Annotating a Spring view (annotated with **@SpringView** as described later) also as **@UIScoped** makes the view retain the same instance when the user navigates away and back to the view.

@ViewScope

The annotation enables the view scope in a bean. The lifecycle of such a bean starts when the user navigates to a view referring to the object and ends when the user navigates out of the view (or when the UI is closed or expires).

Views themselves are by default view-scoped, so a new instance is created every time the user navigates to the view.

11.17.7. Access Control

Access control for views can be implemented by registering beans implementing ViewAccessControl or ViewInstanceAccessControl, which can restrict access to the view either before or after a view instance is created.

Integration with authorization solutions, such as Spring Security, is provided by additional unofficial add-ons on top of Vaadin Spring.

Access Denied View

If access to a view is denied by an access control bean, the access denied view is shown for it. For non-existing views, the error view is shown. You can set up an "Access Denied" view that is shown if the access is denied with setAccessDeniedViewClass() in **SpringViewProvider**, and an error view with setErrorView() in **SpringNavigator**. The same view can also be used both as an access denied view and as an error view to hide the existence of views the user is not allowed to access.

```
@Autowired
SpringViewProvider viewProvider;
@Autowired
SpringNavigator navigator;

@Override
protected void init(VaadinRequest request) {
    // Set up access denied view
    viewProvider.setAccessDeniedViewClass(
        MyAccessDeniedView.class);
    // Set up error view
    navigator.setErrorView(MyErrorView.class);
```

Note that the error view can also be a class with which an error view bean is found. In this case, the error view must be UI scoped.

11.17.8. Deploying Spring UIs and Servlets

Vaadin Spring hooks into Vaadin framework by using a special **SpringVaadinServlet**. As described earlier, you do not need to map an URL path to a UI, as it is handled by Vaadin Spring. However, in the following, we go through some cases where

you need to customize the servlet or use Spring with non-Spring servlets and UIs in a web application.

Custom Servlets

When customizing the Vaadin servlet, as outlined in Section 5.8.2, "Vaadin Servlet, Portlet, and Service", you simply need to extend **com.vaadin.spring.server.SpringVaadinServlet** instead of **com.vaadin.servlet.VaadinServlet**.

```
@WebServlet(value = "/", asyncSupported = true)
public class MySpringServlet extends SpringVaadinServlet { }
```

The custom servlet must not have **@VaadinServletConfiguration**, as you would normally with a Vaadin servlet, as described in Section 5.9, "Deploying an Application".

Defining Servlet Root

Spring UIs are managed by a Spring servlet (**SpringVaadinServlet**), which is by default mapped to the root of the application context. For example, if the name of a Spring UI is "my-spring-ui" and application context is /myproject, the UI would by default have URL "/myproject/my-spring-ui". If you do not want to have the servlet mapped to context root, you can use a **@WebServlet** annotation for the servlet or a web.xml definition to map all Spring UIs to a sub-path.

For example, if we have a root UI and another:

```
@SpringUI(path "") // At Spring servlet root
public class MySpringRootUI extends UI {...}
```

```
@SpringUI("another")
public class AnotherUI extends UI {...}
```

Then define a path for the servlet by defining a custom servlet:

```
@WebServlet(value = "/myspringuis/*", asyncSupported = true)
public class MySpringServlet extends SpringVaadinServlet { }
```

These two UIs would have URLs /myproject/myspringuis and /myproject/myspringuis/another, respectively.

You can also map the Spring servlet to another URL in servlet definition in web.xml, as described the following.

Mixing With Other Servlets

The **SpringVaadinServlet** is normally used as the default servlet, but if you have other servlets in the application, such as for non-Spring UIs, you need to define the Spring servlet explicitly in the web.xml. You can map the servlet to any URL path, but perhaps typically, you define it as the default servlet as follows, and map the other servlets to other URL paths:

```
<web-app>
...
<servlet>
  <servlet-name>Default</servlet-name>
  <servlet-class>
    com.vaadin.spring.server.SpringVaadinServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Default</servlet-name>
  <url-pattern>/myspringuis/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>Default</servlet-name>
  <url-pattern>/VAADIN/*</url-pattern>
</servlet-mapping>
</web-app>
```

With such a setting, paths to Spring UIs would have base path /myapp/myspringuis, to which the (optional) UI path would be appended. The /VAADIN/* only needs to be mapped to the servlet if there are no other Vaadin servlets.

Chapter 12

Portal Integration

12.1. Overview	485
12.2. Creating a Generic Portlet in Eclipse	486
12.3. Developing Vaadin Portlets for Liferay	488
12.4. Portlet UI	497
12.5. Deploying to a Portal	500
12.6. Vaadin IPC for Liferay	507

12.1. Overview

Vaadin supports running UIs as portlets in a portal, as defined in the JSR-286 (Java Portlet API 2.0) standard. A portlet UI is defined just as a regular UI, but deploying to a portal is somewhat different from deployment of regular web applications, requiring special portlet descriptors, etc. Creating the portlet project with the Vaadin Plugin for Eclipse or a Maven archetype automatically generates the necessary descriptors.

In addition to providing user interface through the Vaadin UI, portlets can integrate with the portal to switch between portlet modes and process special portal requests, such as actions and events.

While providing generic support for all portals implementing the standard, Vaadin especially supports the Liferay portal and the needed portal-specific configuration in this chapter is given for Liferay. Vaadin also has a special Liferay IPC add-on to enable communication between portlets.

12.2. Creating a Generic Portlet in Eclipse

This section has not yet been updated for Vaadin Framework 8.

Here we describe the creation of a generic portlet project in Eclipse. You can use the Maven archetypes also in other IDEs or without an IDE.

For Liferay portlet development, you may instead want to use the Maven archetype or Liferay IDE to create the project, as described in Section 12.3, “Developing Vaadin Portlets for Liferay”.

12.2.1. Creating a Project with Vaadin Plugin

The Vaadin Plugin for Eclipse has a wizard for easy creation of generic portlet projects. It creates a UI class and all the necessary descriptor files.

Creating a portlet project is almost identical to the creation of a regular Vaadin servlet application project. For a full treatment of the New Project Wizard and the possible options, please see Section 3.4.1, “Creating a Maven Project”.

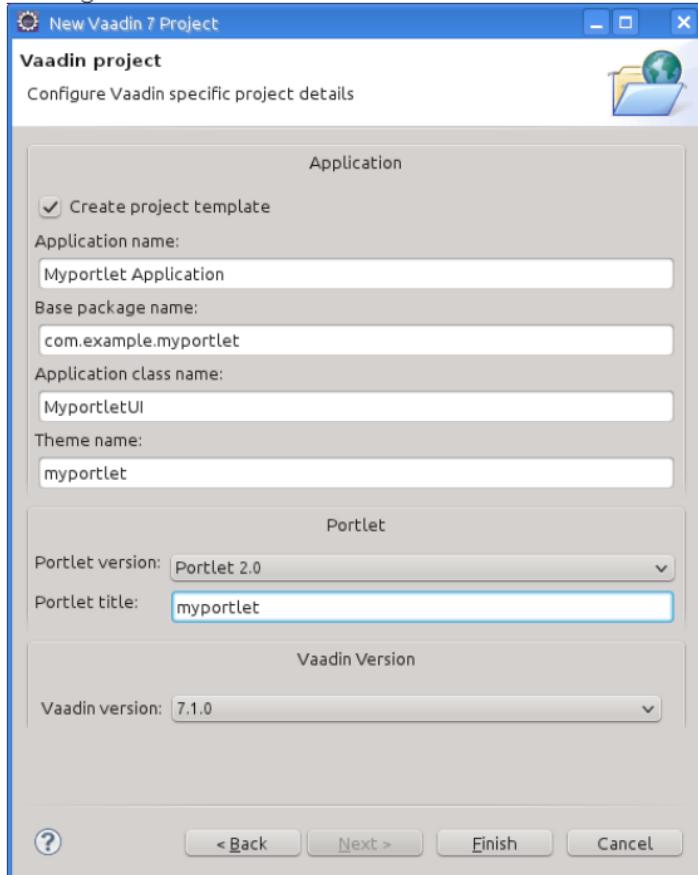
1. Start creating a new project by selecting from the menu **File □ New □ Project...+**
1. In the **New Project** window that opens, select **Web □ Vaadin 7 Project** and click **Next**.
2. In the **Vaadin Project** step, you need to set the basic web project settings. You need to give at least the project name, the runtime, select **Generic Portlet** for the **Deployment configuration**: the default values should be good for the other settings.

You can click **Finish** here to use the defaults for the rest of the settings, or click **Next**.

3. The settings in the **Web Module** step define the basic servlet-related settings and the structure of the web application project. All the settings are pre-filled, and

you should normally accept them as they are and click **Next**.

4. The **Vaadin project** step page has various Vaadin-specific application settings. These are largely the same as for regular applications. Setting them here is easiest - later some of the changes require changes in several different files. The **Create portlet template** option should be automatically selected. You can give another portlet title if you want. You can change most of the settings afterward.



Create project template

Creates a UI class and all the needed portlet deployment descriptors.

Application name

The application name is used in the title of the browser window, which is usually invisible in portlets, and as an identifier, either as is or with a suffix, in various deployment descriptors.

Base package name

Java package for the UI class.

Application class name

Name of the UI class. The default is derived from the project name.

Theme name

Name of the custom portlet theme to use.

Portlet version

Same as in the project settings.

Portlet title

The portlet title, defined in portlet.xml, can be used as the display name of the portlet (at least in Liferay). The default value is the project name. The title is also used as a short description in liferay-plugin-package.properties.

Vaadin version

Same as in the project settings.

Finally, click **Finish** to create the project.

5. Eclipse may ask you to switch to J2EE perspective. A Dynamic Web Project uses an external web server and the J2EE perspective provides tools to control the server and manage application deployment. Click **Yes**.

12.3. Developing Vaadin Portlets for Liferay

This section has not yet been updated for Vaadin Framework 8.

A Vaadin portlet requires resources such as the server-side Vaadin libraries, a theme, and a widget set. You have two basic ways to deploy these: either globally in Liferay, so that the re-

sources are shared between all Vaadin portlets, or as self-contained WARs, where each portlet carries their own resources.

The self-contained way is easier and more flexible to start with, as the different portlets may have different versions of the resources. Currently, the latest Maven archetypes support the self-contained portlets, while with portlets created with the Vaadin Plugin for Eclipse only support globally deployed resources.

Using shared resources is more efficient when you have multiple Vaadin portlets on the same page, as they can share the common resources. However, they must use exactly same Vaadin version. This is recommended for production environments, where you can even serve the theme and widget set from a front-end server. You can install the shared resources as described in Section 12.3.5, "Installing Vaadin Resources".

At the time of writing, the latest Liferay release 6.2 is bundled with a version of Vaadin release 6. If you want to use Vaadin 7 portlets with shared resources, you first need to remove the old ones as described in Section 12.3.4, "Removing the Bundled Installation".

12.3.1. Defining Liferay Profile for Maven

When creating a Liferay portlet project with a Maven archetype or the Liferay IDE, you need to define a Liferay profile. With the Liferay IDE, you can create it when you create the project, as described in Section 12.3.3, "Creating a Portlet Project in Liferay IDE", but for creating a project from the Maven archetype, you need to define it manually.

Defining Profile in settings.xml

Liferay profile can be defined either in the user or in the global settings.xml file for Maven. The global settings file is located in \${MAVEN_HOME}/conf/settings.xml and the user settings file in \${USER_HOME}/.m2/settings.xml. To create a user settings file, copy at least the relevant headers and root element from the global settings file.

```
...  
<profile>
```

```

<id>liferay</id>
<properties>
<liferayInstall>/opt/liferay-portal-6.2-ce-ga2
</liferayInstall>
<plugin.type>portlet</plugin.type>
<liferay.version>6.2.1</liferay.version>
<liferay.maven.plugin.version>6.2.1
</liferay.maven.plugin.version>
<liferay.auto.deploy.dir>$[liferayInstall]/deploy
</liferay.auto.deploy.dir>

<!-- Application server version - here for Tomcat -->
<liferay.tomcat.version>7.0.42</liferay.tomcat.version>
<liferay.tomcat.dir>
    ${liferayInstall}/tomcat-$[liferay.tomcat.version]
</liferay.tomcat.dir>

<liferay.app.server.deploy.dir>$[liferay.tomcat.dir]/webapps
</liferay.app.server.deploy.dir>
<liferay.app.server.lib.global.dir>$[liferay.tomcat.dir]/lib/ext
</liferay.app.server.lib.global.dir>
<liferay.app.server.portal.dir>$[liferay.tomcat.dir]/webapps/ROOT
</liferay.app.server.portal.dir>
</properties>
</profile>

```

The parameters are as follows:

liferayinstall

Full (absolute) path to the Liferay installation directory.

liferay.version

Liferay version by the Maven version numbering scheme.

The first two (major and minor) numbers are same as in the installation package. The third (maintenance) number starts from 0 with first GA (general availability) release.

liferay.maven.plugin.version

This is usually the same as the Liferay version.

liferay.auto.deploy.dir

The Liferay auto-deployment directory. It is by default deploy under the Liferay installation path.

liferay.tomcat.version(optional)

If using Tomcat, its version number.

`liferay.tomcat.dir`

Full (absolute) path to Tomcat installation directory. For Tomcat bundled with Liferay, this is under the Liferay installation directory.

`liferay.app.server.deploy.dir`

Directory where portlets are deployed in the application server used for Liferay. This depends on the server - for Tomcat it is the webapps directory under the Tomcat installation directory.

`liferay.app.server.lib.global.dir`

Library path where libraries globally accessible in the application server should be installed.

`liferay.app.server.portal.dir`

Deployment directory for static resources served by the application server, under the root path of the server.

If you modify the settings after the project is created, you need to touch the POM file in the project to have the settings re-loaded.

Activating the Maven Profile

The Maven 2 Plugin for Eclipse (m2e) must know which Maven profiles you use in a project. This is configured in the Maven section of the project properties. In the **Active Maven Profiles** field, enter the profile ID defined in the settings.xml file, as illustrated in Figure 12.1, "Activating Maven Liferay Profile".

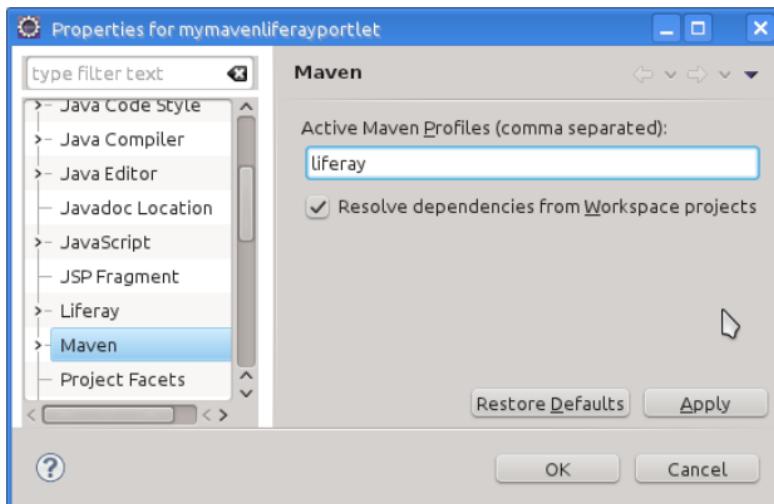


Figure 12.1. Activating Maven Liferay Profile

12.3.2. Creating a Portlet Project with Maven

Creation of Vaadin a Maven project is described in Section 3.7, "Creating a Project with Maven". For a Liferay project, you should use the `vaadin-archetype-liferay-portlet`.

Archetype Parameters

The archetype has a number of parameters. If you use Maven Plugin for Eclipse (m2e) to create the project, you get to enter the parameters after selecting the archetype, as shown in Figure 12.2, "Liferay Project Archetype Parameters".

Minimally, you just need to enter the artifact ID. To activate the Maven profile created as described earlier in Section 12.3.1, "Defining Liferay Profile for Maven", you need to specify the profile in the **Profiles** field under the **Advanced** section.

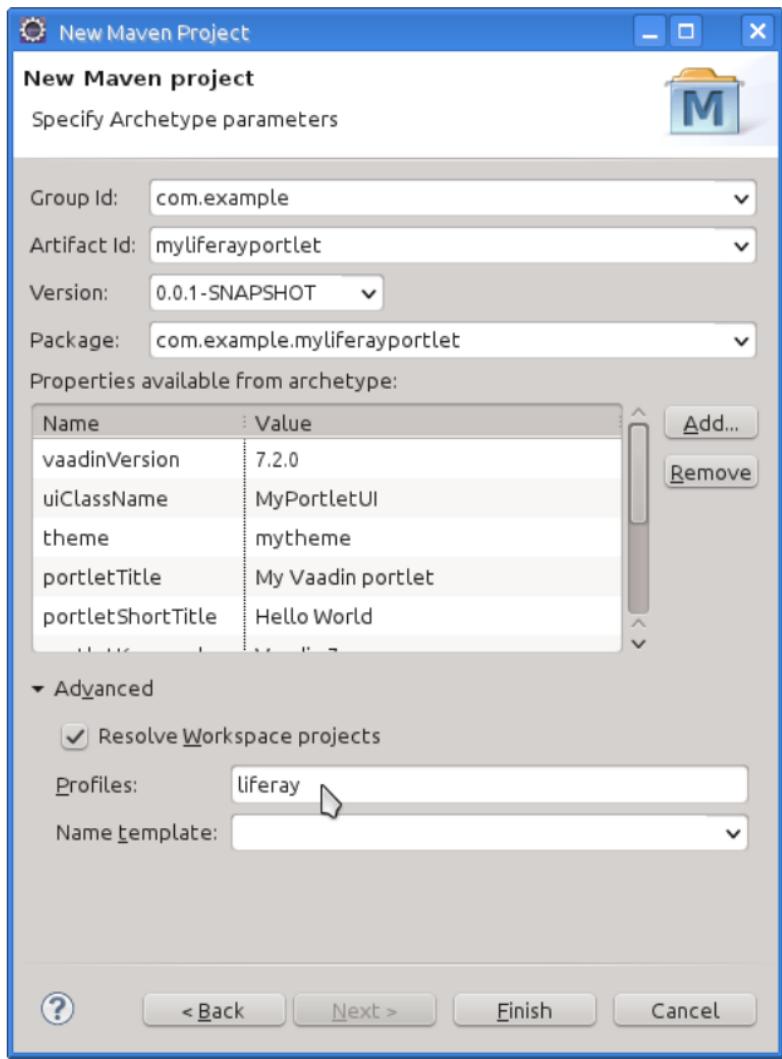


Figure 12.2. Liferay Project Archetype Parameters

The other parameters are the following:

vaadinVersion

Vaadin release version for the Maven dependency.

uiClassName

Class name of the UI class stub to be created.

theme

Theme to use. You can use either a project theme, which must be compiled before deployment, or use one of the default themes.

portletTitle

Title shown in the portlet title bar.

portletShortTitle

Title shown in contexts where a shorter title is preferred.

portletKeywords

Keywords for finding the portlet in Liferay.

portletDescription

A description of the portlet.

portletName

Identifier for the portlet, used for identifying it in the configuration files.

portletDisplayName

Name of the portlet for contexts where it is displayed.

12.3.3. Creating a Portlet Project in Liferay IDE

Liferay IDE, which you install in Eclipse as plugins just like the Vaadin plugin, enables a development environment for Liferay portlets. Liferay IDE allows integrated deployment of portlets to Liferay, just like you would deploy servlets to a server in Eclipse. The project creation wizard supports creation of Vaadin portlets.

Loading widget sets, themes, and the Vaadin JAR from a portlet is possible as long as you have a single portlet, but causes a problem if you have multiple portlets. To solve this, Vaadin portlets need to use a globally installed widget set, theme, and Vaadin libraries.

Liferay 6.2, which is the latest Liferay version at the time of publication of this book, comes bundled with an older Vaadin 6 version. If you want to use Vaadin 7, you need to remove the bundled version and install the newer one manually as described in this chapter.

In these instructions, we assume that you use Liferay bundled with Apache Tomcat, although you can use almost any other application server with Liferay just as well. The Tomcat installation is included in the Liferay installation package, under the tomcat-x.x.x directory.

12.3.4. Removing the Bundled Installation

Before installing a new Vaadin version, you need to remove the version bundled with Liferay. You need to remove the Vaadin library JAR from the library directory of the portal and the VAADIN directory from under the root context. For example, with Liferay bundled with Tomcat, they are usually located as follows:

- tomcat-x.x.x/webapps/ROOT/html/VAADIN
- tomcat-x.x.x/webapps/ROOT/WEB-INF/lib/vaadin.jar

12.3.5. Installing Vaadin Resources

To use common resources needed by multiple Vaadin portlets, you can install them globally as shared resources as described in the following.

If you are installing Vaadin in a Liferay version that comes bundled with an older version of Vaadin, you first need to remove the resources as described in Section 12.3.4, “Removing the Bundled Installation”.

In the following, we assume that you use only the built-in “reindeer” theme in Vaadin and the default widget set.

1. Get the Vaadin installation package from the Vaadin download page
2. Extract the following Vaadin JARs from the installation package: vaadin-server.jar and vaadin-shared.jar, as well as the vaadin-shared-deps.jar and jsoup.jar dependencies from the lib folder
3. Rename the JAR files as they were listed above, without the version number

4. Put the libraries in tomcat-*x.x.x*/webapps/ROOT/WEB-INF/lib/
5. Extract the VAADIN folders from vaadin-server.jar, vaadin-themes.jar, and vaadin-client-compiled.jar and copy their contents to tomcat-*x.x.x*/webapps/ROOT/html/VAADIN.

```
$ cd tomcat-x.x.x/webapps/ROOT/html  
$ unzip path-to/vaadin-server-7.1.0.jar 'VAADIN/*'  
$ unzip path-to/vaadin-themes-7.1.0.jar 'VAADIN/*'  
$ unzip path-to/vaadin-client-compiled-7.1.0.jar 'VAADIN/*'
```

You need to define the widget set, the theme, and the JAR in the portal-ext.properties configuration file for Liferay, as described earlier. The file should normally be placed in the Liferay installation directory. See Liferay documentation for details on the configuration file.

Below is an example of a portal-ext.properties file:

```
# Path under which the VAADIN directory is located.  
# (/html is the default so it is not needed.)  
# vaadin.resources.path=/html  
  
# Portal-wide widget set  
vaadin.widgetset=com.vaadin.server.DefaultWidgetSet  
  
# Theme to use  
# This is the default theme if nothing is specified  
vaadin.theme=reindeer
```

The allowed parameters are:

vaadin.resources.path

Specifies the resource root path under the portal context. This is /html by default. Its actual location depends on the portal and the application server; in Liferay with Tomcat it would be located at webapps/ROOT/html under the Tomcat installation directory.

`vaadin.widgetset`

The widget set class to use. Give the full path to the class name in the dot notation. If the parameter is not given, the default widget set is used.

`vaadin.theme`

Name of the theme to use. If the parameter is not given, the default theme is used, which is reindeer.

You will need to restart Liferay after creating or modifying the portal-ext.properties file.

12.4. Portlet UI

A portlet UI is just like in a regular Vaadin application, a class that extends `com.vaadin.ui.UI`.

```
@Theme("myportlet")
public class MyportletUI extends UI {
    @Override
    protected void init(VaadinRequest request) {
        final VerticalLayout layout = new VerticalLayout();
        layout.setMargin(true);
        setContent(layout);

        Button button = new Button("Click Me");
        button.addClickListener(new Button.ClickListener() {
            public void buttonClick(ClickEvent event) {
                layout.addComponent(
                    new Label("Thank you for clicking"));
            }
        });
        layout.addComponent(button);
    }
}
```

If you created the project as a Servlet 3.0 project, the generated UI stub includes a static servlet class annotated with **@WebServlet**, as described in Section 3.4.2, “Exploring the Project”.

```
@WebServlet(value = "/", asyncSupported = true)
@VaadinServletConfiguration(productionMode = false,
                           ui = MyportletUI.class)
public static class Servlet extends VaadinServlet {
```

This enables running the portlet UI in a servlet container while developing it, which may be easier than deploying to a portal. For Servlet 2.4 projects, a web.xml is created.

The portlet theme is defined with the **@Theme** annotation as usual. The theme for the UI must match a theme installed in the portal. You can use any of the built-in themes in Vaadin. If you use a custom theme, you need to compile it to CSS with the theme compiler and install it in the portal under the VAADIN/themes context to be served statically.

In addition to the UI class, you need the portlet descriptor files, Vaadin libraries, and other files as described later. Figure 12.3, “Portlet Project Structure in Eclipse” shows the complete project structure under Eclipse.

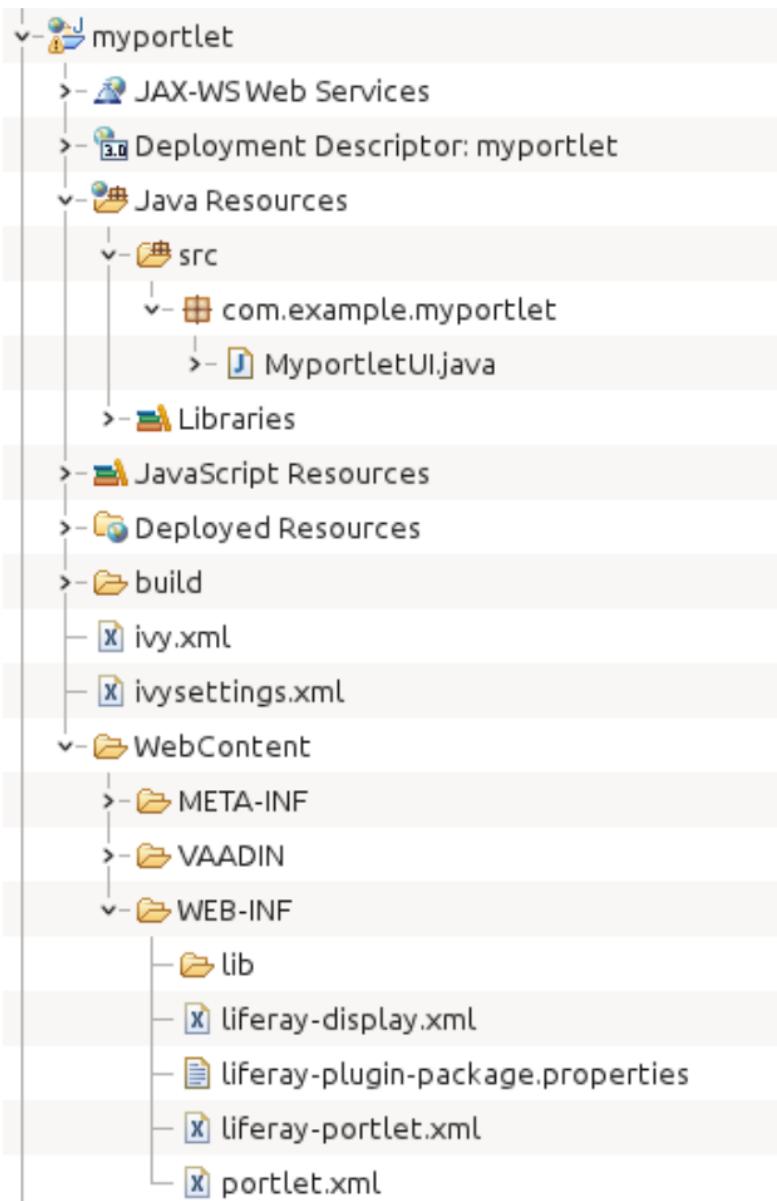


Figure 12.3. Portlet Project Structure in Eclipse

Installed as a portlet in Liferay from the **Add Application** menu, the application will show as illustrated in Figure 12.4. "Hello World Portlet".



Figure 12.4. Hello World Portlet

12.5. Deploying to a Portal

To deploy a portlet WAR in a portal, you need to provide a portlet.xml descriptor specified in the Java Portlet API 2.0 standard (JSR-286). In addition, you may need to include possible portal vendor specific deployment descriptors. The ones required by Liferay are described below.

Deploying a Vaadin UI as a portlet is essentially just as easy as deploying a regular application to an application server. You do not need to make any changes to the UI itself, but only the following:

- Application packaged as a WAR
- WEB-INF/portlet.xml descriptor
- WEB-INF/liferay-portlet.xml descriptor for Liferay
- WEB-INF/liferay-display.xml descriptor for Liferay
- WEB-INF/liferay-plugin-package.properties for Liferay
- Widget set installed to portal (optional)
- Themes installed to portal (optional)
- Vaadin libraries installed to portal (optional)

- Portal configuration settings (optional)

The Vaadin Plugin for Eclipse creates these files for you, when you create a portlet project as described in Section 12.2, "Creating a Generic Portlet in Eclipse".

Installing the widget set and themes to the portal is required for running two or more Vaadin portlets simultaneously in a single portal page. As this situation occurs quite easily, we recommend installing them in any case. Instructions for Liferay are given in Section 12.3, "Developing Vaadin Portlets for Liferay" and the procedure is similar for other portals.

In addition to the Vaadin libraries, you will need to have the portlet.jar in your project classpath. However, notice that you must *not* put the portlet.jar in the same WEB-INF/lib directory as the Vaadin JAR or otherwise include it in the WAR to be deployed, because it would create a conflict with the internal portlet library of the portal. The conflict would cause errors such as " ClassCastException: ...VaadinPortlet cannot be cast to javax.portlet.Portlet".

12.5.1. Portlet Deployment Descriptor

The portlet WAR must include a portlet descriptor located at WEB-INF/portlet.xml. A portlet definition includes the portlet name, mapping to a servlet, modes supported by the portlet, and other configuration. Below is an example of a simple portlet definition in portlet.xml descriptor.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<portlet-app
  xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="2.0"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd
     http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd">

<portlet>
  <portlet-name>Portlet Example portlet</portlet-name>
  <display-name>Vaadin Portlet Example</display-name>

  <!-- Map portlet to a servlet -->
  <portlet-class>
    com.vaadin.server.VaadinPortlet
  </portlet-class>
  <init-param>
```

```

<name>UI</name>

<!-- The application class with package name. -->
<value>com.example.myportlet.MyportletUI</value>
</init-param>

<!-- Supported portlet modes and content types. -->
<supports>
<mime-type>text/html</mime-type>
<portlet-mode>view</portlet-mode>
<portlet-mode>edit</portlet-mode>
<portlet-mode>help</portlet-mode>
</supports>

<!-- Not always required but Liferay requires these. -->
<portlet-info>
<title>Vaadin Portlet Example</title>
<short-title>Portlet Example</short-title>
</portlet-info>
</portlet>
</portlet-app>

```

The mode definitions enable the corresponding portlet controls in the portal. The portlet controls allow changing the mode of the portlet, as described later.

12.5.2. Liferay Portlet Descriptor

Liferay requires a special liferay-portlet.xml descriptor file that defines Liferay-specific parameters. Especially, Vaadin portlets must be defined as "*instanceable*", but not "*ajaxable*".

Below is an example descriptor for the earlier portlet example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE liferay-portlet-app PUBLIC
  "-//Liferay//DTD Portlet Application 4.3.0//EN"
  "http://www.liferay.com/dtd/liferay-portlet-app_4_3_0.dtd">

<liferay-portlet-app>
<portlet>
  <!-- Matches definition in portlet.xml. -->
  <!-- Note: Must not be the same as servlet name. -->
  <portlet-name>Portlet Example portlet</portlet-name>

  <instanceable>true</instanceable>
  <ajaxable>false</ajaxable>
</portlet>
</liferay-portlet-app>

```

See Liferay documentation for further details on the liferay-portlet.xml deployment descriptor.

12.5.3. Liferay Display Descriptor

The WEB-INF/liferay-display.xml file defines the portlet category under which portlets are located in the **Add Application** window in Liferay. Without this definition, portlets will be organized under the "Undefined" category.

The following display configuration, which is included in the demo WAR, puts the Vaadin portlets under the "Vaadin" category, as shown in Figure 12.5, "Portlet Categories in Add Application Window".

```
<?xml version="1.0"?>
<!DOCTYPE display PUBLIC
"-//Liferay//DTD Display 4.0.0//EN"
"http://www.liferay.com/dtd/liferay-display_4_0_0.dtd">

<display>
  <category name="Vaadin">
    <portlet id="Portlet Example portlet" />
  </category>
</display>
```

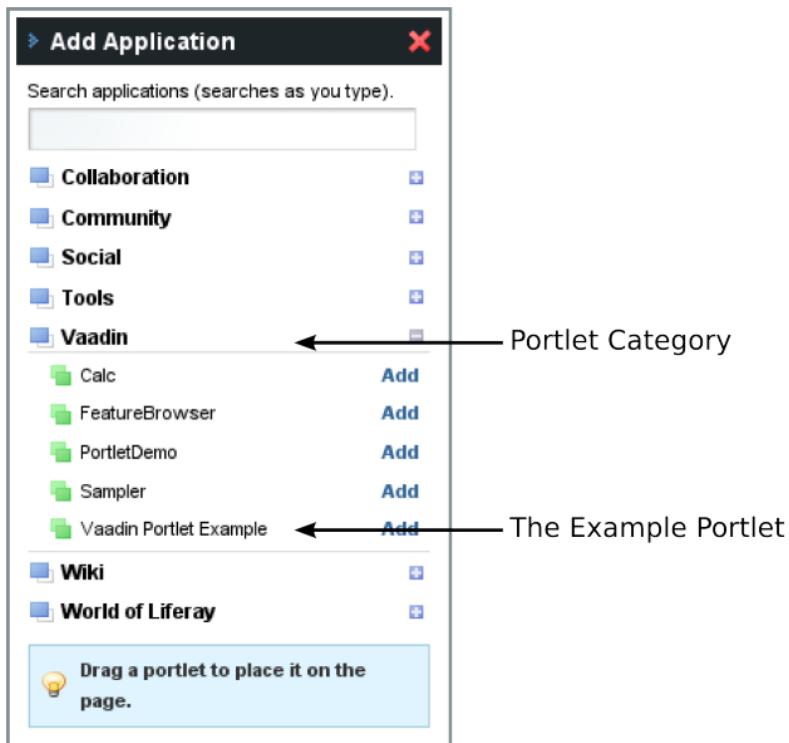


Figure 12.5. Portlet Categories in Add Application Window

See Liferay documentation for further details on how to configure the categories in the `liferay-display.xml` deployment descriptor.

12.5.4. Liferay Plugin Package Properties

The `liferay-plugin-package.properties` file defines a number of settings for the portlet, most importantly the Vaadin JAR to be used.

```
name=Portlet Example portlet
short-description=myporlet
module-group-id=Vaadin
module-incremental-version=1
#change-log=
#page-uri=
#author=
license=Proprietary
```

```
portal-dependency-jars=\n    vaadin.jar
```

name

The plugin name must match the portlet name.

short-description

A short description of the plugin. This is by default the project name.

module-group-id

The application group, same as the category id defined in liferay-display.xml.

license

The plugin license type: "proprietary" by default.

portal-dependency-jars

The JAR libraries on which this portlet depends. This should have value vaadinjar, unless you need to use a specific version. The JAR must be installed in the portal, for example, in Liferay bundled with Tomcat to tomcat-x.x.x/webapps/ROOT/WEB-INF/lib/vaadin.jar.

12.5.5. Using a Single Widget Set

If you have just one Vaadin application that you ever need to run in your portal, you can just deploy the WAR as described above and that's it. However, if you have multiple applications, especially ones that use different custom widget sets, you run into problems, because a portal window can load only a single Vaadin widget set at a time. You can solve this problem by combining all the different widget sets in your different applications into a single widget set using inheritance or composition.

For example, if using the default widget set for portlets, you should have the following for all portlets so that they will all use the same widget set:

```
<portlet>\n    ...\n    <!-- Use the portal default widget set for all portal demos. -->\n    <init-param>\n        <name>widgetset</name>\n        <value>com.vaadin.portal.PortalDefaultWidgetSet</value>
```

```
</init-param>
```

```
...
```

The **PortalDefaultWidgetSet** extends **SamplerWidgetSet**, which extends the **DefaultWidgetSet**. The **DefaultWidgetSet** is therefore essentially a subset of **PortalDefaultWidgetSet**, which contains also the widgets required by the Sampler demo. Other applications that would otherwise require only the regular **DefaultWidgetSet**, and do not define their own widgets, can just as well use the larger set, making them compatible with the demos. The **PortalDefaultWidgetSet** will also be the default Vaadin widgetset bundled in Liferay 5.3 and later.

If your portlets are contained in multiple WARs, which can happen quite typically, you need to install the widget set and theme portal-wide so that all the portlets can use them. See Section 12.3, “Developing Vaadin Portlets for Liferay” on configuring the widget sets in the portal itself.

12.5.6. Building the WAR Package

To deploy the portlet, you need to build a WAR package. For production deployment, you probably want to either use Maven or an Ant script to build the package. In Eclipse, you can right-click on the project and select **Export ▾ WAR**. Choose a name for the package and a target. If you have installed Vaadin in the portal as described in Section 12.3, “Developing Vaadin Portlets for Liferay”, you should exclude all the Vaadin libraries, as well as widget set and themes from the WAR.

12.5.7. Deploying the WAR Package

How you actually deploy a WAR package depends on the portal. In Liferay, you simply drop it to the deploy subdirectory under the Liferay installation directory. The deployment depends on the application server under which Liferay runs; for example, if you use Liferay bundled with Tomcat, you will find the extracted package in the webapps directory under the Tomcat installation directory included in Liferay.

12.6. Vaadin IPC for Liferay

Portlets rarely live alone. A page can contain multiple portlets and when the user interacts with one portlet, you may need to have the other portlets react to the change immediately. This is not normally possible with Vaadin portlets, as Vaadin applications need to get an Ajax request from the client-side to change their user interface. On the other hand, the regular inter-portlet communication (IPC) mechanism in Portlet 2.0 Specification requires a complete page reload, but that is not appropriate with Vaadin or in general Ajax applications, which do not require a page reload. One solution is to communicate between the portlets on the server-side and then use a server-push mechanism to update the client-side.

The Vaadin IPC for Liferay Add-on takes another approach by communicating between the portlets through the client-side. Events (messages) are sent through the **LiferayIPC** component and the client-side widget relays them to the other portlets, as illustrated in Figure 12.6, "Vaadin IPC for Liferay Architecture".

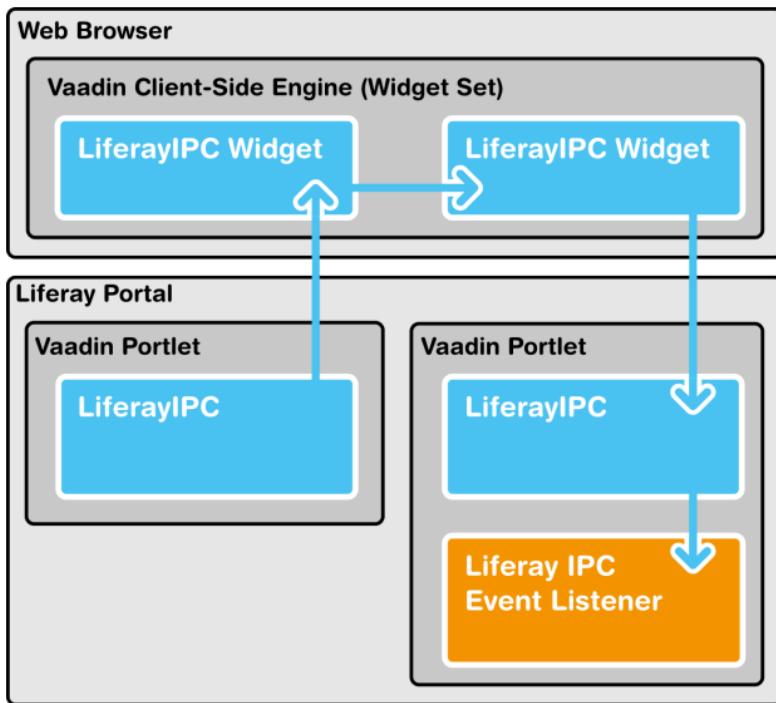


Figure 12.6. Vaadin IPC for Liferay Architecture

Vaadin IPC for Liferay uses the Liferay JavaScript event API for client-side inter-portlet communication, so you can communicate just as easily with other Liferay portlets.

Notice that you can use this communication only between portlets on the same page.

Figure 12.7, "Vaadin IPC Add-on Demo with Two Portlets" shows Vaadin IPC for Liferay in action. Entering a new item in one portlet is updated interactively in the other.

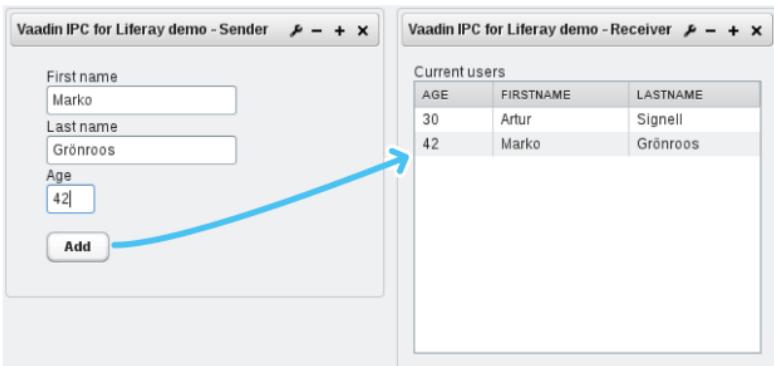


Figure 12.7. Vaadin IPC Add-on Demo with Two Portlets

12.6.1. Installing the Add-on

The Vaadin IPC for Liferay add-on is available from the Vaadin Directory as well as from a Maven repository. To download the installation package or find out the Maven or Ivy dependency, see the add-on page at Vaadin Directory, and install the add-on as described in Chapter 17, *Using Vaadin Add-ons*.

The contents of the installation package are as follows:

`vaadin-ipc-for-liferay-x.x.x.jar`

The add-on JAR in the installation package must be installed in the WEB-INF/lib directory under the root context. The location depends on the server - for example in Liferay running in Tomcat it is located under the webapps/ROOT folder of the server.

`doc`

The documentation folder includes a README.TXT file that describes the contents of the installation package briefly, and licensing.txt and license-asl-2.0.txt, which describe the licensing under the Apache License 2.0. Under the doc/api folder is included the complete JavaDoc API documentation for the add-on.

`vaadin-ipc-for-liferay-x.x.x-demo.war`

A WAR containing demo portlets. After installing the add-on library and compiling the widget set, as described below, you can deploy the WAR to Liferay and

add the two demo portlets to a page, as shown in Figure 12.7, "Vaadin IPC Add-on Demo with Two Portlets". The source of the demo is available at dev.vaadin.com/svn addons/IPCforLiferay/trunk/.

The add-on contains a widget set, which you must compile into the Vaadin widget set installed in the portal.

12.6.2. Basic Communication

LiferayIPC is an invisible user interface component that can be used to send messages between two or more Vaadin portlets. You add it to an application layout as you would any regular user interface component.

```
LiferayIPC liferayipc = new LiferayIPC();
layout.addComponent(liferayipc);
```

You should be careful not to remove the invisible component from the portlet later if you modify the layout of the portlet.

The component can be used both for sending and receiving messages, as described next.

Sending Events

You can send an event (a message) with the `sendEvent()` method, which takes an event ID and the message data as parameters. The event is broadcast to all listening portlets. The event ID is a string that can be used to identify the recipient of an event or the event type.

```
liferayipc.sendEvent("hello", "This is Data");
```

If you need to send more complex data, you need to format or serialize it to a string representation as described in Section 12.6.5, "Serializing and Encoding Data".

Receiving Events

A portlet wishing to receive events (messages) from other portlets needs to register a listener in the component with `addListener()`. The listener receives the messages in a **LiferayIPCEvent** object. Filtering events by the ID is built in into the listener handler, you give the listened event ID as the

first parameter for the `addListener()`. The actual message data is held in the `data` property, which you can read with `getData()`.

```
liferayipc.addListener("hello", new LiferayIPCEventListener() {  
    public void eventReceived(LiferayIPCEvent event) {  
        // Do something with the message data  
        String data = event.getData();  
        Notification.show("Received hello: " + data);  
    }  
});
```

A listener added to a **LiferayIPC** can be removed with `removeListener()`.

12.6.3. Considerations

Both security and efficiency should be considered with inter-portlet communications when using the Vaadin IPC for Liferay.

Browser Security

As the message data is passed through the client-side (browser), any code running in the browser has access to the data. You should be careful not to expose any security-critical data in client-side messaging. Also, malicious code running in the browser could alter or fake messages. Sanitization can help with the latter problem and encryption to solve the both issues. You can also share the sensitive data through session attributes or a database and use the client-side IPC only to notify that the data is available.

