BOOK OF VAADIN

Vaadin 14 Edition

vaadin}>

Book of Vaadin

Vaadin Team
Vaadin Ltd

Revision 1
Published: 2019-07-05
Vaadin 14.0

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.

Table of Contents

Preface

This book provides an overview of Vaadin 14 (released in mid-2019). It covers the topics that you encounter while developing applications with Vaadin. The book is a compilation of the most important documentation available at https://vaadin.com/docs. Detailed documentation about the individual classes, interfaces, and methods is available in the Vaadin API Reference at https://vaadin.com/api.

Downloadable versions of the book, in various formats, are available at https://vaadin.com. These are easier to search than the printed book. The web edition also has additional technical content, such as code examples and additional sections that you may need when developing applications. The slightly abridged print edition is intended to be an introductory textbook to Vaadin that fits in your pocket.

Who is this Book for?

This book is aimed at software developers who use, or are considering using, Vaadin to develop web applications.

The book assumes that you have some experience with programming in Java, but this is not essential. It is as easy to learn Java with Vaadin as it is with any other UI framework. Knowledge of desktop-oriented user interface (UI) frameworks for Java, such as AWT, Swing, or SWT, or libraries such as Qt for C++, is also useful, but is not necessary, to understand the scope of Vaadin, the event-driven programming model, and other common concepts of UI frameworks.

Basic HTML and CSS knowledge can help you when developing client-side components and presentation themes for your application, but it is also not required to understand the concepts in this book.

Book of Vaadin PDF Version

This book is available as hard copy and PDF. You can download the PDF version for free at vaadin.com/book. The PDF version contains all the content in the hard copy version plus more advanced topics.

Some of the additional topics covered in PDF version are:

- Creating Vaadin Components
- Manipulating DOM with Element API
- Integrating Web Components
- Packaging applications for production
- OSGi Support
- Migrating from Vaadin 8 to Vaadin 10
- Migrating from Vaadin 10-13 to Vaadin 14
- Vaadin Designer
- Vaadin Charts
- Vaadin TestBench
- Vaadin Multiplatform Runtime
- Advanced Topics

Supplementary Material

The Vaadin website offers plenty of material to help you understand what Vaadin is, what you can do with it, and how you can do it. Here’s a list of additional resources you can find at https://vaadin.com:

- https://vaadin.com/tutorials/getting-started-with-flow: A step-by-step tutorial that shows you how to create a Java web application using Vaadin.
- https://vaadin.com/components: The list of available UI components, including code snippets and common usage scenarios.
- https://vaadin.com/tutorials: Short hands-on tutorials with full code examples.
- https://vaadin.com/training/courses: Online training video courses.
- https://github.com/vaadin: Vaadin developer’s site on GitHub which hosts the source code, issue system for reporting bugs and suggesting improvements, releases, and activity timeline.

Getting Support

If you get stuck with a problem, the Vaadin Community and the Vaadin company are there to support all of your needs.

The public developer community forum is at https://vaadin.com/forum. Please use this forum to discuss any problems you encounter. The answer to your question may already be in the forum archives, so searching the discussions is typically the best way to begin. Always make sure the information you find matches your version of Vaadin. If you can’t find a solution for the version you are using, don’t hesitate to ask for one: the Vaadin community is highly active and its members are always ready to help others. When you gain more Vaadin experience, or if you already have it, you might want to consider contributing back to the community, by answering questions or participating in the discussions that take place in the forum.

If you find a bug in Vaadin itself, the demo applications, or the documentation, you can report it by filing an issue in the corresponding repository at https://github.com/vaadin. To avoid duplication, please check existing issues before filing a new one. You can also open an issue to request for a new feature, or to suggest modifications to an existing feature.

In addition to the free support you can get from the Vaadin Community, Vaadin offers full commercial support (https://vaadin.com/pricing), training (https://vaadin.com/training), and consulting services (https://vaadin.com/consulting).

1. Introduction

Vaadin is an open-source framework for developing high-quality, modern web applications. It comes with a comprehensive suite of user interface (UI) components and tools designed to make creation and maintenance of web-based user interfaces easy.

Vaadin supports various programming models. You can use the server-side Java API, the client-side HTML Web Components, or a mixture of both. The server-side programming model allows you to program web user interfaces entirely in Java, or any other language for the JVM, much like you would program a desktop application with toolkits such as AWT and Swing. The client-side HTML Web Components model allows you to use the Vaadin's UI components library with any other web framework compatible with Web Components or even in any HTML document, without having to use a web framework at all. Each component in Vaadin has a server-side Java API and a client-side web component API to suit the programming model you want to use. You can also combine the server-side and client-side models to implement part or all of your UI using HTML templates and automated client-server communication provided by Vaadin.

Vaadin includes a set of tools to ease web development. Vaadin Designer is a drag and drop visual editor for IntelliJ IDEA and Eclipse that you can use to implement web user interfaces. It allows you to see what your web application will look like while you implement it. The editor allows you to drag components from a palette and drop them on a canvas, where you can further adjust component properties such as size and caption. Vaadin TestBench is a tool that automates user interface testing. It allows you to ensure regressions are caught before deploying to production environments. You write the tests in Java, which gives the advantages associated with static typing to your testing code. You can perform pixel-perfect testing, simulate user interactions in real-time, and run your tests without opening up a browser in headless mode to automate test execution in Continuous Integration environments.

1.1. Core concepts

1.1.1. Everything is a Component

When using the server-side programming model, everything is a UI component. If you need a button, you can write `new Button()`. If you need a text field, you can write `new TextField()`. You can build your own components and views by combining existing components and layouts. The following is a small but complete Vaadin application written in Java:

```java
@Route("")
public class MainView extends VerticalLayout {
    public MainView() {
        add(new H1("Hello, World!"));
    }
}
```

Vaadin uses a component-based programming model. In the previous example, the application is a UI component that extends one of Vaadin's basic layouts –`VerticalLayout`. In the constructor, we add a `H1` component (that corresponds to a `<h1>` HTML tag) to the layout to say hello to the entire world. We map the view to an empty route with the `@Route("")` annotation, so when this application is deployed to a local server, the view is available on your machine at `http://localhost:8080/`.

1.1.2. Listen to Events to Make your App Interactive

To make the applications interactive, Vaadin provides an event-driven programming model. For example, you can add a listener to a button with `addClickListener()` or get notified of a selection change in a select component with `addValueChangeListener()`.

1.2. Why Vaadin?

There is one feature that makes Vaadin unique – you can implement browser-based user interfaces by writing only server-side Java. Strictly speaking, there’s no need to use or even learn HTML and JavaScript to implement a web application with Vaadin. Besides the Java Programming Language, you can use other languages for the JVM as alternatives.

When the whole application is written in Java, the UI code is object-oriented, making it easy to apply design patterns to all of your code. Moreover, you can use the tools provided by IDEs such as IntelliJ IDEA, Eclipse, and NetBeans, to debug the app with the same debugger you would use for the backend. You can also perform actions such as refactorings, or inspect all the available methods and Javadocs in the Vaadin classes, with ease and without having to leave the IDE.

Vaadin’s server-side programming model automates communication between the client and the server. It takes care of managing the user interface in the browser and any required communication with the server. There is no need to implement RESTful web services, for example, to connect the UI with business logic.

Since the UI code is in Java, you can use any of the libraries available in the huge Java ecosystem in your presentation layer. There is official open-source support to integrate Vaadin in Spring and Jakarta EE applications, and there are multiple third-party integrations with other technologies.

Vaadin includes a coherent set of UI components that work and look well together. You can use them with the Java web framework provided with Vaadin, or as Web Components with other frameworks. All UI components were designed with accessibility in mind, are compatible with mobile devices, and can be styled with CSS. When using the server-side programming model, you can use push to update the UI from the server and enable some of the features of Progressive Web Applications (PWA).

Vaadin also offers the possibility to implement the UI using HTML, together with automated browser-server communication that takes care of data binding, and server-side and client-side method invocation from the counterpart.

2. Developing Vaadin Applications

This section gets you started with application development with Vaadin. It covers the tools you need and explains how to create a new project to start coding with Vaadin.

2.1. Development Toolchain

To develop server-side web applications with Vaadin, you need to install three things:

- The Java Development Kit (JDK)
- Maven
- Node.js
- An Integrated Development Environment (IDE) compatible with Java

2.1.1. The JDK

The JDK, together with the Java Runtime Environment (JRE) and the Java Virtual Machine (JVM), are at the core of Java application development. It's important to understand the difference between these three elements of the Java development environment, before you start developing Java applications.

The JDK is used to convert plain text files with the `.java` extension (that you create using an IDE) to binary files with the `.class` extension, using the `javac` program, through Maven, or through IDE actions. This process is known as compilation. The compiled `.class` file contains something called bytecode – a set of instructions that the JVM can execute in a computer. The JVM is created by the JRE using the `java` program. Simply put, you need to install the JRE to run Java programs and the JDK to develop them. Since running Java programs is part of software development with Java, the JDK includes the JRE.

You can download a free, open-source JDK at https://jdk.java.net. Make sure you download and install version 8 or later of the JDK. The installation process varies depending on your operating system and JDK distribution. One thing you need to make sure of is that the `bin` directory in the JDK installation directory is in the operating system's path, so that the `java` and `javac` programs can run from any working directory in the command line and hence the IDE.

To check that the JDK is installed correctly, run `javac -version` in a terminal. You should get an output similar to the following:

```
javac 1.8.0_181
```

Instructions on how to install the JDK in each operating system are out of the scope of this documentation, but there are many tutorials with detailed and up-to-date instructions online.

2.1.2. IDE Support

You can essentially develop Vaadin applications in any Integrated Development Environment (IDE) that is compatible with Java, such as IntelliJ IDEA, Eclipse, and NetBeans. If you plan to use Vaadin Designer, keep in mind that (at the time of writing) it is only available for IntelliJ IDEA and Eclipse. Vaadin Designer is a productivity tool that helps you to implement UIs faster, but it's not required to develop applications with Vaadin.