Efficiency

Sending data through the browser requires loading and sending it in HTTP requests. The data is held in the memory space of the browser, and handling large data in the client-side JavaScript code can take time. Noticeably large message data can therefore reduce the responsiveness of the application and could, in extreme cases, go over browser limits for memory consumption or JavaScript execution time.

12.6.4. Communication Through Session Attributes

In many cases, such as when considering security or efficiency, it is better to pass the bulk data on the server-side and use the client-side IPC only for notifying the other portlet(s) that

the data is available. Session attributes are a convenient way of sharing data on the server-side. You can also share objects through them, not just strings.

The session variables have a *scope*, which should be *APPLICATION_SCOPE*. The "application" refers to the scope of the Java web application (WAR) that contains the portlets.

If the communicating portlets are in the same Java web application (WAR), no special configuration is needed. You can also communicate between portlets in different WARs, in which case you need to disable the *private-session-attributes* parameter in *liferay-portlet.xml* by setting it to false. Please see Liferay documentation for more information regarding the configuration.

You can also share Java objects between the portlets in the same WAR, not just strings. If the portlets are in different WARs, they normally have different class loaders, which could cause incompatibilities, so you can only communicate with strings and any object data needs to be serialized.

Session attributes are accessible through the **PortletSession** object, which you can access through the portlet context from the Vaadin **Application** class.

```
Person person = new Person(firstname, lastname, age);
```

```
...
```

```
PortletSession session =
((PortletApplicationContext2)getContext()).
getPortletSession();
```

```
// Share the object
String key = "IPCDEMO_person";
session.setAttribute(key, person,
PortletSession.APPLICATION_SCOPE);
```

```
// Notify that it's available
liferayipc.sendEvent("ipc_demodata_available", key);
```

You can then receive the attribute in a **LiferayIPCEventListener** as follows:

```
public void eventReceived(LiferayIPCEvent event) {
    String key = event.getData();
```

```

PortletSession session =
    ((PortletApplicationContext2)getContext()).
        getPortletSession();

// Get the object reference
Person person = (Person) session.getAttribute(key);

// We can now use the object in our application
BeanItem<Person> item = new BeanItem<Person>(person);
form.setItemDataSource(item);
}

```

Notice that changes to a shared object bound to a user interface component are not updated automatically if it is changed in another portlet. The issue is the same as with double-binding in general.

12.6.5. Serializing and Encoding Data

The IPC events support transmitting only plain strings, so if you have object or other non-string data, you need to format or serialize it to a string representation. For example, the demo application formats the trivial data model as a semicolon-separated list as follows:

```

private void sendPersonViaClient(String firstName,
        String lastName, int age) {
    liferayIPC_1.sendEvent("newPerson", firstName + ":" +
        lastName + ":" + age);
}

```

You can use standard Java serialization for any classes that implement the `Serializable` interface. The transmitted data may not include any control characters, so you also need to encode the string, for example by using Base64 encoding.

```

// Some serializable object
MyBean mybean = new MyBean();
...
// Serialize
ByteArrayOutputStream baost = new ByteArrayOutputStream();
ObjectOutputStream oostr;
try {
    oostr = new ObjectOutputStream(baost);
    oostr.writeObject(mybean); // Serialize the object
    oostr.close();
} catch (IOException e) {

```

```

        Notification.show("IO PAN!"); // Complain
    }

// Encode
BASE64Encoder encoder = new BASE64Encoder();
String encoded = encoder.encode(baostr.toByteArray());

// Send the IPC event to other portlet(s)
liferayipc.sendEvent("mybeanforyou", encoded);

You can then deserialize such a message at the receiving end
as follows:

public void eventReceived(LiferayIPCEvent event) {
    String encoded = event.getData();

    // Decode and deserialize it
    BASE64Decoder decoder = new BASE64Decoder();
    try {
        byte[] data = decoder.decodeBuffer(encoded);
        ObjectInputStream ois =
            new ObjectInputStream(
                new ByteArrayInputStream(data));

        // The deserialized bean
        MyBean serialized = (MyBean) ois.readObject();
        ois.close();

        ... do something with the bean ...

    } catch (IOException e) {
        e.printStackTrace(); // Handle somehow
    } catch (ClassNotFoundException e) {
        e.printStackTrace(); // Handle somehow
    }
}

```

12.6.6. Communicating with Non-Vaadin Portlets

You can use the Vaadin IPC for Liferay to communicate also between a Vaadin application and other portlets, such as JSP portlets. The add-on passes the events as regular Liferay JavaScript events. The demo WAR includes two JSP portlets that demonstrate the communication.

When sending events from non-Vaadin portlet, fire the event using the JavaScript `Liferay.fire()` method with an event ID and message. For example, in JSP you could have:

```
<%@ taglib uri="http://java.sun.com/portlet_2_0"
   prefix="portlet" %>
<portlet:defineObjects />

<script>
function send_message() {
    Liferay.fire('hello', "Hello, I'm here!");
}
</script>

<input type="button" value="Send message"
       onclick="send_message()" />
```

You can receive events using a Liferay JavaScript event handler. You define the handler with the on() method in the Liferay object. It takes the event ID and a callback function as its parameters. Again in JSP you could have:

```
<%@ taglib uri="http://java.sun.com/portlet_2_0"
   prefix="portlet" %>
<portlet:defineObjects />

<script>
Liferay.on('hello', function(event, data) {
    alert("Hello: " + data);
});
</script>
```


Chapter 13

Client-Side Vaadin Development

13.1. Overview	517
13.2. Installing the Client-Side Development Environment	518
13.3. Client-Side Module Descriptor	518
13.4. Compiling a Client-Side Module	520
13.5. Creating a Custom Widget	522
13.6. Debugging Client-Side Code	524

This chapter gives an overview of the Vaadin client-side framework, its architecture, and development tools.

13.1. Overview

As noted in the introduction, Vaadin supports two development models: server-side and client-side. Client-side Vaadin code is executed in the web browser as JavaScript code. The code is written in Java, like all Vaadin code, and then compiled to JavaScript with the *Vaadin Client Compiler*. You can develop client-side widgets and integrate them with server-side counterpart components to allow using them in server-side Vaadin applications. That is how the components in the server-side framework and in most add-ons are done. Alternatively, you can create pure client-side GWT applications.

which you can simply load in the browser from an HTML page and use even without server-side connectivity.

The client-side framework is based on the Google Web Toolkit (GWT), with added features and bug fixes. Vaadin is compatible with GWT to the extent of the basic GWT feature set. Vaadin Ltd is a member of the GWT Steering Committee, working on the future direction of GWT together with Google and other supporters of GWT.



Widgets and Components

Google Web Toolkit uses the term *widget* for user interface components. In this book, we use the term *widget* to refer to client-side components, while using the term *component* in a general sense and also in the special sense for server-side components.

The main idea in server-side Vaadin development is to render the server-side components in the browser with the Client-Side Engine. The engine is essentially a set of widgets paired with *connectors* that serialize their state and events with the server-side counterpart components. The client-side engine is technically called a *widget set*, to describe the fact that it mostly consists of widgets and that widget sets can be combined, as described later.

13.2. Installing the Client-Side Development Environment

The installation of the client-side development libraries is described in Chapter 3, *Creating a Vaadin Application*. You especially need the `vaadin-client` library, which contains the client-side Java API, and `vaadin-client-compiler`, which contains the Vaadin Client Compiler for compiling Java to JavaScript.

13.3. Client-Side Module Descriptor

Client-side Vaadin modules, such as the Vaadin Client-Side Engine (widget set) or pure client-side applications, that are

to be compiled to JavaScript, are defined in a *module descriptor* (.gwt.xml) file.

When defining a widget set to build the Vaadin client-side engine, the only necessary task is to inherit a base widget set. If you are developing a regular widget set, you should normally inherit the **DefaultWidgetSet**.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC
  "-//Google Inc.//DTD Google Web Toolkit 1.7.0//EN"
  "http://google-web-toolkit.googlecode.com/svn/tags/1.7.0/distro-source/core/src/gwt-module.dtd">

<module>
  <!-- Inherit the default widget set -->
  <inherits name="com.vaadin.DefaultWidgetSet" />
</module>
```

If you are developing a pure client-side application, you should instead inherit **com.vaadin.Vaadin**, as described in Chapter 14, *Client-Side Applications*. In that case, the module descriptor also needs an entry-point.

If you are using the Eclipse IDE, the New Vaadin Widget wizard will automatically create the GWT module descriptor. See Section 16.2.1, “Creating a Widget” for detailed instructions.

13.3.1. Specifying a Stylesheet

A client-side module can include CSS stylesheets. When the module is compiled, these stylesheets are copied to the output target. In the module descriptor, define a stylesheet element.

For example, if you are developing a custom widget and want to have a default stylesheet for it, you could define it as follows:

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

The specified path is relative to the *public* folder under the folder of the module descriptor.

13.3.2. Limiting Compilation Targets

Compiling widget sets takes considerable time. You can reduce the compilation time significantly by compiling the widget sets only for your browser, which is useful during development. You can do this by setting the *user.agent* property in the module descriptor.

```
<set-property name="user.agent" value="gecko1_8"/>
```

The `value` attribute should match your browser. The browsers supported by GWT depend on the GWT version. below is a list of browser identifiers supported by GWT.

Table 13.1. GWT User Agents

Identifier	Name
ie6	Internet Explorer 6
ie8	Internet Explorer 8
gecko1_8	Mozilla Firefox 1.5 and later
safari	Apple Safari and other Webkit-based browsers including Google Chrome
opera	Opera
ie9	Internet Explorer 9

For more information about the GWT Module XML Format, please see Google Web Toolkit Developer Guide.

13.4. Compiling a Client-Side Module

A client-side module, either a Vaadin widget set or a pure client-side module, needs to be compiled to JavaScript using the Vaadin Client Compiler.

Widget set compilation is most often needed when using add-ons. In that case, the widget sets from different add-ons are compiled into an *application widget set*, as described in Chapter 17, *Using Vaadin Add-ons*.

When doing client-side development, you need to compile the widget set every time you modify the client-side code.

13.4.1. Vaadin Compiler Overview

The Vaadin Client Compiler compiles Java to JavaScript. It is provided as the executable `vaadin-client-compiler` JAR. It requires the `vaadin-client` JAR, which contains the **DefaultWidgetSet**, Vaadin client-side framework.

The compiler compiles a *client module*, which can either be a pure client-side module or a Vaadin widget set, that is, the Vaadin Client-Side Engine that includes the widgets used in the application. The client module is defined with a module descriptor, which was described in Section 13.3, “Client-Side Module Descriptor”. The module descriptor for application widget sets is automatically generated.

While you can compile client modules individually, in Vaadin applications you normally combine them in one application widget set. The application widget set includes any add-on and custom widgets. The compiler scans the class path for any widget sets to include in the application widget set.

Compilation Result

The compiler writes the compilation result to a target folder that will include the compiled JavaScript with any static resources included in the module.

13.4.2. Compiling in Maven Projects

The Vaadin Maven Plugin, which is enabled in all Vaadin archetypes, makes Maven to automatically compile the widget set when necessary.

Compilation Modes

The Vaadin Maven Plugin can compile widget sets either locally or online by using a cloud service. The online compilation requires that all the widget sets are available in certain public Maven repositories. As this is not the case when developing custom widgets, you must use the local mode.

Local compilation is the default mode, so you only need to enable it if you have changed the mode to use the online service.

See Section 17.2.3, “Widget Set Modes” for more information.

Compiling

You can explicitly compile the widget set with the `vaadin:compile` goal.

On command-line:

```
$ mvn vaadin:compile
```

If there is no widget set defined, but you have add-on dependencies that need a custom widget set, the Vaadin Maven plugin will automatically generate a widget set definition for you.

13.4.3. Compiling in Eclipse

When you have the Vaadin Plugin installed in Eclipse, you can simply click the **Compile Vaadin Widgetset** button in the toolbar.



Figure 13.1. Widget set compilation button in Eclipse

It will compile the widget set it finds from the project. If the project has multiple widget sets, such as one for custom widgets and another one for the project, you need to select the module descriptor of the widget set to compile before clicking the button.

Compiling with the Vaadin Plugin for Eclipse currently requires that the module descriptor has suffix Widgetset.gwt.xml, although you can use it to compile also other client-side modules than widget sets.

The result is written under WebContent/VAADIN/widgetsets folder.

13.5. Creating a Custom Widget

Creating a new Vaadin component usually begins from making a client-side widget, which is later integrated with a server-side counterpart to enable server-side development. In addition, you can also choose to make pure client-side widgets, a possibility which we also describe later in this section.

13.5.1. A Basic Widget

All widgets extend the **Widget** class or some of its subclasses. You can extend any core GWT or supplementary Vaadin widgets. Perhaps typically, an abstraction such as **Composite**. The basic GWT widget component hierarchy is illustrated in Figure 13.2, "GWT widget base class hierarchy". Please see the GWT API documentation for a complete description of the widget classes.

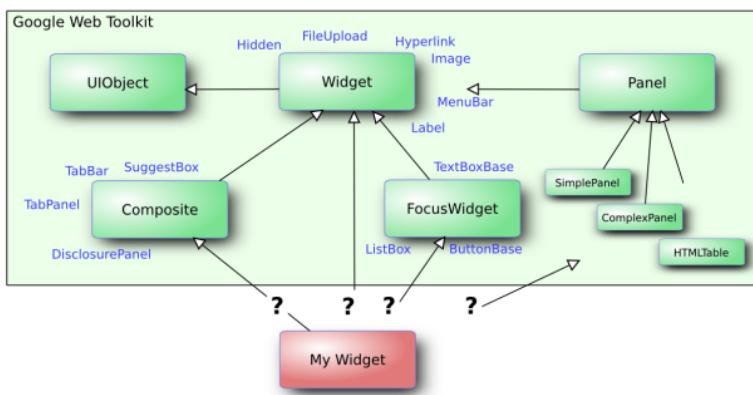


Figure 13.2. GWT widget base class hierarchy

For example, we could extend the **Label** widget to display some custom text.

```
package com.example.myapp.client;

import com.google.gwt.user.client.ui.Label;

public class MyWidget extends Label {
    public static final String CLASSNAME = "mywidget";

    public MyWidget() {
        setStyleName(CLASSNAME);
        setText("This is MyWidget");
    }
}
```

The above example is largely what the Eclipse plugin generates as a widget stub. It is a good practice to set a distinctive style class for the widget, to allow styling it with CSS.

The client-side source code *must* be contained in a client package under the package of the descriptor file, which is covered later.

13.5.2. Using the Widget

You can use a custom widget just like you would use any widget, possibly integrating it with a server-side component, or in pure client-side modules such as the following:

```
public class MyEntryPoint implements EntryPoint {  
    @Override  
    public void onModuleLoad() {  
        // Use the custom widget  
        final MyWidget mywidget = new MyWidget();  
        RootPanel.get().add(mywidget);  
    }  
}
```

13.6. Debugging Client-Side Code

Vaadin currently includes SuperDevMode for debugging client-side code right in the browser.

The predecessor of SuperDevMode, the GWT Development Mode, no longer works in recent versions of Firefox and Chrome, because of certain API changes in the browsers. There exists workarounds on some platforms, but for the sake of simplicity, we recommend using the SuperDevMode.

13.6.1. Launching SuperDevMode

The SuperDevMode is much like the old Development Mode, except that it does not require a browser plugin. Compilation from Java to JavaScript is done incrementally, reducing the compilation time significantly. It also allows debugging JavaScript and even Java right in the browser (currently only supported in Chrome).

You can enable SuperDevMode as follows:

1. You need to set a redirect property in the .gwt.xml module descriptor as follows:

```
<set-configuration-property name="devModeRedirectEnabled" value="true" />
```

In addition, you need the xsiframe linker. It is included in the **com.vaadin.DefaultWidgetSet** as well as in the **com.vaadin.Vaadin** module. Otherwise, you need to include it with:

```
<add-linker name="xsiframe" />
```

2. Compile the module (that is, the widget set), for example by clicking the button in Eclipse.
3. If you are using Eclipse, create a launch configuration for the SuperDevMode by clicking the **Create SuperDevMode launch** in the **Vaadin** section of the project properties.
 - a. The main class to execute should be **com.google.gwt.dev.codeserver.CodeServer**.
 - b. The application takes the fully-qualified class name of the module (or widget set) as parameter, for example, **com.example.myproject.widgetset.MyprojectWidgetset**.
 - c. Add project sources to the class path of the launch if they are not in the project class path.

The above configuration only needs to be done once to enable the SuperDevMode. After that, you can launch the mode as follows:

1. Run the SuperDevMode Code Server with the launch configuration that you created above. This performs the initial compilation of your module or widget set.
2. Launch the servlet container for your application, for example, Tomcat.
3. Open your browser with the application URL and add ?superdevmode parameter to the URL (see the notice below if you are not extending **DefaultWidgetSet**). This recompiles the code, after which the page is reloaded with the SuperDevMode. You can also use the ?debug parameter and then click the **SDev** button in the debug console.

If you make changes to the client-side code and refresh the page in the browser, the client-side is recompiled and you see the results immediately.

The Step 3 above assumes that you extend **DefaultWidgetSet** in your module. If that is not the case, you need to add the following at the start of the `onModuleLoad()` method of the module:

```
if (SuperDevMode.enableBasedOnParameter()) { return; }
```

Alternatively, you can use the bookmarklets provided by the code server. Go to <http://localhost:9876/> and drag the bookmarklets "**Dev Mode On**" and "**Dev Mode Off**" to the bookmarks bar

13.6.2. Debugging Java Code in Chrome

Chrome supports source maps, which allow debugging Java source code from which the JavaScript was compiled.

Open the Chrome Inspector by right-clicking and selecting **Inspect Element**. Click the settings icon in the upper right corner of the window and check the **Scripts** **Enable source maps** option. Refresh the page with the Inspector open, and you will see Java code instead of JavaScript in the scripts tab.

Chapter 14

Client-Side Applications

14.1. Overview	527
14.2. Client-Side Module Entry-Point	530
14.3. Compiling and Running a Client-Side Application ...	531
14.4. Loading a Client-Side Application	532

This chapter describes how to develop client-side Vaadin applications.

We only give a brief introduction to the topic in this book. Please refer to the GWT documentation for a more complete treatment of the many GWT features.

14.1. Overview

Vaadin allows developing client-side modules that run in the browser. Client-side modules can use all the GWT widgets and some Vaadin-specific widgets, as well as the same themes as server-side Vaadin applications. Client-side applications run in the browser, even with no further server communications. When paired with a server-side service to gain access to data storage and server-side business logic, client-side applications can be considered "fat clients", in comparison to the "thin client" approach of the server-side Vaadin applications. The services can use the same back-end services as

server-side Vaadin applications. Fat clients are useful for a range of purposes when you have a need for highly responsive UI logic, such as for games or for serving a huge number of clients with possibly stateless server-side code.

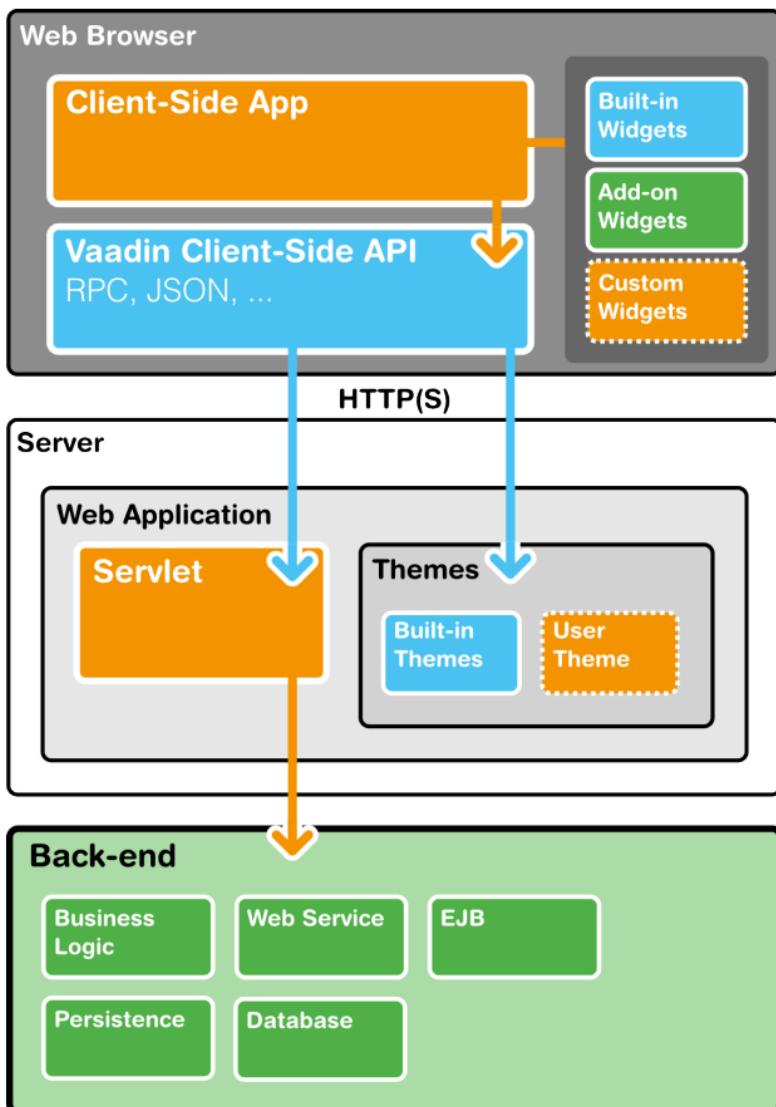


Figure 14.1. Client-Side Application Architecture

A client-side application is defined as a *module*, which has an *entry-point* class. Its `onModuleLoad()` method is executed when the JavaScript of the compiled module is loaded in the browser.

Consider the following client-side application:

```
public class HelloWorld implements EntryPoint {  
    @Override  
    public void onModuleLoad() {  
        RootPanel.get().add(new Label("Hello, world!"));  
    }  
}
```

The user interface of a client-side application is built under a HTML *root element*, which can be accessed by `RootPanel.get()`. The purpose and use of the entry-point is documented in more detail in Section 14.2, "Client-Side Module Entry-Point". The user interface is built from *widgets* hierarchically, just like with server-side Vaadin UIs. The built-in widgets and their relationships are catalogued in Chapter 15, *Client-Side Widgets*. You can also use many of the widgets in Vaadin add-ons that have them, or make your own.

A client-side module is defined in a *module descriptor*, as described in Section 13.3, "Client-Side Module Descriptor". A module is compiled from Java to JavaScript using the Vaadin Compiler, of which use was described in Section 13.4, "Compiling a Client-Side Module". The Section 14.3, "Compiling and Running a Client-Side Application" in this chapter gives further information about compiling client-side applications. The resulting JavaScript can be loaded to any web page, as described in Section 14.4, "Loading a Client-Side Application".

The client-side user interface can be built declaratively using the included *UI Binder*.

The servlet for processing RPC calls from the client-side can be generated automatically using the included compiler.

Even with regular server-side Vaadin applications, it may be useful to provide an off-line mode if the connection is closed. An off-line mode can persist data in a local store in the browser, thereby avoiding the need for server-side storage, and transmit the data to the server when the connection is

again available. Such a pattern is commonly used with Vaadin TouchKit.

14.2. Client-Side Module Entry-Point

A client-side application requires an *entry-point* where the execution starts, much like the init() method in server-side Vaadin UIs.

Consider the following application:

```
package com.example.myapp.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.ui.RootPanel;
import com.vaadin.ui.VButton;

public class MyEntryPoint implements EntryPoint {
    @Override
    public void onModuleLoad() {
        // Create a button widget
        Button button = new Button();
        button.setText("Click me!");
        button.addClickHandler(new ClickHandler() {
            @Override
            public void onClick(ClickEvent event) {
                mywidget.setText("Hello, world!");
            }
        });
        RootPanel.get().add(button);
    }
}
```

Before compiling, the entry-point needs to be defined in a module descriptor, as described in the next section.

14.2.1. Module Descriptor

The entry-point of a client-side application is defined, along with any other configuration, in a client-side module descriptor, described in Section 13.3, “Client-Side Module Descriptor”. The descriptor is an XML file with suffix .gwt.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC
```

```
*-//Google Inc//DTD Google Web Toolkit 1.7.0//EN*
'http://google-web-toolkit.googlecode.com/svn/tags/1.7.0/distro-source/core/src/gwt-module.dtd'
<module>
  <!-- Built-in Vaadin and GWT widgets -->
  <inherits name='com.vaadin.Vaadin' />
  <!-- The entry-point for the client-side application -->
  <entry-point class='com.example.myapp.client.MyEntryPoint' />
</module>
```

You might rather want to inherit the **com.google.gwt.user.User** to get just the basic GWT widgets, and not the Vaadin-specific widgets and classes, most of which are unusable in pure client-side applications.

You can put static resources, such as images or CSS stylesheets, in a public folder (not a Java package) under the folder of the descriptor file. When the module is compiled, the resources are copied to the output folder. Normally in pure client-side application development, it is easier to load them in the HTML host file or in a **ClientBundle** (see GWT documentation), but these methods are not compatible with server-side component integration, if you use the resources for that purpose as well.

14.3. Compiling and Running a Client-Side Application

Compilation of client-side modules other than widget sets with the Vaadin Plugin for Eclipse has recent changes and limitations at the time of writing of this edition and the information given here may not be accurate.

The application needs to be compiled into JavaScript to run it in a browser. For deployment, and also initially for the first time when running the Development Mode, you need to do the compilation with the Vaadin Client Compiler, as described in Section 13.4, “Compiling a Client-Side Module”.

During development, it is easiest to use the SuperDevMode, which also quickly launches the client-side code and also allows debugging. See Section 13.6, “Debugging Client-Side Code” for more details.

14.4. Loading a Client-Side Application

You can load the JavaScript code of a client-side application in an HTML *host page* by including it with a `<script>` tag, for example as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type"
    content="text/html; charset=UTF-8" />

  <title>Embedding a Vaadin Application in HTML Page</title>

  <!-- Load the Vaadin style sheet -->
  <link rel="stylesheet"
    type="text/css"
    href="/myproject/VAADIN/themes/reindeer/legacy-styles.css"/>
</head>

<body>
  <h1>A Pure Client-Side Application</h1>

  <script type="text/javascript" language="javascript"
    src="clientside/com.example.myapp.MyModule/
      com.example.myapp.MyModule.nocache.js">
  </script>
</body>
</html>
```

The JavaScript module is loaded in a `<script>` element. The `src` parameter should be a relative link from the host page to the compiled JavaScript module.

If the application uses any supplementary Vaadin widgets, and not just core GWT widgets, you need to include the Vaadin theme as was done in the example. The exact path to the style file depends on your project structure - the example is given for a regular Vaadin application where themes are contained in the VAADIN folder in the WAR.

In addition to CSS and scripts, you can load any other resources needed by the client-side application in the host page.

Chapter 15

Client-Side Widgets

15.1. Overview	533
15.2. GWT Widgets	534
15.3. Vaadin Widgets	534
15.4. Grid	535

This chapter gives basic documentation on the use of the Vaadin client-side framework for the purposes of creating client-side applications and writing your own widgets.

We only give a brief introduction to the topic in this chapter. Please refer to the GWT documentation for a more complete treatment of the GWT widgets.

15.1. Overview

The Vaadin client-side API is based on the Google Web Toolkit. It involves *widgets* for representing the user interface as Java objects, which are rendered as a HTML DOM in the browser. Events caused by user interaction with the page are delegated to event handlers, where you can implement your UI logic.

In general, the client-side widgets come in two categories - basic GWT widgets and Vaadin-specific widgets. The library includes *connectors* for integrating the Vaadin-specific widgets with the server-side components, thereby enabling the server-side development model of Vaadin. The integration is described in Chapter 16, *Integrating with the Server-Side*.

The layout of the client-side UI is managed with *panel*/widgets, which correspond in their function with layout components in the Vaadin server-side API.

In addition to the rendering API, the client-side API includes facilities for making HTTP requests, logging, accessibility, internationalization, and testing.

For information about the basic GWT framework, please refer to <https://developers.google.com/web-toolkit/overview>.

15.2. GWT Widgets

GWT widgets are user interface elements that are rendered as HTML. Rendering is done either by manipulating the HTML Document Object Model (DOM) through the lower-level DOM API, or simply by injecting the HTML with `setInnerHTML()`. The layout of the user interface is managed using special panel widgets.

For information about the basic GWT widgets, please refer to the GWT Developer's Guide at <https://developers.google.com/web-toolkit/doc/latest/DevGuideUi>.

15.3. Vaadin Widgets

Vaadin comes with a number of Vaadin-specific widgets in addition to the GWT widgets, some of which you can use in pure client-side applications. The Vaadin widgets have somewhat different feature set from the GWT widgets and are foremost intended for integration with the server-side components, but some may prove useful for client-side applications as well.

```
public class MyEntryPoint implements EntryPoint {  
    @Override  
    public void onModuleLoad() {  
        // Add a Vaadin button  
        VButton button = new VButton();  
        button.setText("Click me!");  
        button.addClickHandler(new ClickHandler() {  
            @Override  
            public void onClick(ClickEvent event) {  
                mywidget.setText("Clicked!");  
            }  
        });  
    }  
}
```

```
        }
    }):
    RootPanel.get().add(button);
}
}
```

15.4. Grid

The **Grid** widget is the client-side counterpart for the server-side **Grid** component described in Section 6.21, “**Grid**”.

The client-side API is almost identical to the server-side API, so its documentation is currently omitted here and we refer you to the API documentation. In the following, we go through some customization features of **Grid**.

15.4.1. Renderers

As described in Section 6.21.6, “Column Renderers”, renderers draw the visual representation of data values on the client-side. They implement `Renderer` interface and its `render()` method. The method gets a reference to the element of the grid cell, as well as the data value to be rendered. An implementation needs to modify the element as needed.

For example, **TextRenderer** is implemented simply as follows:

```
public class TextRenderer implements Renderer<String> {
    @Override
    public void render(RendererCellReference cell,
                      String text) {
        cell.getElement().setInnerText(text);
    }
}
```

The server-side renderer API should extend **AbstractRenderer** or **ClickableRenderer** with the data type accepted by the renderer. The data type also must be given for the superclass constructor.

```
public class TextRenderer extends AbstractRenderer<String> {
    public TextRenderer() {
        super(String.class);
    }
}
```

The client-side and server-side renderer need to be connected with a connector extending from **AbstractRendererConnector**.

```
@Connect(com.vaadin.ui.renderer.TextRenderer.class)
public class TextRendererConnector
    extends AbstractRendererConnector<String> {
    @Override
    public TextRenderer getRenderer() {
        return (TextRenderer) super.getRenderer();
    }
}
```

Renderers can have parameters, for which normal client-side communication of extension parameters can be used. Please see the implementations of different renderers for examples.

Chapter 16

Integrating with the Server-Side

16.1. Overview	538
16.2. Starting It Simple With Eclipse	541
16.3. Creating a Server-Side Component	545
16.4. Integrating the Two Sides with a Connector	546
16.5. Shared State	547
16.6. RPC Calls Between Client- and Server-Side	553
16.7. Component and UI Extensions	555
16.8. Styling a Widget	558
16.9. Component Containers	559
16.10. Advanced Client-Side Topics	559
16.11. Creating Add-ons	561
16.12. Migrating from Vaadin 6	564
16.13. Integrating JavaScript Components and Extensions	565

This chapter describes how you can integrate client-side widgets or JavaScript components with a server-side component. The client-side implementations of all standard server-side components in Vaadin use the same client-side interfaces and patterns.

16.1. Overview

Vaadin components consist of two parts: a server-side and a client-side component. The latter are also called *widgets* in Google Web Toolkit (GWT) parlance. A Vaadin application uses the API of the server-side component, which is rendered as a client-side widget in the browser. As on the server-side, the client-side widgets form a hierarchy of layout widgets and regular widgets as the leaves.

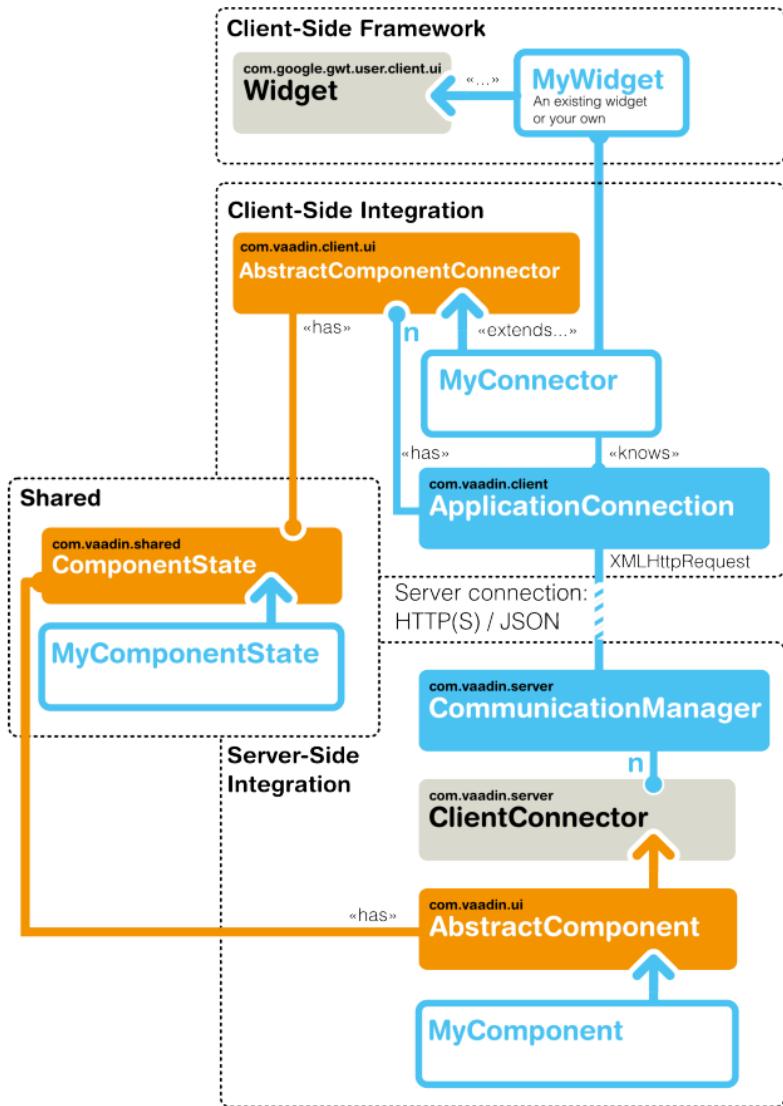


Figure 16.1. Integration of Client-Side Widgets

The communication between a client-side widget and a server-side component is managed with a *connector* that handles synchronizing the widget state and events to and from the server-side.

When rendering the user interface, a client-side connector and a widget are created for each server-side component. The mapping from a component to a connector is defined in the connector class with a @Connect annotation, and the widget is created by the connector class.

The state of a server-side component is synchronized automatically to the client-side widget using a *shared state* object. A shared state object extends **AbstractComponentState** and it is used both in the server-side and the client-side component. On the client-side, a connector always has access to its state instance, as well to the state of its parent component state and the states of its children.

The state sharing assumes that state is defined with standard Java types, such as primitive and boxed primitive types, **String**, arrays, and certain collections (**List**, **Set**, and **Map**) of the supported types. Also the Vaadin **Connector** and some special internal types can be shared.

In addition to state, both server- and client-side can make remote procedure calls (RPC) to the other side. RPC is used foremost for event notifications. For example, when a client-side connector of a button receives a click, it sends the event to the server-side using RPC.

16.1.1. Project Structure

Widget set compilation, as described in Section 13.3, "Client-Side Module Descriptor", requires using a special project structure, where the client-side classes are located under a client package under the package of the module descriptor. Any static resources, such as stylesheets and images, should be located under a public folder (not Java package). The source for the server-side component may be located anywhere, except not in the client-side package.

The basic project structure is illustrated in Figure 16.2, "Basic Widget Integration Project Structure".

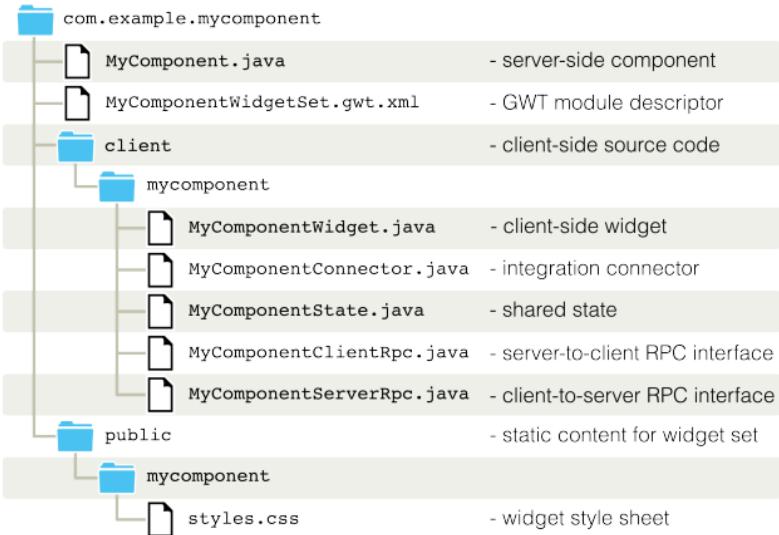


Figure 16.2. Basic Widget Integration Project Structure

The Eclipse wizard, described in Section 16.2, "Starting It Simple With Eclipse", creates a widget integration skeleton with the above structure.

16.1.2. Integrating JavaScript Components

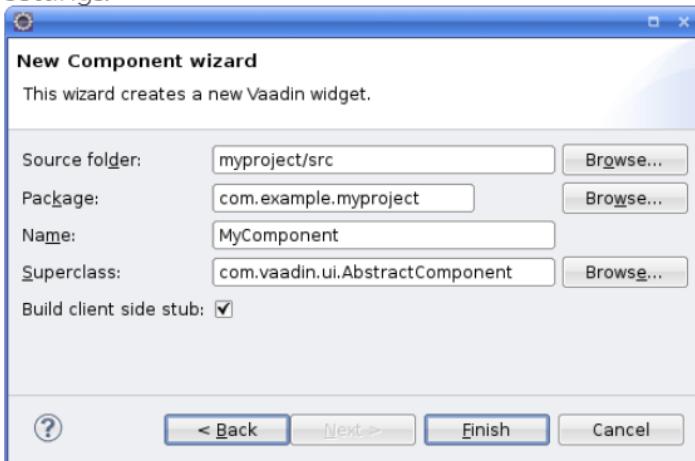
In addition to the GWT widget integration, Vaadin offers a simplified way to integrate pure JavaScript components. The JavaScript connector code is published from the server-side. As the JavaScript integration does not involve GWT programming, no widget set compilation is needed.

16.2. Starting It Simple With Eclipse

Let us first take the easy way and create a simple component with Eclipse. While you can develop new widgets with any IDE or even without, you may find Eclipse and the Vaadin Plugin for it useful, as it automates all the basic routines of widget development, most importantly the creation of new widgets.

16.2.1. Creating a Widget

1. Right-click the project in the Project Explorer and select **New ▾ Other...**.
2. In the wizard selection, select **Vaadin ▾ Vaadin Widget** and click **Next**.
3. In the **New Component Wizard**, make the following settings.



Source folder

The root folder of the entire source tree. The default value is the default source tree of your project, and you should normally leave it unchanged unless you have a different project structure.

Package

The parent package under which the new server-side component should be created. If the project does not already have a widget set, one is created under this package in the widgetset subpackage. The subpackage will contain the .gwt.xml descriptor that defines the widget set and the new widget stub under the widgetset.client subpackage.

Name

The class name of the new *server-side component*. The name of the client-side widget stub will be

the same but with "- **Widget**" suffix, for example, **MyComponentWidget**. You can rename the classes afterwards.

Superclass

The superclass of the server-side component. It is **AbstractComponent** by default, but **com.vaadin.ui.AbstractField** or **com.vaadin.ui.AbstractSelect** are other commonly used superclasses. If you are extending an existing component, you should select it as the superclass. You can easily change the superclass later.

Template

Select which template to use. The default is **Full fledged**, which creates the server-side component, the client-side widget, the connector, a shared state object, and an RPC object. The **Connector only** leaves the shared state and RPC objects out.

Finally, click **Finish** to create the new component.

The wizard will:

- Create a server-side component stub in the base package
- If the project does not already have a widget set, the wizard creates a GWT module descriptor file (.gwt.xml) in the base package and modifies the servlet class or the web.xml deployment descriptor to specify the widget set class name parameter for the application
- Create a client-side widget stub (along with the connector and shared state and RPC stubs) in the client.componentname package under the base package

The structure of the server-side component and the client-side widget, and the serialization of component state between them, is explained in the subsequent sections of this chapter.

To compile the widget set, click the **Compile widget set** button in the Eclipse toolbar. See Section 16.2.2, "Compiling the Widget Set" for details. After the compilation finishes, you

should be able to run your application as before, but using the new widget set. The compilation result is written under the WebContent/VAADIN/widgets folder. When you need to recompile the widget set in Eclipse, see Section 16.2.2, "Compiling the Widget Set". For detailed information on compiling widget sets, see Section 13.4, "Compiling a Client-Side Module".

The following setting is inserted in the web.xml deployment descriptor to enable the widget set:

```
<init-param>
  <description>Application widgetset</description>
  <param-name>widgetset</param-name>
  <param-value>com.example.myproject.widgetset.MyprojectApplicationWidgetset</param-value>
</init-param>
```

You can refactor the package structure if you find need for it, but GWT compiler requires that the client-side code *must* always be stored under a package named "client" or a package defined with a source element in the widget set descriptor.

16.2.2. Compiling the Widget Set

After you edit a widget, you need to compile the widget set. The Vaadin Plugin for Eclipse automatically suggests to compile the widget set in various situations, such as when you save a client-side source file. If this gets annoying, you can disable the automatic recompilation in the Vaadin category in project settings, by selecting the **Suspend automatic widgetset builds** option.

You can compile the widget set manually by clicking the **Compile widgetset** button in the Eclipse toolbar, shown in Figure 16.3, "The **Compile Widgetset** Button in Eclipse Toolbar", while the project is open and selected. If the project has multiple widget set definition files, you need to select the one to compile in the Project Explorer.

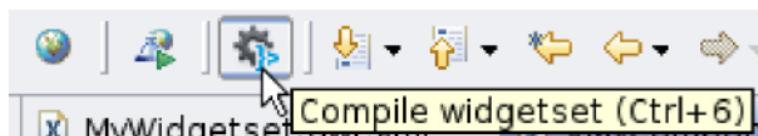
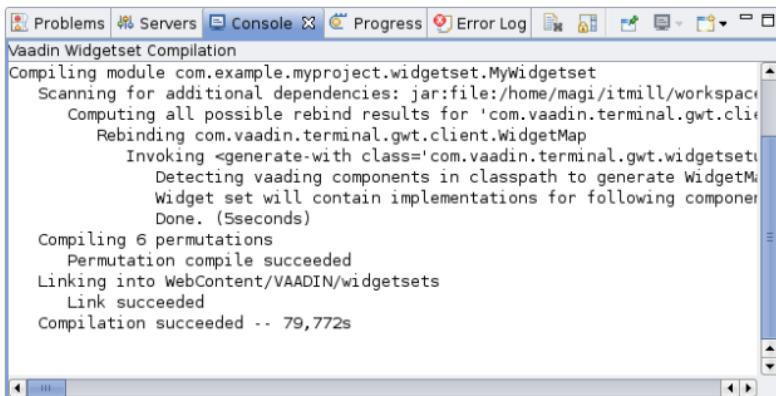


Figure 16.3. The **Compile Widgetset** Button in Eclipse Toolbar

The compilation progress is shown in the **Console** panel in Eclipse, illustrated in Figure 16.4, "Compiling a Widget Set". You should note especially the list of widget sets found in the class path.



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected in the top navigation bar. The title bar says 'Vaadin Widgetset Compilation'. The console output is as follows:

```
Compiling module com.example.myproject.widgetset.MyWidgetset
Scanning for additional dependencies: jar:file:/home/magi/itmill/workspace/
Computing all possible rebind results for 'com.vaadin.terminal.gwt.client.WidgetMap'
Rebinding com.vaadin.terminal.gwt.client.WidgetMap
Invoking <generate-with class='com.vaadin.terminal.gwt.widgetsets.WidgetSetGenerator'>
Detecting vaadin components in classpath to generate WidgetMap
Widget set will contain implementations for following components
Done. (5seconds)
Compiling 6 permutations
Permutation compile succeeded
Linking into WebContent/VAADIN/widgetsets
Link succeeded
Compilation succeeded -- 79,772s
```

Figure 16.4. Compiling a Widget Set

The compilation output is written under the WebContent/VAADIN/widgetsets folder, in a widget set specific folder.

You can speed up the compilation significantly by compiling the widget set only for your browser during development. The generated .gwt.xml descriptor stub includes a disabled element that specifies the target browser. See Section 13.3.2, "Limiting Compilation Targets" for more details on setting the user-agent property.

For more information on compiling widget sets, see Section 13.4, "Compiling a Client-Side Module". Should you compile a widget set outside Eclipse, you need to refresh the project by selecting it in **Project Explorer** and pressing F5.

16.3. Creating a Server-Side Component

Typical server-side Vaadin applications use server-side components that are rendered on the client-side using their counterpart widgets. A server-side component must manage state synchronization between the widget on the client-side, in addition to any server-side logic.

16.3.1. Basic Server-Side Component

The component state is usually managed by a *shared state*, described later in Section 16.5, "Shared State".