When installing and first running the IDE, you might have to set up the JDK. This is as simple as configuring the directory where the JDK is installed, so that the IDE can use it to compile and run the Java programs you develop. In some IDEs, the JDK is referenced as SDK (Software Development Kit).

2.1.3. Maven

Vaadin is distributed through several JAR dependencies available in the Maven Central Repository. You can use any build or dependency management system that can access Maven repositories, for example Maven and Gradle, to include the Vaadin dependencies in your Java project.

Although some IDEs include a bundled distribution of Maven, you might want to use it from the command line as well. You can find instructions on how to download and install Maven at http://maven.apache.org. To check that Maven is installed correctly, run `mvn -v` and confirm that you get output similar to the following:

```
Apache Maven 3.6.0
(97c98ed64a1fdfee7767ce5cfb20918da4f719f3; 2018-10-
24T21:41:47+03:00)
Maven home: /Users/demo-user/Applications/apache-maven-
3.6.0
Java version: 1.8.0_181, vendor: Oracle Corporation,
runtime:
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Conten
ts/Home/jre
Default locale: en_FI, platform encoding: UTF-8
OS name: "mac os x", version: "10.14.3", arch: "x86_64",
family: "mac"
```

2.1.4. Node.js

Since Vaadin version 14, *npm* is used by the framework to manage front-end dependencies. npm is distributed with Node.js–when you install Node.js, you install npm as well. Follow the instructions at https://nodejs.org/en/download to install Node.js. To check that npm works correctly run `npm -v` and confirm that the installed version is displayed.

2.1.5. Installing a Web Server

You can run applications developed with Vaadin in any Java server that supports the Servlet API version 3 or later. Such servers include servlet containers such as Apache Tomcat, Jetty, and application servers such as Wildfly, Glassfish, TomEE, and WebLogic.

When you create a new Vaadin project, using the starters available at https://vaadin.com/start or the official Maven archetypes, you can use the Jetty server during development. This server is configured as a Maven plugin so you need Maven to run the application. Using the Jetty Maven Plugin is convenient during development. However, in production environments, you probably do not want to install Maven, since it is a development tool. In production, you should install the Java web server of your choice. The application is typically packaged as a WAR file that you can deploy to your Java web server.

In later stages of the development process, you might want to use the same Java web server in development and production environments. Most IDEs include integrated features or plugins to configure the server, deploy the application, and start or stop the server. You can find a number of online resources on how to do this with the IDE and server of your choice.

2.2. Starters and Maven Archetypes

There are two ways to create a new Vaadin project. You can use an online tool to configure and download the project, or create it in your development machine using a Maven archetype. Once you have created the project, you can import the project into your IDE.

2.2.1. Creating a New Project Using the Vaadin Starters

The easiest way to create a new Vaadin project from scratch is by generating one at https://vaadin.com/start. If you are new to Vaadin, Create a new empty project with the *Plain Java Servlet* technology stack to generate a minimal project that serves well as a starting point. The code is packaged as a ZIP file that contains a Maven project that you can import into your IDE. You can find detailed instructions on how to import the project at:

- IntelliJ IDEA: https://vaadin.com/tutorials/import-maven-project-intellij-idea
- Eclipse: https://vaadin.com/tutorials/import-maven-project-eclipse
- NetBeans: https://vaadin.com/tutorials/import-maven-project-netbeans

2.2.2. Creating a Project Using the Vaadin Maven Archetype

If you prefer, you can use Maven to generate a new Vaadin project, as an alternative. In the command line run the following, changing `LATEST` to the version of Vaadin you want to use:

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

NOTE You can find the latest version of Vaadin at https://vaadin.com.

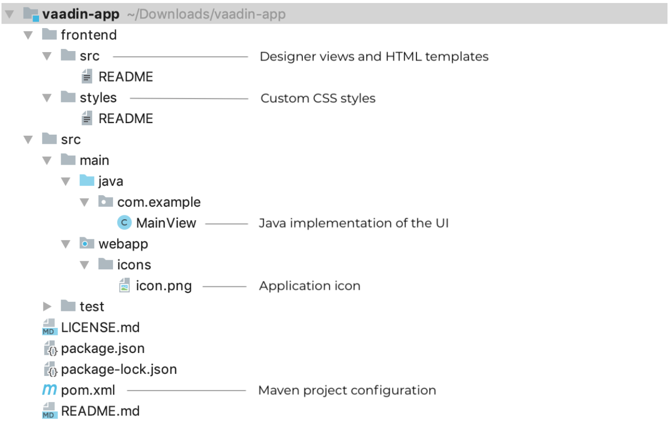
You can also use a Maven archetype directly in your IDE, if you don't want to use the command line. You might have to configure the Maven archetype, `vaadin-archetype-application`, before you use it for the first time. Details on how to do this depends on which IDE you use. Typically, you have to create a new Maven project from an archetype. Look for an option to add a new archetype and specify the following when prompted:

- GroupId: `com.vaadin`
- ArtifactId: `vaadin-archetype-application`
- ArchetypeVersion: `LATEST`

The IDE will perform the corresponding Maven command to create a new Maven project using the configured archetype.

2.3. Exploring the Project

Once you have created the project, using either the starter or the Maven archetype, and imported it into your IDE, you will find the following directory structure:



```
▼ vaadin-app ~/Downloads/vaadin-app
  ▼ frontend
    ▼ src ——————————— Designer views and HTML templates
        README
    ▼ styles ———————— Custom CSS styles
        README
  ▼ src
    ▼ main
      ▼ java
        ▼ com.example
            MainView ——— Java implementation of the UI
      ▼ webapp
        ▼ icons
            icon.png ——— Application icon
    ▶ test
    LICENSE.md
    package.json
    package-lock.json
    pom.xml ————————— Maven project configuration
    README.md
```

NOTE The way the project files are grouped varies depending of the IDE you use. Your project structure may be slightly different.

2.3.1. The MainView Class

When you create a new project with Vaadin, the most interesting part is the `MainView` java class. This class contains the code that implements the UI. If you open this class in your IDE, you will see something similar to the following:

```java
@Route("")
@PWA(name = "Project Base for Vaadin",
     shortName = "Project Base")
public class MainView extends VerticalLayout {

    public MainView() {
        Button button = new Button("Click me",
                event -> Notification.show("Clicked!"));
        add(button);
    }
}
```

The `@Route("")` annotation makes the view visible when you invoke `http://localhost:8080`, for example. You can change the annotation in the `MainView` class or create new classes with different `@Route` annotations to add more views (accessible through different URLs), if you want to. For example, if you change the annotation to `@Route("demo")`, the view will be shown when you point your browser to `http://localhost:8080/demo`. When you have several views, you can programmatically *navigate* between them or let the user invoke them by manually specifying the URL, clicking on a link, or using a saved browser bookmark.

The `@PWA` annotation activates automatic PWA features, such as showing a custom message when the application is not available (offline), or offering to install the application in the home screen to allow users to access the app with one click or touch. This annotation is optional and you can safely remove it, if you do not want these features. Notice that the `MainView` class extends `VerticalLayout`. In order to make the view available in the browser, you need to extend an existing UI component (in addition to annotating the class with `@Route`). You can extend a UI component provided by Vaadin, or use a custom UI component that you created before. A `VerticalLayout` is a convenient starting point, as it allows you to add other UI components that are placed vertically in the order in which they are added. You might also want to experiment with `HorizontalLayout`, `FormLayout`, and `SplitLayout`.

When you invoke the application (for example with `http://localhost:8080/`), Vaadin creates a new instance of the `MainView` which in turn invokes the constructor of the class. In the constructor, a new `Button` is created with a *click listener* that shows a notification when the button is clicked.

2.3.2. Static Resources

In the `frontend` directory, you can add CSS styles (`frontend/styles/`) and views created with Vaadin Designer and HTML templates (`frontend/src/`). These resources are optional, since you can create the views using Java instead.

You can also replace the icon provided in the `src/main/webapp/icons/` directory with your own. This icon is used when you activate PWA.

2.4. Running and Debugging

There are several ways to run the project. Since both projects generated by the starter or the Maven Archetype, include the Jetty Maven Plugin, you can use Maven to run the application. This is done by executing the following command:

```
mvn jetty:run
```

NOTE You need to use a modern web browser to use the application (IE 11 is not supported).

During development, the best approach is to run the `jetty:run` Maven goal inside your IDE. You can find detailed instructions on how to do this with different IDEs at:

- IntelliJ IDEA: https://vaadin.com/tutorials/import-maven-project-intellij-idea
- Eclipse: https://vaadin.com/tutorials/import-maven-project-eclipse
- NetBeans: https://vaadin.com/tutorials/import-maven-project-netbeans

Most, if not all, IDEs have two options to run a Maven goal: **Run** and **Debug**. The **Debug** option allows you to use the IDE's debugger to, for example, add breakpoints to interrupt the execution of the program to inspect the values of variables,or trace the execution of the code line by line.

NOTE Keep in mind that the first time you compile the project, it might take some time. This can happen automatically when you import the project or when running the project for the first time, depending on the method you used to run the application. Maven needs to download all the required dependencies (JAR files) to the local Maven repository in your machine, but once they are downloaded, subsequent compilations are much faster.

3. Understanding Vaadin

Vaadin connects the Java ecosystem to your web platform. This section provides an overview of the architecture of Vaadin and introduces the key concepts in the framework.

3.1. Vaadin Architecture

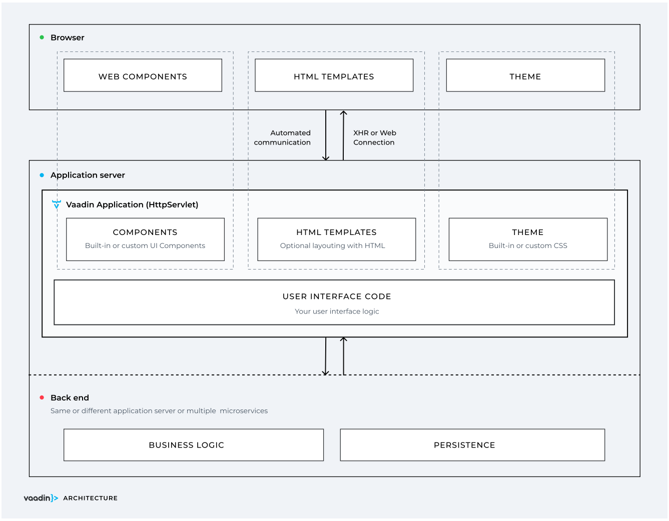
Working with front-end web technologies, such as HTML, CSS and JavaScript, can be challenging and time-consuming for Java developers. In Vaadin, all UI elements are componentized into Web Components[1]. This makes development easier than ever before, because each element is decoupled and sandboxed.