```
public class MyComponent extends AbstractComponent {  
    public MyComponent() {  
        getState().setText("This is MyComponent");  
    }  
  
    @Override  
    protected MyComponentState getState() {  
        return (MyComponentState) super.getState();  
    }  
}
```

16.4. Integrating the Two Sides with a Connector

A client-side widget is integrated with a server-side component with a *connector*. A connector is a client-side class that communicates changes to the widget state and events to the server-side.

A connector normally gets the state of the server-side component by the *shared state*, described later in Section 16.5, "Shared State".

16.4.1. A Basic Connector

The basic tasks of a connector is to hook up to the widget and handle events from user interaction and changes received from the server. A connector also has a number of routine infrastructure methods which need to be implemented.

```
@Connect(MyComponent.class)  
public class MyComponentConnector  
    extends AbstractComponentConnector {  
  
    @Override  
    public MyComponentWidget getWidget() {  
        return (MyComponentWidget) super.getWidget();  
    }  
  
    @Override  
    public MyComponentState getState() {  
        return (MyComponentState) super.getState();  
    }  
}
```

```
}

@Override
public void onStateChanged(StateChangeEvent stateChangeEvent)
{
    super.onStateChanged(stateChangeEvent);

    // Do something useful
    final String text = getState().text;
    getWidget().setText(text);
}
}
```

Here, we handled state change with the crude `onStateChanged()` method that is called when any of the state properties is changed. A finer and simpler handling is achieved by using the **@OnStateChange** annotation on a handler method for each property, or by **@DelegateToWidget** on a shared state property, as described later in Section 16.5, “Shared State”.

16.4.2. Communication with the Server-Side

The main task of a connector is to communicate user interaction with the widget to the server-side and receive state changes from the server-side and relay them to the widget.

Server-to-client communication is normally done using a *shared state*, as described in Section 16.5, “Shared State”, as well as RPC calls. The serialization of the state data is handled completely transparently.

For client-to-server communication, a connector can make remote procedure calls (RPC) to the server-side. Also, the server-side component can make RPC calls to the connector. For a thorough description of the RPC mechanism, refer to Section 16.6, “RPC Calls Between Client- and Server-Side”.

16.5. Shared State

The basic communication from a server-side component to its the client-side widget counterpart is handled using a *shared state*. The shared state is serialized transparently. It should be considered read-only on the client-side, as it is not serialized back to the server-side.

A shared state object simply needs to extend the **AbstractComponentState**. The member variables should normally be declared as public.

```
public class MyComponentState extends AbstractComponentState {  
    public String text;  
}
```

A shared state should never contain any logic. If the members have private visibility for some reason, you can also use public setters and getters, in which case the property must not be public.

16.5.1. Location of Shared-State Classes

The shared-state classes are used by both server- and client-side classes, but widget set compilation requires that they must be located in a client-side source package. The default location is under a client package under the package of the .gwt.xml descriptor. If you wish to organize the shared classes separately from other client-side code, you can define separate client-side source packages for pure client-side classes and any shared classes. In addition to shared state classes, shared classes could include enumerations and other classes needed by shared-state or RPC communication.

For example, you could have the following definitions in the .gwt.xml descriptor:

```
<source path="client" />  
<source path="shared" />
```

The paths are relative to the package containing the descriptor.

16.5.2. Accessing Shared State on Server-Side

A server-side component can access the shared state with the getState() method. It is required that you override the base implementation with one that returns the shared state object cast to the proper type, as follows:

```
@Override  
public MyComponentState getState() {
```

```
        return (MyComponentState) super.getState();
    }
```

You can then use the `getState()` to access the shared state object with the proper type.

```
public MyComponent() {
    getState().setText("This is the initial state");
    ...
}
```

16.5.3. Handling Shared State in a Connector

A connector can access a shared state with the `getState()` method. The access should be read-only. It is required that you override the base implementation with one that returns the proper shared state type, as follows:

```
@Override
public MyComponentState getState() {
    return (MyComponentState) super.getState();
}
```

State changes made on the server-side are communicated transparently to the client-side. When a state change occurs, the `onStateChanged()` method in the connector is called. You should always call the superclass method before anything else to handle changes to common component properties.

```
@Override
public void onStateChanged(StateChangeEvent stateChangeEvent) {
    super.onStateChanged(stateChangeEvent);

    // Copy the state properties to the widget properties
    final String text = getState().getText();
    getWidget().setText(text);
}
```

The crude `onStateChanged()` method is called when any of the state properties is changed, allowing you to have even complex logic in how you manipulate the widget according to the state changes. In most cases, however, you can handle the property changes more easily and also more efficiently by using instead the **@OnStateChange** annotation on the handler methods for each property, as described next in Section 16.5.4, “Handling Property State Changes with **@OnStateChange**”, or by delegating the property value directly to

the widget, as described in Section 16.5.5, “Delegating State Properties to Widget”.

16.5.4. Handling Property State Changes with @OnStateChange

The **@OnStateChange** annotation can be used to mark a connector method that handles state change on a particular property, given as parameter for the annotation. In addition to higher clarity, this avoids handling all property changes if a state change occurs in only one or some of them. However, if a state change can occur in multiple properties, you can only use this technique if the properties do not have interaction that prevents handling them separately in arbitrary order.

We can replace the `onStateChange()` method in the earlier connector example with the following:

```
@OnStateChange("text")
void updateText() {
    getWidget().setText(getState().text);
}
```

If the shared state property and the widget property have same name and do not require any type conversion, as is the case in the above example, you could simplify this even further by using the **@DelegateToWidget** annotation for the shared state property, as described in Section 16.5.5, “Delegating State Properties to Widget”.

16.5.5. Delegating State Properties to Widget

The **@DelegateToWidget** annotation for a shared state property defines automatic delegation of the property value to the corresponding widget property of the same name and type, by calling the respective setter for the property in the widget.

```
public class MyComponentState extends AbstractComponentState {
    @DelegateToWidget
    public String text;
}
```

This is equivalent to handling the state change in the connector, as done in the example in Section 16.5.4, “Handling Property State Changes with **@OnStateChange**”.

If you want to delegate a shared state property to a widget property of another name, you can give the property name as a string parameter for the annotation.

```
public class MyComponentState extends AbstractComponentState {  
    @DelegateToWidget("description")  
    public String text;  
}
```

16.5.6. Referring to Components in Shared State

While you can pass any regular Java objects through a shared state, referring to another component requires special handling because on the server-side you can only refer to a server-side component, while on the client-side you only have widgets. References to components can be made by referring to their connectors (all server-side components implement the Connector interface).

```
public class MyComponentState extends AbstractComponentState {  
    public Connector otherComponent;  
}
```

You could then access the component on the server-side as follows:

```
public class MyComponent {  
    public void MyComponent(Component otherComponent) {  
        getState().otherComponent = otherComponent;  
    }  
  
    public Component getOtherComponent() {  
        return (Component)getState().otherComponent;  
    }  
  
    // And the cast method  
    @Override  
    public MyComponentState getState() {  
        return (MyComponentState)super.getState();  
    }  
}
```

On the client-side, you should cast it in a similar fashion to a **ComponentConnector**, or possibly to the specific connector type if it is known.

16.5.7. Sharing Resources

Resources, which commonly are references to icons or other images, are another case of objects that require special handling. A Resource object exists only on the server-side and on the client-side you have an URL to the resource. You need to use the `setResource()` and `getResource()` on the server-side to access a resource, which is serialized to the client-side separately.

Let us begin with the server-side API:

```
public class MyComponent extends AbstractComponent {  
    ...  
  
    public void setMyIcon(Resource myIcon) {  
        setResource("myIcon", myIcon);  
    }  
  
    public Resource getMyIcon() {  
        return getResource("myIcon");  
    }  
}
```

On the client-side, you can then get the URL of the resource with `getResourceUrl()`.

```
@Override  
public void onStateChanged(StateChangeEvent stateChangeEvent) {  
    super.onStateChanged(stateChangeEvent);  
    ...  
  
    // Get the resource URL for the icon  
    getWidget().setMyIcon(getResourceUrl("myIcon"));  
}
```

The widget could then use the URL, for example, as follows:

```
public class MyWidget extends Label {  
    ...  
  
    Element imgElement = null;  
  
    public void setMyIcon(String url) {
```

```
if (imgElement == null) {  
    imgElement = DOM.createImg();  
    getElement().appendChild(imgElement);  
}  
  
DOM.setElementAttribute(imgElement, "src", url);  
}  
}
```

16.6. RPC Calls Between Client- and Server-Side

Vaadin supports making Remote Procedure Calls (RPC) between a server-side component and its client-side widget counterpart. RPC calls are normally used for communicating stateless events, such as button clicks or other user interaction, in contrast to changing the shared state. Either party can make an RPC call to the other side. When a client-side widget makes a call, a server request is made. Calls made from the server-side to the client-side are communicated in the response of the server request during which the call was made.

If you use Eclipse and enable the "Full-Fledged" widget in the New Vaadin Widget wizard, it automatically creates a component with an RPC stub.

16.6.1. RPC Calls to the Server-Side

RPC calls from the client-side to the server-side are made through an RPC interface that extends the `ServerRpc` interface. A server RPC interface simply defines any methods that can be called through the interface.

For example:

```
public interface MyComponentServerRpc extends ServerRpc {  
    public void clicked(String buttonName);  
}
```

The above example defines a single `clicked()` RPC call, which takes a **String** object as the parameter.

You can pass the most common standard Java types, such as primitive and boxed primitive types, **String**, and arrays and some collections (**List**, **Set**, and **Map**) of the supported types.

Also the Vaadin **Connector** and some special internal types can be passed.

An RPC method must return void - the widget set compiler should complain if it doesn't.

Making a Call

Before making a call, you need to instantiate the server RPC object with `RpcProxy.create()`. This is usually done transparently by using `getRpcProxy()`. After that, you can make calls through the server RPC interface that you defined, for example as follows:

```
@Connect(MyComponent.class)
public class MyComponentConnector
    extends AbstractComponentConnector {

    public MyComponentConnector() {
        getWidget().addClickHandler(new ClickHandler() {
            public void onClick(ClickEvent event) {
                final MouseEventDetails mouseDetails =
                    MouseEventDetailsBuilder
                        .buildMouseEventDetails(
                            event.getNativeEvent(),
                            getWidget().getElement());
                MyComponentServerRpc rpc =
                    getRpcProxy(MyComponentServerRpc.class);

                // Make the call
                rpc.clicked(mouseDetails.getButtonName());
            }
        });
    }
}
```

Handling a Call

RPC calls are handled in a server-side implementation of the server RPC interface. The call and its parameters are serialized and passed to the server in an RPC request transparently.

```
public class MyComponent extends AbstractComponent {
    private MyComponentServerRpc rpc =
        new MyComponentServerRpc() {
            private int clickCount = 0;

            public void clicked(String buttonName) {
                Notification.show("Clicked " + buttonName);
            }
        }
}
```

```
    }
}:

public MyComponent() {
    ...
    registerRpc(rpc);
}
}
```

16.7. Component and UI Extensions

Adding features to existing components by extending them by inheritance creates a problem when you want to combine such features. For example, one add-on could add spell-check to a **TextField**, while another could add client-side validation. Combining such add-on features would be difficult if not impossible. You might also want to add a feature to several or even to all components, but extending all of them by inheritance is not really an option. Vaadin includes a component plug-in mechanism for these purposes. Such plug-ins are simply called *extensions*.

Also a UI can be extended in a similar fashion. In fact, some Vaadin features such as the JavaScript execution are UI extensions.

Implementing an extension requires defining a server-side extension class and a client-side connector. An extension can have a shared state with the connector and use RPC, just like a component could.

16.7.1. Server-Side Extension API

The server-side API for an extension consists of class that extends (in the Java sense) the **AbstractExtension** class. It typically has an *extend()* method, a constructor, or a static helper method that takes the extended component or UI as a parameter and passes it to *super.extend()*.

For example, let us have a trivial example with an extension that takes no special parameters, and illustrates the three alternative APIs:

```
public class CapsLockWarning extends AbstractExtension {
    // You could pass it in the constructor
```

```
public CapsLockWarning(PasswordField field) {
    super.extend(field);
}

// Or in an extend() method
public void extend(PasswordField field) {
    super.extend(field);
}

// Or with a static helper
public static addTo(PasswordField field) {
    new CapsLockWarning().extend(field);
}
}
```

The extension could then be added to a component as follows:

```
PasswordField password = new PasswordField("Give it");

// Use the constructor
new CapsLockWarning(password);

// ... or with the extend() method
new CapsLockWarning().extend(password);

// ... or with the static helper
CapsLockWarning.addTo(password);

layout.addComponent(password);
```

Adding a feature in such a "reverse" way is a bit unusual in the Vaadin API, but allows type safety for extensions, as the method can limit the target type to which the extension can be applied, and whether it is a regular component or a UI.

16.7.2. Extension Connectors

An extension does not have a corresponding widget on the client-side, but only an extension connector that extends the **AbstractExtensionConnector** class. The server-side extension class is specified with a @Connect annotation, just like in component connectors.

An extension connector needs to implement the extend() method, which allows hooking to the extended component. The normal extension mechanism is to modify the extended

component as needed and add event handlers to it to handle user interaction. An extension connector can share a state with the server-side extension as well as make RPC calls, just like with components.

In the following example, we implement a "Caps Lock warning" extension. It listens for changes in Caps Lock state and displays a floating warning element over the extended component if the Caps Lock is on.

```
@Connect(CapsLockWarning.class)
public class CapsLockWarningConnector
    extends AbstractExtensionConnector {

    @Override
    protected void extend(ServerConnector target) {
        // Get the extended widget
        final Widget pw =
            ((ComponentConnector) target).getWidget();

        // Preparations for the added feature
        final VOverlay warning = new VOverlay();
        warning.setOwner(pw);
        warning.add(new HTML("Caps Lock is enabled!"));

        // Add an event handler
        pw.addDomHandler(new KeyPressHandler() {
            public void onKeyPress(KeyPressEvent event) {
                if (isEnabled() && isCapsLockOn(event)) {
                    warning.showRelativeTo(passwordWidget);
                } else {
                    warning.hide();
                }
            }
        }, KeyPressEvent.getType());
    }

    private boolean isCapsLockOn(KeyPressEvent e) {
        return e.isShiftKeyDown() ^
            Character.isUpperCase(e.getCharCode());
    }
}
```

The `extend()` method gets the connector of the extended component as the parameter, in the above example a **PasswordFieldConnector**. It can access the widget with the `getWidget()`.

An extension connector needs to be included in a widget set. The class must therefore be defined under the client package of a widget set, just like with component connectors.

16.8. Styling a Widget

To make your widget look stylish, you need to style it. There are two basic ways to define CSS styles for a component: in the widget sources and in a theme. A default style should be defined in the widget sources, and different themes can then modify the style.

16.8.1. Determining the CSS Class

The CSS class of a widget element is normally defined in the widget class and set with `setStyleName()`. A widget should set the styles for its sub-elements as it desires.

For example, you could style a composite widget with an overall style and with separate styles for the sub-widgets as follows:

```
public class MyPickerWidget extends ComplexPanel {  
    public static final String CLASSNAME = "mypicker";  
  
    private final TextBox textBox = new TextBox();  
    private final PushButton button = new PushButton("...");  
  
    public MyPickerWidget() {  
        setElement(Document.get().createDivElement());  
        setStylePrimaryName(CLASSNAME);  
  
        textBox.setStylePrimaryName(CLASSNAME + "-field");  
        button.setStylePrimaryName(CLASSNAME + "-button");  
  
        add(textBox.getElement());  
        add(button.getElement());  
  
        button.addClickHandler(new ClickHandler() {  
            public void onClick(ClickEvent event) {  
                Window.alert("Calendar picker not yet supported!");  
            }  
        });  
    }  
}
```

In addition, all Vaadin components get the `v-widget` class. If it extends an existing Vaadin or GWT widget, it will inherit CSS classes from that as well.

16.8.2. Default Stylesheet

A client-side module, which is normally a widget set, can include stylesheets. They must be placed under the public folder under the folder of the widget set, a described in Section 13.3.1, "Specifying a Stylesheet".

For example, you could style the widget described above as follows:

```
.mypicker {  
    white-space: nowrap;  
}  
  
.mypicker-button {  
    display: inline-block;  
    border: 1px solid black;  
    padding: 3px;  
    width: 15px;  
    text-align: center;  
}
```

Notice that some size settings may require more complex handling and calculating the sizes dynamically.

16.9. Component Containers

Component containers, such as layout components, are a special group of components that require some consideration. In addition to handling state, they need to manage communicating the hierarchy of their contained components to the other side.

The easiest way to implement a component container is extend the **AbstractComponentContainer**, which handles the synchronization of the container server-side components to the client-side.

16.10. Advanced Client-Side Topics

In the following, we mention some topics that you may encounter when integrating widgets.

16.10.1. Client-Side Processing Phases

Vaadin's client-side engine reacts to changes from the server in a number of phases, the order of which can be relevant for a connector. The processing occurs in the `handleUIDLMessage()` method in **ApplicationConnection**, but the logic can be quite overwhelming, so we describe the phases in the following summary.

1. Any dependencies defined by using **@JavaScript** or **@StyleSheet** on the server-side class are loaded. Processing does not continue until the browser confirms that they have been loaded.
2. New connectors are instantiated and `init()` is run for each Connector.
3. State objects are updated, but no state change event is fired yet.
4. The connector hierarchy is updated, but no hierarchy change event is fired yet. `setParent()` and `setChildren()` are run in this phase.
5. Hierarchy change events are fired. This means that all state objects and the entire hierarchy are already up to date when this happens. The DOM hierarchy should in theory be up to date after all hierarchy events have been handled, although there are some built-in components that for various reasons do not always live up to this promise.
6. Captions are updated, causing `updateCaption()` to be invoked on layouts as needed.
7. **@DelegateToWidget** is handled for all changed state objects using the annotation.
8. State change events are fired for all changed state objects.
9. `updateFromUIDL()` is called for legacy connectors.
10. All RPC methods received from the server are invoked.

11. Connectors that are no longer included in the hierarchy are unregistered. This calls `onUnregister()` on the Connector.
12. The layout phase starts, first checking the sizes and positions of all elements, and then notifying any `ElementResizeListener`s, as well as calling the appropriate layout method for the connectors that implement either `[classname]#SimpleManagedLayout` or **DirectionalManagedLayout** interface.

16.11. Creating Add-ons

Add-ons are the most convenient way to reuse Vaadin code, either commercially or free. Vaadin Directory serves as the store for the add-ons. You can distribute add-ons both as JAR libraries and Zip packages.

Creating a typical add-on package involves the following tasks:

- Compile server-side classes
- Compile JavaDoc (optional)
- Build the JAR
- Include Vaadin add-on manifest
- Include the compiled server-side classes
- Include the compiled JavaDoc (optional)
- Include sources of client-side classes for widget set compilation (optional)
- Include any JavaScript dependency libraries (optional)
- Exclude any test or demo code in the project

The exact contents depend on the add-on type. Component add-ons often include a widget set, but not always, such as JavaScript components or pure server-side components. You can also have data container and theme add-ons, as well as various tools.

It is common to distribute the JavaDoc in a separate JAR, but you can also include it in the same JAR.

16.11.1. Exporting Add-on in Eclipse

If you use the Vaadin Plugin for Eclipse for your add-on project, you can simply export the add-on from Eclipse.

1. Select the project and then **File → Export** from the menu
2. In the export wizard that opens, select **Vaadin → Vaadin Add-on Package**, and click **Next**
3. In the **Select the resources to export** panel, select the content that should be included in the add-on package. In general, you should include sources in src folder (at least for the client-side package), compiled server-side classes, themes in WebContent/VAADIN/themes. These are all included automatically. You probably want to leave out any demo or example code.

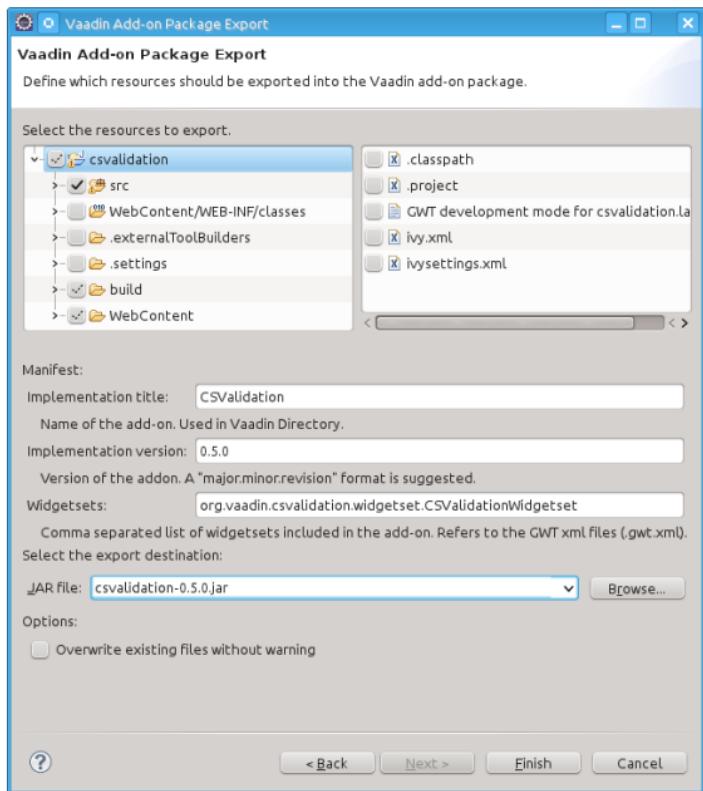


Figure 16.5. Exporting a Vaadin Add-on

If you are submitting the add-on to Vaadin Directory, the **Implementation title** should be exactly the name of the add-on in Directory. The name may contain spaces and most other letters. Notice that *it is not possible to change the name later*.

The **Implementation version** is the version of your add-on. Typically experimental or beta releases start from 0.1.0, and stable releases from 1.0.0.

The **Widgetsets** field should list the widget sets included in the add-on, separated by commas. The widget sets should be listed by their class name, that is, without the .gwt.xml extension.

The **JAR file** is the file name of the exported JAR file. It should normally include the version number of the add-on. You should follow the Maven format for the name, such as myaddon-1.0.0.jar.

Finally, click **Finish**.

16.12. Migrating from Vaadin 6

The client-side architecture was redesigned almost entirely in Vaadin 7. In Vaadin 6, state synchronization was done explicitly by serializing and deserializing the state on the server- and client-side. In Vaadin 7, the serialization is handled automatically by the framework using state objects.

In Vaadin 6, a server-side component serialized its state to the client-side using the `Paintable` interface in the client-side and deserialized the state through the `VariableOwner` interface. In Vaadin 7, these are done through the `ClientConnector` interface.

On the client-side, a widget deserialized its state through the `Paintable` interface and sent state changes through the `ApplicationConnection` object. In Vaadin 7, these are replaced with the `ServerConnector`.

In addition to state synchronization, Vaadin 7 has an RPC mechanism that can be used for communicating events. They are especially useful for events that are not associated with a state change, such as a button click.

The framework ensures that the connector hierarchy and states are up-to-date when listeners are called.

16.12.1. Quick (and Dirty) Migration

Vaadin 7 has a compatibility layer that allows quick conversion of a widget.

1. Create a connector class, such as `MyConnector`, that extends `LegacyConnector`. Implement the `getWidget()` method.

2. Move the @ClientWidget(MyWidget.class) from the server-side component, say **MyComponent**, to the **MyConnector** class and make it @Connect(MyComponent.class).
3. Have the server-side component implement the LegacyComponent interface to enable compatibility handling.
4. Remove any calls to super.paintContent()
5. Update any imports on the client-side

16.13. Integrating JavaScript Components and Extensions

Vaadin allows simplified integration of pure JavaScript components, as well as component and UI extensions. The JavaScript connector code is published from the server-side. As the JavaScript integration does not involve GWT programming, no widget set compilation is needed.

16.13.1. Example JavaScript Library

There are many kinds of component libraries for JavaScript. In the following, we present a simple library that provides one object-oriented JavaScript component. We use this example later to show how to integrate it with a server-side Vaadin component.

The example library includes a single **MyComponent** component, defined in mylibrary.js.

```
// Define the namespace
var mylibrary = mylibrary || {};

mylibrary.MyComponent = function (element) {
    element.innerHTML =
        "<div class='caption>Hello, world!</div>" +
        "<div class='textinput>Enter a value: " +
        "<input type='text' name='value'/>" +
        "<input type='button' value='Click'/" +
        "</div>";

// Style it
element.style.border = "thin solid red";
```

```

element.style.display = "inline-block";

// Getter and setter for the value property
this.getValue = function () {
    return element;
    getElementsByTagName("input")[0].value;
};

this.setValue = function (value) {
    element.getElementsByTagName("input")[0].value =
        value;
};

// Default implementation of the click handler
this.click = function () {
    alert("Error: Must implement click() method");
};

// Set up button click
var button = element.getElementsByTagName("input")[1];
var self = this; // Can't use this inside the function
button.onclick = function () {
    self.click();
};

```

When used in an HTML page, the library would be included with the following definition:

```
<script type="text/javascript"
       src="mylibrary.js"></script>
```

You could then use it anywhere in the HTML document as follows:

```
<!-- Placeholder for the component -->
<div id="foo"></div>

<!-- Create the component and bind it to the placeholder -->
<script type="text/javascript">
    window.foo = new mylibrary.MyComponent(
        document.getElementById("foo"));
    window.foo.click = function () {
        alert("Value is " + this.getValue());
    }
</script>
```



Figure 16.6. A JavaScript Component Example

You could interact with the component with JavaScript for example as follows:

```
<a href="javascript:foo.setValue('New value')>Click here</a>
```

16.13.2. A Server-Side API for a JavaScript Component

To begin integrating such a JavaScript component, you would need to sketch a bit how it would be used from a server-side Vaadin application. The component should support writing the value as well as listening for changes to it.

```
final MyComponent mycomponent = new MyComponent();

// Set the value from server-side
mycomponent.setValue("Server-side value");

// Process a value input by the user from the client-side
mycomponent.addValueChangeListener(
    new MyComponent.ValueChangeListener() {
        @Override
        public void valueChange() {
            Notification.show("Value: " + mycomponent.getValue());
        }
    });
}

layout.addComponent(mycomponent);
```

Basic Server-Side Component

A JavaScript component extends the **AbstractJavaScriptComponent**, which handles the shared state and RPC for the component.

```
package com.vaadin.book.examples.client.js;

@JavaScript(["mylibrary.js", "mycomponent-connector.js"])
public class MyComponent extends AbstractJavaScriptComponent {
    public interface ValueChangeListener extends Serializable {
        void valueChange();
    }
    ArrayList<ValueChangeListener> listeners =
        new ArrayList<ValueChangeListener>();
```

```
public void addValueChangeListener(  
    ValueChangeListener listener) {  
    listeners.add(listener);  
}  
  
public void setValue(String value) {  
    getState().value = value;  
}  
  
public String getValue() {  
    return getState().value;  
}  
  
@Override  
protected MyComponentState getState() {  
    return (MyComponentState) super.getState();  
}  
}
```

Notice later when creating the JavaScript connector that its name must match the package name of this server-side class.

The shared state of the component is as follows:

```
public class MyComponentState extends JavaScriptComponentState {  
    public String value;  
}
```

If the member variables are private, you need to have public setters and getters for them, which you can use in the component.

16.13.3. Defining a JavaScript Connector

A JavaScript connector is a function that initializes the JavaScript component and handles communication between the server-side and the JavaScript code.

A connector is defined as a connector initializer function that is added to the window object. The name of the function must match the server-side class name, with the full package path. Instead of the Java dot notation for the package name, underscores need to be used as separators.

The Vaadin client-side framework adds a number of methods to the connector function. The `this.getElement()` method returns the HTML DOM element of the component. The `this.getState()` returns a shared state object with the current state as synchronized from the server-side.

```

window.com_vaadin_book_examples_client_js_MyComponent =
function() {
    // Create the component
    var mycomponent =
        new mylibrary.MyComponent(this.getElement());

    // Handle changes from the server-side
    this.onStateChange = function() {
        mycomponent.setValue(this.getState().value);
    };

    // Pass user interaction to the server-side
    var self = this;
    mycomponent.click = function() {
        self.onClick(mycomponent.getValue());
    };
};

```

In the above example, we pass user interaction using the JavaScript RPC mechanism, as described in the next section.

16.13.4. RPC from JavaScript to Server-Side

User interaction with the JavaScript component has to be passed to the server-side using an RPC (Remote Procedure Call) mechanism. The JavaScript RPC mechanism is almost equal to regular client-side widgets, as described in Section 16.6, "RPC Calls Between Client- and Server-Side".

Handling RPC Calls on the Server-Side

Let us begin with the RPC function registration on the server-side. RPC calls are handled on the server-side in function handlers that implement the `JavaScriptFunction` interface. A server-side function handler is registered with the `addFunction()` method in **AbstractJavaScriptComponent**. The server-side registration actually defines a JavaScript method that is available in the client-side connector object.

Continuing from the server-side **MyComponent** example we defined earlier, we add a constructor to it that registers the function.

```

public MyComponent() {
    addFunction("onClick", new JavaScriptFunction() {
        @Override
        public void call(JsonArray arguments) {

```

```
    getState().setValue(arguments.getString(0));
    for (ValueChangeListener listener: listeners)
        listener.valueChange();
    }
}:
```

Making an RPC Call from JavaScript

An RPC call is made simply by calling the RPC method in the connector. In the constructor function of the JavaScript connector, you could write as follows (the complete connector code was given earlier):

```
window.com_vaadin_book_examples_gwt_js_MyComponent =
function() {
    ...
    var connector = this;
    mycomponent.click = function() {
        connector.onClick(mycomponent.getValue());
    };
}
```

Here, the mycomponent.click is a function in the example JavaScript library, as described in Section 16.13.1, “Example JavaScript Library”. The onClick() is the method we defined on the server-side. We pass a simple string parameter in the call.

You can pass anything that is valid in JSON notation in the parameters.

Chapter 17

Using Vaadin Add-ons

17.1. Overview	571
17.2. Using Add-ons in a Maven Project	573
17.3. Installing Commercial Vaadin Add-on License	582
17.4. Troubleshooting	585

This chapter describes the installation of add-on components, themes, containers, and other tools from the Vaadin Directory and the use of commercial add-ons offered by Vaadin.

17.1. Overview

In addition to the components, layouts, themes, and data sources built in into the core Vaadin library, many others are available as add-ons. Vaadin Directory provides a rich collection of add-ons for Vaadin, and you may find others from independent sources. Add-ons are also one way to share your own components between projects.

17.1.1. Installing

Installing add-ons from Vaadin Directory is simple, just adding a Maven or an Ivy dependency, or downloading the JAR package and dropping it in the web library folder of the project.

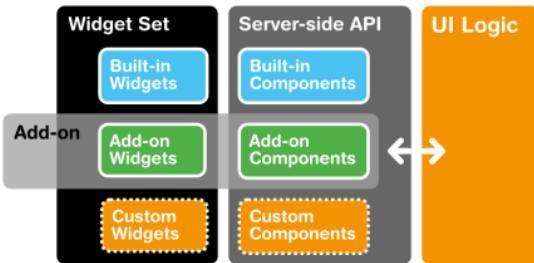


Figure 17.1. Role of the widget set

Most add-ons include *widgets*, client-side counterparts of the server-side components used in the Vaadin Java API, as illustrated in Figure 17.1, “Role of the widget set”. The *widget set* needs to be compiled into the application widget set.

Adding the dependency in Maven projects and compiling the widget set is described in Section 17.2, “Using Add-ons in a Maven Project”. The section also describes how to use the online compilation and CDN services during development.

For Eclipse projects that use Ivy for dependency management, see ??? You can also download and install add-ons from a ZIP-package, as described in ???.

17.1.2. Add-on Licenses

Add-ons available from Vaadin Directory are distributed under different licenses, of which some are commercial. While the add-ons can be downloaded directly, you should note their license and other terms and conditions. Many are offered under a dual licensing agreement so that they can be used in open source projects for free, and many have a trial period for closed-source development. Commercial Vaadin add-ons distributed under the CVAL license require installing a license key as instructed in Section 17.3, “Installing Commercial Vaadin Add-on License”.

17.1.3. Feedback and Support

After trying out an add-on, you can give some feedback to the author of the add-on by rating the add-on with one to five stars and optionally leaving a comment. Most add-ons

also have a discussion forum thread for user feedback and questions.

17.2. Using Add-ons in a Maven Project

To use add-ons in a Maven project, you simply have to add them as dependencies in the project POM.

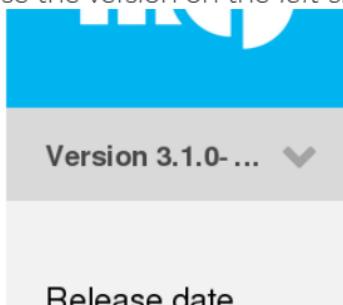
Most add-ons include client-side widgets, counterparts of the server-side components. The add-on widgets will be included and compiled into the *application widget set*. Compiling the widget set is handled by the Vaadin Maven Plugin. It is enabled in Maven projects created from the Vaadin archetypes, as described in Chapter 3, *Creating a Vaadin Application*.

The plugin will attempt to automatically detect if you need to compile the application widget set. It will generate a target/generated-sources/gwt/AppWidgetset.gwt.xml widget set descriptor, update it when necessary, and use it for compiling the widget set.

17.2.1. Adding a Dependency

Vaadin Directory provides a Maven repository for all the add-ons in the Directory.

1. Open the add-on page in Vaadin Directory.
2. Choose the version on the *left-side menu bar*.



The maven
Learn about
vaadin.com

The latest version is selected by default, but you can choose another version from the drop-down menu.

3. Click **Install** to display the dependency declarations.



If the add-on is available with multiple licenses, you will be prompted to select a license for the dependency.

4. Select the **Maven** tab.



5. Open the pom.xml file. In single-module projects, you only have one, located in the root folder of the project. In multi-module projects, open the one in your Vaadin application module.

Eclipse IDE

Right-click on the pom.xml file and select **Open With ➔ XML Editor**. You can also left-click it, which opens the Maven POM editor, but directly editing the XML code is usually easier. You can also use the XML editor tab of the POM editor.

6. Copy and paste the dependency declaration to under the dependencies element.

```
...
<dependencies>
    ...
    <dependency>
        <groupId>[replaceable]com.vaadin.addon</groupId>
        <artifactId>[replaceable]vaadin-charts</artifactId>
        <version>[replaceable]1.0.0</version>
```

```
</dependency>  
</dependencies>
```

You can use an exact version number, as is done in the example above, or LATEST to always use the latest version of the add-on.

The POM excerpt given in Directory includes also a repository definition, but if you have used the Vaadin archetypes to create your project, it already includes the definition.

7. *For commercial add-ons*, you need a license key.

Click **Activate** to buy a license, obtain a trial license key, or get the key from your Pro Tools subscription.



For official Vaadin add-ons, see Section 17.3, "Installing Commercial Vaadin Add-on License" for more information.

8. *In Vaadin 7.6 and older*: You need to compile the widget set as described in Section 17.2.2, "Compiling the Application Widget Set".

17.2.2. Compiling the Application Widget Set



Note

The widget set is automatically compiled in Vaadin 7.7 and later. The plugin will attempt to detect any add-ons that need the widget set to be compiled.

Just note that it can take a bit time to compile.

To speed up, instead of compiling it locally, you can also use a public cloud service to compile it for you, and use it directly from a CDN service.

See Section 17.2.3, "Widget Set Modes" for instructions.

In projects that use Vaadin 7.6 or older, you need to manually compile the widget set as follows.

Enabling Widget Set Compilation

Compiling the widget set in Maven projects requires the Vaadin Maven plugin. It is included in Maven projects created with a current Vaadin archetype.

If all is well, you are set to go.

If you have used the Vaadin archetypes to create the project, the POM should include all necessary declarations to compile the widget set.

However, if your Maven project has been created otherwise, you may need to enable widget set compilation manually. The simplest way to do that is to copy the definitions from a POM created with the archetype. Specifically, you need to copy the vaadin-maven-plugin definition in the plugin section, as well as the Vaadin dependencies and any relevant settings.

Compiling the Widget Set

The widget set compilation occurs in standard Maven build phase, such as with *package* or *install* goal.

Eclipse IDE

Click the **Compile Vaadin Widgetset** button in the Eclipse toolbar.



Command-line

Simply run the package goal.

```
$ mvn package
```

Then, just deploy the WAR to your application server.

Recompiling the Widget Set

The Vaadin plugin for Maven tries to avoid recompiling the widget set unless necessary, which sometimes means that it is not compiled even when it should. Running the clean goal usually helps, but causes a full recompilation. You can compile the widget set manually by running the `vaadin:compile` goal.

Eclipse IDE

Click the **Compile Vaadin Widgetset** button in the Eclipse toolbar.

Command-line

Run the `vaadin:compile` goal.

```
$ mvn vaadin:compile
```

Updating the Widget Set

Note that the `vaadin:compile` goal does not update the project widget set by searching new widget sets from the class path. It must be updated when you, for example, add or remove add-ons. You can do that by running the `vaadin:update-widget-set` goal in the project directory.