TIP

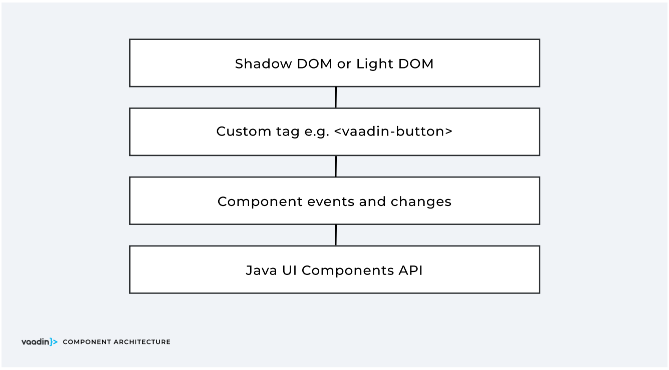
Watch the Vaadin 10+: Intro[2] free training video to learn more about the Vaadin framework, basic Vaadin application architecture and how Vaadin components work.

Vaadin includes:

- A type-safe Java UI Component API on the server side that facilitates the use of the Web Components.
- Automated bi-directional communication between the server and the browser, that:
    - Gives Java developers full access to all modern web enhancements.
    - Makes it easier to connect the UI to data via a robust Java backend, instead of using traditional REST-based communication.
- Two-way data binding: when the UI changes on either the client or the server, the changes automatically reflect on the other side.



Vaadin allows you to access browser APIs, Web Components, and even simple DOM elements, directly from the server-side Java. It is not necessary to understand how the client-to-server communication or Web Components work. This leaves you free to focus on creating components that work at a higher-abstraction level.


Shadow DOM or Light DOM
Custom tag e.g. <vaadin-button>
Component events and changes
Java UI Components API
vaadin}> COMPONENT ARCHITECTURE

3.2. Building UIs with Components

UI components that are designed to build interactive web apps are the core of Vaadin. In addition, Vaadin provides powerful abstraction layers that you can use to create new components.

3.2.1. Vaadin Components

The high abstraction layers provided by Vaadin's Java Web Components API allow you to build UIs in an extremely productive way. You do not need any HTML or JavaScript knowledge, and only enough CSS experience to style the look and feel of your app.

Example: Using the `TextField` component.

```java
TextField textField = new TextField();
// Simple HTML inline text
Span greeting = new Span("Hello stranger");

textField.addValueChangeListener(event ->
      greeting.setText("Hello " + event.getValue()));

VerticalLayout layout = new VerticalLayout(
      textField, greeting);
```

The API includes prebuilt server-side components and most native HTML elements.

See Components[3], for a full set of available Vaadin components.

3.2.2. Creating New Components in Java

On the higher abstraction layers, you can easily create custom components by adapting or combining existing components to meet your requirements.

The light-weight component architecture and the ability to access the DOM and browser APIs from the server side, simplifies component customization. While staying on the server side you can perfect customizations and eliminate bugs, by leveraging Vaadin's automated communication layer between the browser and the server.

Example: Extending `Component` to create a custom component.

```
@Tag("my-label")
public class MyLabel extends Component {
    public void setText(String text) {
        getElement().setText(text);
    }

    public String getText() {
        return getElement().getText();
    }
}
```

See the tutorials in Creating Components[4] to learn how to build components with a reusable API, and Element API[5] to learn how to access and customize the DOM from the server side.

3.2.3. Integrating a Web Component

Vaadin allows you to create a Java API for any available Web Component and then use the API in your projects.

Example: Importing the `game-card` Web Component into the `GameCard` Java class.

```
@Tag("game-card")
@JsModule("./game-card.js")
public class GameCard extends Component {

}
```

See the tutorials in Integrating a Web Component[6] for more.

You can also find prebuilt Java APIs for Web Components that have been published by the Vaadin Community in the Vaadin Directory.[7]

3.2.4. Building Components with HTML Templates

Another way to create components is to separate the layout from the UI logic. The best way to do this is to use JavaScript modules and HTML templates together with Java classes. The JavaScript module contain the layout and (if needed) pure client-side logic, while the Java classes takes care of the server-side logic, like event handling.

You can use these components in the same way as any other component in your Java environment. Vaadin does not distinguish between pure Java or HTML/Java combined components.

Example: @Id injection in a component.

```
static get template() {
    return html`
        <vaadin-vertical-layout>
            <vaadin-text-field id="textField">
            </vaadin-text-field>
            <label id="greeting">Hello stranger</label>

            <input type="color"
                  on-input="updateFavoriteColor">
            <label>Favorite color: </label>
        </vaadin-vertical-layout>`;
}
```

```
private @Id("textField") TextField textField;
private @Id("greeting") Label greeting;

// Setting things up in the component's constructor
textField.addValueChangeListener(event ->
      greeting.setText("Hello " + event.getValue()));

// Instance method in the component published to the
// client
@EventHandler
private void updateFavoriteColor(
      @EventData("event.target.value") String color) {
    getModel().setColorCode(color);
}
```

See the tutorials in Creating Polymer Templates[8] for more.

3.3. Routing and Navigation

Vaadin provides the `Router` class to structure the navigation of your web app or site into logical parts.

You can use the `@Route` annotation to register navigation targets. You can specify a path, and optionally a parent layout class to display the component.

Example: Using the `@Route` annotation.

```
// register the component to url/company and show it
// inside the main layout
@Route(value = "company", layout = MainLayout.class)
@Tag("div")
public class CompanyComponent extends Component {
}

public class MainLayout extends Div
        implements RouterLayout {
}
```

See the tutorials in Routing and Navigation for more.

3.4. How Vaadin Components Work

Vaadin allows Java code to control the DOM in the web browser, with a server-side Java representation of the same DOM tree. All changes are automatically synchronized to the real DOM tree in the browser.

The DOM tree is built up from `Element` instances: each instance represents a DOM element in the browser. The root of the server-side DOM tree is the `Element` of the `UI` instance. You can access it using the `ui.getElement()` method. This element represents the `<body>` tag.

Elements on the server are implemented as flyweight instances. This means that you cannot compare elements using the `==` and `!=` operators. Instead, you need to use the `element.equals(otherElement)` method to check whether two instances refer to the same DOM element in the browser.

3.4.1. Element Hierarchy

A web app is structured as a tree of elements, with the `UI` instance element as the root. An element can be added as a child of another element, using methods such as:

- `element.appendChild(Element)` to add an element at the end of a parent's child list, or
- `element.insertChild(int, Element)` to add an element to any position in a child list.

You can use `element.getParent()` to navigate upwards in the element hierarchy, and `element.getChildren()` to navigate downwards.

3.4.2. Component Hierarchy

The `Component` class wraps the `Element` and provides a higher level of abstraction. You can obtain the element representation of a component using the `Component.getElement()` method.

The component's element can optionally contain any number of child elements. In addition to the low-level element, the component itself can also support child components, and methods similar to `Component.add(Component… )` are provided for this purpose.

You can navigate through the component's hierarchy using `component.getParent()` to navigate upwards, and `component.getChildren()` to navigate downwards.

The component hierarchy is constructed based on the element hierarchy. Changes in the component hierarchy are reflected in the element hierarchy (but not vice versa).

3.4.3. HTML Templates

As an alternative to creating the DOM in Java, you can use HTML templates. In this case, Java is only used for server-side control and interaction with elements, for example via event listeners.

Possible benefits of this approach include:

- A clearer overview of the structure of the component.
- Improved performance. Because the same template definition is used for all component instances using the same template file, less memory is used on the server and less data needs to be sent to the browser.

NEXT: Follow the tutorial to build your first Vaadin application: **Getting started with Vaadin**[9]

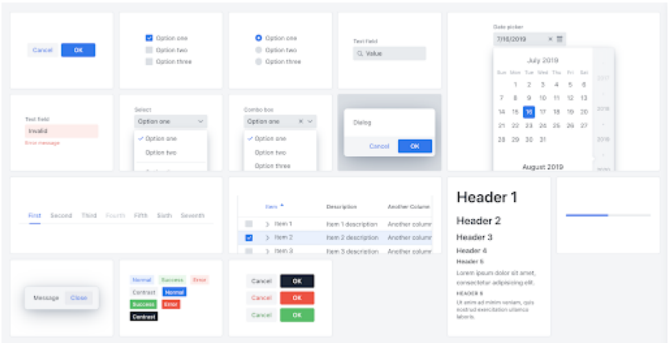
----

[1] https://www.webcomponents.org/
[2] https://vaadin.com/training/course/view/v10-intro
[3] https://vaadin.com/components/browse
[4] https://vaadin.com/docs/flow/creating-components/tutorial-component-basic.html
[5] https://vaadin.com/docs/flow/element-api/tutorial-event-listener.html
[6] https://vaadin.com/docs/flow/web-components/integrating-a-web-component.html
[7] https://vaadin.com/directory/search?framework=Vaadin%2010
[8] https://vaadin.com/docs/flow/polymer-templates/tutorial-template-basic.html
[9] https://vaadin.com/tutorials/getting-started-with-flow

4. Using Vaadin Components

In addition to built-in components, you can:imagesdir: ./images Vaadin offers a **comprehensive set of ready-made** UI components that you can use out of the box to create beautiful web applications with all the functionality expected today.

The set includes ready-to-use components for every conceivable part of a modern UI, including grids, layouts, form fields, upload and similar functions, and more.



The components are provided in various formats and can be used in either HTML or Java. In this section we focus only on Java.