```
$ mvn vaadin:update-widgetset
[INFO] auto discovered modules [your.company.gwt.ProjectNameWidgetSet]
[INFO] Updating widgetset your.company.gwt.ProjectNameWidgetSet
[ERROR] 27/10/2011 19:22:34 com.vaadin.terminal.gwt.widgetsetutils.ClassPathExplorer getAvailableWidgetSets
[ERROR] INFO: Widgetsets found from classpath:
```

Do not mind the "ERROR" labels, they are just an issue with the Vaadin Plugin for Maven.

After running the update, you need to run the `vaadin:compile` goal to actually compile the widget set.

17.2.3. Widget Set Modes

The application widget set is by default compiled locally. You can also have it compiled in a public cloud service provided by Vaadin, and either use it directly from a CDN service or download it to serve it from your development server.

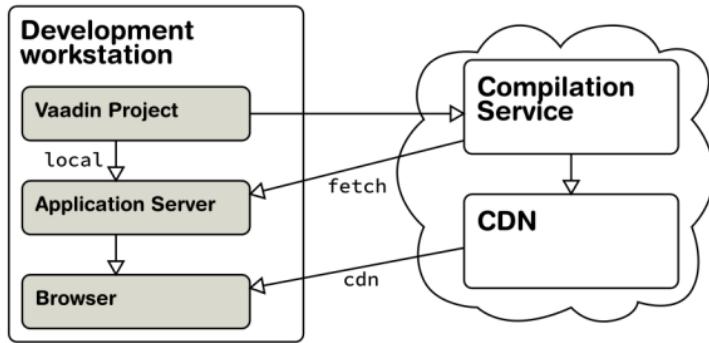


Figure 17.2. Widget set modes

The widget set mode, defined in the project POM, determines how the widget set is compiled.

local (default)

The widget set is compiled locally in your development workstation.

cdn

Compilation is done in the public cloud service. It is served directly from the CDN (Content Delivery Network) service.

Using CDN is recommended for development.

fetch

Compilation is done in the public cloud service. The widget set is then downloaded and deployed with the rest of the application to the application server.

The mode is set with a `vaadin.widgetset.mode` property in the properties section at the beginning of the project POM.

Local Widget Set Compilation

If add-ons are detected, an `AppWidgetset.gwt.xml` descriptor file is generated into the generated-resources folder, and later updated automatically. The compiler uses the descriptor to compile the widget set, which is included in the web application archive.

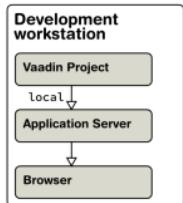


Figure 17.3. Local widget set compilation

Local compilation is needed in projects that have custom widgets or widget sets that are not available from the Maven central repository or from the Vaadin add-ons or pre-releases repositories. Local compilation is necessary for completely offline work.

Local compilation is currently the default mode. It is therefore not necessary to set it explicitly, unless you have made global Maven settings and want to override them in a particular project. You can set the local parameter in the properties section of your pom.xml:

```
<properties>
...
<vaadin.widgetset.mode>local</vaadin.widgetset.mode>
</properties>
...
```

Online Widget Set Compilation and CDN

The online compilation service makes it easier to use add-on components in Vaadin applications, by avoiding compilation of widget sets locally. It caches the widget sets, so often one is available immediately. A widget set can combine widgets from multiple add-ons and if a particular combination does not already exist, the service automatically compiles it.

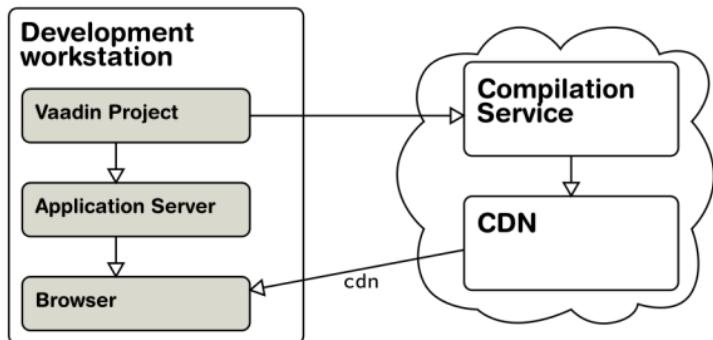


Figure 17.4. Online widget set compilation and CDN

The CDN (Content Delivery Network) is a global network of web servers that serve the cached widget sets directly when you load your application in your browser. Avoiding local compilation can speed up development significantly. In development teams, all can use the shared widget sets immediately.

Tip

While this gives benefits, the application will not work without online connectivity. When you need to avoid the demo effect, either use the local or fetch mode.

The cloud service compiles a widget set with a given Vaadin version and a list of add-ons installed in your project. The Maven plug-in receives a public URL where the widget set is available and generates an `AppWidgetset.java` file that configures your application to use the online version. The file is generated to the default Java package.

Note

Online compilation and CDN can only be used with publicly available add-ons. This means that the add-on dependencies must be available in the Maven central repository or in the Vaadin add-ons or pre-releases repositories. If you have

custom widget implementations or non-public add-ons in your sources, you cannot use the online compilation and CDN service.

Enabling Compilation

To enable online compilation and delivery from the CDN, set the widget set mode to cdn in the properties section of your pom.xml:

```
<properties>
```

```
    ...<vaadin.widgetset.mode>cdn</vaadin.widgetset.mode>
</properties>
```

When using the online compilation service, a WidgetsetInfo implementation is generated for your project; this makes it possible for your application to find the widget set from the correct location.



Note

The CDN is not meant to be used in production.

It is meant for speeding up development for yourself and your team. It is also useful if you maintain your source code in GitHub or a similar service, so that your globally working development team can immediately use the widget sets without need to compile them.

For production, especially in intranet applications, you should normally use the local or fetch modes. This ensures that separating the availability of the Vaadin CDN service from availability of the application server does not add an extra point of failure.

They can be used for production if your application is intended as globally available, you want to gain the global delivery benefit of the

Vaadin CDN, and the availability tradeoff is not significant.

Serving Remotely Compiled Widget Set Locally

If you want to use online compilation, but still serve the files as part of your application, you can use the fetch mode.

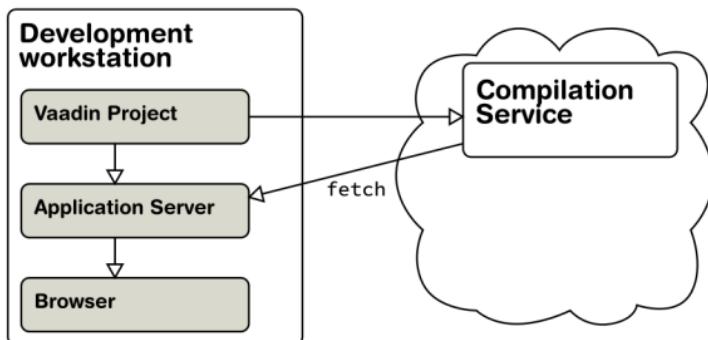


Figure 17.5. Fetching online widget set compilation

The Maven plugin downloads the compiled widget set into the project as it was compiled locally. It generates an **AppWidgetset** class and used to provide a correct URL to the locally stored widget set.

To enable the fetch mode, in the properties section of your pom.xml:

```
<properties>
...
<vaadin.widgetset.mode>fetch</vaadin.widgetset.mode>
</properties>
...
```

17.3. Installing Commercial Vaadin Add-on License

The commercial Vaadin add-ons require installing a license key before using them. The license keys are development licenses and checked during widget set compilation, or in

Vaadin TestBench when executing tests, so you do not need them when deploying the application.

17.3.1. Obtaining License Keys

You can purchase add-ons or obtain a free trial key from the Vaadin website. You need to register in the website to obtain a key.

You can get license keys from vaadin.com/pro/licenses.

1. Select in the Vaadin website **My Account → Licenses** or directly **Licenses** if you are a Pro Tools subscriber.



Licenses

Vaadin Framework 6 7

Select which version of Vaadin Framework are you using.

Subscription Licenses (CVAL)

Official Vaadin add-ons are licensed under the [Commercial Vaadin Add-on License](#).

Get your subscription licenses below.

License key

	Spreadsheet 1.0.0.beta1	Spreadsheet component for business applications	
	TestBench 4.0.2	An environment for automated user interface regression testing of Vaadin applications on multiple platforms and browsers	
	Charts 2.0.0	The most comprehensive visualization library available for Vaadin.	
	TouchKit 4.0.0	Build touch-enabled applications for iOS and Android mobile devices using Vaadin	

2. Click on a license key to obtain the purchased or trial key.

A screenshot of a web page showing the 'Spreadsheet 1.0.0.beta1' license key. The key is displayed as 'L1cen5e-c0de' in a large font. To the right of the key is a blue 'Download' button. Below the key, there is a note: 'Copy-paste the license key to a file called vaadin.spreadsheet.developer.license or download it directly.' and 'After downloading you need to copy the file to your home directory (/home/<username>/).'

17.3.2. Installing License Key in License File

To install the license key in a development workstation, you can copy and paste it verbatim to a file in your home directory.

License for each product has a separate license file as follows:

Vaadin Charts

.vaadin.charts.developer.license

Vaadin Spreadsheet

.vaadin.spreadsheet.developer.license

Vaadin TestBench

.vaadin.testbench.developer.license

Vaadin TouchKit

.vaadin.touchkit.developer.license

For example, in Linux and OS X:

```
$ echo "L1cen5e-c0de" > ~/vaadin.<product>.developer.license
```

17.3.3. Passing License Key as System Property

You can also pass the key as a system property to the widget set compiler, usually with a -D option. For example, on the command-line:

```
$ java -Dvaadin.<product>.developer.license=L1cen5e-c0de ...
```

where the <product> is the product ID, such as charts, spreadsheet, designer, or testbench.

Passing License Key in Different Environments

How you actually pass the parameter to the widget set compiler depends on the development environment and the build system that you use to compile the widget set. Below are listed a few typical environments:

Eclipse IDE

To install the license key for all projects, select **Window** → **Preferences** and navigate to the **Java** → **Installed JREs** section. Select the JRE version that you use for the ap-

plication and click **Edit**. In the **Default VM arguments**, give the *-D* expression as shown above.

Apache Ant

If compiling the project with Apache Ant, you could set the key in the Ant script as follows:

```
<sysproperty key="vaadin.<product>.developer.license"
    value="L1cen5e-c0de"/>
```

However, you should never store license keys in a source repository, so if the Ant script is stored in a source repository, you should pass the license key to Ant as a property that you then use in the script for the value argument of the `<sysproperty>` as follows:

```
<sysproperty key="vaadin.<product>.developer.license"
    value="${vaadin.<product>.developer.license}">
```

When invoking Ant from the command-line, you can pass the property with a *-D* parameter to Ant.

Apache Maven

If building the project with Apache Maven, you can pass the license key with a *-D* parameter to Maven:

```
$ mvn -Dvaadin.<product>.developer.license=L1cen5e-c0de package
```

where the `<product>` is the product ID, such as charts, spreadsheet, designer, or testbench.

Continuous Integration Systems

In CIS systems, you can pass the license key to build runners as a system property in the build configuration. However, this only passes it to a runner. As described above, Ant does not pass it to sub-processes implicitly, so you need to forward it explicitly as described earlier.

17.4. Troubleshooting

If you experience problems with using add-ons, you can try the following:

- Check the `.gwt.xml` descriptor file under the the project root package. For example, if the project root package

is com.example.myproject, the widget set definition file is typically at com/example/project/AppWidgetset.gwt.xml. The location is not fixed and it can be elsewhere, as long as references to it match. See Section 13.3, “Client-Side Module Descriptor” for details on the contents of the client-side module descriptor, which is used to define a widget set.

- Check the WEB-INF/web.xml deployment descriptor and see that the servlet for your UI has a widget set parameter, such as the following:

```
<init-param>
  <description>UI widgetset</description>
  <param-name>widgetset</param-name>
  <param-value>com.example.project.AppWidgetSet</param-value>
</init-param>
```

Check that the widget set class corresponds with the .gwt.xml file in the source tree.

- See the VAADIN/widgetsets directory and check that the widget set appears there. You can remove it and recompile the widget set to see that the compilation works properly.
- Use the **Net** tab in Firebug to check that the widget set (and theme) is loaded properly.
- Use the ?debug parameter for the application to open the debug window and check if there is any version conflict between the widget set and the Vaadin library, or the themes. See Section 11.3, “Debug Mode and Window” for details.
- Refresh and recompile the project. In Eclipse, select the project and press F5, stop the server, clean the server temporary directories, and restart it.
- Check the Error Log view in Eclipse (or in the IDE you use).

Chapter 18

Vaadin Charts

18.1. Overview	587
18.2. Installing Vaadin Charts	590
18.3. Basic Use	593
18.4. Chart Types	603
18.5. Chart Configuration	634
18.6. Chart Data	645
18.7. Advanced Uses	652
18.8. Timeline	654

This chapter provides the documentation for the Vaadin Charts add-on.

18.1. Overview

Vaadin Charts is a feature-rich interactive charting library for Vaadin. It provides a **Chart** component. The **Chart** can visualize one- and two-dimensional numeric data in many available chart types. The charts allow flexible configuration of all the chart elements as well as the visual style. The library includes a number of built-in visual themes, which you can extend further. The basic functionalities allow the user to interact with the chart elements in various ways, and you can define custom interaction with click events.



Figure 18.1. Vaadin Charts

The data displayed in a chart can be one- or two dimensional tabular data, or scatter data with free X and Y values. Data displayed in range charts has minimum and maximum values instead of singular values.

This chapter covers the basic use of Vaadin Charts and the chart configuration. For detailed documentation of the configuration parameters and classes, please refer to the JavaDoc API documentation of the library.

In the following basic examples, which we study further in Section 18.3, “Basic Use” and Section 18.3, “Basic Use”, we demonstrate how to display one-dimensional data in a column graph and customize the X and Y axis labels and titles.

Java:

```
Chart chart = new Chart(ChartType.BAR);
chart.setWidth("400px");
chart.setHeight("300px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Planets");
conf.setSubTitle("The bigger they are the harder they pull");
conf.getLegend().setEnabled(false); // Disable legend

// The data
ListSeries series = new ListSeries("Diameter");
series.setData(4900, 12100, 12800,
             6800, 143000, 125000,
             51100, 49500);
conf.addSeries(series);

// Set the category labels on the axis correspondingly
XAxis xaxis = new XAxis();
xaxis.setCategories("Mercury", "Venus", "Earth",
                    "Mars", "Jupiter", "Saturn",
                    "Uranus", "Neptune");
xaxis.setTitle("Planet");
conf.addXAxis(xaxis);

// Set the Y axis title
YAxis yaxis = new YAxis();
yaxis.setTitle("Diameter");
```

```

yaxis.getLabels().setFormatter(
    "function() {return Math.floor(this.value/1000) + '\u00a0Mm';}");
yaxis.getLabels().setStep(2);
conf.addyAxis(yaxis);

layout.addComponent(chart);

```

Web Components:

```

<!DOCTYPE html>
<html>
<head>
<script src="bower_components/webcomponentsjs/webcomponents-lite.min.js"></script>
<link rel="import" href="bower_components/vaadin-charts/vaadin-bar-chart.html">
</head>
<body>
<div style="width:400px; height:300px">
<vaadin-bar-chart>
    <chart-title>Planets</chart-title>
    <subtitle>The bigger they are the harder they pull</subtitle>
    <legend>
        <enabled>false</enabled>
    </legend>
    <x-axis>
        <chart-title>Planet</chart-title>
        <categories>Mercury, Venus, Earth, Mars,
        Jupiter, Saturn, Uranus, Neptune</categories>
    </x-axis>
    <y-axis>
        <chart-title>Diameter</chart-title>
        <labels formatter = "function () { return this.value / 1000 + 'Mm';}">
            <step>2</step>
        </labels>
    </y-axis>
    <data-series name="Diameter">
        <data>
            4900, 12100, 12800, 6800,
            143000, 125000, 51100, 49500
        </data>
    </data-series>
</vaadin-bar-chart>
</div>
</body>
</html>

```

The resulting chart is shown in Figure 18.2, “Basic Chart Example”.

Planets

The bigger they are the harder they pull

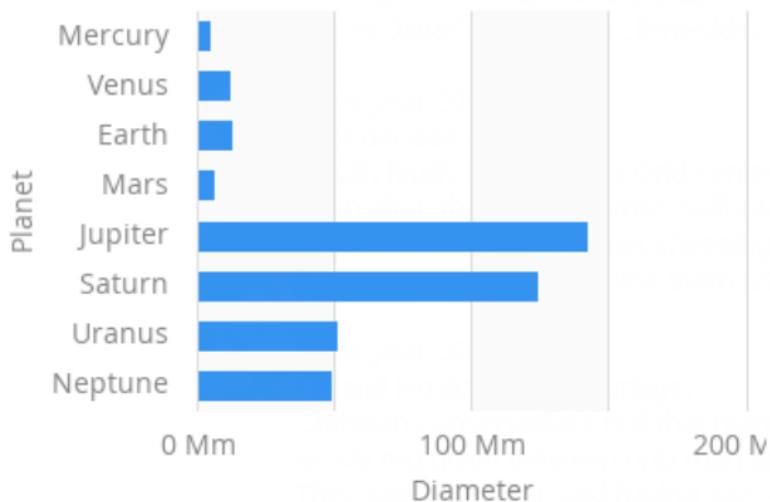


Figure 18.2. Basic Chart Example

18.1.1. Licensing

Vaadin Charts is a commercial product licensed under the CVAL License (Commercial Vaadin Add-On License). A license needs to be purchased for all use, including web deployments as well as intranet use.

The commercial licenses can be purchased from the Vaadin Directory, where you can also find the license details and download the Vaadin Charts.

18.2. Installing Vaadin Charts

As with most Vaadin add-ons, you can install Vaadin Charts as a Maven or Ivy dependency in your project, or from an installation package. For general instructions on installing add-ons, please see Chapter 17, *Using Vaadin Add-ons*.

Vaadin Charts 4 requires Vaadin 8.0 or later.

Using Vaadin Charts requires a license key, which you must install before compiling the widget set. The widget set must be compiled after setting up the dependency or library JARs.

18.2.1. Maven Dependency

The Maven dependency for Vaadin Charts is as follows:

```
<dependency>
  <groupId>com.vaadin</groupId>
  <artifactId>vaadin-charts</artifactId>
  <version>4.0.0</version>
</dependency>
```

You also need to define the Vaadin add-ons repository if not already defined:

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

18.2.2. Ivy Dependency

The Ivy dependency, to be defined in ivy.xml, would be as follows:

```
<dependency org="com.vaadin" name="vaadin-charts"
  rev="4.0.0" />
```

It is generally recommended to use a fixed version number, but you can also use latest.release to get the latest release.

18.2.3. Installing License Key

You need to install a license key before compiling the widget set. The license key is checked during widget set compilation, so you do not need it when deploying the application.

You can purchase Vaadin Charts or obtain a free trial key from the Vaadin Charts download page in Vaadin Directory. You need to register in Vaadin Directory to obtain the key.

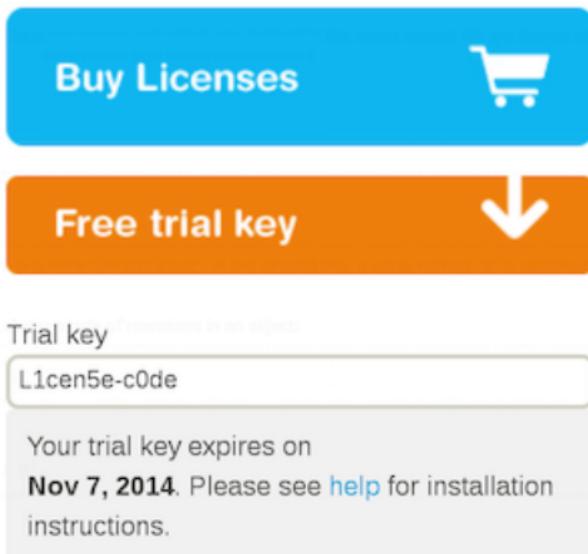


Figure 18.3. Obtaining License Key from Vaadin Directory

To install the license key in a development workstation, you can copy and paste it verbatim to a `vaadin.charts.developer.license` file in your home directory. For example, in Linux and OS X:

```
$ echo "L1cen5e-c0de" > ~/vaadin.charts.developer.license
```

You can also pass the key as a system property to the widget set compiler, usually with a `-D` option. For example, on the command-line:

```
$ java -Dvaadin.charts.developer.license=L1cen5e-c0de ...
```

Passing License Key in Different Environments

How you actually pass the parameter to the widget set compiler depends on the development environment and the build system that you use to compile the widget set. Below are listed a few typical environments:

Eclipse IDE

To install the license key for all projects, select **Window** **Preferences** and navigate to the **Java** **Installed JREs** section. Select the JRE version that you use for the ap-

plication and click **Edit**. In the **Default VM arguments**, give the *-D* expression as shown above.

Apache Ant

If compiling the project with Apache Ant, you could set the key in the Ant script as follows:

```
<sysproperty key="vaadin.charts.developer.license"  
            value="L1cen5e-c0de"/>
```

However, you should never store license keys in a source repository, so if the Ant script is stored in a source repository, you should pass the license key to Ant as a property that you then use in the script for the value argument of the `<sysproperty>` as follows:

```
<sysproperty key="vaadin.charts.developer.license"  
            value="${vaadin.charts.developer.license}" />
```

When invoking Ant from the command-line, you can pass the property with a *-D* parameter to Ant.

Apache Maven

If building the project with Apache Maven, you can pass the license key with a *-D* parameter to Maven:

```
$ mvn -Dvaadin.charts.developer.license=L1cen5e-c0de package
```

Continuous Integration Systems

In CIS systems, you can pass the license key to build runners as a system property in the build configuration. However, this only passes it to a runner. As described above, Ant does not pass it to sub-processes implicitly, so you need to forward it explicitly as described earlier.

18.3. Basic Use

The **Chart** is a regular Vaadin component, which you can add to a layout. You can give the chart type in the constructor or set it later in the chart model. A chart has a height of 400 pixels and takes full width by default, which settings you may often need to customize.

```
Chart chart = new Chart(ChartType.COLUMN);  
chart.setWidth("400px"); //100% by default
```

```
chart.setHeight("300px"); // 400px by default
```

```
...  
layout.addComponent(chart);
```

The chart types are described in Section 18.4, “Chart Types”. The main parts of a chart are illustrated in Figure 18.4, “Chart Elements”.

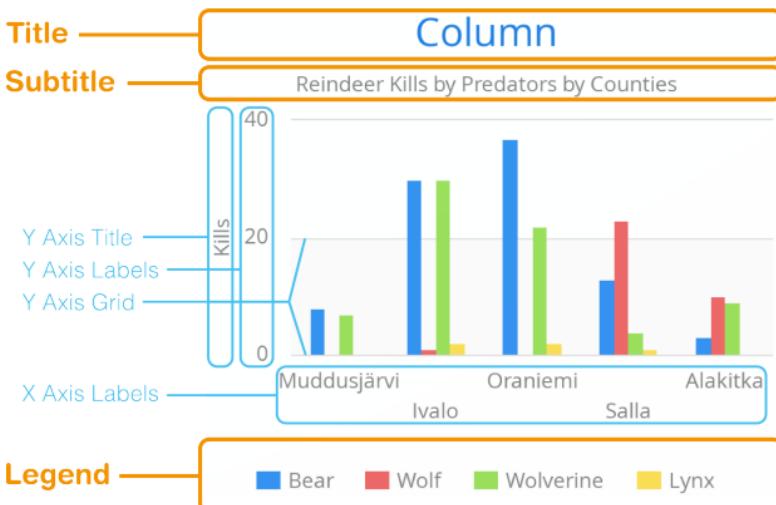


Figure 18.4. Chart Elements

To actually display something in a chart, you typically need to configure the following aspects:

- Basic chart configuration
- Configure *plot options* for the chart type
- Configure one or more *data series* to display
- Configure axes

The plot options can be configured for each data series individually, or for different chart types in mixed-type charts.

18.3.1. Basic Chart Configuration

After creating a chart, you need to configure it further. At the least, you need to specify the data series to be displayed in the configuration.

Most methods available in the **Chart** object handle its basic Vaadin component properties. All the chart-specific properties are in a separate **Configuration** object, which you can access with the `getConfiguration()` method.

```
Configuration conf = chart.getConfiguration();
conf.setTitle("Reindeer Kills by Predators");
conf.setSubTitle("Kills Grouped by Counties");
```

The configuration properties are described in more detail in Section 18.5, "Chart Configuration".

18.3.2. Plot Options

Many chart settings can be configured in the *plot options* of the chart or data series. Some of the options are chart type specific, as described later for each chart type, while many are shared.

For example, for line charts, you could disable the point markers as follows:

```
// Disable markers from lines
PlotOptionsLine plotOptions = new PlotOptionsLine();
plotOptions.setMarker(new Marker(false));
conf.setPlotOptions(plotOptions);
```

You can set the plot options for the entire chart or for each data series separately, allowing also mixed-type charts, as described in Section 18.3.6, "Mixed Type Charts".

The shared plot options are described in Section 18.5.1, "Plot Options".

18.3.3. Chart Data Series

The data displayed in a chart is stored in the chart configuration as a list of **Series** objects. A new data series is added in a chart with the `addSeries()` method.

```
ListSeries series = new ListSeries("Diameter");
series.setData(4900, 12100, 12800,
              6800, 143000, 125000,
              5100, 49500);
conf.addSeries(series);
```

The data can be specified with a number of different series types **DataSeries**, **ListSeries**, **AreaListSeries**, **RangeSeries**, and **ContainerDataSeries**.

Data point features, such as color and size, can be defined in the versatile **DataSeries**, which contains **DataSeriesItem** items. Special chart types, such as box plots and 3D scatter charts require using their own special data point type.

The data series configuration is described in more detail in Section 18.6, "Chart Data".

18.3.4. Axis Configuration

One of the most common tasks for charts is customizing its axes. At least, you usually want to set the axis titles. Usually you also want to specify labels for data values in the axes.

When an axis is categorical rather than numeric, you can define category labels for the items. They must be in the same order and the same number as you have values in your data series.

```
XAxis xaxis = new XAxis();
xaxis.setCategories("Mercury", "Venus", "Earth",
                    "Mars", "Jupiter", "Saturn",
                    "Uranus", "Neptune");
xaxis.setTitle("Planet");
conf.addXAxis(xaxis);
```

Formatting of numeric labels can be done with JavaScript expressions, for example as follows:

```
// Set the Y axis title
YAxis yaxis = new YAxis();
yaxis.setTitle("Diameter");
yaxis.getLabels().setFormatter(
    "function() {return Math.floor(this.value/1000) + '\u00a0Mm\u00a0';}");
yaxis.getLabels().setStep(2);
conf.addYAxis(yaxis);
```

18.3.5. Displaying Multiple Series

The simplest data, which we saw in the examples earlier in this chapter, is one-dimensional and can be represented with a single data series. Most chart types support multiple data series, which are used for representing two-dimensional data. For example, in line charts, you can have multiple lines and in column charts the columns for different series are grouped by category. Different chart types can offer alternative display modes, such as stacked columns. The legend displays the symbols for each series.

```
// The data
// Source: V. Maijala, H. Norberg, J. Kumpula, M. Nieminen
// Calf production and mortality in the Finnish
// reindeer herding area, 2002.
String predators[] = {"Bear", "Wolf", "Wolverine", "Lynx"};
int kills[][] = {
    {8, 0, 7, 0}, // Muddusjarvi
    {30, 1, 30, 2}, // Ivalo
    {37, 0, 22, 2}, // Oraniemi
    {13, 23, 4, 1}, // Salla
    {3, 10, 9, 0} // Alakitka
};

// Create a data series for each numeric column in the table
for (int predator = 0; predator < 4; predator++) {
    ListSeries series = new ListSeries();
    series.setName(predators[predator]);

    // The rows of the table
    for (int location = 0; location < kills.length; location++)
        series.addData(kills[location][predator]);
    conf.addSeries(series);
}
```

The result for both regular and stacked column chart is shown in Figure 18.5, “Multiple Series in a Chart”. Stacking is enabled with `setStacking()` in **PlotOptionsColumn**.



Figure 18.5. Multiple Series in a Chart

18.3.6. Mixed Type Charts

You can enable mixed charts by setting the chart type in the **PlotOptions** object for a data series, which overrides the default chart type set in the **Chart** object. You can also make color and other settings for the series in the plot options.

For example, to get a line chart, you need to use **PlotOptionsLine**.

// A data series as column graph

```
DataSource series1 = new DataSource();
PlotOptionsColumn options1 = new PlotOptionsColumn();
options1.setColor(SolidColor.BLUE);
series1.setPlotOptions(options1);
series1.setData(4900, 12100, 12800,
    6800, 143000, 125000, 51100, 49500);
conf.addSeries(series1);
```

// A data series as line graph

```
ListSeries series2 = new ListSeries("Diameter");
```

```
PlotOptionsLine options2 = new PlotOptionsLine();
options2.setColor(SolidColor.RED);
series2.setPlotOptions(options2);
series2.setData(4900, 12100, 12800,
               6800, 143000, 125000, 51100, 49500);
conf.addSeries(series2);
```

In the above case, where we set the chart type for each series, the overall chart type is irrelevant.

18.3.7. 3D Charts

Most chart types can be made 3-dimensional by adding 3D options to the chart. You can rotate the charts, set up the view distance, and define the thickness of the chart features, among other things. You can also set up a 3D axis frame around a chart.

Planets – In 3D!

The bigger they are the harder they pull

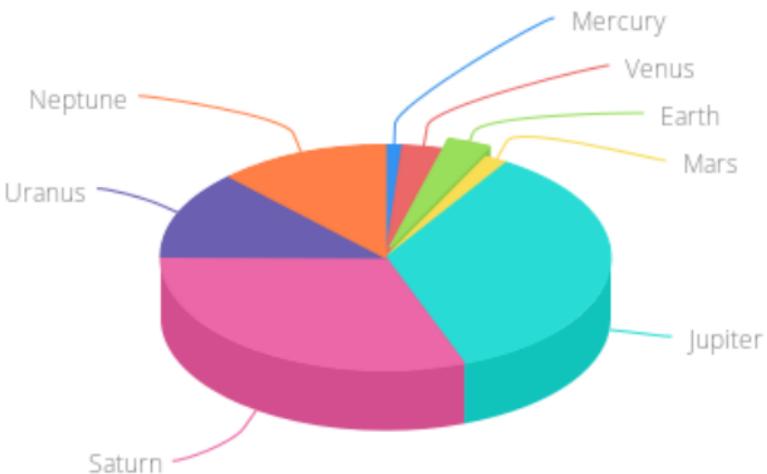


Figure 18.6. 3D Charts

3D Options

3D view has to be enabled in the **Options3d** configuration, along with other parameters. Minimally, to have some 3D effect, you need to rotate the chart according to the *alpha* and *beta* parameters.

Let us consider a basic scatter chart for an example. The basic configuration for scatter charts is described elsewhere, but let us look how to make it 3D.

```
Chart chart = new Chart(ChartType.SCATTER);
Configuration conf = chart.getConfiguration();
... other chart configuration ...
```

// In 3D!

```
Options3d options3d = new Options3d();
options3d.setEnabled(true);
options3d.setAlpha(10);
options3d.setBeta(30);
options3d.setDepth(135); // Default is 100
options3d.setViewDistance(100); // Default
conf.getChart().setOptions3d(options3d);
```

The 3D options are as follows:

alpha

The vertical tilt (pitch) in degrees.

beta

The horizontal tilt (yaw) in degrees.

depth

Depth of the third (Z) axis in pixel units.

enabled

Whether 3D plot is enabled. Default is *false*.

frame

Defines the 3D frame, which consists of a back, bottom, and side panels that display the chart grid.

+

```
Frame frame = new Frame();
Back back=new Back();
back.setColor(SolidColor.BEIGE);
back.setSize();
frame.setBack(back);
options3d.setFrame(frame);
```

[parameter]#viewDistance#: View distance **for** creating perspective distortion. Default is 100.

3D Plot Options

The above sets up the general 3D view, but you also need to configure the 3D properties of the actual chart type. The 3D plot options are chart type specific. For example, a pie has *depth* (or thickness), which you can configure as follows:

// Set some plot options

```
PlotOptionsPie options = new PlotOptionsPie();
... Other plot options for the chart ...
```

```
options.setDepth(45); // Our pie is quite thick
```

```
conf.setPlotOptions(options);
```

3D Data

For some chart types, such as pies and columns, the 3D view is merely a visual representation for one- or two-dimensional data. Some chart types, such as scatter charts, also feature a third, *depth axis*, for data points. Such data points can be given as **DataSeriesItem3d** objects.

The Z parameter is *depth* and is not scaled: there is no configuration for the depth or Z axis. Therefore, you need to handle scaling yourself as is done in the following.

// Orthogonal data points in 2x2x2 cube
double[][] points = { {0.0, 0.0, 0.0}, // x, y, z

```
{1.0, 0.0, 0.0},
{0.0, 1.0, 0.0},
{0.0, 0.0, 1.0},
{-1.0, 0.0, 0.0},
{0.0, -1.0, 0.0},
{0.0, 0.0, -1.0}};
```

```
DataSeries series = new DataSeries();
```

```
for (int i=0; i<points.length; i++) {
```

```
    double x = points[i][0];
    double y = points[i][1];
    double z = points[i][2];
```

// Scale the depth coordinate, as the depth axis is
// not scaled automatically

```
DataSeriesItem3d item = new DataSeriesItem3d(x, y,
```

```
    z * options3d.getDepth().doubleValue());
    series.add(item);
}
conf.addSeries(series);
```

Above, we defined 7 orthogonal data points in the $2 \times 2 \times 2$ cube centered in origo. The 3D depth was set to 135 earlier. The result is illustrated in Figure 18.7, "3D Scatter Chart".

Scatter – in 3D!

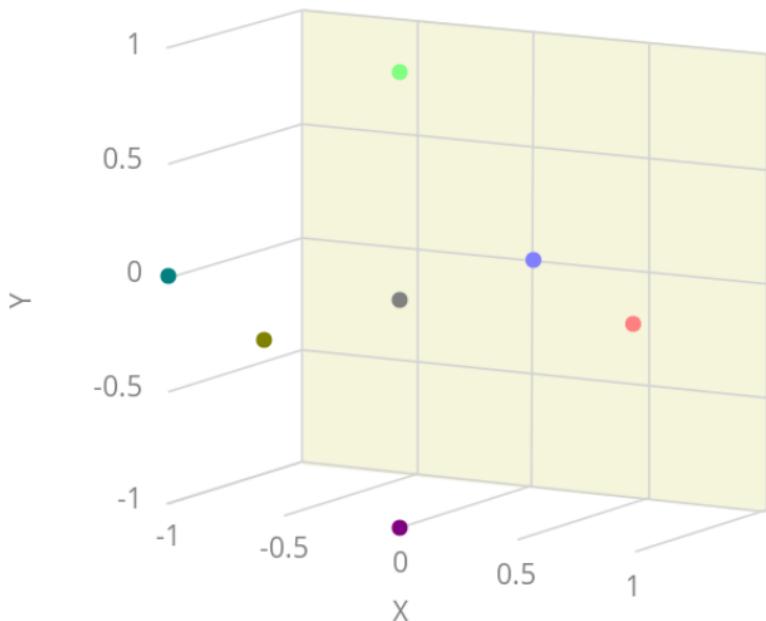


Figure 18.7. 3D Scatter Chart

18.3.8. Chart Themes

The visual style and essentially any other chart configuration can be defined in a *theme*. All charts shown in a UI may have only one theme, which can be set with `setTheme()` in the **ChartOptions**.

The **ChartOptions** is a **UI** extension that is created and referenced by calling the `get()` as follows:

```
// Set Charts theme for the current UI  
ChartOptions.get().setTheme(new SkiesTheme());
```

The **VaadinTheme** is the default chart theme in Vaadin Charts. Other available themes are **GrayTheme**, **GridTheme**, **SkiessTheme**, and **HighChartsDefaultTheme**.

A theme is a Vaadin Charts configuration that is used as a template for the configuration when rendering the chart.

18.3.9. Chart Language

Lang class provides an opportunity for internationalizing charts. You can specify a custom decimal point, names of months, weekdays and their abbreviated forms. You can also specify the text to display when the chart contains no data.

```
final Lang fi = new Lang();  
  
// Set language properties  
fi.setDecimalPoint(",");  
  
fi.setMonths(new String[] { "Tammikuu", "Helmikuu", "Maaliskuu",  
    "Huhtikuu", "Toukokuu", "Kesäkuu",  
    "Heinäkuu", "Elokuu", "Syyskuu",  
    "Lokakuu", "Marraskuu", "Joulukuu" });  
  
fi.setShortMonths(new String[] { "Tamm", "Helmi", "Maalis",  
    "Huhti", "Touko", "Kesä",  
    "Heina", "Elo", "Sys",  
    "Loka", "Marras", "Joulu" });  
  
fi.setWeekdays(new String[] { "Ma", "Ti", "Ke", "To", "Pe", "La", "Su" });  
fi.setNoData("Data puuttuu");
```

Lang option is global and can be set with `setLang()` in the **ChartOptions**:

```
ChartOptions.get().setLang(fi);
```

18.4. Chart Types

Vaadin Charts comes with over a dozen different chart types. You normally specify the chart type in the constructor of the **Chart** object. The available chart types are defined in the **ChartType** enum. You can later read or set the chart type with the `chartType` property of the chart model, which you can get with `getConfiguration().getChart()`.

The supported chart types are:

area	arearange	areaspline	areasplinerange
bar	boxplot	bubble	column
columnrange	errorbar	flags	funnel
gauge	heatmap	line	pie
polygon	pyramid	scatter	solidgauge
sparkline	spline	treemap	waterfall

Each chart type has its specific plot options and support its specific collection of chart features. They also have specific requirements for the data series.

The basic chart types and their variants are covered in the following subsections.

18.4.1. Line and Spline Charts

Line charts connect the series of data points with lines. In the basic line charts the lines are straight, while in spline charts the lines are smooth polynomial interpolations between the data points.

Table 18.1. Line Chart Subtypes

ChartType	Plot Options Class
LINE	PlotOptionsLine
SPLINE	PlotOptionsSpline

Plot Options

The *color* property in the line plot options defines the line color, *lineWidth* the line width, and *dashStyle* the dash pattern for the lines.

See Section 18.4.6, "Scatter Charts" for plot options regarding markers and other data point properties. The markers can also be configured for each data point.

18.4.2. Area Charts

Area charts are like line charts, except that they fill the area between the line and some threshold value on Y axis. The threshold depends on the chart type. In addition to the base type, chart type combinations for spline interpolation and ranges are supported.

Table 18.2. Area Chart Subtypes

ChartType	Plot Options Class
AREA	PlotOptionsArea
AREASPLINE	PlotOptionsAreaSpline
AREARANGE	PlotOptionsAreaRange
AREASPLINERANGE	PlotOptionsAreaSplineRange

In area range charts, the area between a lower and upper value is painted with a transparent color. The data series must specify the minimum and maximum values for the Y coordinates, defined either with **RangeSeries**, as described in Section 18.6.3, “Range Series”, or with **DataSeries**, described in Section 18.6.2, “Generic Data Series”.

Plot Options

Area charts support *stacking*, so that multiple series are piled on top of each other. You enable stacking from the plot options with `setStacking()`. The `Stacking.NORMAL` stacking mode does a normal summative stacking, while the `Stacking.PERCENT` handles them as proportions.

The `fillColor` property for the area is defined with the `fillColor` property and its transparency with `fillOpacity` (the opposite of transparency) with a value between 0.0 and 1.0.

The `color` property in the line plot options defines the line color, `lineWidth` the line width, and `dashStyle` the dash pattern for the lines.

See Section 18.4.6, “Scatter Charts” for plot options regarding markers and other data point properties. The markers can also be configured for each data point.

18.4.3. Column and Bar Charts

Column and bar charts illustrate values as vertical or horizontal bars, respectively. The two chart types are essentially equivalent, just as if the orientation of the axes was inverted.

Multiple data series, that is, two-dimensional data, are shown with thinner bars or columns grouped by their category, as described in Section 18.3.5, “Displaying Multiple Series”. Enabling stacking with `setStacking()` in plot options stacks the columns or bars of different series on top of each other.

You can also have *COLUMNRANGE* charts that illustrate a range between a lower and an upper value, as described in Section 18.4.11, “Area and Column Range Charts”. They require the use of **RangeSeries** for defining the lower and upper values.

Table 18.3. Column and Bar Chart Subtypes

ChartType	Plot Options Class
COLUMN	PlotOptionsColumn
COLUMNRANGE	PlotOptionsColumnRange
BAR	PlotOptionsBar

See the API documentation for details regarding the plot options.

18.4.4. Error Bars

An error bars visualize errors, or high and low values, in statistical data. They typically represent high and low values in data or a multitude of standard deviation, a percentile, or a quantile. The high and low values are represented as horizontal lines, or “whiskers”, connected by a vertical stem.

While error bars technically are a chart type (`ChartType.ERRORBAR`), you normally use them together with some primary chart type, such as a scatter or column chart.

Average and Extreme Temperatures in Turku

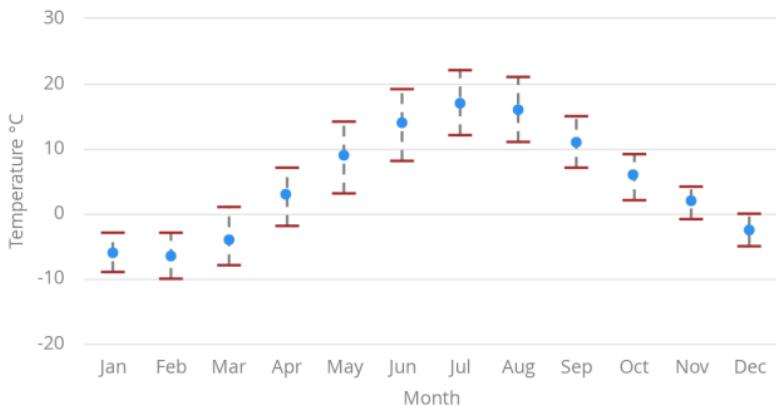


Figure 18.8. Error Bars in a Scatter Chart

To display the error bars for data points, you need to have a separate data series for the low and high values. The data series needs to use the **PlotOptionsErrorBar** plot options type.

```
// Create a chart of some primary type
Chart chart = new Chart(ChartType.SCATTER);
chart.setWidth("600px");
chart.setHeight("400px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Average Temperatures in Turku");
conf.getLegend().setEnabled(false);

// The primary data series
ListSeries averages = new ListSeries();
-6, -6.5, -4, 3, 9, 14, 17, 16, 11, 6, 2, -2.5;

// Error bar data series with low and high values
DataSeries errors = new DataSeries();
errors.add(new DataSeriesItem(0, -9, -3));
errors.add(new DataSeriesItem(1, -10, -3));
errors.add(new DataSeriesItem(2, -8, 1));
...

// Configure the stem and whiskers in error bars
PlotOptionsErrorbar barOptions = new PlotOptionsErrorbar();
barOptions.setStemColor(SolidColor.GREY);
barOptions.setStemWidth(2);
barOptions.setStemDashStyle(DashStyle.DASH);
barOptions.setWhiskerColor(SolidColor.BROWN);
barOptions.setWhiskerLength(80, Sizeable.Unit.PERCENTAGE); // 80% of category width
barOptions.setWhiskerWidth(2); // Pixels
errors.setPlotOptions(barOptions);

// The errors should be drawn lower
conf.addSeries(errors);
conf.addSeries(averages);
```

Note that you should add the error bar series first, to have it rendered lower in the chart.

Plot Options

Plot options for error bar charts have type **PlotOptionsErrorBar**. It has the following chart-specific plot option properties:

whiskerColor, *whiskerWidth*, and *whiskerLength*

The color, width (vertical thickness), and length of the horizontal "whiskers" that indicate high and low values.

stemColor, *stemWidth*, and *stemDashStyle*

The color, width (thickness), and line style of the vertical "stems" that connect the whiskers. In box plot charts, which also have stems, they extend from the quadrantile box.

18.4.5. Box Plot Charts

Box plot charts display the distribution of statistical variables. A data point has a median, represented with a horizontal line, upper and lower quartiles, represented by a box, and a low and high value, represented with T-shaped "whiskers". The exact semantics of the box symbols are up to you.

Box plot chart is closely related to the error bar chart described in Section 18.4.4, "Error Bars", sharing the box and whisker elements.

Orienteering Split Times

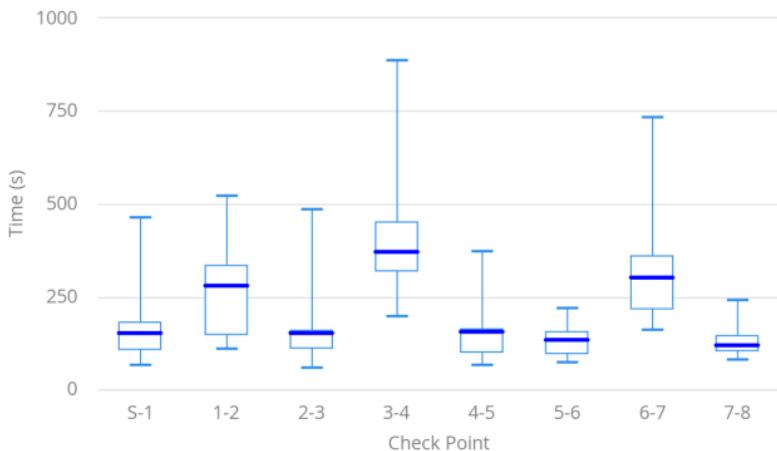


Figure 18.9. Box Plot Chart

The chart type for box plot charts is `ChartType.BOXPLOT`. You normally have just one data series, so it is meaningful to disable the legend.

```
Chart chart = new Chart(ChartType.BOXPLOT);
chart.setWidth("400px");
chart.setHeight("300px");
```

```
// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Orienteering Split Times");
conf.getLegend().setEnabled(false);
```

Plot Options

The plot options for box plots have type `PlotOptionsBoxPlot`, which extends the slightly more generic `PlotOptionsErrorBar`. They have the following plot option properties:

`medianColor, medianWidth`

Color and width (vertical thickness) of the horizontal median indicator line.

For example:

```
// Set median line color and thickness
PlotOptionsBoxplot plotOptions = new PlotOptionsBoxplot();
```

```
plotOptions.setMedianColor(SolidColor.BLUE);
plotOptions.setMedianWidth(3);
conf.setPlotOptions(plotOptions);
```

Data Model

As the data points in box plots have five different values instead of the usual one, they require using a special **Box-PlotItem**. You can give the different values with the setters, or all at once in the constructor.

```
// Orienteering control point times for runners
double data[][] = orienteeringdata();

DataSeries series = new DataSeries();
for (double cpointtimes[]: data) {
    StatAnalysis analysis = new StatAnalysis(cpointtimes);
    series.add(new BoxPlotItem(analysis.low(),
        analysis.firstQuartile(),
        analysis.median(),
        analysis.thirdQuartile(),
        analysis.high())));
}
conf.setSeries(series);
```

If the "low" and "high" attributes represent an even smaller quantile, or a larger multiple of standard deviation, you can have outliers. You can plot them with a separate data series, with

18.4.6. Scatter Charts

Scatter charts display a set of unconnected data points. The name refers to freely given X and Y coordinates, so the **Data-Series** or **ContainerSeries** are usually the most meaningful data series types for scatter charts.

Random Sphere

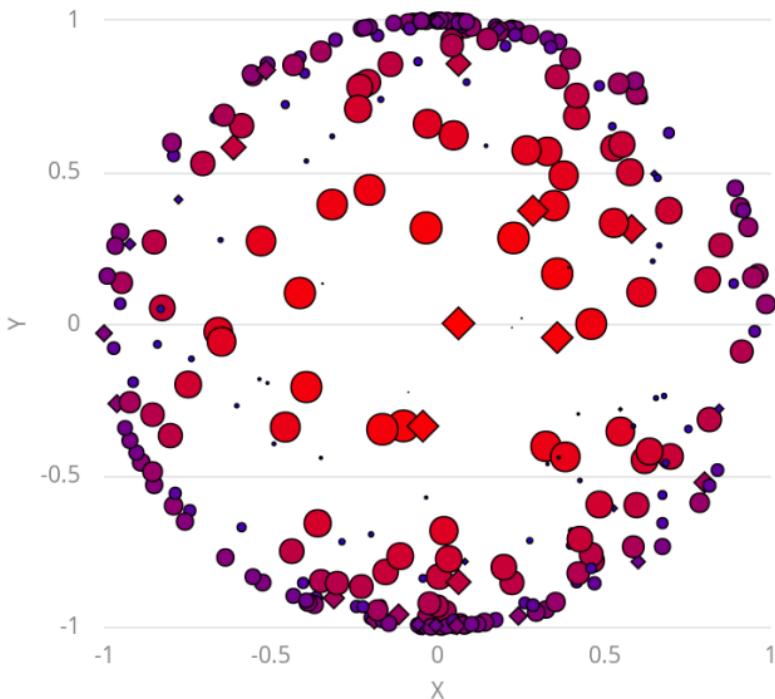


Figure 18.10. Scatter Chart

The chart type of a scatter chart is `ChartType.SCATTER`. Its options can be configured in a **PlotOptionsScatter** object, although it does not have any chart-type specific options.

```
Chart chart = new Chart(ChartType.SCATTER);
chart.setWidth("500px");
chart.setHeight("500px");
```

```
// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Random Sphere");
conf.getLegend().setEnabled(false); // Disable legend
```

```
PlotOptionsScatter options = new PlotOptionsScatter();
// ... Give overall plot options here ...
conf.setPlotOptions(options);
```

```
DataSeries series = new DataSeries();
```

```

for (int i=0; i<300; i++) {
    double lng = Math.random() * 2 * Math.PI;
    double lat = Math.random() * Math.PI - Math.PI/2;
    double x = Math.cos(lat) * Math.sin(lng);
    double y = Math.sin(lat);
    double z = Math.cos(lng) * Math.cos(lat);

    DataSeriesItem point = new DataSeriesItem(x,y);
    Marker marker = new Marker();
    // Make settings as described later
    point.setMarker(marker);
    series.add(point);
}
conf.addSeries(series);

```

The result was shown in Figure 18.10, "Scatter Chart".

Data Point Markers

Scatter charts and other charts that display data points, such as line and spline charts, visualize the points with *markers*. The markers can be configured with the **Marker** property objects available from the plot options of the relevant chart types, as well as at the level of each data point, in the **Data-SeriesItem**. You need to create the marker and apply it with the `setMarker()` method in the plot options or the data series item.

For example, to set the marker for an individual data point:

```

DataSeriesItem point = new DataSeriesItem(x,y);
Marker marker = new Marker();
// ... Make any settings ...
point.setMarker(marker);
series.add(point);

```

Marker Shape Properties

A marker has a *lineColor* and a *fillColor*, which are set using a **Color** object. Both solid colors and gradients are supported. You can use a **SolidColor** to specify a solid fill color by RGB values or choose from a selection of predefined colors in the class.

```

// Set line width and color
marker.setLineWidth(1); // Normally zero width

```

```
marker.setLineColor(SolidColor.BLACK);

// Set RGB fill color
int level = (int) Math.round((l-z)*127);
marker.setFillColor(
    new SolidColor(255-level, 0, level));
point.setMarker(marker);
series.add(point);
```

You can also use a color gradient with **GradientColor**. Both linear and radial gradients are supported, with multiple color stops.

Marker size is determined by the *radius* parameter, which is given in pixels. The actual visual radius includes also the line width.

```
marker.setRadius((z+1)*5);
```

Marker Symbols

Markers are visualized either with a shape or an image symbol. You can choose the shape from a number of built-in shapes defined in the **MarkerSymbolEnum** enum (*CIRCLE*, *SQUARE*, *DIAMOND*, *TRIANGLE*, or *TRIANGLE_DOWN*). These shapes are drawn with a line and fill, which you can set as described above.

```
marker.setSymbol(MarkerSymbolEnum.DIAMOND);
```

You can also use any image accessible by a URL by using a **MarkerSymbolUrl** symbol. If the image is deployed with your application, such as in a theme folder, you can determine its URL as follows:

```
String url = VaadinServlet.getCurrent().getServletContext()
    .getContextPath() + "/VAADIN/themes/mytheme/img/smiley.png";
marker.setSymbol(new MarkerSymbolUrl(url));
```

The line, radius, and color properties are not applicable to image symbols.

18.4.7. Bubble Charts

Bubble charts are a special type of scatter charts for representing three-dimensional data points with different point

sizes. We demonstrated the same possibility with scatter charts in Section 18.4.6, “Scatter Charts”, but the bubble charts make it easier to define the size of a point by its third (Z) dimension, instead of the radius property. The bubble size is scaled automatically, just like for other dimensions. The default point style is also more bubbly.



Figure 18.11. Bubble Chart

The chart type of a bubble chart is `ChartType.BUBBLE`. Its options can be configured in a **PlotOptionsBubble** object, which has a single chart-specific property, `displayNegative`, which controls whether bubbles with negative values are displayed at all. More typically, you want to configure the bubble marker. The bubble tooltip is configured in the basic configuration. The Z coordinate value is available in the formatter JavaScript with this `point.z` reference.

The bubble radius is scaled linearly between a minimum and maximum radius. If you would rather scale by the area of the bubble, you can approximate that by taking square root of the Z values.

18.4.8. Pie Charts

A pie chart illustrates data values as sectors of size proportionate to the sum of all values. The pie chart is enabled with `ChartType.PIE` and you can make type-specific settings in the **PlotOptionsPie** object as described later.

```
Chart chart = new Chart(ChartType.PIE);
Configuration conf = chart.getConfiguration();
...

```

A ready pie chart is shown in Figure 18.12, "Pie Chart".

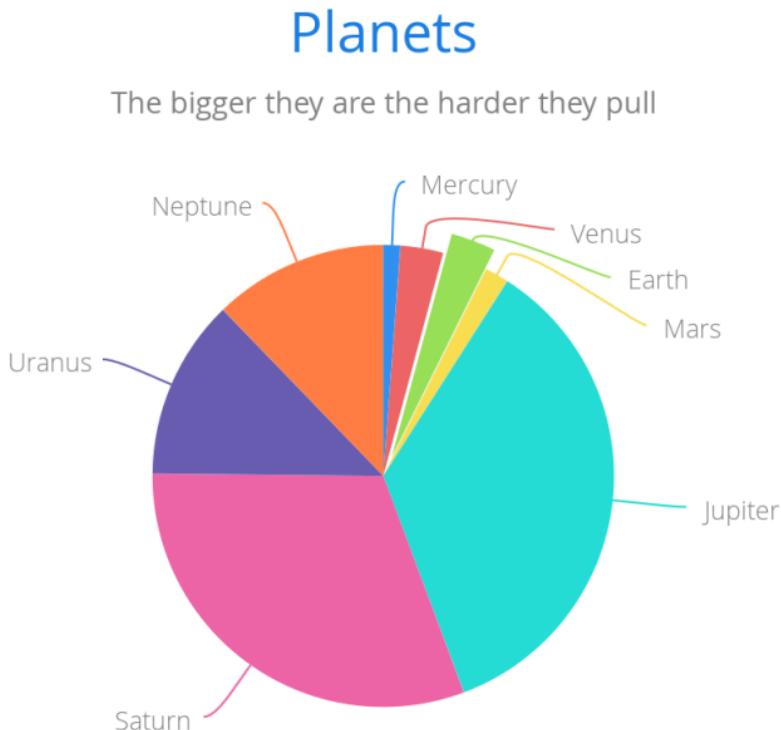


Figure 18.12. Pie Chart

Plot Options

The chart-specific options of a pie chart are configured with a **PlotOptionsPie**.

```
PlotOptionsPie options = new PlotOptionsPie();
options.setInnerSize("0");
options.setSize("75%"); // Default
options.setCenter("50%", "50%"); // Default
conf.setPlotOptions(options);
```

innerSize

A pie with inner size greater than zero is a "donut". The inner size can be expressed either as number of pixels or as a relative percentage of the chart area with a string (such as "60%") See the section later on donuts.

size

The size of the pie can be expressed either as number of pixels or as a relative percentage of the chart area with a string (such as "80%"). The default size is 75%, to leave space for the labels.

center

The X and Y coordinates of the center of the pie can be expressed either as numbers of pixels or as a relative percentage of the chart sizes with a string. The default is "50%", "50%".

Data Model

The labels for the pie sectors are determined from the labels of the data points. The **DataSet**s or **ContainerDataSet**s, which allow labeling the data points, should be used for pie charts.

```
DataSet series = new DataSet();
series.add(new DataSetItem("Mercury", 4900));
series.add(new DataSetItem("Venus", 12100));
...
conf.addSeries(series);
```

If a data point, as defined as a **DataSetItem** in a **DataSet**, has the *sliced* property enabled, it is shown as slightly cut away from the pie.

```
// Slice one sector out
DataSetItem earth = new DataSetItem("Earth", 12800);
earth.setSliced(true);
series.add(earth);
```

Donut Charts

Setting the *innerSize* of the plot options of a pie chart to a larger than zero value results in an empty hole at the center of the pie.

```
PlotOptionsPie options = new PlotOptionsPie();
options.setInnerSize("60%");
conf.setPlotOptions(options);
```

As you can set the plot options also for each data series, you can put two pie charts on top of each other, with a smaller one fitted in the "hole" of the donut. This way, you can make pie charts with more details on the outer rim, as done in the example below:

```
// The inner pie
DataSeries innerSeries = new DataSeries();
innerSeries.setName("Browsers");
PlotOptionsPie innerPieOptions = new PlotOptionsPie();
innerPieOptions.setSize("60%");
innerSeries.setPlotOptions(innerPieOptions);
...
DataSeries outerSeries = new DataSeries();
outerSeries.setName("Versions");
PlotOptionsPie outerSeriesOptions = new PlotOptionsPie();
outerSeriesOptions.setInnerSize("60%");
outerSeries.setPlotOptions(outerSeriesOptions);
...
```

The result is illustrated in Figure 18.13. "Overlaid Pie and Donut Chart".

Browser market share, April, 2011

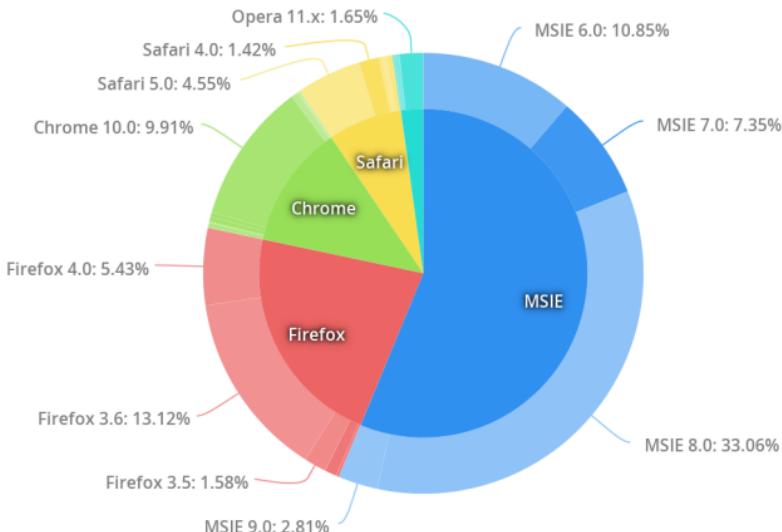


Figure 18.13. Overlaid Pie and Donut Chart

18.4.9. Gauges

A gauge is an one-dimensional chart with a circular Y-axis, where a rotating pointer points to a value on the axis. A gauge can, in fact, have multiple Y-axes to display multiple scales.

A *solid gauge* is otherwise like a regular gauge, except that a solid color arc is used to indicate current value instead of a pointer. The color of the indicator arc can be configured to change according to color stops.

Let us consider the following gauge:

```
Chart chart = new Chart(ChartType.GAUGE);
chart.setWidth("400px");
chart.setHeight("400px");
```

After the settings done in the subsequent sections, it will show as in Figure 18.14, "A Gauge".

Speedometer



Figure 18.14. A Gauge

Gauge Configuration

The start and end angles of the gauge can be configured in the **Pane** object of the chart configuration. The angles can be given as -360 to 360 degrees, with 0 at the top of the circle.

```
Configuration conf = chart.getConfiguration();
conf.setTitle("Speedometer");
conf.getPane().setStartAngle(-135);
conf.getPane().setEndAngle(135);
```

Axis Configuration

A gauge has only an Y-axis. You need to provide both a minimum and maximum value for it.

```

YAxis yaxis = new YAxis();
yaxis.setTitle("km/h");

// The limits are mandatory
yaxis.setMin(0);
yaxis.setMax(100);

// Other configuration
yaxis.getLabels().setStep(1);
yaxis.setTickInterval(10);
yaxis.setTickLength(10);
yaxis.setTickWidth(1);
yaxis.setMinorTickInterval(1);
yaxis.setMinorTickLength(5);
yaxis.setMinorTickWidth(1);
yaxis.setPlotBands(new PlotBand[]{  

    new PlotBand(0, 60, SolidColor.GREEN),  

    new PlotBand(60, 80, SolidColor.YELLOW),  

    new PlotBand(80, 100, SolidColor.RED)});  

yaxis.setGridLineWidth(0); // Disable grid  

conf.addYAxis(yaxis);

```

You can do all kinds of other configuration to the axis - please see the API documentation for all the available parameters.

Setting and Updating Gauge Data

A gauge only displays a single value, which you can define as a data series of length one, such as follows:

```

ListSeries series = new ListSeries("Speed", 80);
conf.addSeries(series);

```

Gauges are especially meaningful for displaying changing values. You can use the updatePoint() method in the data series to update the single value.

```

final TextField tf = new TextField("Enter a new value");
layout.addComponent(tf);

Button update = new Button("Update", new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        Integer newValue = new Integer((String)tf.getValue());
        series.updatePoint(0, newValue);
    }
});
layout.addComponent(update);

```

18.4.10. Solid Gauges

A solid gauge is much like a regular gauge described previously; a one-dimensional chart with a circular Y-axis. However, instead of a rotating pointer, the value is indicated by a rotating arc with solid color. The color of the indicator arc can be configured to change according to the value using color stops.

Let us consider the following gauge:

```
Chart chart = new Chart(ChartType.SOLIDGAUGE);
chart.setWidth("400px");
chart.setHeight("400px");
```

After the settings done in the subsequent sections, it will show as in Figure 18.15, "A Solid Gauge".

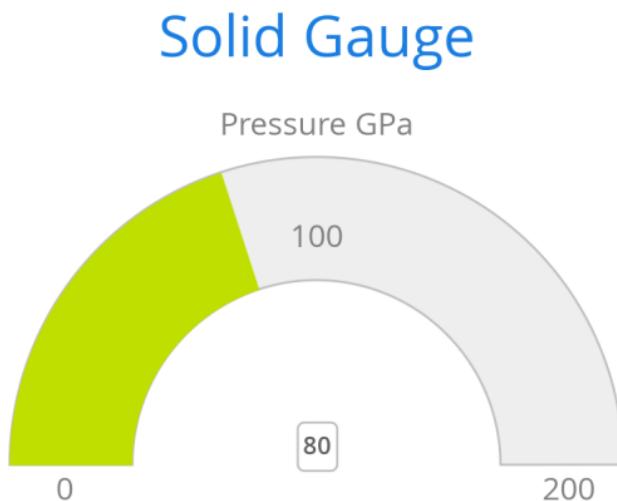


Figure 18.15. A Solid Gauge

While solid gauge is much like a regular gauge, the configuration differs

Configuration

The solid gauge must be configured in the drawing **Pane** of the chart configuration. The gauge arc spans an angle, which is specified as -360 to 360 degrees, with 0 degrees at the top of the arc. Typically, a semi-arc is used, where you use -90 and

90 for the angles, and move the center lower than you would have with a full circle. You can also adjust the size of the gauge pane: enlargening it allows positioning tick labels better.

```
Configuration conf = chart.getConfiguration();
conf.setTitle("Solid Gauge");

Pane pane = conf.getPane();
pane.setSize("125%");           // For positioning tick labels
pane.setCenter("50%", "70%");   // Move center lower
pane.setStartAngle(-90);        // Make semi-circle
pane.setEndAngle(90);          // Make semi-circle
```

The shape of the gauge display is defined as the background of the pane. You at least need to set the shape as either "arc" or "solid". You typically also want to set background color and inner and outer radius.

```
Background bkg = new Background();
bkg.setBackgroundColor(new SolidColor("#eeeeee")); // Gray
bkg.setInnerRadius("60%"); // To make it an arc and not circle
bkg.setOuterRadius("100%"); // Default - not necessary
bkg.setShape("arc");      // solid or arc
pane.setBackground(bkg);
```

Axis Configuration

A gauge only has an Y-axis. You must define the value range (*min* and *max*).

```
YAxis yaxis = new YAxis();
yaxis.setTitle("Pressure GPa");
yaxis.getTitle().setY(-80); // Move 70 px upwards from center

// The limits are mandatory
yaxis.setMin(0);
yaxis.setMax(200);

// Configure ticks and labels
yaxis.setTickInterval(100); // At 0, 100, and 200
yaxis.getLabels().setY(-16); // Move 16 px upwards
yaxis.setGridLineWidth(0); // Disable grid
```

You can configure color stops for the indicator arc. The stops are defined with **Stop** objects having stop points from 0.0 to 1.0 and color values.

```
yaxis.setStops(new Stop(0.1f, SolidColor.GREEN),
               new Stop(0.5f, SolidColor.YELLOW),
               new Stop(0.9f, SolidColor.RED));
```

```
conf.addyAxis(yaxis);
```

Setting `yaxis.getLabels().setRotationPerpendicular()` makes gauge labels rotate perpendicular to the center.

You can do all kinds of other configuration to the axis - please see the API documentation for all the available parameters.

Plot Options

Solid gauges do not currently have any chart type specific plot options. See Section 18.5.1, "Plot Options" for common options.

```
PlotOptionsSolidgauge options = new PlotOptionsSolidgauge();
```

// Move the value display box at the center a bit higher

```
Labels dataLabels = new Labels();
dataLabels.setY(-20);
options.setDataLabels(dataLabels);

conf.setPlotOptions(options);
```

Setting and Updating Gauge Data

A gauge only displays a single value, which you can define as a data series of length one, such as as follows:

```
ListSeries series = new ListSeries("Pressure MPa", 80);
conf.addSeries(series);
```

Gauges are especially meaningful for displaying changing values. You can use the `updatePoint()` method in the data series to update the single value.

```
final TextField tf = new TextField("Enter a new value");
layout.addComponent(tf);

Button update = new Button("Update").new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        Integer newValue = new Integer((String)tf.getValue());
        series.updatePoint(0, newValue);
    }
});
layout.addComponent(update);
```

18.4.11. Area and Column Range Charts

Ranged charts display an area or column between a minimum and maximum value, instead of a singular data point. They require the use of **RangeSeries**, as described in Section 18.6.3, “Range Series”. An area range is created with *AREARANGE* chart type, and a column range with *COLUMNRANGE* chart type.

Consider the following example:

```
Chart chart = new Chart(ChartType.AREARANGE);
chart.setWidth("400px");
chart.setHeight("300px");

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Extreme Temperature Range in Finland");
...

// Create the range series
// Source: http://ilmatieteenlaitos.fi/lampotilaennatyksia
RangeSeries series = new RangeSeries("Temperature Extremes",
    new Double[]{-51.5,10.9},
    new Double[]{-49.0,11.8},
    ...
    new Double[]{-47.0,10.8}); // ...
conf.addSeries(series);
```

The resulting chart, as well as the same chart with a column range, is shown in Figure 18.16, “Area and Column Range Chart”.

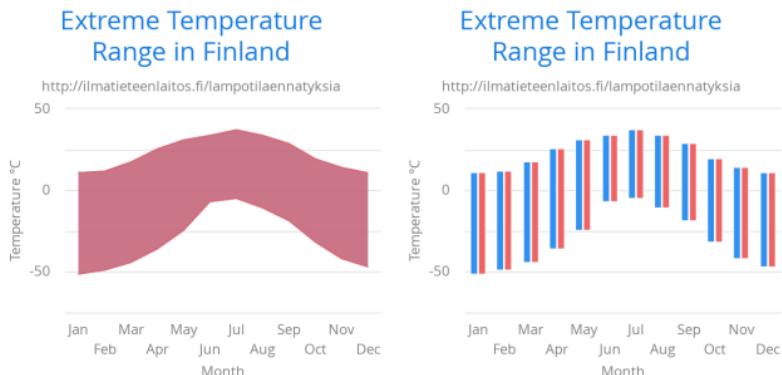


Figure 18.16. Area and Column Range Chart

18.4.12. Polar, Wind Rose, and Spiderweb Charts

Most chart types having two axes can be displayed in *polar* coordinates, where the X axis is curved on a circle and Y axis from the center of the circle to its rim. Polar chart is not a chart type in itself, but can be enabled for most chart types with `setPolar(true)` in the chart model parameters. Therefore all chart type specific features are usable with polar charts.

Vaadin Charts allows many sorts of typical polar chart types, such as *wind rose*, a polar column graph, or *spiderweb*, a polar chart with categorical data and a more polygonal visual style.

// Create a chart of some type

```
Chart chart = new Chart(ChartType.LINE);
```

// Enable the polar projection

```
Configuration conf = chart.getConfiguration();
conf.getChart().setPolar(true);
```

You need to define the sector of the polar projection with a **Pane** object in the configuration. The sector is defined as degrees from the north direction. You also need to define the value range for the X axis with `setMin()` and `setMax()`.

// Define the sector of the polar projection

```
Pane pane = new Pane(0, 360); // Full circle
conf.addPane(pane);
```

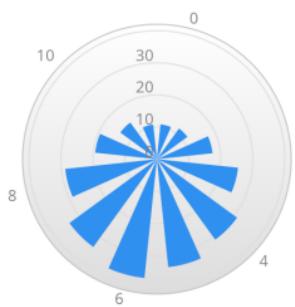
// Define the X axis and set its value range

```
XAxis axis = new XAxis();
axis.setMin(0);
axis.setMax(360);
```

The polar and spiderweb charts are illustrated in Figure 18.17, "Wind Rose and Spiderweb Charts".

Extreme Temperature Range in Finland

<http://ilmatieteenlaitos.fi/lampotilaennatyksia>



Extreme Temperature Range in Finland

<http://ilmatieteenlaitos.fi/lampotilaennatyksia>

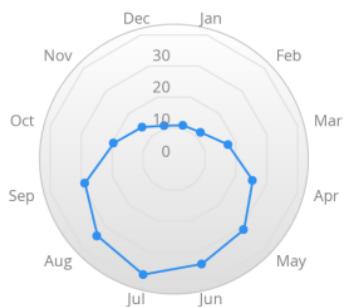


Figure 18.17. Wind Rose and Spiderweb Charts

Spiderweb Charts

A *spiderweb* chart is a commonly used visual style of a polar chart with a polygonal shape rather than a circle. The data and the X axis should be categorical to make the polygonal interpolation meaningful. The sector is assumed to be full circle, so no angles for the pane need to be specified.

18.4.13. Funnel and Pyramid Charts

Funnel and pyramid charts are typically used to visualize stages in a sales processes, and for other purposes to visualize subsets of diminishing size. A funnel or pyramid chart has layers much like a stacked column: in funnel from top-to-bottom and in pyramid from bottom-to-top. The top of the funnel has width of the drawing area of the chart, and diminishes in size down to a funnel "neck" that continues as a column to the bottom. A pyramid diminishes from bottom to top and does not have a neck.

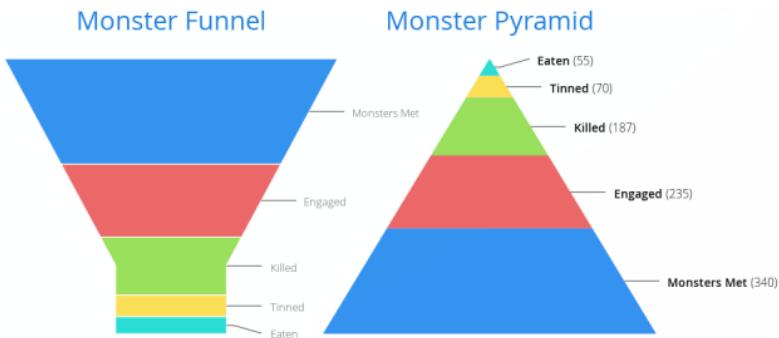


Figure 18.18. Funnel and Pyramid Charts

Funnels have chart type *FUNNEL*, pyramids have *PYRAMID*.

The labels of the funnel blocks are by default placed on the right side of the blocks, together with a connector. You can configure their style in the plot options.

18.4.14. Waterfall Charts

Waterfall charts are used for visualizing level changes from an initial level to a final level through a number of changes in the level. The changes are given as delta values, and you can have a number of intermediate totals, which are calculated automatically.

Changes in Reindeer Population in 2011

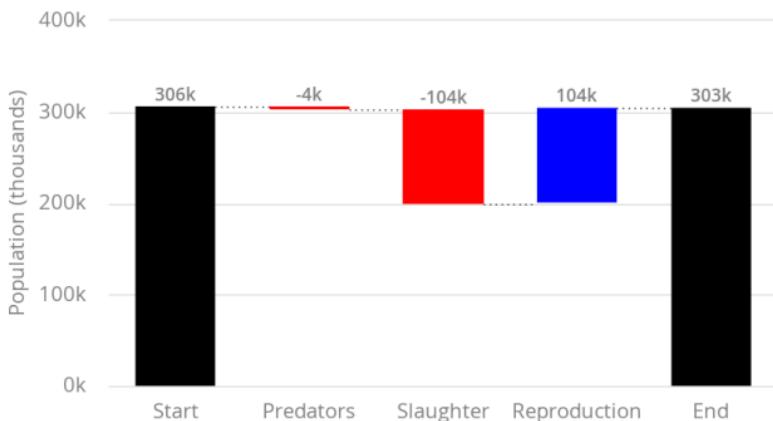


Figure 18.19. Waterfall Charts

Waterfall charts have chart type WATERFALL.

18.4.15. Heat Maps

A heat map is a two-dimensional grid, where the color of a grid cell indicates a value.

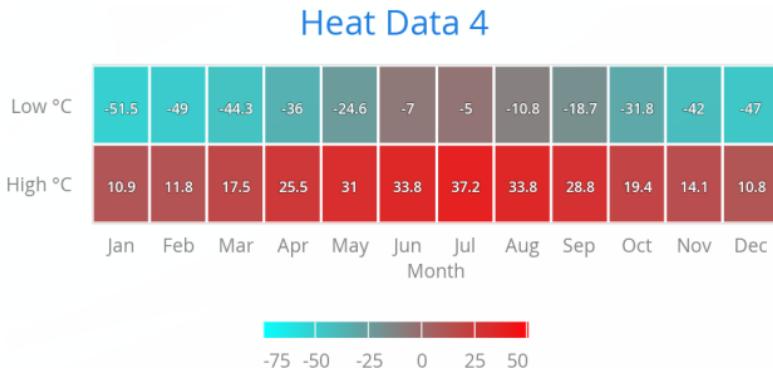


Figure 18.20. Heat Maps

Heat maps have chart type HEATMAP.

18.4.16. Tree Maps

A tree map is used to display hierarchical data. It consists of a group of rectangles that contains other rectangles, where the size of a rectangle indicates the item value.

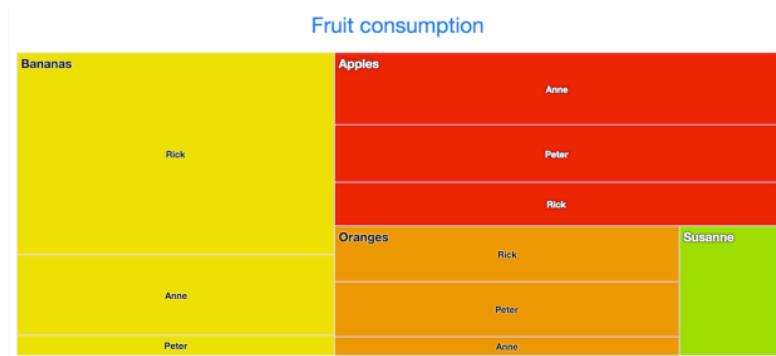


Figure 18.21. Tree Maps

Tree maps have chart type TREEMAP.

18.4.17. Polygons

A polygon can be used to draw any freeform filled or stroked shape in the Cartesian plane.

Polygons consist of connected data points. The **DataSeries** or **ContainerSeries** are usually the most meaningful data series types for polygon charts. In both cases, the *x* and *y* properties should be set.

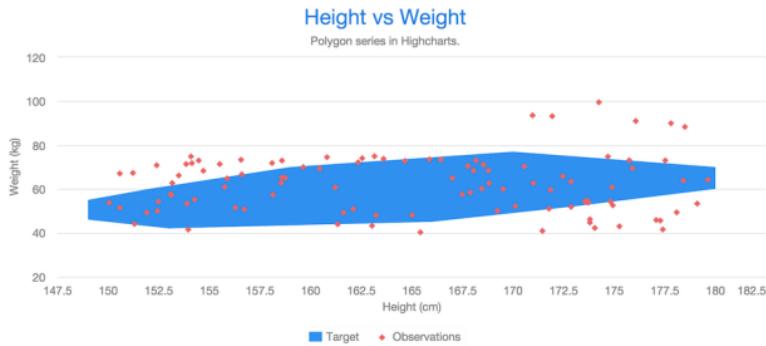


Figure 18.22. Polygon combined with Scatter

Polygons have chart type POLYGON.

18.4.18. Flags

Flags is a special chart type for annotating a series or the X axis with callout labels. Flags indicate interesting points or events on the series or axis. The flags are defined as items in a data series separate from the annotated series or axis.

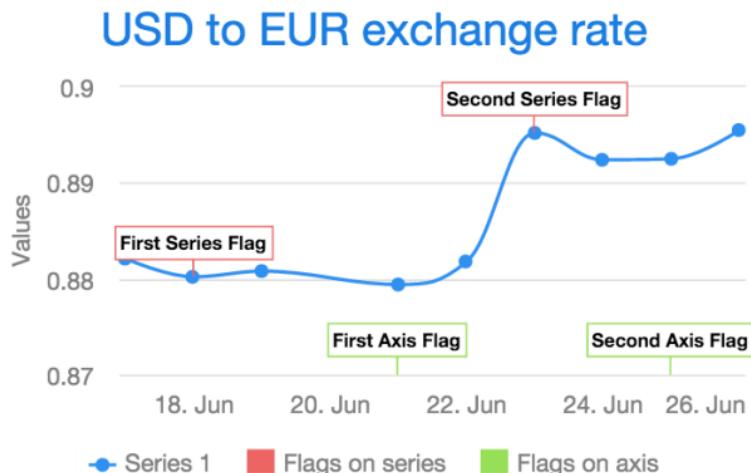


Figure 18.23. Flags placed on an axis and a series

Flags are normally used in a chart that has one or more normal data series.

Plot Options

The flags are defined in a series that has its chart type specified by setting its plot options as **PlotOptionsFlags**. In addition to the common plot options properties, flag charts also have the following properties:

shape

defines the shape of the marker. It can be one of FLAG, CIRCLEPIN, SQUAREPIN, or CALLOUT.

stackDistance

defines the vertical offset between flags on the same value in the same series. Defaults to 12.

onSeries

defines the ID of the series where the flags should be drawn on. If no ID is given, the flags are drawn on the X axis.

onKey

in chart types that have multiple keys (Y values) for a data point, the property defines on which key the flag is placed. Line and column series have only one key, y. In range, OHLC, and candlestick series, the flag can be placed on the open, high, low, or close key. Defaults to y.

Data

The data for flags series require x and title properties, but can also have text property indicating the tooltip text. The easiest way to set these properties is to use **FlagItem**.

18.4.19. OHLC and Candlestick Charts

An Open-High-Low-Close (OHLC) chart displays the change in price over a period of time. The OHLC charts have chart type OHLC. An OHLC chart consists of vertical lines, each having a horizontal tickmark both on the left and the right side. The top and bottom ends of the vertical line indicate the highest and lowest prices during the time period. The tickmark on

the left side of the vertical line shows the opening price and the tickmark on the right side the closing price.

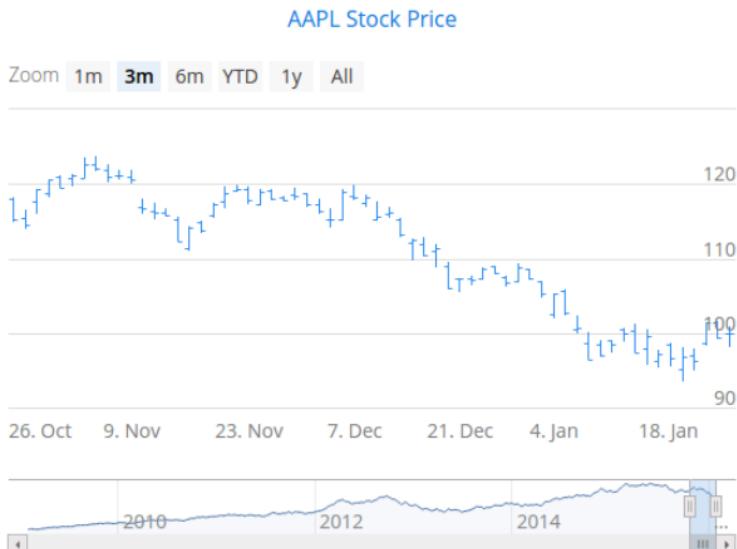


Figure 18.24. OHLC Chart.

A candlestick chart is another way to visualize OHLC data. A candlestick has a body and two vertical lines, called *wicks*. The body represents the opening and closing prices. If the body is filled, the top edge of the body shows the opening price and the bottom edge shows the closing price. If the body is unfilled, the top edge shows the closing price and the bottom edge the opening price. In other words, if the body is filled, the opening price is higher than the closing price, and if not, lower. The upper wick represents the highest price during the time period and the lower wick represents the lowest price. A candlestick chart has chart type CANDLESTICK.



Figure 18.25. Candlestick Chart.

To attach data to an OHLC or a candlestick chart, you need to use a **DataSeries** or a **ContainerSeries**. See Section 18.6, "Chart Data" for more details. A data series for an OHLC chart must contain **OhlcItem** objects. An **OhlcItem** contains a date and the open, highest, lowest, and close price on that date.

```
Chart chart = new Chart(ChartType.OHLC);
chart.setTimeline(true);

Configuration configuration = chart.getConfiguration();
configuration.setTitle().setText("AAPL Stock Price");
DataSeries dataSeries = new DataSeries();
for (StockPrices.OhlcData data : StockPrices.fetchAaplOhlcPrice()) {
    OhlcItem item = new OhlcItem();
    item.setX(data.getDate());
    item.setLow(data.getLow());
    item.setHigh(data.getHigh());
    item.setClose(data.getClose());
    item.setOpen(data.getOpen());
    dataSeries.add(item);
}
configuration.setSeries(dataSeries);
chart.drawChart(configuration);
```

When using **DataProviderSeries**, you need to specify the functions used for retrieving OHLC properties: `setX()`, `setOpen()`, `setHigh()` `setLow()`, and `setClose()`.

```

Chart chart = new Chart(ChartType.OHLC);
Configuration configuration = chart.getConfiguration();

// Create a DataProvider filled with stock price data
DataProvider<OhlcData> dataProvider = initDataProvider();
// Wrap the container in a data series
DataProviderSeries<OhlcData> dataSeries = new DataProviderSeries<>(dataProvider);
dataSeries.setX(OhlcData::getDate);
dataSeries.setLow(OhlcData::getLow);
dataSeries.setHigh(OhlcData::getHigh);
dataSeries.setClose(OhlcData::getClose);
dataSeries.setOpen(OhlcData::getOpen);

PlotOptionsOhlc plotOptionsOhlc = new PlotOptionsOhlc();
plotOptionsOhlc.setTurboThreshold(0);
dataSeries.setPlotOptions(plotOptionsOhlc);

configuration.setSeries(dataSeries);

```

Typically the OHLC and candlestick charts contain a lot of data, so it is useful to use them with the timeline feature enabled. The timeline feature is described in Section 18.8, “Timeline”.

Plot Options

You can use a **DataGrouping** object to configure data grouping properties. You set it in the plot options with `setDataGrouping()`. If the data points in a series are so dense that the spacing between two or more points is less than value of the `groupPixelWidth` property in the **DataGrouping**, the points will be grouped into appropriate groups so that each group is more or less two pixels wide. The approximation property in **DataGrouping** specifies which data point value should represent the group. The possible values are: average, open, high, low, close, and sum.

Using `setUpColor()` and `setUpLineColor()` allow setting the fill and border colors of the candlestick that indicate rise in the values. The default colors are white.

18.5. Chart Configuration

All the chart content configuration of charts is defined in a *chart model* in a **Configuration** object. You can access the model with the `getConfiguration()` method.

The configuration properties in the **Configuration** class are summarized in the following:

- credits: **Credits** (text, position, href, enabled)

- labels: **HTMLLabels** (html, style)
- lang: **Lang** (decimalPoint, thousandsSep, loading)
- legend: **Legend** (see Section 18.5.3, “Legend”)
- pane: **Pane**
- plotoptions: **PlotOptions** (see Section 18.5.1, “Plot Options”)
- series: Series
- subTitle: **SubTitle**
- title: **Title**
- tooltip: **Tooltip**
- xAxis: **XAxis** (see Section 18.5.2, “Axes”)
- yAxis: **YAxis** (see Section 18.5.2, “Axes”)

For data configuration, see Section 18.6, “Chart Data”.

18.5.1. Plot Options

The plot options are used to configure the data series in the chart. For example, line color could be specified for each line series. Plot options can be set in the configuration of the entire chart or for each data series separately with `setPlotOptions()`. When the plot options are set to the entire chart, it will be applied to all the series in the chart.

For example, the following enables stacking in column charts:

```
Chart chart = new Chart();
Configuration configuration = chart.getConfiguration();
PlotOptionsColumn plotOptions = new PlotOptionsColumn();
plotOptions.setStacking(Stacking.NORMAL);
configuration.setPlotOptions(plotOptions);
```

Chart can contain multiple plot options which can be added dynamically with `addPlotOptions()`.

The developer can specify also the plot options for the particular data series as follows:

```
ListSeries series = new ListSeries(50, 60, 70, 80);
PlotOptionsColumn plotOptions = new PlotOptionsColumn();
plotOptions.setStacking(Stacking.NORMAL);
series.setPlotOptions(plotOptions);
```

The plot options are defined in type-specific options classes or in a **PlotOptionsSeries** class which contains general options for all series types. Type specific classes are applied to all the series with the same type in the chart. If **PlotOptionsSeries** is used, it will be applied to all the series in the chart regardless of the type.

Chart types are divided into several groups with common properties. These groups are presented as abstract classes, that allow to use polymorphism for setting common properties for specific implementations. The abstract classes and groups are the following:

- **AreaOptions** ⊗ **PlotOptionsArea**, **PlotOptionsArearange**,
PlotOptionsAreaspline, **PlotOptionsAreasplinerange**
- **ColumnOptions** ⊗ **PlotOptionsBar**, **PlotOptionsColumn**,
PlotOptionsColumnrange
- **GaugeOptions** ⊗ **PlotOptionsGauge**, **PlotOptionsSolidgauge**
- **PointOptions** ⊗ **PlotOptionsLine**, **PlotOptionsSpline**,
PlotOptionsScatter
- **PyramidOptions** ⊗ **PlotOptionsPyramid**, **PlotOptionsFunnel**
- **OhlcOptions** ⊗ **PlotOptionsOhlc**, **PlotOptionsCandlestick**

For example, to set the same lineWidth for **PlotOptionsLine** and **PlotOptionsSpline** use **PointOptions**.

```
private void setCommonProperties(PointOptions options) {
    options.setLineWidth(5);
    options.setColor(SolidColor.RED);
    options.setAnimation(false);
}

...
PlotOptionsSpline lineOptions = new PlotOptionsSpline();
```

```
PlotOptionsLine splineOptions = new PlotOptionsLine();
setCommonProperties(lineOptions);
setCommonProperties(splineOptions);
configuration.setPlotOptions(lineOptions, splineOptions);
```

See the API documentation of each chart type and its plot options class for more information about the chart-specific options.

Other Options

The following options are supported by some chart types.

width

Defines the width of the chart either by pixels or as a percentual proportion of the drawing area.

height

Defines the height of the chart either by pixels or as a percentual proportion of the drawing area.

depth

Specifies the thickness of the chart in 3D mode.

allowPointSelect

Specifies whether data points, in whatever way they are visualized in the particular chart type, can be selected by clicking on them. Defaults to *false*.

borderColor

Defines the border color of the chart elements.

borderWidth

Defines the width of the border in pixels.

center

Defines the center of the chart within the chart area by left and top coordinates, which can be specified either as pixels or as a percentage (as string) of the drawing area. The default is top 50% and left 50%.

slicedOffset

In chart types that support slices, such as pie and pyramid charts, specifies the offset for how far a slice is

detached from other items. The amount is given in pixels and defaults to 10 pixels.

visible

Specifies whether or not a chart is visible. Defaults to true.

18.5.2. Axes

Different chart types may have one, two, or three axes; in addition to X and Y axes, some chart types may have a color axis. These are represented by **XAxis**, **YAxis**, and **ColorAxis**, respectively. The X axis is usually horizontal, representing the iteration over the data series, and Y vertical, representing the values in the data series. Some chart types invert the axes and they can be explicitly inverted with `getChart().setInverted()` in the chart configuration. An axis has a caption and tick marks at intervals indicating either numeric values or symbolic categories. Some chart types, such as gauge, have only Y-axis, which is circular in the gauge, and some such as a pie chart have none.

The basic elements of X and Y axes are illustrated in Figure 18.26, "Chart Axis Elements".

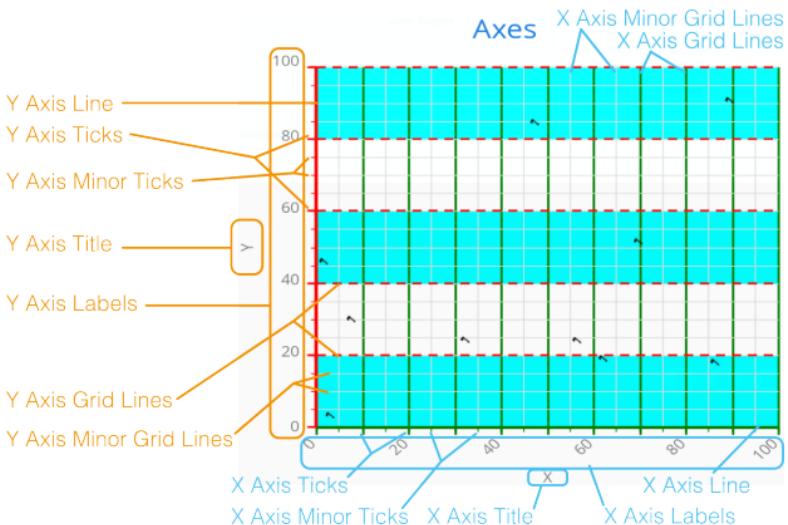


Figure 18.26. Chart Axis Elements

Axis objects are created and added to the configuration object with addXAxis() and addYAxis().

```
XAxis xaxis = new XAxis();
xaxis.setTitle("Axis title");
conf.addXAxis(xaxis);
```

A chart can have more than one Y-axis, usually when different series displayed in a graph have different units or scales. The association of a data series with an axis is done in the data series object with setYAxis().

For a complete reference of the many configuration parameters for the axes, please refer to the JavaDoc API documentation of Vaadin Charts.

Axis Type

Axes can be one of the following types, which you can set with setType(). The axis types are enumerated under **AxisType**. *LINEAR* is the default.

LINEAR(default)

For numeric values in linear scale.

LOGARITHMIC

For numerical values, as in the linear axis, but the axis will be scaled in the logarithmic scale. The minimum for the axis *must* be a positive non-zero value (log(0) is not defined, as it has limit at negative infinity when the parameter approaches zero).

DATETIME

Enables date/time mode in the axis. The date/time values are expected to be given either as a **Date** object or in milliseconds since the Java (or Unix) date epoch on January 1st 1970 at 00:00:00 GMT. You can get the millisecond representation of Java **Date** with getTime().

CATEGORY

Enables using categorical data for the axis, as described in more detail later. With this axis type, the category labels are determined from the labels of the data points

in the data series, without need to set them explicitly with `setCategories()`.

Categories

The axes display, in most chart types, tick marks and labels at some numeric interval by default. If the items in a data series have a symbolic meaning rather than numeric, you can associate *categories* with the data items. The category label is displayed between two axis tick marks and aligned with the data point. In certain charts, such as column chart, where the corresponding values in different data series are grouped under the same category. You can set the category labels with `setCategories()`, which takes the categories as (an ellipsis) parameter list, or as an iterable. The list should match the items in the data series.

```
XAxis xaxis = new XAxis();
xaxis.setCategories("Mercury", "Venus", "Earth",
    "Mars", "Jupiter", "Saturn",
    "Uranus", "Neptune");
```

You can only set the category labels from the data point labels by setting the axis type to `CATEGORY`, as described earlier.

Labels

The axes display, in most chart types, tick marks and labels at some numeric interval by default. The format and style of labels in an axis is defined in a **Labels** object, which you can get with `getLabels()` from the axis.

```
XAxis xaxis = new XAxis();
...
Labels xlabel = xaxis.getLabels();
xlabel.setAlign(HorizontalAlign.CENTER); //Default
xlabel.getStyle().setColor(SolidColor.GREEN);
xlabel.getStyle().setFontWeight(FontWeight.BOLD);
xlabel.setRotation(-45);
xlabel.setStep(2); //Every 2 major tick
```

Axis labels have the following configuration properties:

align

Defines the alignment of the labels relative to the centers of the ticks. On left alignment, the left edges of labels are aligned at the tickmarks, and correspondingly the right side on right alignment. The default is determined automatically based on the direction of the axis and rotation of the labels.

distance(only in polar charts)

Distance of labels from the perimeter of the plot area, in pixels.

enabled

Whether labels are enabled or not. Defaults to *true*.

format

Formatting string for labels, as described in Section 18.5.4, "Formatting Labels". Defaults to "`{value}`".

formatter

A JavaScript formatter for the labels, as described in Section 18.5.4, "Formatting Labels". The value is available in the `this.value` property. The `this` object also has `axis`, `chart`, `isFirst`, and `isLast` properties. Defaults to:

```
function() {return this.value;}
```

maxStaggerLines(only horizontal axis)

When labels on the horizontal (usually X) axis are displayed so densely that they would overlap, they are automatically placed on alternating lines in "staggered" fashion. When number of lines is not set manually with `staggerLines`, this parameter defines the maximum number of such lines: value 1 disables automatic staggering. Default is 5 lines.

rotation

Defines rotation of labels in degrees. A positive value indicates rotation in clockwise direction. Labels are rotated at their alignment point. Defaults to 0.