- Each component consists of a client-side Web Component (with CSS styles), and an associated Java class, providing the server-side API.
- All components are designed responsively and work equally well on mobile touch devices and desktops.
- All components comply with accessibility WAI-ARIA standards and support keyboard shortcuts and screen

readers.

- You can configure component behavior in many ways to fit your application. Each component has its own set of configuration properties like width, height, label, description, error messages, and so on.
- Field components provide sophisticated data binding with data conversion and validation.

You can find the up-to-date component listing and documentation of Vaadin components at https://vaadin.com/components.

4.1. Form Input Fields

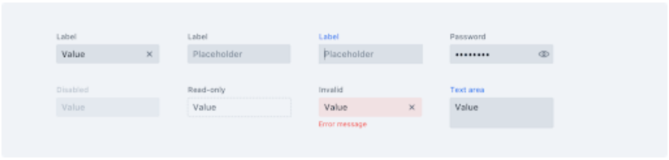
4.1.1. Text Field

The Text field component allows the user to enter text and is probably the most common component found in web forms.

Text Field is the parent component of various additional text field components that extend it to provide specific behavior. These components include Text Area Field, Password Field, Email Field, and Number Field.

Text Field provides a single line input area. You can use the Text Area Field if you need a bigger input area.

You can configure a placeholder text, a clear button, minimum and maximum length, autofocus, autocomplete (where available), validation against a against a regular expression pattern, and more.



Usage:

```
TextField textField = new TextField();
textField.setLabel("Text field label");
textField.setPlaceholder("placeholder text");
```

4.1.2. Email Field

The Email Field component is a Text field that expects email input. It supports browser autocomplete (where available), and validates that the input is an email address by checking for the ampersand character (@).

The Email Field is useful for collecting valid email addresses and verifying a user's identity.

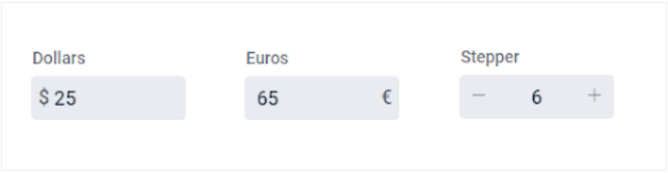
Usage:

```
EmailField emailField = new EmailField("Email");
emailField.addValueChangeListener(event -> message
.setText(
    String.format(
        "Email field value changed from '%s' to '%s'",
        event.getOldValue(), event.getValue())
    )
);
```

4.1.3. Number Field

The Number Field component is a Text field that only accepts numeric values.

You can set an initial value and default value, allow null values, apply functions, set decrease and increase controls, define minimum and maximum values, set prefix and suffix values, and more. On mobile devices the browser shows dedicated input controls.



Usage:

```
NumberField dollarField = new NumberField("Dollars");
dollarField.setPrefixComponent(new Span("$"));

NumberField euroField = new NumberField("Euros");
euroField.setSuffixComponent(new Span("€"));

NumberField stepperField = new NumberField("Stepper");
stepperField.setValue(1d);
stepperField.setMin(0);
stepperField.setMax(10);
stepperField.setHasControls(true);
```

4.1.4. Password Field

The Password Field component is a Text field that allows the user to safely enter their password. The field masks the actual characters with dots, and displays an eye icon that allows the user to toggle password visibility. It supports browser autocomplete (where available).

Usage:

```
PasswordField passwordField = new PasswordField();
passwordField.setLabel("Passwordl");
```

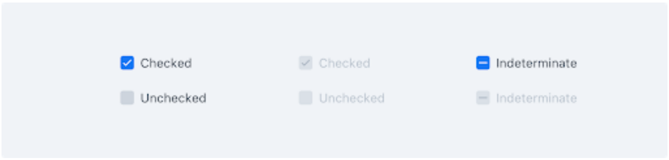
4.1.5. Checkbox

The Checkbox field consists of a single selection box or tick box. When selected, a tick or check mark displays in the box.

Checkbox is an interactive component that allows the user to indicate a choice, by toggling the box on and off. Checked indicates a positive response, and unchecked a negative response.

You can use a string or HTML to set the label and configure the box to be marked as indeterminate.

Single check boxes are useful to answer Yes/No questions, and are frequently used to allow the user to enable and disable a setting or choice, for example.



The Checkbox Group field is provided as a separate component. This field groups and displays multiple checkboxes, and allows the user to make multiple selections in the same field. You can configure the options to display vertically or horizontally.

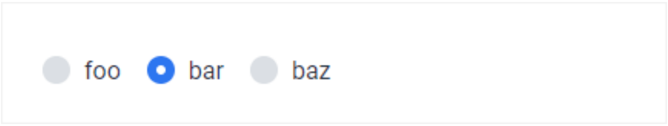
Usage:

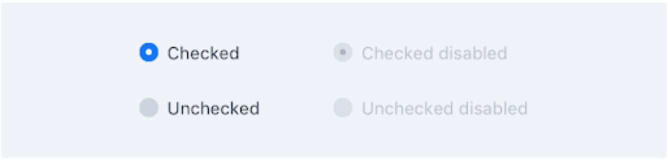
```
Checkbox checkbox = new Checkbox();
checkbox.setLabel("My Label");
```

4.1.6. Radio Button Group

The Radio Button Group field consists of multiple radio button options. The options are mutually exclusive in that only one selection is possible. If the user makes a second selection, the first is automatically deselected.