```
Labels xlabels = xaxis.getLabels();
xlabels.setAlign(HorizontalAlign.RIGHT);
xlabels.setRotation(-45); // Tilt 45 degrees CCW
```

staggerLines

Defines number of lines for placing the labels to avoid overlapping. By default undefined, and the number of lines is automatically determined up to *maxStaggerLines*.

step

Defines tick interval for showing labels, so that labels are shown at every *n*th tick. The default step is automatically determined, along with staggering, to avoid overlap.

```
Labels xlabels = xaxis.getLabels();
xlabels.setStep(2); // Every 2 major tick
```

style

Defines style for labels. The property is a **Style** object, which has to be created and set.

```
Labels xlabels = xaxis.getLabels();
Style xlabelstyle = new Style();
xlabelstyle.setColor(SolidColor.GREEN);
xlabels.setStyle(xlabelstyle);
```

useHTML

Allows using HTML in custom label formats. Otherwise, HTML is quoted. Defaults to false.

x,y

Offsets for the label's position, relative to the tick position. X offset defaults to 0, but Y to null, which enables automatic positioning based on font size.

Gauge, pie, and polar charts allow additional properties.

For a complete reference of the many configuration parameters for the labels, please refer to the JavaDoc API documentation of Vaadin Charts.

Axis Range

The axis range is normally set automatically to fit the data, but can also be set explicitly. The *extremes* property in the axis configuration defines the minimum and maximum values of the axis range. You can set them either individually with *setMin()* and *setMax()*, or together with *setExtremes()*. Changing

the extremes programmatically requires redrawing the chart with `drawChart()`.

18.5.3. Legend

The legend is a box that describes the data series shown in the chart. It is enabled by default and is automatically populated with the names of the data series as defined in the series objects, and the corresponding color symbol of the series.

alignment

Specifies the horizontal alignment of the legend box within the chart area. Defaults to `HorizontalAlign.CENTER`.

enabled

Enables or disables the legend. Defaults to true.

layout

Specifies the layout direction of the legend items. Defaults to `LayoutDirection.HORIZONTAL`.

title

Specifies the title of the legend.

verticalAlign

Specifies the vertical alignment of the legend box within the chart area. Defaults to `VerticalAlign.BOTTOM`.

```
Legend legend = configuration.getLegend();
legend.getTitle().setText("City");
legend.setLayout(LayoutDirection.VERTICAL);
legend.setAlign(HorizontalAlign.LEFT);
legend.setVerticalAlign(VerticalAlign.TOP);
```

The result can be seen in Figure 18.27, "Legend example".

Monthly Average Temperature

Source: WorldClimate.com

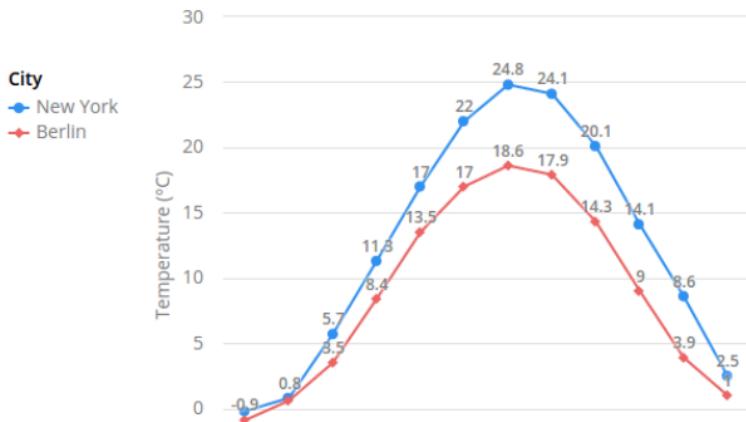


Figure 18.27. Legend example

18.5.4. Formatting Labels

Data point values, tooltips, and tick labels are formatted according to formatting configuration for the elements, with configuration properties described earlier for each element. Formatting can be set up in the overall configuration, for a data series, or for individual data points. The format can be defined either by a format string or by JavaScript formatter, which are described in the following.

Using Format Strings

A formatting string contain free-form text mixed with variables. Variables are enclosed in brackets, such as "Here {point.y} is a value at {point.x}". In different contexts, you have at least the following variables available:

- *value* in axis labels
- *point.x*, *pointx* in data points and tooltips
- *series.name* in data points and tooltips
- *series.color* in data points and tooltips

Values can be formatted according to a formatting string, separated from the variable name by a colon.

For numeric values, a subset of C printf formatting specifiers is supported. For example, "`{point.y:%02.2f}`" would display a floating-point value with two decimals and two leading zeroes, such as 02.30.

For dates, you can use a subset of PHP strftime() formatting specifiers. For example, "`{value:%Y-%m-%d %H:%M:%S}`" would format a date and time in the ISO 8601 format.

Using a JavaScript Formatter

A JavaScript formatter is given in a string that defines a JavaScript function that returns the formatted string. The value to be formatted is available in `this.value` for axis labels, or `this.x`, `this.y` for data points.

For example, to format tick labels on a chart axis, you could have:

```
YAxis yaxis = new YAxis();
Labels ylabels = yaxis.getLabels();
ylabels.setFormatter("function() {return this.value + ' km';}");
```

Simplified Formatting

Some contexts that display labels allow defining simple formatting for the labels. For example, data point tooltips allow defining prefix, suffix, and floating-point precision for the values.

18.6. Chart Data

Chart data is stored in a data series model that contains information about the visual representation of the data points in addition to their values. There are a number of different types of series - **DataSeries**, **ListSeries**, **AreaListSeries**, and **RangeSeries**.

18.6.1. List Series

The **ListSeries** is essentially a helper type that makes the handling of simple sequential data easier than with **DataSer-**

ies. The data points are assumed to be at a constant interval on the X axis, starting from the value specified with the pointStart property (default is 0) at intervals specified with the pointInterval property (default is 1.0). The two properties are defined in the **PlotOptions** for the series.

The Y axis values are given as constructor parameters or using the setData() method.

```
ListSeries series = new ListSeries(  
    "Total Reindeer Population",  
    181091, 201485, 188105);  
PlotOptionsLine plotOptions = new PlotOptionsLine();  
plotOptions.setPointStart(1959);  
series.setPlotOptions(plotOptions);  
conf.addSeries(series);
```

You can also add them one by one with the addData() method.

If the chart has multiple Y axes, you can specify the axis for the series by its index number using setyAxis().

18.6.2. Generic Data Series

The **DataSet**s can represent a sequence of data points at an interval as well as scatter data. Data points are represented with the **DataSetItem** class, which has *x* and *y* properties for representing the data value. Each item can be given a category name.

```
DataSet series = new DataSet();  
series.setName("Total Reindeer Population");  
series.add(new DataSetItem(1959, 181091));  
series.add(new DataSetItem(1960, 201485));  
series.add(new DataSetItem(1961, 188105));  
series.add(new DataSetItem(1962, 177206));  
  
// Modify the color of one point  
series.get(2).getMarker().setFillColor(SolidColor.RED);  
conf.addSeries(series);
```

Data points are associated with some visual representation parameters: marker style, selected state, legend index, and dial style (for gauges). Most of them can be configured at the level of individual data series items, the series, or in the overall plot options for the chart. The configuration options are de-

scribed in Section 18.5, “Chart Configuration”. Some parameters, such as the sliced option for pie charts is only meaningful to configure at item level.

Adding and Removing Data Items

New **DataSeriesItem** items are added to a series with the `add()` method. The basic method takes just the data item, but the other method takes also two boolean parameters. If the `updateChart` parameter is false, the chart is not updated immediately. This is useful if you are adding many points in the same request.

The `shift` parameter, when true, causes removal of the first data point in the series in an optimized manner, thereby allowing an animated chart that moves to left as new points are added. This is most meaningful with data with even intervals.

You can remove data points with the `remove()` method in the series. Removal is generally not animated, unless a data point is added in the same change, as is caused by the `shift` parameter for the `add()`.

Updating Data Items

If you update the properties of a **DataSeriesItem** object, you need to call the `update()` method for the series with the item as the parameter. Changing data in this way causes animation of the change.

Range Data

Range charts expect the Y values to be specified as minimum-maximum value pairs. The **DataSeriesItem** provides `setLow()` and `setHigh()` methods to set the minimum and maximum values of a data point, as well as a number of constructors that accept the values.

```
RangeSeries series =  
    new RangeSeries("Temperature Extremes");  
  
// Give low-high values in constructor  
series.add(new DataSeriesItem(0, -51.5, 10.9));  
series.add(new DataSeriesItem(1, -49.0, 11.8));
```

```
// Set low-high values with setters
DataSeriesItem point = new DataSeriesItem();
point.setX(2);
point.setLow(-44.3);
point.setHigh(17.5);
series.add(point);
```

The **RangeSeries** offers a slightly simplified way of adding ranged data points, as described in Section 18.6.3, “Range Series”.

18.6.3. Range Series

The **RangeSeries** is a helper class that extends **DataSeries** to allow specifying interval data a bit easier, with a list of minimum-maximum value ranges in the Y axis. You can use the series in range charts, as described in Section 18.4.11, “Area and Column Range Charts”.

For the X axis, the coordinates are generated at fixed intervals starting from the value specified with the `pointStart` property (default is 0) at intervals specified with the `pointInterval` property (default is 1.0).

Setting the Data

The data in a **RangeSeries** is given as an array of minimum-maximum value pairs for the Y value axis. The pairs are also represented as arrays. You can pass the data using the ellipsis in the constructor or using `setData()`:

```
RangeSeries series =
    new RangeSeries("Temperature Ranges",
    new Double[]{ -51.5, 10.9 },
    new Double[]{ -49.0, 11.8 },
    ...
    new Double[]{ -47.0, 10.8 });
conf.addSeries(series);
```

18.6.4. Data Provider Series

DataProviderSeries is an adapter for using a Vaadin DataProvider as a **DataSeries** in a chart. Using `setPointName()`, `setX()`, and `setY()` you can define which parts of the bean in the DataProvider are used in the chart.



Note

DataProviderSeries is based on the data model in Vaadin Framework 8. It replaces **Container-DataSeries**, which allowed binding to a Container data model in Vaadin Framework 7.

Let us consider an example, where we have a DataProvider which provides items of type **Order**. The **Order** class has get-Description(), getUnitPrice(), and getQuantity() to be used for the chart:

```
public class Order {  
    private String description;  
    private int quantity;  
    private double unitPrice;  
  
    public Order(String description, int quantity, double unitPrice) {  
        this.description = description;  
        this.quantity = quantity;  
        this.unitPrice = unitPrice;  
    }  
  
    public String getDescription() {  
        return description;  
    }  
  
    public int getQuantity() {  
        return quantity;  
    }  
  
    public double getUnitPrice() {  
        return unitPrice;  
    }  
  
    public double getTotalPrice() {  
        return unitPrice * quantity;  
    }  
}
```

If we have a data provider containing a list of **Order** instances:

```
// The data  
List<Order> orders = new ArrayList<>();  
orders.add(new Order("Domain Name", 3, 7.99));  
orders.add(new Order("SSL Certificate", 1, 119.00));  
orders.add(new Order("Web Hosting", 1, 19.95));  
orders.add(new Order("Email Box", 20, 0.15));  
orders.add(new Order("E-Commerce Setup", 1, 25.00));  
orders.add(new Order("Technical Support", 1, 50.00));  
  
DataProvider<Order, ?> dataProvider = new ListDataProvider<>(orders);
```

We can display the data in a **Chart** as follows:

```
// Create a chart and use the data provider  
Chart chart = new Chart(ChartType.COLUMN);
```

```
Configuration configuration = chart.getConfiguration();
DataProviderSeries<Order> series = new DataProviderSeries<>(dataProvider, Order::getTotalPrice);
configuration.addSeries(series);
```



Note

The **DataProviderSeries** constructor takes the y value provider as an optional argument. It can also be set using `setY`.

To make the chart look nicer, we can add a name for the series and show the order description when hovering points:

```
series.setName("Order item quantities");
series.setX(Order::getDescription);
```

To show the description also as x axis labels, we need to set the x axis type to category as the labels are strings:

```
configuration.getxAxis().setType(AxisType.CATEGORY);
```

The result, with some added titles, is shown in Figure 18.28, "Chart Bound to a DataProvider".

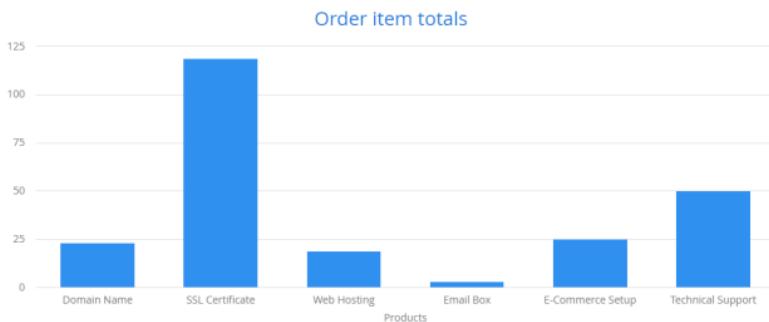


Figure 18.28. Chart Bound to a DataProvider

18.6.5. Drill-Down

Vaadin Charts allows drilling down from a chart to a more detailed view by clicking an item in the top-level view. To enable the feature, you need to provide a separate data series for each of the detailed views by calling the `addItemWithDrill-down()` method. When the user clicks on a drill-down item, the current series is animated into the linked drill-down

series. A customizable back button is provided to navigate back to the main series, as shown in Figure 18.29, "Detailed series after a drill-down".

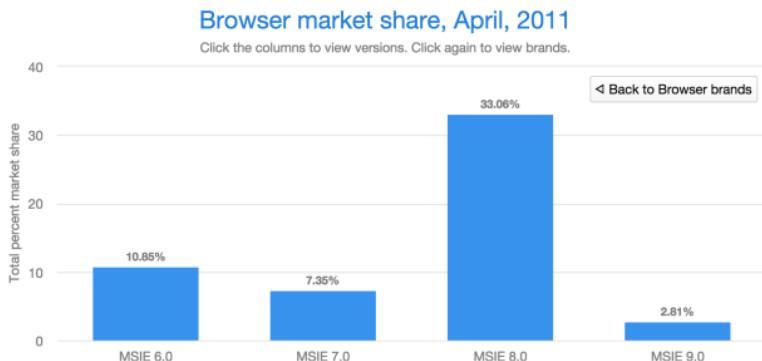


Figure 18.29. Detailed series after a drill-down

There are two ways to use drill-down: synchronous and asynchronous.

Synchronous

When using synchronous drill-down, you provide the top-level series and all the series below it beforehand. The data is transferred to the client-side at the same time and no client-server communication needs to happen for the drill-down. The drill-down series must have an identifier, set with setId(), as shown below.

```
Series series = new Series();
SeriesItem mainItem = new SeriesItem("MSIE", 55.11);
Series drillDownSeries = new Series("MSIE versions");
drillDownSeries.setId("MSIE");
drillDownSeries.addItem(new SeriesItem("MSIE 6.0", 10.85));
drillDownSeries.addItem(new SeriesItem("MSIE 7.0", 7.35));
drillDownSeries.addItem(new SeriesItem("MSIE 8.0", 33.06));
drillDownSeries.addItem(new SeriesItem("MSIE 9.0", 2.81));
series.addItemWithDrilldown(mainItem, drillDownSeries);
```

Asynchronous

When using asynchronous drill-down, you omit the drill-down series parameter. Instead, you provide a callback method with `Chart.setDrillDownCallback()`. When the user clicks an item in the series, the callback is called to provide a drill-down series.

```
    DataSeries series = new DataSeries();

    DataSeriesItem mainItem = new DataSeriesItem("MSIE", 55.11);
    series.addItemWithDrilldown(mainItem);

    chart.setDrilldownCallback(new DrilldownCallback() {
        @Override
        public Series handleDrilldown(DrilldownEvent event) {
            DataSeries drillDownSeries = new DataSeries("MSIE versions");

            drillDownSeries.add(new DataSeriesItem("MSIE 6.0", 10.85));
            drillDownSeries.add(new DataSeriesItem("MSIE 7.0", 7.35));
            drillDownSeries.add(new DataSeriesItem("MSIE 8.0", 33.06));
            drillDownSeries.add(new DataSeriesItem("MSIE 9.0", 2.81));

            return drillDownSeries;
        }
    });
}
```

You can use the event to decide what kind of series you want to return. The event contains, for example, a reference to the item that was clicked. Note that the same callback is used for all items. The callback can also return null, which will prevent a drilldown.

18.7. Advanced Uses

18.7.1. Server-Side Rendering and Exporting

In addition to using charts in Vaadin UIs, you may also need to provide them as images or in downloadable documents. Vaadin Charts can be rendered on the server-side using a headless JavaScript execution environment, such as PhantomJS.

Vaadin Charts supports a HighCharts remote export service, but the SVG Generator based on PhantomJS is almost as easy to use and allows much more powerful uses.

Using a Remote Export Service

Vaadin Charts has a simple built-in export functionality that does the export in a remote export server. Vaadin Charts provides a default export service, but you can also configure your own.

You can enable the built-in export function by setting setExporting(true) in the chart configuration.

```
chart.getConfiguration().setExporting(true);
```

To configure it further, you can provide a **Exporting** object with custom settings.

```
// Create the export configuration  
Exporting exporting = new Exporting(true);  
  
// Customize the file name of the download file  
exporting.setFilename("mychartfile");  
  
// Use the exporting configuration in the chart  
chart.getConfiguration().setExporting(exporting);
```

The functionality uses a HighCharts export service by default. To use your own, you need to set up one and then configure it in the exporting configuration as follows:

```
exporting.setUrl("http://my.own.server.com");
```

Using the SVG Generator

The **SVGGenerator** in Vaadin Charts provides an advanced way to render the Chart into SVG format on the server-side. SVG is well supported by many applications, can be converted to virtually any other graphics format, and can be passed to PDF report generators.

The generator uses PhantomJS to render the chart on the server-side. You need to install it from phantomjs.org. After installation, PhantomJS should be in your system path. If not, you can set the *phantom.exec* system property for the JRE to point to the PhantomJS binary.

To generate the SVG image content as a string (it's XML), simply call the `generate()` method in the **SVGGenerator** singleton and pass it the chart configuration.

```
String svg = SVGGenerator.getInstance()  
    .generate(chart.getConfiguration());
```

You can then use the SVG image as you like, for example, for download from a **StreamResource**, or include it in a HTML, PDF, or other document. You can use SVG tools such as the Batik or iText libraries to generate documents. For a complete example, you can check out the Charts Export Demo from the Subversion repository at <https://github.com/vaadin/charts/tree/master/chart-export-demo>.

18.8. Timeline

A charts timeline feature allows selecting different time ranges for which to display the chart data, as well as navigating between such ranges. It is especially useful when working with large time Section 18.3.3, "Chart Data Series". Adding a timeline to your chart is very easy - just set the 'timeline' property to 'true', that is, call `setTimeline(true)`. You can enable the timeline in a chart that displays one or more time series. Most of the chart types support the timeline. There are few exceptions which are listed here: Section 18.4.8, "Pie Charts", Section 18.4.9, "Gauges", Section 18.4.10, "Solid Gauges", Section 18.4.13, "Funnel and Pyramid Charts", and Section 18.4.13, "Funnel and Pyramid Charts". Enabling the timeline in these chart types will raise a runtime exception.

You can change the time range using the navigator at the bottom of the chart. To be able to use the navigator, the X values of the corresponding data series should be of the type **Date**. Also integer values can be used, in which case they are interpreted as milliseconds since the 01/01/1970 epoch. If you have multiple series, the first one is presented in the navigator.

Range selector



Figure 18.30. Vaadin chart with a timeline.

Another way to change the time range is to use the range selector. The range selector includes a set of predefined time ranges for easier navigation, for example, 1 month, 3 month, 6 month etc. To specify a custom time range, you can use range selector text fields for setting start and end of the time interval.

You can configure the range navigator and selector in the chart configuration. To show or hide the navigator, call `setEnabled()`. You can use **Navigator** and **PlotOptionsSeries** to change the appearance of the navigator.

```
Navigator navigator = configuration.getNavigator();
navigator.setEnabled(true);
navigator.setMargin(75);
PlotOptionsSeries plotOptions=new PlotOptionsSeries();
plotOptions.setColor(SolidColor.BROWN);
navigator.setSeries(plotOptions);
```

You can change the style of the range selector buttons with the `setButtonTheme(theme)` method and specify the index of the button to appear pre-selected with the `setSelected(index)` method.

```

RangeSelector rangeSelector = new RangeSelector();
rangeSelector.setSelected(4);
ButtonTheme theme = new ButtonTheme();
Style style = new Style();
style.setColor(new SolidColor("#0766d8"));
style.setFontWeight(FontWeight.BOLD);
theme.setStyle(style);
rangeSelector.setButtonTheme(theme);

Chart chart = new Chart();
chart.setTimeline(true);
Configuration configuration = chart.getConfiguration();
configuration.setRangeSelector(rangeSelector);
chart.drawChart(configuration);

```

You can customize the date format for the time range input fields by specifying formatter strings for displaying and editing the dates, as well as a corresponding JavaScript parser function to parse edited values:

```

RangeSelector rangeSelector = new RangeSelector();
rangeSelector.setInputDateFormat("%Y-%M-%D.%H.%M");
rangeSelector.setInputEditDateFormat("%Y-%M-%D.%H.%M");
rangeSelector.setInputDateParser(
    "function(value) {" +
    "value = value.split(/[-]/);\n" +
    "return Date.UTC(\n" +
        "    parseInt(value[0], 10),\n" +
        "    parseInt(value[1], 10),\n" +
        "    parseInt(value[2], 10),\n" +
        "    parseInt(value[3], 10),\n" +
        "    parseInt(value[4], 10),\n" +
    ");");
configuration.setRangeSelector(rangeSelector);

```

Timeline charts allow comparing the charts series against each other. Setting the compare property to either Compare.PERCENT or Compare.VALUE will show the difference between charts data series in percentage or absolute values respectively.

```

PlotOptionsSeries plotOptions = new PlotOptionsSeries();
plotOptions.setCompare(Compare.PERCENT);
configuration.setPlotOptions(plotOptions);

```

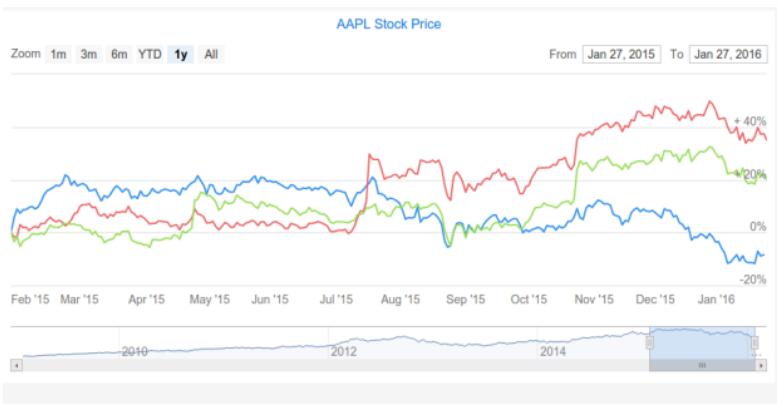


Figure 18.31. Vaadin chart with a percentage comparison between series.

You can find more examples in the Timeline section of Vaadin Charts Demo.

Chapter 19

Vaadin Spreadsheet

19.1. Overview	659
19.2. Installing Vaadin Spreadsheet	664
19.3. Basic Use	665
19.4. Spreadsheet Configuration	667
19.5. Cell Content and Formatting	669
19.6. Context Menus	671
19.7. Tables Within Spreadsheets	672

This chapter describes how to use Vaadin Spreadsheet, a Vaadin add-on component for displaying and editing Excel spreadsheets within any Vaadin application.

19.1. Overview

Spreadsheet applications have been the sonic screwdriver of business computation and data collection for decades. In recent years, spreadsheet web services have become popular with cloud-based services that offer better collaboration, require no installation, and some are even free to use. However, both desktop and third-party cloud-based services are difficult to integrate well with web applications. Being a Vaadin UI component, Vaadin Spreadsheet allows complete integration with Vaadin applications and further with the overall system. The ability to work on Excel spreadsheets allows desktop interoperability and integration with document management.

By eliminating the dependency on third-party cloud-based services, Vaadin Spreadsheet also gives control over the privacy of documents. Growing security concerns over cloud-based information storage have increased privacy requirements, with lowering trust in global third-party providers. Vaadin applications can run on private application servers, and also in a cloud if necessary, allowing you to prioritize between privacy and local and global availability.

The screenshot shows a spreadsheet application window. At the top left, the cell A2 is selected. The title "Loan calculator" is centered in cell A1. In cell C3, the text "Edit these!" is displayed. Below it, there are several data entries:

		Loan amount	\$10 000,00
		Interest rate	5,00%
		Period (years)	2
		Start date	1/1/15
		Monthly payment	\$438,71
		Number of payments	24
		Total interest	\$529,13
		Last payment	12/1/16
		Total price payed	\$10 529,13

At the bottom of the spreadsheet, there are navigation icons: back, forward, plus, and a tab labeled "Sheet1".

Figure 19.1. Demo for Vaadin Spreadsheet

Vaadin Spreadsheet is a UI component that you use much like any other component. It has full size by default, to use all the available space in the containing layout. You can directly modify the cell data in the active worksheet by entering textual and numerical values, as well as using Excel formulas for spreadsheet calculations.

```
Spreadsheet sheet = new Spreadsheet();
sheet.setWidth("400px"); // Full size by default
sheet.setHeight("250px");
```

```

// Put customary greeting in a cell
sheet.createCell(0, 0, "Hello, world");

// Have some numerical data
sheet.createCell(1, 0, 6);
sheet.createCell(1, 1, 7);

// Perform a spreadsheet calculation
sheet.createCell(1, 2, ""); // Set a dummy value
sheet.getCell(1, 2).setCellFormula("A2*B2");

// Resize a column to fit the cell data
sheet.autofitColumn(0);

layout.addComponent(sheet);
layout.setSizeFull(); // Typically

```

The result is shown in Figure 19.2, “Simple Spreadsheet”.

	A1		Hello, world		
	A	B	C	D	E
1	Hello, world				
2	6	7		42	
3					
4					
5					
6					
7					
8					

<< < > >> + Sheet0

Figure 19.2. Simple Spreadsheet

Cell values and formulas can be set, read, and styled through the server-side API, so you can easily implement custom editing features through menus or toolbars.

Full integration with a Vaadin application is reached through the server-side access to the spreadsheet data as well as visual styling. Changes in cell values can be handled in the

Vaadin application and you can use almost any Vaadin components within a spreadsheet. Field components can be bound to cell data.

Vaadin Spreadsheet uses Apache POI to work on Microsoft Excel documents. You can access the Apache POI data model to perform low-level tasks, although you should note that if you make modifications to the data model, you have the responsibility to notify the spreadsheet to update itself.

19.1.1. Features

The basic features of Vaadin Spreadsheet are as follows:

- Support for touch devices
- Excel XLSX files, limited support for XLS files
- Support for Excel formulas
- Excel-like editing with keyboard support
- Lazy loading of cell data from server to browser in large spreadsheets
- Protected cells and sheets

The following features support integration with Vaadin Framework and add-ons:

- Handle changes in cell data
- Vaadin components in spreadsheet cells
- Support for Vaadin declarative format
- Vaadin TestBench element API for UI testing

19.1.2. Spreadsheet Demo

The Vaadin Spreadsheet Demo showcases the most important Spreadsheet features. You can try it out at <http://demo.vaadin.com/spreadsheet>.

19.1.3. Requirements

- Vaadin 7.4 or later
- Same browser requirements as Vaadin Framework, except Internet Explorer 9 or later is required

19.1.4. Limitations

Vaadin Spreadsheet 1.2 has the following limitations:

- No provided toolbars, menus, or other controls for formatting cells
- Limited support for the older XSL format
- Limitations of Apache POI
- Using a table as a named range in formulas is not supported. As a workaround, you can manually convert tables to ranges. Open the spreadsheet in Excel, right-click on the table, select the **Convert to Range** action, and save the spreadsheet.
- The SUBTOTAL formula is limited to aggregate functions that do not ignore hidden values (function codes from 1 to 7, as well as 9), because they are not implemented in Apache POI
- Strict OOXML format is not supported by Apache POI

19.1.5. Licensing

Vaadin Spreadsheet is a commercial product licensed under the CVAL license (Commercial Vaadin Add-On License). Development licenses need to be purchased for each developer working with Vaadin Spreadsheet, regardless of whether the resulting applications using it are deployed publicly or privately in an intranet.

A Vaadin Pro Tools subscription includes a subscription license for Vaadin Spreadsheet. Perpetual licenses can be purchased from the Vaadin Spreadsheet product page, where you can also find the licensing details. For evaluation purposes, a free

trial key allows using Vaadin Spreadsheet for a 30-day evaluation period.

19.2. Installing Vaadin Spreadsheet

You can download and install Spreadsheet from Vaadin Directory at vaadin.com/addon/vaadin-spreadsheet as an installation package, or get it with Maven or Ivy. You can purchase the required CVAL license or get a free trial key from Vaadin Directory or subscribe to the Pro Tools at vaadin.com/pro.

Add-on installation is described in detail in Chapter 17, *Using Vaadin Add-ons*. The add-on includes both a widget set and a theme, so you need to compile the widget sets and themes in your project.

19.2.1. Installing a License Key

You need to install a license key before compiling the widget set. The license key is checked during widget set compilation, so you do not need it when deploying the application.

You can purchase Vaadin Spreadsheet or obtain a free trial key from the Vaadin Spreadsheet download page in Vaadin Directory. You need to register in Vaadin Directory to obtain the key.

See Section 17.3, "Installing Commercial Vaadin Add-on License" for detailed instructions on obtaining and installing the license key.

19.2.2. Compiling the Widget Set

Compile the widget set as instructed in Section 17.2.2, "Compiling the Application Widget Set". Widget set compilation should automatically update your project widget set to include the Spreadsheet widget set:

```
<inherits name="com.vaadin.addon.spreadsheet.Widgetset"/>
```

If you have set the widget set to be manually edited, you need to add the element yourself.

19.2.3. Compiling Theme

Compile the theme as instructed in Section 9.4, "Compiling Sass Themes". If you compile in Eclipse or with Maven, the addons.scss file in your theme should be automatically updated to include the Spreadsheet theme:

```
@import "../VAADIN/addons/spreadsheet/spreadsheet.scss";
@mixin addons {
    @include spreadsheet;
}
```

If you are compiling the theme otherwise, or the theme addons are not automatically updated for some reason, you need to add the statements yourself.

19.2.4. Importing the Demo

The Demo, illustrated in Figure 19.1, "Demo for Vaadin Spreadsheet" in the overview, showcases most of the functionality in Vaadin Spreadsheet. You can try out the demo online at demo.vaadin.com/spreadsheet or check the sources from the GitHub page.

19.3. Basic Use

Spreadsheet is a Vaadin component, which you can use as you would any component. You can create it, or load from an Excel file, create cells and new sheets, define formulas, and so forth. In the following, we go through these basic steps.

19.3.1. Creating a Spreadsheet

The default constructor for **Spreadsheet** creates an empty spreadsheet with a default sheet. The spreadsheet component has full size by default, so the containing layout must have defined size in both dimensions; a spreadsheet may not have undefined size.

In the following example, we create an empty spreadsheet with a fixed size and add it to a layout.

```
import com.vaadin.addon.spreadsheet.Spreadsheet;
...
private Spreadsheet spreadsheet = null;
```

```
@Override  
protected void init(VaadinRequest request) {  
    VerticalLayout layout = new VerticalLayout();  
    layout.setSizeFull();  
    setContent(layout);  
  
    spreadsheet = new Spreadsheet();  
    layout.addComponent(spreadsheet);  
}
```

An empty spreadsheet automatically gets an initial worksheet with some default size and settings.

Loading an Excel Spreadsheet

To open an existing Excel file in Spreadsheet, you need to pass an Excel file to the **Spreadsheet** constructor.

```
import com.vaadin.addon.spreadsheet.Spreadsheet;  
...  
private Spreadsheet spreadsheet;  
  
@Override  
protected void init(VaadinRequest request) {  
    final VerticalLayout layout = new VerticalLayout();  
    layout.setSizeFull();  
    setContent(layout);  
  
    File sampleFile = new File("C:/Users/John/Calculator.xlsx");  
    try {  
        spreadsheet = new Spreadsheet(sampleFile);  
    } catch (IOException e) {  
        e.printStackTrace();  
        return;  
    }  
    layout.addComponent(spreadsheet);  
}
```

You can load an Excel file from the local filesystem with a **File** reference or from memory or other source with an **InputStream**.

Working with an Apache POI Workbook

If you have an Apache POI workbook created otherwise, you can wrap it to **Spreadsheet** with the respective constructor.

You can access the underlying workbook with `getWorkbook()`. However, if you make modifications to the workbook, you

must allow the spreadsheet update itself by calling appropriate update methods for the modified element or elements.

19.3.2. Working with Sheets

A "spreadsheet" in reality works on a *workbook*, which contains one or more *worksheets*. You can create new sheets and delete existing ones with `createNewSheet()` and `deleteSheet()`, respectively.

```
// Recreate the initial sheet to configure it
Spreadsheet sheet = new Spreadsheet();
sheet.createNewSheet("New Sheet", 10, 5);
sheet.deleteSheet(0);
```

When a sheet is deleted, the index of the sheets with a higher index is decremented by one. When the active worksheet is deleted, the next one by index is set as active, or if there are none, the previous one.

All operations on the spreadsheet content are done through the currently active worksheet. You can set an existing sheet as active with `setActiveSheetIndex()`.

19.4. Spreadsheet Configuration

Spreadsheet has a number of configuration parameters and sections that affect the overall appearance and function of the entire spreadsheet. The most important ones are mentioned in the following and further configuration is provided through the Apache POI API.

19.4.1. Spreadsheet Elements

The **Spreadsheet** object provides the following configuration of various UI elements:

Grid lines

Cells are by default separated by grid lines. You can control their visibility with `setGridlinesVisible()`.

Column and row headings

Headings for rows and columns display the row and column indexes, and allow selecting and resizing the

rows and columns. You can control their visibility with `setRowColHeadingsVisible()`.

Top bar

The top bar displays the address of the currently selected cell and an editor for cell content. You can control its visibility with `setFunctionBarVisible()`.

Bottom bar

The bottom bar displays the address of the currently selected cell and an editor for cell content. You can control its visibility with `setSheetSelectionBarVisible()`.

Report mode

In the report mode, both the top and bottom bars are hidden. You can enable the mode with `setReportStyle()`.

19.4.2. Frozen Row and Column Panes

You can define panes of rows and columns that are frozen in respect to scrolling. You can create the pane for the current worksheet with `createFreezePane()`, which takes the number of frozen rows and columns as parameters.

`sheet.createFreezePane(1, 2);`

The result is shown in Figure 19.3, "Frozen Row and Column Panes".

The screenshot shows a spreadsheet interface with a frozen header row. The header row contains the column labels 'First Name' and 'Last Name' under column A, and 'Born' under column C. The first data row (row 2) contains 'Nicolaus' under 'First Name', 'Copernicus' under 'Last Name', and '1999-03-19' under 'Born'. The second data row (row 3) contains 'Galileo' under 'First Name', 'Galilei' under 'Last Name', and '1964-03-15' under 'Born'. Row 4 is partially visible with a red border around the first two columns. The bottom navigation bar includes icons for navigating between sheets and a button to add a new sheet, labeled 'Sheet1'.

A4				
	A	B	C	D
1	First Name	Last Name	Born	
2	Nicolaus	Copernicus	1999-03-19	
3	Galileo	Galilei	1964-03-15	
4				
5				
6				

Figure 19.3. Frozen Row and Column Panes

19.5. Cell Content and Formatting

In the following, we go through various user interface features in the table cells.

19.5.1. Cell Formatting

Formatting cell values can be accomplished by using cell styles. A cell style must be created in the workbook by using `createCellStyle()`. Cell data format is set for the style with `setDataFormat()`.

```
// Define a cell style for dates
```

```
CellStyle dateStyle = sheet.getWorkbook().createCellStyle();
DataFormat format = sheet.getWorkbook().createDataFormat();
dateStyle.setDataFormat(format.getFormat("yyyy-mm-dd"));
```

```
// Add some data rows
```

```
sheet.createCell(1, 0, "Nicolaus");
sheet.createCell(1, 1, "Copernicus");
sheet.createCell(1, 2,
    new GregorianCalendar(1999,2,19).getTime());
```

```
// Style the date cell
```

```
sheet.getCell(1,2).setCellStyle(dateStyle);
```

19.5.2. Cell Font Style

Cells can be styled by different fonts. A font definition not only includes a particular typeface, but also weight (bold or normal), emphasis, underlining, and other such font attributes.

A font definition is managed by **Font** class in the Apache POI API. A new font can be created with `createFont()` in the workbook.

For example, in the following we make a header row in a spreadsheet bold:

```
// Create some column captions in the first row
```

```
sheet.createCell(0, 0, "First Name");
sheet.createCell(0, 1, "Last Name");
sheet.createCell(0, 2, "Born");
```

```
// Create a bold font
```

```
Font bold = sheet.getWorkbook().createFont();
bold.setBold(true);
```

```
// Set the cells in the first row as bold  
for (int col=0; col <= 2; col++)  
    sheet.getCell(0, col).getCellStyle().setFont(bold);
```

19.5.3. Cell Comments

Cells may have comments that are indicated by ticks in the corner of the cells, and the comment is shown when mouse is hovered on the cells. The **SpreadsheetDefaultActionHandler** described in Section 19.6.1, "Default Context Menu" enables adding comments from the context menu.

A new comment can be added through the POI API of a cell, with addComment(). For a detailed example for managing cell comments, we refer to the **InsertDeleteCellCommentAction** and **EditCellCommentAction** classes employed by the default action handler.

19.5.4. Merging Cells

You can merge spreadsheet cells with any variant of the addMergedRegion() method in **Spreadsheet**.

The **SpreadsheetDefaultActionHandler** described in Section 19.6.1, "Default Context Menu" enables merging selected cells from the context menu.

Previously merged cells can be unmerged with removeMergedRegion(). The method takes as its parameter a region index. You can search for a particular region through the POI **Sheet** API for the active sheet, which you can obtain with getActiveSheet(). The getMergedRegion() returns a merged region by index and you can iterate through them by knowing the number of regions, which you can find out with getNumMergedRegions().

19.5.5. Components in Cells

You can have Vaadin components in spreadsheet cells and bind field components to the cell data. The components can be shown all the time or work as editors that appear when a cell is activated for editing.

Components in a spreadsheet must be generated by a `SpreadsheetComponentFactory`, which you need to implement.

19.5.6. Hyperlinks

Hyperlinks in cells can point to other worksheets in the current workbook, or to external URLs. Links must be added through the POI API. Spreadsheet provides default handling for hyperlink clicks, which can be overridden with a custom `HyperlinkCellClickHandler`, which you assign with `setHyperlinkCellClickHandler()`.

19.5.7. Popup Buttons in Cells

You can add a popup button in a cell, clicking which opens a drop-down popup overlay, which can contain any Vaadin components. You can add a popup button with any of the `setPopup()` methods for different cell addressing forms. A popup button is a `PopupButton` object, which is a regular Vaadin component container, so you can add any components to it by `addComponent()`.

You can create popup buttons for a row of cells in a cell range by defining a `table`, as described in Section 19.7, “Tables Within Spreadsheets”.

19.6. Context Menus

Grid has a context menu, which can be used for various editing functions. The context menu uses the standard Vaadin action handler mechanism.

19.6.1. Default Context Menu

The `SpreadsheetDefaultActionHandler` provides a ready set of common spreadsheet editing operations.

- Add, delete, hide, and show rows and columns
- Insert, edit, and delete cell comments
- Merge and unmerge cells
- Add filter table to selected range

The default context menu is not enabled by default; to enable it, you need to create and set it as the action handler explicitly.

```
sheet.addActionHandler(new SpreadsheetDefaultActionHandler());
```

19.6.2. Custom Context Menus

You can implement custom context menus either by implementing an ActionHandler or extending the **SpreadsheetDefaultActionHandler**. The source code of the default handler should serve as a good example of implementing context menu items for different purposes.

19.7. Tables Within Spreadsheets

A cell range in a worksheet can be configured as a *table*, which adds popup menu buttons in the header row of the range. The popup menus contain Vaadin components, which you can use to implement various functionalities in the table, such as sorting or filtering. Vaadin Spreadsheet does not include any implementations of such features, merely the UI elements to enable them.

Such a table is defined by a **SpreadsheetTable** or a **SpreadsheetFilterTable** added to the spreadsheet.

19.7.1. Creating a Table

For example, let us assume that you have a sheet with some data in a cell region. The first row of the region should contain column captions for the region.

```
Spreadsheet sheet = new Spreadsheet();
```

```
// Have a header row in a region of the sheet
sheet.createCell(1, 1, "First Name");
sheet.createCell(1, 2, "Last Name");
sheet.createCell(1, 3, "Born");
sheet.createCell(1, 4, "Died");
```

```
... insert the data ...
```

```
// Define the range
CellRangeAddress range =
    new CellRangeAddress(1, 7, 1, 4);
```

A table is created for a cell range and then registered in the spreadsheet with registerTable(); you can unregister it with unregisterTable(). The first row of the cell range should contain the table captions.

```
// Create a table in the range
SpreadsheetTable table = new SpreadsheetTable(sheet, range);
sheet.registerTable(table);

// Enable hiding each column in the popup
for (int col = range.getFirstColumn();
     col <= range.getLastColumn(); col++) {
    final int c = col; // For access in the lambda
    table.getPopupButton(col).addComponent(
        new Button("Hide Column", e -> { // Java 8
            sheet.setColumnHidden(c, true);
            table.getPopupButton(c).closePopup();
        }));
}
```

The popup buttons are typically used for performing operations on columns, such as sorting.

19.7.2. Filtering With a Table

SpreadsheetFilterTable is a spreadsheet table that allows filtering the rows in the table in the popup menus. The menu is filled with checkboxes for each unique value in the column. Deselecting the items causes hiding the respective rows in the spreadsheet.

Unresolved directive in y/website/_spreadsheet/chapter-spreadsheet.asciidoc - include::spreadsheet-migration-vaadin8.asciidoc[leveloffset=+2]

Chapter 20

Vaadin TestBench

20.1. Overview	675
20.2. Quickstart	681
20.3. Installing Vaadin TestBench	684
20.4. Developing JUnit Tests	688
20.5. Creating a Test Case	694
20.6. Querying Elements	697
20.7. Element Selectors	700
20.8. Special Testing Topics	702
20.9. Creating Maintainable Tests	708
20.10. Taking and Comparing Screenshots	713
20.11. Running Tests	719
20.12. Running Tests in a Distributed Environment	722
20.13. Parallel Execution of Tests	730
20.14. Headless Testing	732
20.15. Behaviour-Driven Development	734
20.16. Integration Testing with Maven	736
20.17. Known Issues	740
20.18. TestBench 4 to TestBench 5 Migration Guide	742

This chapter describes the installation and use of the Vaadin TestBench.

20.1. Overview

Testing is one of the cornerstones of modern software development. Extending throughout the development process, testing is the thread that binds the product to the require-

ments. In agile and other iterative development processes, with ever shorter release cycles and continuous integration, the automation of integration, regression, endurance, and acceptance testing is paramount. Further, UI automation may be needed for integration purposes, such as for assistive technologies. The special nature of web applications creates many unique requirements for both testing and UI automation.

Vaadin TestBench allows controlling the browser from Java code, as illustrated in Figure 20.1, "Controlling the Browser with TestBench". It can open a new browser window to start the application, interact with the UI components, for example, by clicking them, and then get the HTML element values.

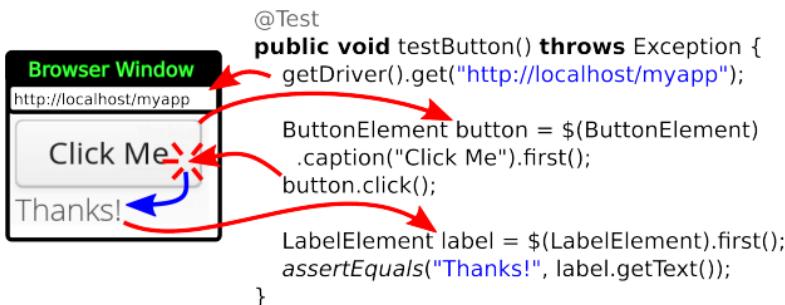


Figure 20.1. Controlling the Browser with TestBench

Before going further into feature details, you may want to try out Vaadin TestBench yourself. You just need to create a new Vaadin project either with the Eclipse plugin or the Maven archetype. Both create a simple application stub that includes TestBench test cases for testing the UI. You also need to install an evaluation license. For instructions, jump to Section 20.2, "Quickstart" and, after trying it out, come back.

20.1.1. Vaadin TestBench in Software Development

Vaadin TestBench can work as the centerpiece of the software development process, for testing the application at all modular levels and in all the phases of the development cycle:

- Automated acceptance tests

- Unit tests
- End-to-end integration tests
- Regression tests

Let us look at each of these topics separately.

Any methodological software development, agile or not, is preceded by specification of requirements, which define what the software should actually do. *Acceptance tests* ensure that the product conforms to the requirements. In agile development, their automation allows continuous tracking of progress towards iteration goals, as well as detecting regressions. The importance of requirements is emphasized in *test-driven development* (TDD), where tests are written before actual code. In Section 20.15, “Behaviour-Driven Development”, we show how to use Vaadin TestBench for *behaviour-driven development* (BDD), a form of TDD that concentrates on the formal behavioural specification of requirements.

Unit testing is applied to the smallest scale of software components; in Vaadin applications these are typically individual UI components or view classes. You may also want to generate many different kinds of inputs for the application and check that they produce the desired outputs. For complex composites, such as views, you can use the Page Object Pattern described in Section 20.9.2, “The Page Object Pattern”. The pattern simplifies and modularizes testing by separating low-level details from the more abstract UI logic. In addition to serving the purpose of unit tests, it creates an abstraction layer for higher-level tests, such as acceptance and end-to-end tests.

Integration tests ensure that software units work together at different levels of modularization. At the broadest level, *end-to-end tests* extend through the entire application lifecycle from start to finish, going through many or all user stories. The aim is not just to verify the functional requirements for user interaction, but also that data integrity is maintained. For example, in a messaging application, a user would log in, both send and receive messages, and finally log out. Such test workflows could include configuration, registration, interaction between users, administrative tasks, deletion of user accounts, and so forth.

In *regression testing*, you want to ensure that only intended changes occur in the behaviour after modifying the code. There are two lines of defence against such regressions. The primary source of regression tests are the acceptance, unit, and integration tests that validate that the displayed values in the UI's HTML representation are logically correct. Yet, they are not sufficient for detecting visual regressions, for example, because of invalid UI rendering or theme problems. Comparing screenshots to reference images forms a more sensitive layer to detect regressions, at the expense of losing robustness for changes in layout and theming. The costs of the tradeoff can be minimized by careful application of screenshot comparison only at critical points and by making the analysis of such regressions as easy as possible. As described in Section 20.10, "Taking and Comparing Screenshots", Vaadin TestBench automatically highlights differences in screenshots and allows masking irrelevant areas from image comparison.

You can develop such tests along with your application code, for example with JUnit, which is a widely used Java unit testing framework. You can run the tests as many times as you want in your workstation or in a distributed grid setup.

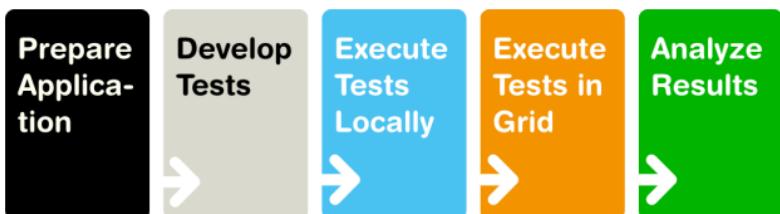


Figure 20.2. TestBench Workflow

20.1.2. Features

The main features of Vaadin TestBench are:

- Control a browser from Java
- Generate component selectors in debug window
- Validate UI state by assertions and screen capture comparison

- Screen capture comparison with difference highlighting
- Distributed test grid for running tests
- Integration with unit testing
- Test with browsers on mobile devices

Execution of tests can be distributed over a grid of test nodes, which speeds up testing. The grid nodes can run different operating systems and have different browsers installed. In a minimal setup, such as for developing the tests, you can use Vaadin TestBench on just a single computer.

20.1.3. Based on Selenium

Vaadin TestBench is based on the Selenium web browser automation library, especially Selenium WebDriver, which allows you to control browsers straight from Java code.

Selenium is augmented with Vaadin-specific extensions, such as:

- Proper handling of Ajax-based communications of Vaadin
- A high-level, statically typed element query API for Vaadin components
- Performance testing of Vaadin applications
- Screen capture comparison
- Finding HTML elements by a Vaadin selector

20.1.4. TestBench Components

The TestBench library includes WebDriver, which provides API to control a browser like a user would. This API can be used to build tests, for example, with JUnit. It also includes the grid hub and node servers, which you can use to run tests in a grid configuration.

Vaadin TestBench Library provides the central control logic for:

- Executing tests with the WebDriver
- Additional support for testing Vaadin-based applications
- Comparing screen captures with reference images
- Distributed testing with grid node and hub services

20.1.5. Requirements

Requirements for developing and running tests are:

- Java JDK 1.6 or newer
- Browsers installed on test nodes as supported by Selenium WebDriver
 - Google Chrome
 - Internet Explorer
 - Mozilla Firefox (ESR version recommended)
 - Opera
 - Mobile browsers: Android, iPhone
- A build system, such as Ant or Maven, to automate execution of tests during build process (recommended)

Note that running tests on an Extended Support Release (ESR) version of Firefox is recommended because of the frequent release cycle of Firefox, which often cause tests to fail. Download an ESR release of Firefox from <http://www.mozilla.org/en-US/firefox/organizations/all.html>. Install it alongside your normal Firefox install (do not overwrite).

For Mac OS X, note the issue mentioned in Section 20.17.2, "Running Firefox Tests on Mac OS X".

20.1.6. Continuous Integration Compatibility

Continuous integration means automatic compilation and testing of applications frequently, typically at least daily, but ideally every time when code changes are committed to the

source repository. This practice allows catching integration problems early and finding the changes that first caused them to occur.

You can make unit tests with Vaadin TestBench just like you would do any other Java unit tests, so they work seamlessly with continuous integration systems. Vaadin TestBench is tested to work with at least TeamCity and Hudson/Jenkins build management and continuous integration servers, which all have special support for the JUnit unit testing framework.



Figure 20.3. Continuous Integration Workflow

Figure 20.3, “Continuous Integration Workflow” illustrates a typical development setup. Both changes to application and test sources are checked in into a source repository, from where the CIS server checks them out, compiles, and deploys the web application to a server. Then, it runs the tests and collects the results.

20.1.7. Licensing and Trial Period

You can download Vaadin TestBench from Vaadin Directory and try it out for a free 30-day trial period, after which you are required to acquire the needed licenses. You can purchase licenses from the Directory. A license for Vaadin TestBench is also included in the Vaadin Pro Account subscription.

20.2. Quickstart

This section walks you through the steps needed to add a TestBench test to your Vaadin application. We use Maven to set up the project and handle the dependencies. If you do

not have a Vaadin application, you can generate it using Maven archetype:

```
$ mvn -B archetype:generate \
-DarchetypeGroupId=com.vaadin \
-DarchetypeArtifactId=vaadin-archetype-application \
-DarchetypeVersion=7.7.x \
-DgroupId=org.test \
-DartifactId=vaadin-app \
-Dversion=0.1 \
-Dpackaging=war
```

Running tests on different browsers requires a web driver for the browser to be installed, see Section 20.3.3, “Installing Browser Drivers”.

A blue circular icon containing a white lowercase letter 'i', representing an informational note.

Note

Before TestBench version 5.0.0, Firefox did not require a web driver setup. Since TestBench version 5.0.0, Geckodriver is used to run tests on Firefox. Geckodriver requires a separate installation.

A blue circular icon containing a white lowercase letter 'i', representing an informational note.

Note

Geckodriver used for the latest Firefox versions (48 and newer) does not support actions API. It is recommended you use Firefox ESR (currently at 45.3) to execute tests on Firefox until this has been fixed.

20.2.1. Adding the Maven Dependency

Let us add the Vaadin TestBench dependency to the project. Open the pom.xml file and find the dependencies tag. Add the following dependency declaration just before the end tag (/dependencies):

```
<dependency>
<groupId>com.vaadin</groupId>
<artifactId>vaadin-testbench</artifactId>
<version>4.1.0</version>
```

```
<scope>test</scope>
</dependency>
```

20.2.2. Installing License Key

Before running tests, you need to install a license key. You can purchase a license or obtain a free trial key from the Vaadin TestBench download page in Vaadin Directory. The easiest way to install the license key is to copy and paste it to a .vaadin.testbench.developer.license file in your home directory. Other options of installing license key are described ???.

20.2.3. Create a Sample Test Class

In the Maven world, all test classes and resources live in the src/test directory, so create that directory. Continue by creating a java directory under that so that you end up with an src/test/java directory structure. Create a new file called MyAppTest.java in the src/test/java directory.

To create a TestBench test, you need to perform the following steps:

- Set the browser driver and open the tested web page.
- Simulate user actions by using TestbenchElement classes.
- Compare expected and actual results.
- Close the browser window.

To give an example, we create a test that clicks the button in the UI and checks that the value of the label has changed.

```
import org.junit.Assert;
import org.junit.Test;

public class MyAppTest extends ParallelTest {

    private static final String URL="http://localhost";
    private static final String PORT="8080";

    @Before
    public void setup() throws Exception {
        setDriver(new FirefoxDriver());
        //Open the web page
        getDriver().get(URL+":"+PORT);
    }
}
```

```

@After
public void tearDown() throws Exception {
    //close the browser window
    getDriver().quit();
}

@Test
public void test() {
    //Get a reference to the button
    ButtonElement button = $(ButtonElement.class).first();

    //Simulate button click
    button.click();

    //Get text field value
    String actualValue = $(LabelElement.class).first().getText();

    //Check that the value is not empty
    Assert.assertFalse(actualValue.isEmpty());
}
}

```

Now you have your first TestBench test case ready.

20.2.4. Running Tests

Before running the test, you should start your Vaadin application under test.

`mvn jetty:run`

You can now run the test by issuing Maven command line:

`mvn test`

or run it as a Junit Test from you IDE.

20.3. Installing Vaadin TestBench

As with most Vaadin add-ons, you can install Vaadin TestBench as a Maven or Ivy dependency in your project, or from an installation package. The installation package contains some extra material, such as documentation, as well as the standalone library, which you use for testing in a grid.

The component element classes are Vaadin version specific and they are packaged in a `vaadin-testbench-api` library JAR, separately from the `vaadin-testbench-core` runtime library, which is needed for executing the tests.

Additionally, you may need to install drivers for the browsers you are using as described in Section 20.3.3, “Installing Browser Drivers”.

20.3.1. Test Development Setup

In a typical test development setup, you develop tests in a Java project and run them on the development workstation. You can run the same tests in a dedicated test server, such as a continuous integration system.

In a test development setup, you do not need a grid hub or nodes. However, if you develop tests for a grid, you can run the tests, the grid hub, and one node all in your development workstation. A distributed setup is described later.

Maven Dependency

Add to the Vaadin TestBench dependency to your pom.xml. Remember to use the test scope as you don’t want the testing libraries to be deployed with your application.

```
<dependency>
  <groupId>com.vaadin</groupId>
  <artifactId>vaadin-testbench</artifactId>
  <version>4.1.0.beta2</version>
  <scope>test</scope>
</dependency>
```

You also need to define the Vaadin add-ons repository if not already defined:

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

The vaadin-archetype-application archetype includes the repository declarations.

Ivy Dependency

The Ivy dependency, to be defined in ivy.xml, would be as follows:

```
<dependency org="com.vaadin" name="vaadin-testbench"  
rev="latest.release" conf=nodeploy->default/>
```

The optional nodeploy->default configuration mapping requires a nodeploy configuration in the Ivy module; it is automatically created for new Vaadin projects.

A new Vaadin project created with the Vaadin Plugin for Eclipse includes the dependency.

Code Organization

We generally recommend developing tests in a project or module separate from the web application to be tested to avoid library problems. If the tests are part of the same project, you should at least arrange the source code and dependencies so that the test classes, the TestBench library, and their dependencies would not be deployed unnecessarily with the web application.

20.3.2. A Distributed Testing Environment

Vaadin TestBench supports distributed execution of tests in a grid. See Section 20.12, "Running Tests in a Distributed Environment".

20.3.3. Installing Browser Drivers

Each browser requires a browser specific web driver to be setup before tests can be run.

1. Download the latest browser driver
 - Firefox - install GeckoDriver for your platform from the latest release at:
<https://github.com/mozilla/geckodriver/releases>
 - Chrome - install ChromeDriver (a part of the Chromium project) for your platform from the latest release at:
<https://sites.google.com/a/chromium.org/chromedriver/downloads>

- Microsoft Edge - install MicrosoftWebDriver for your platform from the latest release at:
<https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/>
 - Internet Explorer (Windows only) - install IEDriverServer.exe from the latest Selenium release:
<https://github.com/SeleniumHQ/selenium/wiki/InternetExplorerDriver>
2. Add the driver executable to user PATH. In a distributed testing environment, give it as a command-line parameter to the grid node service, as described in Section 20.12.4, "Starting a Grid Node".

Adding Web Driver to System Path

The driver executable must be included in the operating system PATH or be given with a driver-specific system Java property:

- Google Chrome: `webdriver.chrome.driver`
- Mozilla Firefox: `webdriver.gecko.driver`
- Microsoft Edge: `webdriver.edge.driver`
- Internet Explorer: `webdriver.ie.driver`
- PhantomJS: `phantomjs.ghostdriver.path` and `phantomjs.binary.path`

You can set the property in Java with `System.setProperty(prop, key)` or pass it as a command-line parameter to the Java executable with `-Dwebdriver.chrome.driver=/path/to/driver`.

If you use an ESR version of Firefox, which is recommended for test stability, you need refer to the binary when creating the driver as follows:

```
FirefoxBinary binary =  
    new FirefoxBinary(new File("/path/to/firefox_ESR_45"));  
driver = TestBench.createDriver(  
    new FirefoxDriver(binary, new FirefoxProfile()));
```

Installing ChromeDriver for Ubuntu Chromium

While you can install Google Chrome in Ubuntu, it also has its own Chromium Browser, which is based on the Google Chrome. Chromium has its own version of ChromeDriver, which requires some additional installation steps to be usable.

Install the ChromeDriver:

```
$ sudo apt-get install chromium-chromedriver
```

Add the driver executable to path, such as:

```
$ sudo ln -s /usr/lib/chromium-browser/chromedriver /usr/local/bin/chromedriver
```

The Chromium libraries need to be included in the system library path:

```
$ sudo sh -c 'echo "/usr/lib/chromium-browser/libs" > /etc/ld.so.conf.d/chrome_libs.conf'
```

```
$ sudo ldconfig
```

20.4. Developing JUnit Tests

JUnit is a popular unit testing framework for Java development. Most Java IDEs, build systems, and continuous integration systems provide support for JUnit. However, while we concentrate on the development of JUnit tests in this chapter, Vaadin TestBench and the WebDriver are in no way specific to JUnit and you can use any test execution framework, or just regular Java applications, to develop TestBench tests.

You may want to keep your test classes in a separate source tree in your application project, or in an altogether separate project, so that you do not have to include them in the web application WAR. Having them in the same project may be nicer for version control purposes.

20.4.1. Basic Test Case Structure

A JUnit test case is defined with annotations for methods in a test case class. With TestBench, the test case class should extend the **TestBenchTestCase** class, which provides the WebDriver and ElementQuery APIs.

```
public class MyTestcase extends TestBenchTestCase {
```

The basic JUnit annotations used in TestBench testing are the following:

@Rule

You can define certain TestBench parameters and other JUnit rules with the @Rule annotation.

For example, to enable taking screenshots on test failures, as described in Section 20.10.2, “Taking Screenshots on Failure”, you would define:

```
@Rule  
public ScreenshotOnFailureRule screenshotOnFailureRule =  
    new ScreenshotOnFailureRule(this, true);
```

Note that if you use this rule, you must *not* call driver.quit() in your @After method, as the method is executed before the screenshot is taken, but the driver must be open to take it.

@Before

The annotated method is executed before each test (annotated with @Test). Normally, you create and set the driver here.

```
@Before  
public void setUp() throws Exception {  
    setDriver(new FirefoxDriver());  
}
```

The driver class should be one of **FirefoxDriver**, **ChromeDriver**, **InternetExplorerDriver**, **SafariDriver**, or **PhantomJSWebDriver**. Please check **RemoteWebDriver** from API documentation for the current list of implementations. Notice that some of the drivers require installing a browser driver, as described in Section 20.3.3, “Installing Browser Drivers”.

The driver instance is stored in the driver property in the test case. While you can access the property directly by the member variable, you should set it only with the setter.

@Test

Annotates a test method. You normally first open the page and then execute commands and make assertions on the content.

```
@Test
public void testClickButton() throws Exception {
    getDriver().get("http://localhost:8080/myproject");

    // Click the button
    ButtonElement button = $(ButtonElement.class).
        caption("Click Me").first();
    button.click();

    // Check that the label text is correct
    LabelElement label = $(LabelElement.class).first();
    assertEquals("Thanks!", label.getText());
}
```

Normally, you would define the URL with a variable that is common for all tests, and possibly concatenate it with a URI fragment to get to an application state.

@After

After each test is finished, you normally need to quit the driver to close the browser.

```
@After
public void tearDown() throws Exception {
    driver.quit();
}
```

However, if you enable grabbing screenshots on failure with the **ScreenshotOnFailureRule**, as described in Section 20.10.2, “Taking Screenshots on Failure”, the rules are executed after @After, but the driver needs to be open when the rule to take the screenshot is executed. Therefore, you should not quit the driver in that case. The rule quits the driver implicitly.

You can use any other JUnit features. Notice, however, that using TestBench requires that the driver has been created and is still open.

A complete test case could be as follows:

```

import com.vaadin.testbench.ScreenshotOnFailureRule;
import com.vaadin.testbench.TestBenchTestCase;
import com.vaadin.testbench.elements.ButtonElement;
import com.vaadin.testbench.elements.LabelElement;

import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.openqa.selenium.firefox.FirefoxDriver;

import java.util.List;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;

public class MyprojectTest extends TestBenchTestCase {
    @Rule
    public ScreenshotOnFailureRule screenshotOnFailureRule =
        new ScreenshotOnFailureRule(this, true);

    @Before
    public void setUp() throws Exception {
        setDriver(new FirefoxDriver()); // Firefox
    }

    /**
     * Opens the URL where the application is deployed.
     */
    private void openTestUrl() {
        getDriver().get("http://localhost:8080/myproject");
    }

    @Test
    public void testClickButton() throws Exception {
        openTestUrl();

        // At first there should be no labels
        assertFalse(${LabelElement.class}.exists());

        // Click the button
        ButtonElement clickMeButton = ${ButtonElement.class}.
            caption("Click Me").first();
        clickMeButton.click();

        // There should now be one label
        assertEquals(1, ${LabelElement.class}.all().size());

        // ... with the specified text
        assertEquals("Thank you for clicking".
            ${LabelElement.class}.first().getText());

        // Click the button again
        clickMeButton.click();

        // There should now be two labels
        List<LabelElement> allLabels =

```

```

$(LabelElement.class).all();
assertEquals(2, allLabels.size());

// ... and the last label should have the correct text
LabelElement lastLabel = allLabels.get(1);
assertEquals("Thank you for clicking",
           lastLabel.getText());
}
}

```

This test case stub is created by the Vaadin project wizard in Eclipse and by the Maven archetype, as described in Section 20.2, “Quickstart”.

20.4.2. Running JUnit Tests in Eclipse

The Eclipse IDE integrates JUnit with nice control features, such as running the tests in the current test source file. The test results are reported visually in the JUnit view in Eclipse.

New Vaadin projects created with the Vaadin Plugin for Eclipse contain the TestBench API dependency, as described in Section 20.2, “Quickstart”, so you can run TestBench tests right away.

To configure an existing project for TestBench testing, you need to do the following:

1. Include the TestBench API dependency in the project.
 - a. If using a project created with the Vaadin Plugin for Eclipse, add the TestBench API library dependency in ivy.xml. It should be as follows:

```

<dependency org="com.vaadin"
           name="vaadin-testbench-api"
           rev="latest.release"
           conf="nodeploy- & default"/>

```

The TestBench API library provides element classes for Vaadin components, so its revision number follows the earliest supported Vaadin release. For old Vaadin versions, you can try using the latest.release as given above.

The project should contain the nodeploy configuration, as created for new Vaadin projects. See ??? for more details.

- a. Otherwise, add the vaadin-testbench-api and vaadin-testbench-core JARs from the installation package to a library folder in the project, such as lib. You should not put the library in WEB-INF/lib as it is not used by the deployed Vaadin web application. Refresh the project by selecting it and pressing F5.
2. Right-click the project in Project Explorer and select **Properties**, and open the **Java Build Path** and the **Libraries** tab. Click **Add JARs**, navigate to the library folder, select the library, and click **OK**.
3. Switch to the **Order and Export** tab in the project properties. Make sure that the TestBench JAR is above the gwt-devjar (it may contain an old httpclient package), by selecting it and moving it with the **Up** and **Down** buttons.
4. Click **OK** to exit the project properties.
5. Right-click a test source file and select **Run As ➔ JUnit Test**.

A JUnit view should appear, and it should open the Firefox browser, launch the application, run the test, and then close the browser window. If all goes well, you have a passed test case, which is reported in the JUnit view area in Eclipse, as illustrated in Figure 20.4, "Running JUnit Tests in Eclipse".



Figure 20.4. Running JUnit Tests in Eclipse

If you are using some other IDE, it might support JUnit tests as well. If not, you can run the tests using Ant or Maven.

20.5. Creating a Test Case

20.5.1. Test Setup

Test configuration is done in a method annotated with @Before. The method is executed before each test case.

The basic configuration tasks are:

- Set TestBench parameters
- Create the web driver
- Do any other initialization

TestBench Parameters

TestBench parameters are defined with static methods in the **com.vaadin.testbench.Parameters** class. The parameters are mainly for screenshots and documented in Section 20.10, "Taking and Comparing Screenshots".

20.5.2. Basic Test Case Structure

A typical test case does the following:

1. Open the URL
2. Navigate to desired state .. Find a HTML element (**WebElement**) for interaction
 - a. Use click() and other commands to interact with the element
 - b. Repeat with different elements until desired state is reached
3. Find a HTML element (**WebElement**) to check
4. Get and assert the value of the HTML element
5. Get a screenshot

The **WebDriver** allows finding HTML elements in a page in various ways, for example, with XPath expressions. The access methods are defined statically in the **By** class.

These tasks are realized in the following test code:

```
@Test  
public void basic() throws Exception {  
    getDriver().get("http://localhost:8080/tobetested");  
  
    // Find an element to interact upon  
    ButtonElement button =  
        $(ButtonElement.class).id("mybutton");  
  
    // Click the button  
    button.click();  
  
    // Check that the label text is correct  
    LabelElement label = $(LabelElement.class).first();  
    assertEquals("Thanks!", label.getText());  
}
```

You can also use URI fragments in the URL to open the application at a specific state.

You should use the JUnit assertion commands. They are static methods defined in the org.junit.Assert class, which you can import (for example) with:

```
import static org.junit.Assert.assertEquals;
```

Please see the Selenium API documentation for a complete reference of the element search methods in the **WebDriver** and **By** classes and for the interaction commands in the **WebElement** class.

TestBench has a collection of its own commands, defined in the TestBenchCommands interface. You can get a command object that you can use by calling testBench(driver) in a test case.

While you can develop tests simply with test cases as described above, for the sake of maintainability it is often best to modularize the test code further, such as to separate testing at the levels of business logic and the page layout. See Sec-

tion 20.9, “Creating Maintainable Tests” for information about using page objects for this purpose.

20.5.3. Creating and Closing a Web Driver

Vaadin TestBench uses Selenium WebDriver to execute tests in a browser. The **WebDriver** instance is created with the static `createDriver()` method in the **TestBench** class. It takes the driver as the parameter and returns it after registering it. The test cases must extend the **TestBenchTestCase** class, which manages the TestBench-specific features. You need to store the driver in the test case with `setDriver()`.

The basic way is to create the driver in a method annotated with the JUnit `@Before` annotation and close it in a method annotated with `@After`.

```
public class AdvancedTest extends TestBenchTestCase {  
    @Before  
    public void setUp() throws Exception {  
        ...  
        setDriver(TestBench.createDriver(new FirefoxDriver()));  
    }  
    ...  
    @After  
    public void tearDown() throws Exception {  
        driver.quit();  
    }  
}
```

This creates the driver for each test you have in the test class, causing a new browser instance to be opened and closed. If you want to keep the browser open between the tests, you can use `@BeforeClass` and `@AfterClass` methods to create and quit the driver. In that case, the methods as well as the driver instance have to be static and you need to set the driver in a `@Before` method.

```
public class AdvancedTest extends TestBenchTestCase {  
    static private WebDriver driver;  
  
    @BeforeClass  
    static public void createDriver() throws Exception {  
        driver = TestBench.createDriver(new FirefoxDriver());  
    }  
  
    @Before  
    public void setUp() throws Exception {  
        setDriver(driver);  
    }  
}
```

```
}

...
@AfterClass
static public void tearDown() throws Exception {
    driver.quit();
}
}
```

Browser Drivers

Please see the API documentation of the WebDriver interface for a complete list of supported drivers, that is, classes implementing the interface.

All supported browsers require a special driver, as was noted in Section 20.3.3, “Installing Browser Drivers”.

20.6. Querying Elements

The high-level ElementQuery API allows querying Vaadin components in the browser according to their component class type, hierarchy, caption, and other properties. Once one or more components are found, they can be interacted upon. The query API forms an domain-specific language (DSL), embedded in the **TestBenchTestCase** class.

The basic idea of element queries match elements and return queries, which can again be queried upon, until terminated by a terminal query that returns one or more elements.

Consider the following query:

```
List<ButtonElement> buttons = $(ButtonElement.class).all();
```

The query returns a list of HTML elements of all the **Button** components in the UI. Every Vaadin component has its corresponding element class, which has methods to interact with the particular component type. We could control the buttons found by the query, for example, by clicking them as follows:

```
for (ButtonElement b: buttons)
    b.click();
```

In the following sub-sections, we look into the details of element queries.

20.6.1. Generating Queries with Debug Window

You can use the debug window to easily generate the element query code to select a particular element in the UI. This should be especially useful when starting to use TestBench, to get the idea what the queries should be like.

First, enable the debug window with the `&debug` parameter for the application, as described in more detail in Section 11.3, "Debug Mode and Window". You can interact with the UI in any way you like before generating the query code, but we recommend that you proceed by following the sequence in which the user would use the UI in each use case, making the queries at each step.

Switch to the TestBench tab in the debug window, and enable the pick mode by clicking the small button. Now, when you hover the mouse pointer on elements, it highlights them, and when you click one, it generates the TestBench element query to find the element. Use of the debug window is illustrated in Figure 20.5, "Using Debug Window to Generate Element Queries".



Figure 20.5. Using Debug Window to Generate Element Queries

You can select and copy and paste the code from the debug window to your editor. To exit the pick mode, click the pick button again.

The debug window feature is available in Vaadin 7.2 and later.

20.6.2. Querying Elements by Component Type (\$)

The \$ method creates an **ElementQuery** that looks for the given element class. The method is available both in **TestBenchTestcase** and **ElementQuery**, which defines the context. The search is done recursively in the context.

```
// Find the first OK button in the UI
ButtonElement button = $(ButtonElement.class)
    .caption("OK").first();

// A nested query where the context of the latter
// component type query is the matching elements
// - matches the first Label inside the "content" layout.
LabelElement label = $(VerticalLayoutElement.class)
    .id("content").$(LabelElement.class).first();
```

20.6.3. Non-Recursive Component Queries (\$\$)

The \$\$ method creates a non-recursive **ElementQuery**. It is a shorthand for first creating a recursive query with \$ and then calling recursive(false) for the query.

20.6.4. Element Classes

Each Vaadin component has a corresponding element class in TestBench, which contains methods for interacting with the particular component. The element classes extend **TestBenchElement**. It implements Selenium WebElement, so the Selenium element API can be used directly. The element classes are distributed in a Vaadin library rather than with TestBench, as they must correspond with the Vaadin version used in the application.

In addition to components, other Vaadin UI elements such as notifications (see Section 20.8.4, “Testing Notifications”) can have their corresponding element class. Add-on libraries may also define their custom element classes.

TestBenchElement is a TestBench command executor, so you can always use an element to create query in the sub-tree of the element. For example, in the following we first find a layout element by its ID and then do a sub-query to find the first label in it:

```
VBoxLayoutElement layout =  
    $(VBoxLayoutElement.class).id("content");  
LabelElement label = layout.$(LabelElement.class).first();
```

20.6.5. ElementQuery Objects

You can use an **ElementQuery** object to either make sub-queries to refine the query, or use a query terminator to finalize the query and get one or more matching elements.

20.6.6. Query Terminators

A query is finalized by a sub-query that returns an element or a collection of elements.

first()

Returns the first found element.

get()

Returns the element by index in the collection of matching elements.

all()

Returns a List of elements of the query type.

id()

Returns the unique element having the given ID. Element IDs must always be unique in the web page. It is therefore meaningless to make a complex query to match the ID, just matching the element class is enough.

Web Elements

A query returns one or more elements extending Selenium **WebElement**. The particular element-specific class offers methods to manipulate the associated Vaadin component, while you can also use the lower-level general-purpose methods defined in **WebElement**.

20.7. Element Selectors

In addition to the high-level ElementQuery API described in the previous section, Vaadin TestBench includes the lower-

level Selenium WebDriver API, with Vaadin extensions. You can find elements also by a plain XPath expression, an element ID, CSS style class, and so on. You can use such selectors together with the element queries. Like the ElementQuery API, it can be considered a domain-specific language (DSL) that is embedded in the **TestBenchTestCase** class.

The available selectors are defined as static methods in the **com.vaadin.testbench.By** class. They create and return a **By** instance, which you can use for the `findElement()` method in **WebDriver**.

The ID, CSS class, and Vaadin selectors are described below. For others, we refer to the Selenium WebDriver API documentation.

Some selectors are not applicable to all elements, for example if an element does not have an ID or is outside the Vaadin application.

20.7.1. Finding by ID

Selecting elements by their HTML element id attribute is a robust way to select elements, as noted in Section 20.9.1, “Increasing Selector Robustness”. It requires that you component IDs for the UI components with `setId()`.

```
Button button = new Button("Push Me!");
button.setId("pushmebutton");
```

The button would be rendered as a HTML element: `<div id="pushmebutton" ...>...</div>`. The element would then be accessible with a low-level WebDriver call:

```
findElement(By.id("pushmebutton")).click();
```

The selector is equivalent to the statically typed element query `$(ButtonElement.class).id("pushmebutton")`.

20.7.2. Finding by CSS Class

An element with a particular CSS style class name can be selected with the `By.className()` method. CSS selectors are useful for elements which have no ID, nor can be found easily

from the component hierarchy, but do have a particular unique CSS style. Tooltips are one example, as they are floating div elements under the root element of the application. Their v-tooltip style makes it possible to select them as follows:

```
// Verify that the tooltip contains the expected text
String tooltipText = findElement(
    By.className("v-tooltip")).getText();
```

you can find the complete example AdvancedCommandsIT-Case.java in TestBench demo.

20.8. Special Testing Topics

In the following, we go through a number of TestBench features for handling special cases, such as tooltips, scrolling, notifications, context menus, and profiling responsiveness. Finally, we look into the Page Object pattern.

20.8.1. Waiting for Vaadin

Selenium, on which Vaadin TestBench is based, is originally intended for regular web applications that load a page that is immediately rendered by the browser. In such applications, you can test the page elements immediately after the page is loaded. In Vaadin and other AJAX applications, rendering is done by JavaScript code asynchronously, so you need to wait until the server has given its response to an AJAX request and the JavaScript code finishes rendering the UI. Selenium supports AJAX applications by having special wait methods to poll the UI until the rendering is finished. In pure Selenium, you need to use the wait methods explicitly, and know what to use and when. Vaadin TestBench works together with the client-side engine of Vaadin framework to immediately detect when the rendering is finished. Waiting is implicit, so you do not normally need to insert any wait commands yourself.

Waiting is automatically enabled, but it may be necessary to disable it in some cases. You can do that by calling disableWaitForVaadin() in the TestBenchCommands interface. You can call it in a test case as follows:

```
testBench(driver).disableWaitForVaadin();
```

When disabled, you can wait for the rendering to finish by calling `waitForVaadin()` explicitly.

```
testBench(driver).waitForVaadin();
```

You can re-enable the waiting with `enableWaitForVaadin()` in the same interface.

20.8.2. Testing Tooltips

Component tooltips show when you hover the mouse over a component. Showing them require special command. Handling them is also special, as the tooltips are floating overlay element, which are not part of the normal component hierarchy.

Let us assume that you have set the tooltip as follows:

```
// Create a button with a component ID  
Button button = new Button("Push Me!");  
button.setId("main.button");  
  
// Set the tooltip  
button.setDescription("This is a tip");
```

The tooltip of a component is displayed with the `showTooltip()` method in the **TestBenchElementCommands** interface. You should wait a little to make sure it comes up. The floating tooltip element is not under the element of the component, but you can find it by `//div[@class='v-tooltip']` XPath expression.

```
@Test  
public void testTooltip() throws Exception {  
    driver.get(appUrl);  
  
    ButtonElement button =  
        $(ButtonElement.class).id("main.button");  
  
    button.showTooltip();  
  
    WebElement ttip = findElement(By.className("v-tooltip"));  
    assertEquals(ttip.getText(), "This is a tip");  
}
```

20.8.3. Scrolling

Some Vaadin components, such as **Table** and **Panel** have a scrollbar. Normally, when you interact with an element within such a scrolling region, TestBench implicitly tries to scroll to the element to make it visible. In some cases, you may wish to scroll a scrollbar explicitly. You can accomplish that with the scroll() (vertical) and scrollLeft() (horizontal) methods in the respective element classes for the scrollable components. The scroll position is given in pixels.

```
// Scroll to a vertical position
PanelElement panel = $(PanelElement.class)
    .caption("Scrolling Panel").first();
panel.scroll(123);
```

20.8.4. Testing Notifications

You can find notification elements by the **NotificationElement** class in the element query API. The element class supports getting the caption with getCaption(), description with getDescription(), and notification type with getType().

Let us assume that you pop the notification up as follows:

```
Button button = new Button("Pop It Up", e -> // Java 8
    Notification.show("The caption", "The description",
        Notification.Type.WARNING_MESSAGE));
```

You could then check for the notification as follows:

```
// Click the button to open the notification
ButtonElement button =
    $(ButtonElement.class).caption("Pop It Up").first();
button.click();

// Verify the notification
NotificationElement notification =
    $(NotificationElement.class).first();
assertEquals("The caption", notification.getCaption());
assertEquals("The description", notification.getDescription());
assertEquals("warning", notification.getType());
notification.close();
```

You need to close the notification box with close() to move forward.

20.8.5. Testing Context Menus

Opening context menus requires special handling. First, to open a menu, you need to "context-click" on a specific sub-element in a component that supports context menus. You can do that with a contextClick() action in a **Actions** object.

A context menu is displayed as a floating element, which is under a special overlays element in the HTML page, not under the component from which it pops up. You can find it from the page by its CSS class v-contextmenu. The menu items are represented as text, and you can find the text with an XPath expression as shown in the example below.

In the following example, we open a context menu in a **Table** component, find an item by its caption text, and click it.

```
// Get a table cell to work on
TableElement table = inExample(TableElement.class).first();
WebElement cell = table.getCell(3, 0); // A cell in the row

// Perform context click action to open the context menu
new Actions(getDriver()).contextClick(cell).perform();

// Find the opened menu
WebElement menu = findElement(By.className("v-contextmenu"));

// Find a specific menu item
WebElement menuitem = menu.findElement(
    By.xpath("//*[text() = 'Add Comment']]");

// Select the menu item
menuitem.click();
```

20.8.6. Profiling Test Execution Time

It is not just that it works, but also how long it takes. Profiling test execution times consistently is not trivial, as a test environment can have different kinds of latency and interference. For example in a distributed setup, timings taken on the test server would include the latencies between the test server, the grid hub, a grid node running the browser, and the web server running the application. In such a setup, you could also expect interference between multiple test nodes, which all might make requests to a shared application server and possibly also share virtual machine resources.

Furthermore, in Vaadin applications, there are two sides which need to be profiled: the server-side, on which the application logic is executed, and the client-side, where it is rendered in the browser. Vaadin TestBench includes methods for measuring execution time both on the server-side and the client-side.

The `TestBenchCommands` interface offers the following methods for profiling test execution time:

`totalTimeSpentServicingRequests()`

Returns the total time (in milliseconds) spent servicing requests in the application on the server-side. The timer starts when you first navigate to the application and hence start a new session. The time passes only when servicing requests for the particular session. The timer is shared in the servlet session, so if you have, for example, multiple portlets in the same application (session), their execution times will be included in the same total.
//TODO Vaadin 7: windows to roots

Notice that if you are also interested in the client-side performance for the last request, you must call the `timeSpentRenderingLastRequest()` before calling this method. This is due to the fact that this method makes an extra server request, which will cause an empty response to be rendered.

`timeSpentServicingLastRequest()`

Returns the time (in milliseconds) spent servicing the last request in the application on the server-side. Notice that not all user interaction through the WebDriver cause server requests.

As with the total above, if you are also interested in the client-side performance for the last request, you must call the `timeSpentRenderingLastRequest()` before calling this method.

`totalTimeSpentRendering()`

Returns the total time (in milliseconds) spent rendering the user interface of the application on the client-side, that is, in the browser. This time only passes when the browser is rendering after interacting with it through

the WebDriver. The timer is shared in the servlet session, so if you have, for example, multiple portlets in the same application (session), their execution times will be included in the same total.

timeSpentRenderingLastRequest()

Returns the time (in milliseconds) spent rendering user interface of the application after the last server request. Notice that not all user interaction through the WebDriver cause server requests.

If you also call the `timeSpentServicingLastRequest()` or `totalTimeSpentServicingRequests()`, you should do so before calling this method. The methods cause a server request, which will zero the rendering time measured by this method.

Generally, only interaction with fields in the *immediate* mode cause server requests. This includes button clicks. Some components, such as **Table**, also cause requests otherwise, such as when loading data while scrolling. Some interaction could cause multiple requests, such as when images are loaded from the server as the result of user interaction.

The following example is given in the `VerifyExecutionTimeIT-Case.java` file in the `TestBench` demo.

```
@Test
public void verifyServerExecutionTime() throws Exception {
    // Get start time on the server-side
    long currentSessionTime = testBench(getDriver())
        .totalTimeSpentServicingRequests();

    // Interact with the application
    calculateOnePlusTwo();

    // Calculate the passed processing time on the serve-side
    long timeSpentByServerForSimpleCalculation =
        testBench().totalTimeSpentServicingRequests() -
        currentSessionTime;

    // Report the timing
    System.out.println("Calculating 1+2 took about "
        + timeSpentByServerForSimpleCalculation
        + "ms in servlets service method.");

    // Fail if the processing time was critically long
    if (timeSpentByServerForSimpleCalculation > 30) {
```

```
        fail("Simple calculation shouldn't take " +
             timeSpentByServerForSimpleCalculation + "ms!");
    }

    // Do the same with rendering time
    long totalTimeSpentRendering =
        testBench().totalTimeSpentRendering();
    System.out.println("Rendering UI took "
        + totalTimeSpentRendering + "ms");
    if (totalTimeSpentRendering > 400) {
        fail("Rendering UI shouldn't take "
            + totalTimeSpentRendering + "ms!");
    }

    // A normal assertion on the UI state
    assertEquals("3.0",
        $(TextFieldElement.class).first()
            .getAttribute("value"));
}
```

20.9. Creating Maintainable Tests

The first important rule in developing tests is to keep them readable and maintainable. Otherwise, when the test fail, such as after refactoring the application code, the developers get impatient in trying to understand them to fix them, and easily disable them. Readability and maintainability can be improved with the Page Object Pattern described below.

The second rule is to run the tests often. It is best to use a continuous integration server to run them at least once a day, or preferably on every commit.

20.9.1. Increasing Selector Robustness

Robustness of tests is important for avoiding failures because of irrelevant changes in the HTML DOM tree. Different selectors have differences in their robustness and it depends on how they are used.

The ElementQuery API uses the logical widget hierarchy to find the HTML elements, instead of the exact HTML DOM structure. This makes them somewhat robust, although still vulnerable to irrelevant changes in the exact component hierarchy of the UI. Also, if you internationalize the application, selecting components by their caption is not viable.

The low-level XPath selector can be highly vulnerable to changes in the DOM path, especially if the path is given down from the body element of the page. The selector is, however, very flexible, and can be used in robust ways, for example, by selecting by HTML element and a CSS class name or an attribute value. You can likewise use a CSS selector to select specific components by CSS class in a robust way.

Using Component IDs to Increase Robustness

To make UIs more robust for testing, you can set a unique *component ID* for specific components with `setId()`, as described in more detail in Section 20.7.1, “Finding by ID”.

Let us consider the following application, in which we set the IDs using a hierarchical notation to ensure that they are unique; in a more modular case you could consider a different strategy.

```
public class UIToBeTested extends UI {  
    @Override  
    protected void init(VaadinRequest request) {  
        setId("myui");  
  
        final VerticalLayout content = new VerticalLayout();  
        content.setMargin(true);  
        content.setId("myui.content");  
        setContent(content);  
  
        // Create a button  
        Button button = new Button("Push Me!");  
  
        // Optional: give the button a unique ID  
        button.setId("myui.content.pushmebutton");  
  
        content.addComponent(button);  
    }  
}
```

After preparing the application this way, you can find the element by the component ID with the `id()` query terminator.

```
// Click the button  
ButtonElement button =  
    $(ButtonElement.class).id("myui.content.pushmebutton");  
button.click();
```

The IDs are HTML element id attributes and must be unique in the UI, as well as in the page in which the UI is running, in

case the page has other content than the particular UI instance. In case there could be multiple UIs, you can include a UI part in the ID, as we did in the example above.

Using CSS Class Names to Increase Robustness

As a similar method to using component IDs, you can add CSS class names to components with `addStyleName()`. This enables matching them with the `findElement(By.className())` selector, as described in Section 20.7.2, “Finding by CSS Class”. You can use the selector in element queries. Unlike IDs, CSS class names do not need to be unique, so an HTML page can have many elements with the same CSS class.

You can use CSS class names also in XPath selectors.

20.9.2. The Page Object Pattern

The Page Object Pattern aims to simplify and modularize testing application views. The pattern follows the design principle of separation of concerns, to handle different concerns in separate modules, while hiding information irrelevant to other tests by encapsulation.

Defining a Page Object

A *page object* has methods to interact with a view or a sub-view, and to retrieve values in the view. You also need a method to open the page and navigate to the proper view.

For example:

```
public class CalculatorPageObject
    extends TestBenchTestCase {
    @FindBy(id = "button_=")
    private WebElement equals;
    ...
    /**
     * Opens the URL where the calculator resides.
     */
    public void open() {
        getDriver().get(
            "http://localhost:8080/?restartApplication");
    }
}
```

```

    /**
     * Pushes buttons on the calculator
     *
     * @param buttons the buttons to push: "123+2", etc.
     * @return The same instance for method chaining.
     */
    public CalculatorPageObject enter(String buttons) {
        for (char numberChar : buttons.toCharArray()) {
            pushButton(numberChar);
        }
        return this;
    }

    /**
     * Pushes the specified button.
     *
     * @param button The character of the button to push.
     */
    private void pushButton(char button) {
        getDriver().findElement(
            By.id("button_" + button)).click();
    }

    /**
     * Pushes the equals button and returns the contents
     * of the calculator "display".
     *
     * @return The string (number) shown in the "display"
     */
    public String getResult() {
        equals.click();
        return display.getText();
    }

    ...
}

```

Finding Member Elements By ID

If you have **WebElement** members annotated with **@FindBy**, they can be automatically filled with the HTML element matching the given component ID, as if done with `driver.findElement(By.id(fieldname))`. To do so, you need to create the page object with **PageFactory** as is done in the following test setup:

```

public class PageObjectExampleITCase {
    private CalculatorPageObject calculator;

    @Before
    public void setUp() throws Exception {
        driver = TestBench.createDriver(new FirefoxDriver());
    }
}

```

```
// Use PageFactory to automatically initialize fields  
calculator = PageFactory.initElements(driver,  
    CalculatorPageObject.class);  
}  
...  
}
```

The members must be typed dynamically as **WebElement**, but you can wrap them to a typed element class with the `wrap()` method:

```
ButtonElement equals = equalsElement.wrap(ButtonElement.class);
```

Using a Page Object

Test cases can use the page object methods at business logic level, without knowing about the exact structure of the views.

For example:

```
@Test  
public void testAddCommentRowToLog() throws Exception {  
    calculator.open();  
  
    // Just do some math first  
    calculator.enter("1+2");  
  
    // Verify the result of the calculation  
    assertEquals("3.0", calculator.getResult());  
  
    ...  
}
```

The Page Object Example

You can find the complete example of the Page Object Pattern in the `src/test/java/com/vaadin/testbenchexample/pageobjectsexample/` folder in the TestBench Demo. The `PageObjectExampleITCase.java` runs tests on the Calc UI (also included in the example sources), using the page objects to interact with the different parts of the UI and to check the results.

The page objects included in the `pageobjects` subfolder are as follows:

- The **CalculatorPageObject** (as outlined in the example code above) has methods to click the buttons in the

calculator and the retrieve the result shown in the "display".

- The **LogPageObject** can retrieve the content of the log entries in the log table, and right-click them to open the comment sub-window.
- The **AddComment** can enter a comment string in the comment editor sub-window and submit it (click the **Add** button).

20.10. Taking and Comparing Screenshots

You can take and compare screenshots with reference screenshots taken earlier. If there are differences, you can fail the test case.

20.10.1. Screenshot Parameters

The screenshot configuration parameters are defined with static methods in the **com.vaadin.testbench.Parameters** class.

screenshotErrorDirectory(default: null)

Defines the directory where screenshots for failed tests or comparisons are stored.

screenshotReferenceDirectory(default: null)

Defines the directory where the reference images for screenshot comparison are stored.

screenshotComparisonTolerance(default: 0.01)

Screen comparison is usually not done with exact pixel values, because rendering in browser often has some tiny inconsistencies. Also image compression may cause small artifacts.

screenshotComparisonCursorDetection(default: false)

Some field component get a blinking cursor when they have the focus. The cursor can cause unnecessary failures depending on whether the blink happens to make the cursor visible or invisible when taking a screenshot. This parameter enables cursor detection that tries to minimize these failures.

`maxScreenshotRetries(default: 2)`

Sometimes a screenshot comparison may fail because the screen rendering has not yet finished, or there is a blinking cursor that is different from the reference screenshot. For these reasons, Vaadin TestBench retries the screenshot comparison for a number of times defined with this parameter.

`screenshotRetryDelay(default: 500)`

Delay in milliseconds for making a screenshot retry when a comparison fails.

For example:

```
@Before  
public void setUp() throws Exception {  
    Parameters.setScreenshotErrorDirectory(  
        "screenshots/errors");  
    Parameters.setScreenshotReferenceDirectory(  
        "screenshots/reference");  
    Parameters.setMaxScreenshotRetries(2);  
    Parameters.setScreenshotComparisonTolerance(1.0);  
    Parameters.setScreenshotRetryDelay(10);  
    Parameters.setScreenshotComparisonCursorDetection(true);  
    Parameters.setCaptureScreenshotOnFailure(true);  
}
```

20.10.2. Taking Screenshots on Failure

Vaadin TestBench can take screenshots automatically when a test fails. To enable the feature, you need to include the **ScreenshotOnFailureRule** JUnit rule with a member variable annotated with **@Rule** in the test case as follows:

```
@Rule  
public ScreenshotOnFailureRule screenshotOnFailureRule =  
    new ScreenshotOnFailureRule(this, true);
```

Notice that you must not call `quit()` for the driver in the `@After` method, as that would close the driver before the rule takes the screenshot.

The screenshots are written to the error directory defined with the `screenshotErrorDirectory` parameter. You can configure it in the test case setup as follows:

```
@Before  
public void setUp() throws Exception {
```

```
Parameters.setScreenshotErrorDirectory("screenshots/errors");
...
}
```

20.10.3. Taking Screenshots for Comparison

Vaadin TestBench allows taking screenshots of the web browser window with the `compareScreen()` command in the **TestBenchCommands** interface. The method has a number of variants.

The `compareScreen([classname]#File) #` takes a **File** object pointing to the reference image. In this case, a possible error image is written to the error directory with the same file name. You can get a file object to a reference image with the static `ImageFileUtil.getReferenceScreenshotFile()` helper method.

```
assertTrue("Screenshots differ",
    testBench(driver).compareScreen(
        ImageFileUtil.getReferenceScreenshotFile(
            "myshot.png")));

```

The `compareScreen([classname]#String) #` takes a base name of the screenshot. It is appended with browser identifier and the file extension.

```
assertTrue(testBench(driver).compareScreen("tooltip"));

```

The `compareScreen([classname]#BufferedImage,String) #` allows keeping the reference image in memory. An error image is written to a file with a name determined from the base name given as the second parameter.

Screenshots taken with the `compareScreen()` method are compared to a reference image stored in the reference image folder. If differences are found (or the reference image is missing), the comparison method returns false and stores the screenshot in the error folder. It also generates an HTML file that highlights the differing regions.

Screenshot Comparison Error Images

Screenshots with errors are written to the error folder, which is defined with the `screenshotErrorDirectory` parameter described in Section 20.10.1, "Screenshot Parameters".

For example, the error caused by a missing reference image could be written to screenshot/errors/tooltip_firefox_12.0.png. The image is shown in Figure 20.6. “A screenshot taken by a test run”.



Figure 20.6. A screenshot taken by a test run

Screenshots cover the visible page area in the browser. The size of the browser is therefore relevant for screenshot comparison. The browser is normally sized with a predefined default size. You can set the size of the browser window in a couple of ways. You can set the size of the browser window with, for example, `driver.manage().window().setSize(new Dimension(1024, 768))`: in the `@Before` method. The size includes any browser chrome, so the actual screenshot size will be smaller. To set the actual view area, you can use `TestBenchCommands.resizeViewPortTo(1024, 768)`.

Reference Images

Reference images are expected to be found in the reference image folder, as defined with the `screenshotReferenceDirectory` parameter described in Section 20.10.1, “Screenshot Parameters”. To create a reference image, just copy a screenshot from the errors/ directory to the reference/ directory.

For example:

```
$ cp screenshot/errors/tooltip_firefox_12.0.png screenshot/reference/
```

Now, when the proper reference image exists, rerunning the test outputs success:

```
$ java ...  
JUnit version 4.5  
  
Time: 18.222  
  
OK (1 test)
```

Masking Screenshots

You can make masked screenshot comparison with reference images that have non-opaque regions. Non-opaque pixels in the reference image, that is, ones with less than 1.0 value in the alpha channel, are ignored in the screenshot comparison.

Please see the ScreenshotITCase.java example in the TestBench Demo for an example of using masked screenshots. The example/Screenshot_Comparison_Tests.pdf document describes how to enable the example and how to create the screenshot masks in an image editor.

Visualization of Differences in Screenshots with Highlighting

Vaadin TestBench supports advanced difference visualization between a captured screenshot and the reference image. A difference report is written to a HTML file that has the same name as the failed screenshot, but with .html suffix. The reports are written to the same errors/ folder as the screenshots from the failed tests.

The differences in the images are highlighted with blue rectangles. Moving the mouse pointer over a square shows the difference area as it appears in the reference image. Clicking the image switches the entire view to the reference image and back. Text "**Image for this run**" is displayed in the top-left corner of the screenshot to distinguish it from the reference image.

Figure 20.7, “The reference image and a highlighted error image” shows a difference report with one difference between the visualized screenshot (bottom) and the reference image (top).

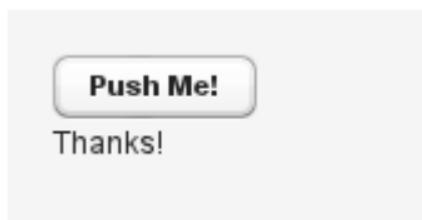


Figure 20.7. The reference image and a highlighted error image

20.10.4. Practices for Handling Screenshots

Access to the screenshot reference image directory should be arranged so that a developer who can view the results can copy the valid images to the reference directory. One possibility is to store the reference images in a version control system and check-out them to the reference/ directory.

A build system or a continuous integration system can be configured to automatically collect and store the screenshots as build artifacts.

20.10.5. Known Compatibility Problems

Screenshots when running Internet Explorer 9 in Compatibility Mode

Internet Explorer prior to version 9 adds a two-pixel border around the content area. Version 9 no longer does this and as a result screenshots taken using Internet Explorer 9 running in compatibility mode (IE7/IE8) will include the two pixel border, contrary to what the older versions of Internet Explorer do.

20.11. Running Tests

During test development, you usually run the tests from your IDE. After that, you want to have them run by a build system, possibly under a continuous integration system. In the following, we describe how to run tests by Ant and Maven.

20.11.1. Running Tests with Ant

Apache Ant has built-in support for executing JUnit tests; you can use the <junit> task in an Ant script to execute JUnit tests. Note that in earlier versions, you need to enable the support, you need to have the JUnit library junitjar and its Ant integration library ant-junitjar in the Ant classpath, as described in the Ant documentation.

The following Ant script allows testing a Vaadin application created with the Vaadin Plugin for Eclipse. It assumes that the test source files are located under a test directory under the current directory and compiles them to the classes directory. The the class path is defined with the classpath reference ID and should contain TestBench and other necessary libraries.

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="run-tests">
    <path id="classpath">
        <fileset dir="lib">
            <include name="vaadin-testbench-*jar"/>
            <include name="junit-*jar"/>
        </fileset>
    </path>

    <!-- This target compiles the JUnit tests. -->
    <target name="compile-tests">
        <mkdir dir="classes" />
        <javac srcdir="test" destdir="classes"
              debug="on" encoding="utf-8"
              includeantruntime="false">
            <classpath>
                <path refid="classpath" />
            </classpath>
        </javac>
    </target>

    <!-- This target calls JUnit -->
    <target name="run-tests" depends="compile-tests">
```

```

<junit fork="yes">
    <classpath>
        <path refid="classpath" />
        <pathelement path="classes" />
    </classpath>

    <formatter type="brief" usefile="false" />

    <batchtest>
        <fileset dir="test">
            <include name="**/**java" />
        </fileset>
    </batchtest>
</junit>
</target>
</project>

```

You also need to deploy the application to test, and possibly launch a dedicated server for it.

Retrieving TestBench with Ivy

To retrieve TestBench and its dependencies with Ivy in the Ant script, first install Ivy to your Ant installation, if necessary. In the build script, you need to enable Ivy with the namespace declaration and include a target for retrieving the libraries, as follows:

```

<project xmlns:ivy="antlib:org.apache.ivy.ant"
         default="run-tests">
    ...
    <!-- Retrieve dependencies with Ivy -->
    <target name="resolve">
        <ivy:retrieve conf="testing" type="jar.bundle"
                      pattern="lib/[artifact]-[revision].[ext]"/>
    </target>

    <!-- This target compiles the JUnit tests. -->
    <target name="compile-tests" depends="resolve">
    ...

```

This requires that you have a " testing" configuration in your ivy.xml and that the TestBench dependency are enabled in the configuration.

```

<ivy-module>
    ...
    <configurations>

```

```
...
<conf name="testing" />
</configurations>

<dependencies>
    ...
    <!-- TestBench 4 -->
    <dependency org="com.vaadin"
        name="vaadin-testbench-api"
        rev="latest.release"
        conf="nodeploy,testing -> default" />
    ...

```

You also need to build and deploy the application to be tested to the server and install the TestBench license key.

20.11.2. Running Tests with Maven

Executing JUnit tests with Vaadin TestBench under Maven requires defining it as a dependency in any POM that needs to execute TestBench tests.

A complete example of a Maven test setup is given in the TestBench demo project available at github.com/vaadin/test-bench-demo. See the README for further instructions.

Defining TestBench as a Dependency

You need to define the TestBench library as a dependency in the Maven POM of your project as follows:

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-testbench</artifactId>
    <version>4.x.x</version>
</dependency>
```

For instructions on how to create a new Vaadin project with Maven, please see Section 3.7, “Creating a Project with Maven”.

Running the Tests

To compile and run the tests, simply execute the test lifecycle phase with Maven as follows:

```
$ mvn test
```

```
...
```

```
-----
```

```
TESTS
```

```
-----  
Running TestBenchExample  
Tests run: 6, Failures: 1, Errors: 0, Skipped: 1, Time elapsed: 36.736 sec <<< FAILURE!
```

Results :

Failed tests:
testDemo(TestBenchExample):
 expected:<[5/17]12> but was:<[17.6.20]12>

Tests run: 6, Failures: 1, Errors: 0, Skipped: 1

The example configuration starts Jetty to run the application that is tested.

If you have screenshot tests enabled, as mentioned in ???, you will get failures from screenshot comparison. The failed screenshots are written to the target/testbench/errors folder. To enable comparing them to "expected" screenshots, you need to copy the screenshots to the src/test/resources/screenshots/reference/ folder. See Section 20.10, "Taking and Comparing Screenshots" for more information regarding screenshots.

20.12. Running Tests in a Distributed Environment

A distributed test environment consists of a test server, a grid hub and a number of test nodes.

The components of a grid setup are illustrated in Figure 20.8, "Vaadin TestBench Grid Setup".

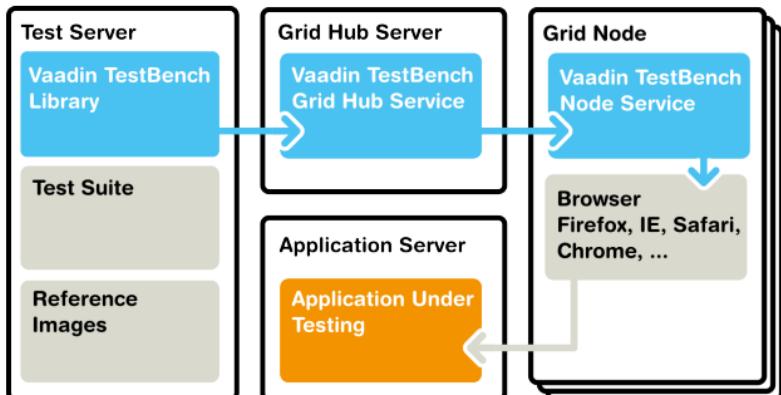


Figure 20.8. Vaadin TestBench Grid Setup

The grid hub is a service that handles communication between the JUnit test runner and the nodes. The hub listens to calls from test runners and delegates them to the grid nodes. The nodes are services that perform the actual execution of test commands in the browser. Different nodes can run on different operating system platforms and have different browsers installed.

The hub requires very little resources, so you would typically run it either in the test server or on one of the nodes. You can run the tests, the hub, and one node all in one host, but in a fully distributed setup, you install the Vaadin TestBench components on separate hosts.

Controlling browsers over a distributed setup requires using a remote WebDriver.

20.12.1. Running Tests Remotely

Remote tests are just like locally executed tests, except instead of using a browser driver, you use a remote web driver that can connect to the hub. The hub delegates the connection to a grid node with the desired capabilities, that is, which browsers are installed in the node.

Instead of creating and handling the remote driver explicitly, as described in the following, you can use the **ParallelTest** framework presented in Section 20.13, “Parallel Execution of Tests”.

An example of remote execution of tests is given in the Test-Bench demo described in [???](#). See the `README.md` file for further instructions.

In the following example, we create and use a remote driver that runs tests in a Selenium cloud at testingbot.com. The desired capabilities of a test node are described with a `DesiredCapabilities` object.

```
public class UsingHubITCase extends TestBenchTestCase {
```

```
private String baseUrl;  
private String clientKey = "INSERT-YOUR-CLIENT-KEY-HERE";  
private String clientSecret = "INSERT-YOUR-CLIENT-KEY-HERE";
```

@Before

```

public void setUp() throws Exception {
    // Create a RemoteDriver against the hub.
    // In you local setup you don't need key and secret.
    // but if you use service like testingbot.com, they
    // can be used for authentication
    URL testingbotdotcom = new URL("http://" +
        clientKey + ":" + clientSecret +
        "@hub.testingbot.com:4444/wd/hub");
    setDriver(new RemoteWebDriver(testingbotdotcom,
        DesiredCapabilities.iphone()));
    baseUrl = "http://demo.vaadin.com/Calc/";
}

@Test
@Ignore("Requires testingbot.com credentials")
public void testOnePlusTwo() throws Exception {
    // run the test just as with "local bots"
    openCalculator();
    $(ButtonElement.class).caption("1").first().click();
    $(ButtonElement.class).caption("+").first().click();
    $(ButtonElement.class).caption("2").first().click();
    $(ButtonElement.class).caption("=").first().click();
    assertEquals("3.0", $(TextFieldElement.class)
        .first().getAttribute("value"));

    // Thats it. Services may provide also some other goodies
    // like the video replay of your test in testingbot.com
}

private void openCalculator() {
    getDriver().get(baseUrl);
}

@After
public void tearDown() throws Exception {
    getDriver().quit();
}
}

```

Please see the API documentation of the **DesiredCapabilities** class for a complete list of supported capabilities.

Running the example requires that the hub service and the nodes are running. Starting them is described in the subsequent sections. Please refer to Selenium documentation for more detailed information.

20.12.2. Starting the Hub

The TestBench grid hub listens to calls from test runners and delegates them to the grid nodes. The grid hub service is included in the Vaadin TestBench JAR and you can start it with the following command:

```
$ java -jar vaadin-testbench-standalone-4.x.x.jar \
    -role hub
```

You can open the control interface of the hub also with a web browser. Using the default port, just open URL `http://localhost:4444/`. Once you have started one or more grid nodes, as instructed in the next section, the "console" page displays a list of the grid nodes with their browser capabilities.

20.12.3. Node Service Configuration

Test nodes can be configured with command-line options, as described later, or in a configuration file in JSON format. If no configuration file is provided, a default configuration is used.

A node configuration file is specified with the `-nodeConfig` parameter to the node service, for example as follows:

```
$ java -jar vaadin-testbench-standalone-4.x.x.jar
    -role node -nodeConfig nodeConfig.json
```

See Section 20.12.4, "Starting a Grid Node" for further details on starting the node service.

Configuration File Format

The test node configuration file follows the JSON format, which defines nested associative maps. An associative map is defined as a block enclosed in curly braces ({}). A mapping is a key-value pair separated with a colon (:). A key is a string literal quoted with double quotes ("key"). The value can be a string literal, list, or a nested associative map. A list a comma-separated sequence enclosed within square brackets ([]).

The top-level associative map should have two associations: capabilities (to a list of associative maps) and configuration (to a nested associative map).

```
{
  "capabilities": [
    {
      "browserName": "firefox",
      ...
    }
  ]
}
```

```
    },
    ...
],
"configuration":  
{
  "port": 5555,  

  ...
}  
}
```

A complete example is given later.

Browser Capabilities

The browser capabilities are defined as a list of associative maps as the value of the capabilities key. The capabilities can also be given from command-line using the `-browser` parameter, as described in Section 20.12.4, “Starting a Grid Node”.

The keys in the map are the following:

platform

The operating system platform of the test node: WINDOWS, XP, VISTA, LINUX, or MAC.

browserName

A browser identifier, any of: android, chrome, firefox, htmlunit, internet explorer, iphone, opera, or phantomjs (as of TestBench 3.1).

maxInstances

The maximum number of browser instances of this type open at the same time for parallel testing.

version

The major version number of the browser.

seleniumProtocol

This should be WebDriver for WebDriver use.

firefox_binary

Full path and file name of the Firefox executable. This is typically needed if you have Firefox ESR installed in a location that is not in the system path.

Server Configuration

The node service configuration is defined as a nested associative map as the value of the configuration key. The configuration parameters can also be given as command-line parameters to the node service, as described in Section 20.12.4, "Starting a Grid Node".

See the following example for a typical server configuration.

Example Configuration

```
{  
  "capabilities":  
  [  
    {  
      "browserName": "firefox",  
      "maxInstances": 5,  
      "seleniumProtocol": "WebDriver",  
      "version": "10",  
      "firefox_binary": "/path/to/firefox10"  
    },  
    {  
      "browserName": "firefox",  
      "maxInstances": 5,  
      "version": "16",  
      "firefox_binary": "/path/to/firefox16"  
    },  
    {  
      "browserName": "chrome",  
      "maxInstances": 5,  
      "seleniumProtocol": "WebDriver"  
    },  
    {  
      "platform": "WINDOWS",  
      "browserName": "internet explorer",  
      "maxInstances": 1,  
      "seleniumProtocol": "WebDriver"  
    }  
  ],  
  "configuration":  
  {  
    "proxy": "org.openqa.grid.selenium.proxy.DefaultRemoteProxy",  
    "maxSession": 5,  
    "port": 5555,  
    "host": ip,  
    "register": true,  
    "registerCycle": 5000,  
    "hubPort": 4444  
  }  
}
```

20.12.4. Starting a Grid Node

A TestBench grid node listens to calls from the hub and is capable of opening a browser. The grid node service is included in the Vaadin TestBench JAR and you can start it with the following command:

```
$ java -jar \
vaadin-testbench-standalone-4.x.x.jar \
-role node \
-hub http://localhost:4444/grid/register
```

The node registers itself in the grid hub. You need to give the address of the hub either with the *-hub* parameter or in the node configuration file as described in Section 20.12.3, "Node Service Configuration".

You can run one grid node in the same host as the hub, as is done in the example above with the localhost address.

Operating system settings

Make any operating system settings that might interfere with the browser and how it is opened or closed. Typical problems include crash handler dialogs.

On Windows, disable error reporting in case a browser crashes as follows:

1. Open **Control Panel** → **System**
2. Select the **Advanced** tab
3. Select **Error reporting**
4. Check that **Disable error reporting** is selected
5. Check that **But notify me when critical errors occur** is not selected

Settings for Screenshots

The screenshot comparison feature requires that the user interface of the browser stays constant. The exact features that

interfere with testing depend on the browser and the operating system.

In general:

- Disable blinking cursor
- Use identical operating system themeing on every host
- Turn off any software that may suddenly pop up a new window
- Turn off screen saver

If using Windows and Internet Explorer, you should give also the following setting:

- Turn on **Allow active content to run in files on My Computer** under **Security settings**

Browser Capabilities

The browsers installed in the node can be defined either with command-line parameters or with a configuration file in JSON format, as described in Section 20.12.3, “Node Service Configuration”.

On command-line, you can issue one or more *-browser* options to define the browser capabilities. It must be followed by a comma-separated list of property-value definitions, such as the following:

```
-browser "browserName=firefox,version=10,firefox_binary=/path/to/firefox10"\n-browser "browserName=firefox,version=16,firefox_binary=/path/to/firefox16"\n-browser "browserName=chrome,maxInstances=5"\n-browser "browserName=internet explorer,maxInstances=1,platform=WINDOWS"
```

The configuration properties are described in Section 20.12.3, “Node Service Configuration”.

Browser Driver Parameters

If you use Chrome or Internet Explorer, their remote driver executables must be in the system path (in the PATH environment variable) or be given with a command-line parameter to the node service:

Internet Explorer

-Dwebdriver.ie.driver=C:\path\to\IEDriverServer.exe

Google Chrome

-Dwebdriver.chrome.driver=/path/to/ChromeDriver

20.12.5. Mobile Testing

Vaadin TestBench includes an iPhone and an Android driver, with which you can test on mobile devices. The tests can be run either in a device or in an emulator/simulator.

The actual testing is just like with any WebDriver, using either the **iPhoneDriver** or the **AndroidDriver**. The Android driver assumes that the hub (android-server) is installed in the emulator and forwarded to port 8080 in localhost, while the iPhone driver assumes port 3001. You can also use the **RemoteWebDriver** with either the `iphone()` or the `android()` capability, and specify the hub URI explicitly.

The mobile testing setup is covered in detail in the Selenium documentation for both the iOS driver and the AndroidDriver.

20.13. Parallel Execution of Tests

The **ParallelTest** class provides an easy way to run tests in parallel locally, as well as remotely in a test grid.

To enable parallel execution of tests, usually during test development, you need to extend **ParallelTest** instead of **TestBenchTestCase**. **ParallelTest** will automatically:

- Grab a screenshot if the test fails
- Terminate the driver after the test ends (remove any calls to `WebDriver.quit()` you might have in your test)
- Take any `@RunLocally` or `@RunOnHub` annotations on the test class, and the corresponding system properties, into account

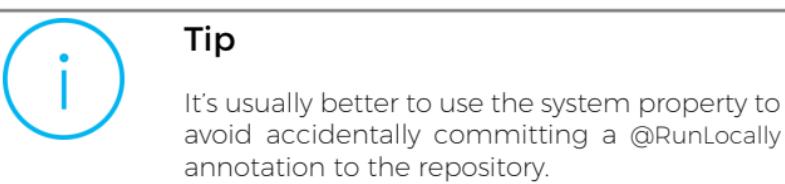
For a parallel test, you **must** define the hub it should be run on (using `@RunOnHub` or `-Dcom.vaadin.testbench.Parameters.hostname=[hubHost]`) or that it should run locally on a given

browser (using @RunLocally or -Dcom.vaadin.testbench.Parameters.runLocally=[browser]-[version]). The system property contains either a browserName, which should (case insensitively) match an enum in com.vaadin.testbench.parallel.Browser or a name-version combination. It can for example be edge or chrome-45.

20.13.1. Local Parallel Execution

When you add a @RunLocally annotation on the test class or define the -Dcom.vaadin.testbench.Parameters.runLocally=[browser]-[version] system property, the ParallelTest will be run using a local browser regardless of any @RunOnHub definition.

```
@RunLocally  
public class MyTest extends ParallelTest {  
    @Test  
    ...  
}
```



When you run the tests, TestBench launches multiple browser windows to run each test in parallel.

The default browser to run locally on is Firefox. You can give another browser as a parameter for the annotation, e.g.

```
@RunLocally(Browser.CHROME)
```

For Chrome and IE, you need to have the browser driver installed, as described in Section 20.3.3, "Installing Browser Drivers".

20.13.2. Multi-Browser Execution in a Grid

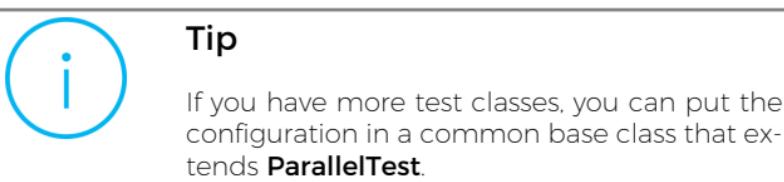
To run tests in multiple different browsers or remotely, you first need to set up and launch a grid hub and one or more grid nodes, as described in Section 20.12, "Running Tests in a Distributed Environment". This enables remote execution in

a test grid, although you can run the hub and a test node also in your development workstation.

To run a test case class in a grid, you simply need to annotate the test case classes with the @RunOnHub annotation or using -Dcom.vaadin.testbench.Parameters.hubHostname=[hubHost]). Both the annotation and the system parameter takes the host address of the hub as a parameter. You need to define the browsers to run on in a method annotated with @BrowserConfiguration. It must return a list of **DesiredCapabilities**.

```
@RunOnHub("hub.testgrid.mydomain.com")
public class MyTest extends ParallelTest {
    @Test
    ...
    @BrowserConfiguration
    public List<DesiredCapabilities> getBrowserConfiguration() {
        List<DesiredCapabilities> browsers =
            new ArrayList<DesiredCapabilities>();
        // Add all the browsers you want to test
        browsers.add(BrowserUtil.firefox());
        browsers.add(BrowserUtil.chrome());
        browsers.add(BrowserUtil.ie11());
        return browsers;
    }
}
```

The actual browsers tested depends on the browser capabilities of the test node or nodes.



20.14. Headless Testing

TestBench (3.1 and later) supports fully-featured headless testing with PhantomJS (<http://phantomjs.org>), a headless browser based on WebKit. It has fast native support for various web standards: JavaScript, DOM handling, CSS selector, JSON, Canvas, and SVG.

Headless testing using PhantomJS allows for around 15% faster test execution without having to start a graphical web browser, even when performing screenshot-based testing! This also makes it possible to run full-scale functional tests on the front-end directly on a build server, without the need to install any web browsers.

It is usually best to use a graphical browser to develop the test cases, as it is possible to see interactively what happens while the tests are being executed. Once the tests are working correctly in a graphical browser, you can migrate them to run on the PhantomJS headless browser.

20.14.1. Basic Setup for Running Headless Tests

The only set up required is to install the PhantomJS binary. Follow the instructions for your operating system at PhantomJS download page, and place the binary in the system path.

The PhantomJSDriver dependency is already included in Vaadin TestBench.

Creating a Headless WebDriver Instance

Creating an instance of the **PhantomJSDriver** is just as easy as creating an instance of **FirefoxDriver**.

```
setDriver(TestBench.createDriver(  
    new PhantomJSDriver()));
```

Some tests may fail because of the small default window size in PhantomJS. Such tests are, for example, tests containing elements that pop up and might go off-screen when the window is small. To make them work better, specify a size for the window:

```
getDriver().manage().window().setSize(  
    new Dimension(1024, 768));
```

Nothing else is needed to run tests headlessly.

20.14.2. Running Headless Tests in a Distributed Environment

Running PhantomJS in a distributed grid is equally easy. First, install PhantomJS in the nodes by following the instructions in Section 20.14.1, “Basic Setup for Running Headless Tests”. Then, start PhantomJS using the following command:

```
phantomjs --webdriver=8080 \
--webdriver-selenium-grid-hub=http://127.0.0.1:4444
```

The above will start PhantomJS in the WebDriver mode and register it with a grid hub running at 127.0.0.1:4444. After this, running tests in the grid is as easy as passing DesiredCapabilities.phantomjs() to the RemoteWebDriver constructor.

```
setDriver(new RemoteWebDriver(
    DesiredCapabilities.phantomjs()));
```

20.15. Behaviour-Driven Development

Behaviour-driven development (BDD) is a development methodology based on test-driven development, where development starts from writing tests for the software-to-be. BDD involves using a *ubiquitous language* to communicate between business goals - the desired behaviour - and tests to ensure that the software fulfills those goals.

The BDD process starts by collection of business requirements expressed as *user stories*, as is typical in agile methodologies. A user with a *role* requests a *feature* to gain a *benefit*.

Stories can be expressed as number of *scenarios* that describe different cases of the desired behaviour. Such a scenario can be formalized with the following three phases:

- *Given* that I have calculator open
- *When* I push calculator buttons
- *Then* the display should show the result

This kind of formalization is realized in the JBehave BDD framework for Java. The TestBench Demo includes a JBehave

example, where the above scenario is written as the following test class:

```
public class CalculatorSteps extends TestBenchTestCase {  
    private WebDriver driver;  
    private CalculatorPageObject calculator;  
  
    @BeforeScenario  
    public void setUpWebDriver() {  
        driver = TestBench.createDriver(new FirefoxDriver());  
        calculator = PageFactory.initElements(driver,  
            CalculatorPageObject.class);  
    }  
  
    @AfterScenario  
    public void tearDownWebDriver() {  
        driver.quit();  
    }  
  
    @Given("I have the calculator open")  
    public void theCalculatorIsOpen() {  
        calculator.open();  
    }  
  
    @When("I push $buttons")  
    public void enter(String buttons) {  
        calculator.enter(buttons);  
    }  
  
    @Then("the display should show $result")  
    public void displayShows(String result) {  
        assertEquals(result, calculator.getResult());  
    }  
}
```

The demo employs the page object defined for the application UI, as described in Section 20.9.2, “The Page Object Pattern”.

Such scenarios are included in one or more stories, which need to be configured in a class extending **JUnitStory** or **JUnitStories**. In the example, this is done in the <https://github.com/vaadin/testbench-demo/blob/master/src/test/java/com/vaadin/testbenchexample/bdd/SimpleCalculation.java> class. It defines how story classes can be found dynamically by the class loader and how stories are reported.

For further documentation, please see JBehave website at jbehave.org.

20.16. Integration Testing with Maven

TestBench is often used in Maven projects, where you want to run tests as part of the build lifecycle. While ordinary Java unit tests are usually executed in the test phase, TestBench tests are usually executed in the integration-test phase (to run the phase properly you need to invoke verify phase as explained later).

??? describes how to use the Vaadin application archetype for Maven provides a TestBench setup. In this section, we largely go through that and also describe how to make such a setup also in Vaadin add-on projects.

20.16.1. Project Structure

In Vaadin Applications

In a typical Vaadin application, such as in the one created by the archetype, you only have one module and you run tests there. The application sources would normally be in src/main and test code, together with TestBench tests, in src/test.

In Libraries and Add-ons

In multi-module projects, you may have libraries in other modules, and the actual web application in another. Here you could do library unit tests in the library modules, and integration tests in the web application module.

The multi-module project structure is recommended for Vaadin add-ons, where you have the add-on library in one module and a demo in another.

20.16.2. Overview of Lifecycle

The Maven lifecycle phases relevant to TestBench execution are as follows:

1. compile
 - Compile server-side
 - Compile widget set

- Compile theme
2. test-compile
 - Compile test classes (both unit and integration tests)
 3. pre-integration-test
 - Start web server (Jetty or other)
 4. integration-test
 - Execute TestBench tests
 5. post-integration-test
 - Stop web server
 6. verify
 - Verify the results of the integration tests

20.16.3. Overview of Configuration

The Maven configuration in the pom.xml should include the following parts, with our reference toolchain given in parentheses:

- Integration test runner (Maven Failsafe Plugin)
- Web server (Jetty)
- Web server runner (Jetty Maven Plugin)
- Vaadin compilation and deployment (Vaadin Maven Plugin)

20.16.4. Vaadin Plugin Configuration

The Vaadin Maven Plugin compiles the widget set with the Vaadin Client Compiler and the theme with the Vaadin Sass compiler in the compile phase. Its configuration should be as is normal for Vaadin projects. The default configuration is created by the Vaadin project archetype, as described in Section 3.7, “Creating a Project with Maven”.

20.16.5. Configuring Integration Testing

Our reference toolchain uses the Maven Failsafe Plugin to execute integration tests with TestBench. The plugin executes tests (JUnit or other) defined in the test files included in the plugin configuration.

To run TestBench tests made with JUnit, you need that dependency in the `<dependencies>` section:

```
<!-- Needed for running TestBench tests -->
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.12</version>
<scope>test</scope>
</dependency>
```

Surefire requires the following plugin configuration (under `<plugins>`):

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-failsafe-plugin</artifactId>
<version>2.17</version>
<configuration>
<includes>
<include>**Tests.java</include>
</includes>
<excludes>
</excludes>
<!-- Here list files that might match to naming conventions unintentionally. We can ignore them from testing. -->
</excludes>
</configuration>
<executions>
<execution>
<id>failsafe-integration-tests</id>
<phase>integration-test</phase>
<goals>
<goal>integration-test</goal>
</goals>
</execution>
<execution>
<id>failsafe-verify</id>
<phase>verify</phase>
<goals>
<goal>verify</goal>
</goals>
</execution>
</executions>
</plugin>
```

Set the include and exclude patterns according to your naming convention. See Failsafe documentation for more details about the configuration, for example, if you want to use other test provider than JUnit.

20.16.6. Configuring Test Server

We use Jetty as our reference test server, as it is a light-weight server that is easy to configure with a Maven plugin.

The dependency for Jetty goes as follows:

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-webapp</artifactId>
  <version>9.2.10.v20150310</version>
  <scope>test</scope>
</dependency>
```

The plugin configuration for running Jetty goes as follows:

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>8.1.16.v20140903</version>

  <configuration>
    <webApp>
      <contextPath>/myapp</contextPath>
    </webApp>
    <stopKey>STOP</stopKey>
    <stopPort>8005</stopPort>
  </configuration>

  <executions>
    <execution>
      <id>start-jetty</id>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>start</goal>
      </goals>
      <configuration>
        <daemon>true</daemon>
        <scanIntervalSeconds>0</scanIntervalSeconds>
      </configuration>
    </execution>
    <execution>
      <id>stop-jetty</id>
      <phase>post-integration-test</phase>
      <goals>
        <goal>stop</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Here you only need to configure the *contextPath* parameter, which is the context path to your web application.

20.17. Known Issues

This section provides information and instructions on a few features that are known to be difficult to use or need modification to work.

20.17.1. Latest Firefox Compatibility

Mozilla Firefox changed important APIs in version 48 that affected their compatibility with WebDriver which directly affects TestBench. Firefox 48 and posterior versions will require the use of GeckoDriver that is not feature complete or fully compatible with Selenium currently. As a workaround it is recommended to use either Firefox 47 or Firefox ESR (currently at 45.3) to execute tests on Firefox until this can be fixed with a stable release of GeckoDriver.

20.17.2. Running Firefox Tests on Mac OS X

Firefox needs to have focus in the main window for any focus events to be triggered. This sometimes causes problems if something interferes with the focus. For example, a **TextField** that has an input prompt relies on the JavaScript onFocus() event to clear the prompt when the field is focused.

The problem occurs when OS X considers the Java process of an application using TestBench (or the node service) to have a native user interface capability, as with AWT or Swing, even when they are not used. This causes the focus to switch from Firefox to the process using TestBench, causing tests requiring focus to fail. To remedy this problem, you need to start the JVM in which the tests are running with the `-Djava.awt.headless=true` parameter to disable the user interface capability of the Java process.

Note that the same problem is present also when debugging tests with Firefox. We therefore recommend using Chrome for debugging tests, unless Firefox is necessary.

20.17.3. Random Failures with Internet Explorer Driver

Unfortunately, a TestBench test may sometimes behave unexpectedly, causing random failures. One of the reasons for these random failures is problems with the Internet Explorer Driver. Internet Explorer specification requires browser window to be focused to receive user events, while Selenium Web Driver specification points out that Web Driver should work without window focus. You can get more information about it from Internet Explorer Driver documentation.

TestBench provides a workaround to this problem. Use **RetryRule(int maxAttempts)** rule to re-run a test several times in case of a random failure. Use *maxAttempts* to specify the maximum number of attempts for running tests. The test passes as soon as one attempt is executed without any errors, i.e. it is only run as many times as needed.

```
public class RandomFailureTest extends TestBenchTestCase {  
    // Run the test two times  
    @Rule  
    public RetryRule rule = new RetryRule(2);  
  
    @Test  
    public void emptyTest() {  
    }  
}
```



Note

RetryRule affects all the test methods in the class and also child classes.



Note

The default value of *maxAttempts* is 1, meaning that tests run only once. You can change the value of *maxAttempts* using the Java system property: *-Dcom.vaadin.testbench.Parameters.maxAttempts=2*. This will affect all the TestBench tests.



Note

Use **RetryRule** when you are sure that the test fails because of the problems with the Web Driver, but not your application. Using **RetryRule** without cautions may hide random problems happening in your application.

20.18. TestBench 4 to TestBench 5 Migration Guide

20.18.1. Introduction

Vaadin TestBench 5 is intended to test Vaadin Framework 8 applications.

Vaadin TestBench 5 is API compatible with Vaadin TestBench 4.x, and can thus be used as a drop-in replacement for any 4.x version provided the tested project has been migrated to Vaadin Framework 8, see Vaadin Framework 8 migration guide. The exceptions to this rule are listed in the API changes section below.

20.18.2. API Changes in TestBench 5

- **CheckBoxGroupElement** added to test **CheckBoxGroup** Component
- **RadioButtonGroupElement** added to test **RadioButtonGroup** Component

20.18.3. Testing Deprecated Vaadin Components

Vaadin Framework 8 provides an optional dependency that includes compatibility components to facilitate the migration between versions 7 and 8.

Compatibility Only Components

Some components only exist in com.vaadin.v7.ui compatibility package like:

- **Table**
- **Tree**
- **TreeTable**
- **Form**
- **PopupDateField**
- **Calendar**
- **OptionGroup**
- **ProgressIndicator**
- **Select**

Such components and their corresponding **TestBenchElement** have been deprecated but can still be used and tested.

Compatibility and Vaadin 8 Components

Other components exist in both com.vaadin.v7.ui and com.vaadin.ui packages. In these cases the same **TestBenchElement** can be used to test either version.

One example of this is **NativeSelectElement** that can be used to test both com.vaadin.v7.ui **NativeSelect** and com.vaadin.ui **NativeSelect**

20.18.4. Changes in dependency

Selenium

Selenium version was upgraded to 3.0.1. TestBench 5 should still work with older (Selenium 2 / TestBench 4 based) test clusters.

Set Index

Symbols

@ApplicationScoped, 471
@CDIUI, 468-469
@Connect, 540
@PreserveOnRefresh, 103
@SessionScoped, 471
@SpringUI, 479
@UIScoped, 470

A

AbstractComponent, 120, 123
AbstractComponentContainer, 121
AbstractComponentState, 540
AbstractField, 121
AbstractListing, 121
add-ons
 creating, 561-564
AJAX, 8, 61
Alignment, 258-261

B

BrowserWindowOpener, 102

C

caption, 124
caption property, 124
CDI, 465-474
 scopes, 469-472
Client-Side Engine, 58, 64
close(), 106
 UI, 104

closableSessions, 105, 114
closing, 104, 106
compatibility, 333
component, 57
Component, 120
Component interface, 122-123
 caption, 124
 description, 126
 enabled, 127
 icon, 128
 locale, 129
 read-only, 132
 style name, 133
 visible, 134
connector, 539
context menus, 705
Contexts and Dependency Injection, 465-474
CSS, 58, 60-61, 321-363
 compatibility, 333
 introduction, 323-333
CSS selections, 362
CSS3, 61
 custom, 103

D

Data Binding, 59
Data Model, 59
DefaultUIProvider, 102
description, 126
description property, 126
DOM, 60-61
Drag and Drop, 437-449
 Accept Criteria, 442-447

E

Eclipse
 widget development, 541-545
enabled, 127
enabled property, 127

events, 58
execute(), 418
expiration, 104-106
extension, 362

F

field, 57
Field, 137-141

G

getLocale(), 130
Google Web Toolkit, 3, 8, 58, 60, 62, 208
theming, 326
widgets, 537-570

H

heartbeat, 106
HorizontalSplitPanel, 240-242
HTML, 60
HTML 5, 61
HTML templates, 58
HTTP, 58

I

icon, 128
icon property, 128
in Component, 129
init(), 103
interfaces, 121
IPC add-on, 507-515
IT Mill Toolkit, 8

J

JavaDoc, 121
JavaScript, 3, 60-61, 541
execute(), 418
print(), 418-419

JavaScript integration, 565-570

L

Liferay
display descriptor, 503-504
plugin properties, 504-505
portlet descriptor, 502
liferay-display.xml, 503-504
liferay-plugin-package.xml, 504-505
loading, 102
locale, 129
locale property
in Component, 129
Log4j, 450
logout, 106

M

Maven
compiling, 52
creating a project, 50-53
using add-ons, 53, 573-582

N

Notification
testing, 704

O

Out of Sync, 114
overflow, 239
overflow CSS property, 211, 234

P

Page
setLocation(), 106
PDF, 419
popup, 397

popup windows, 397
portal integration, 485-515
preserving on refresh, 103
print(), 418-419
printing, 418-420
push, 106

R

read-only, 132
read-only property, 132
redirection, 105-106
responsive extension, 362-363

S

Sampler, 121
Sass, 58, 60
scroll bars, 211, 234-235, 239
Scrollable, 235, 239
scrolling, 704
SCSS, 61
Serializable, 122
server push, 58, 106
servletInitialized(), 102
session, 101
 closing, 106
 expiration, 105-106
 timeout, 105
session-timeout, 114
SessionDestroyListener, 102, 105
SessionInitListener, 102
setComponentAlignment(), 258-261
setLocation(), 106
Sizeable interface, 135
SLF4J, 450
Spreadsheet, 659-673
Spring, 474-483
 scopes, 479-480
state object, 540
style name, 133

style name property, 133
system messages, 106

T

TestBenchElement, 699
testing, 704
Text change events, 154-156
TextField, 152-156
theme, 58, 321-363
themeing, 326
themes, 61
ThreadLocal pattern, 457-458
timeout, 105
 tooltips, 126

U

UI, 104
 closing, 104
 expiration, 104
 heartbeat, 106
 loading, 102
 preserving on refresh, 103
UIProvider, 102
 custom, 103

V

Vaadin 6 Migration
 add-ons, 564-565
Vaadin CDI Add-on, 465-474
Vaadin Data Model, 365-393
Vaadin Designer, 267-292
Vaadin Spring, 474-483
VaadinRequest, 103
VaadinService, 102
VaadinServlet, 58, 102
ValueChangeMode, 154
VerticalSplitPanel, 240-242
visible, 134
visible property, 134

W

widget, 57
widget, definition, 518
widgets, 537
windows
 popup, 397

X

XML, 61
XMLHttpRequest, 61