The Radio Button Group field is similar to the Check Box Group field, except that it allows only a single selection.





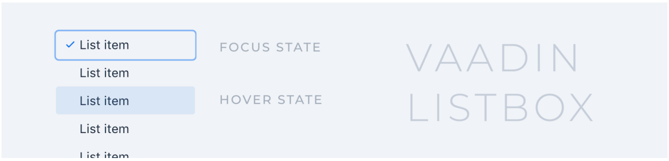
Usage:

```
RadioButtonGroup<String> group =
    new RadioButtonGroup<>();
group.setItems("foo", "bar", "baz");
group.addValueChangeListener(event -> message.setText(
    String.format("Value changed from '%s' to '%s'",
        event.getOldValue(), event.getValue()))
);
```

4.1.7. List Box

The List Box field displays a list of values or choices from which the user can make a single selection.

The field supports separators and arbitrary HTML content.



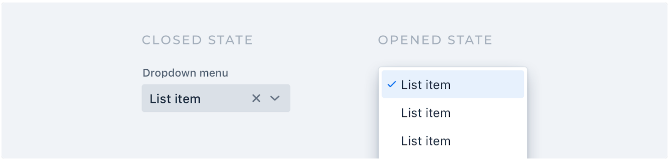
Usage:

```
ListBox<String> listBox = new ListBox<>();
listBox.setItems("Bread", "Butter", "Milk");
```

4.1.8. Select

The Select field allows the user to select a single option from a dropdown list. It is similar to a native browser select element. The field displays a down arrow that activates the dropdown list.

You can add child components, validate input, configure a clear button, and more.



Usage:

```
Select<String> select =
    new Select<>("Option one", "Option two");
select.setPlaceholder("Placeholder");
select.setLabel("Label");
```

4.1.9. Combo Box

The Combo Box field is a combination of a standard list box (dropdown list) and an editable text box. It allows the user to enter text and/or select an option from a dropdown list.

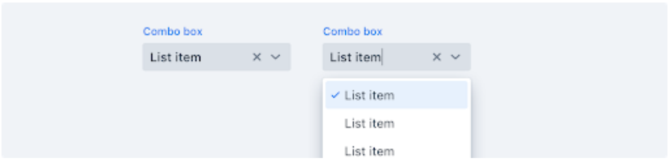
The field filters items in the dropdown list automatically: as the user enters alphanumeric characters, the list automatically offers better possible matches. For example, entering the letters sou in a country field reduces the options to countries containing only the word south.

This field supports lazy loading and you can configure the page size for this purpose. You can also disallow custom values.

Combo box is extremely useful when you need to:

- Provide a large volume of options, for example countries, postal codes and the like.

- Allow the user to add options that are not in the list.



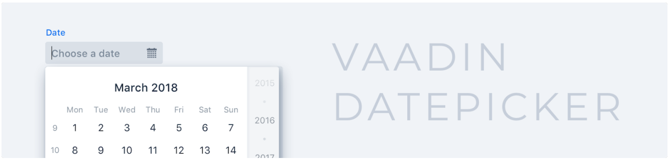
Usage:

```
ComboBox<String> comboBox =
    new ComboBox<>("Browsers");
comboBox.setItems("Google Chrome",
    "Mozilla Firefox", "Opera",
    "Apple Safari", "Microsoft Edge");

comboBox.addValueChangeListener(event -> {
    if (event.getSource().isEmpty()) {
        message.setText("No browser selected");
    } else {
        message.setText("Selected browser: "
            + event.getValue());
    }
});
```

4.1.10. Date Picker

The Date Picker field allows the user to select a date in the calendar. The field displays a calendar icon that activates the popup calendar interface when selected. The calendar UI features dual-speed scrolling and is easy and quick to use.

You can configure the initial visible date, maximum and minimum dates, internationalization (i18n), locales, and more.



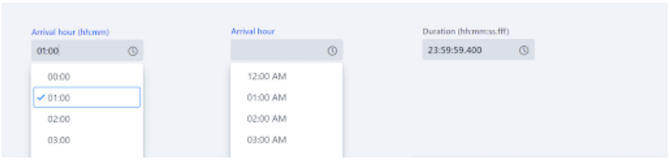
Usage:

```
DatePicker datePicker = new DatePicker();
datePicker.setLabel("Select a day within " +
    "this month");
datePicker.setPlaceholder("Date within " +
    "this month");

LocalDate now = LocalDate.now();

datePicker.setMin(now.withDayOfMonth(1));
datePicker.setMax(now.withDayOfMonth(
    now.lengthOfMonth()));
```

4.1.11. Time Picker

The Time Picker field allows the user to select a time in a dropdown list. The field displays a clock icon and focusing on the field activates a dropdown list.

You can configure the time intervals that display (for example, an option for every 15/30/60 minutes), the time format, minimum and maximum times, and even set time to the user's local time. By default, the field uses ISO 8601 formatting (hh:mm, hh:mm:ss or hh:mm:ss.fff).



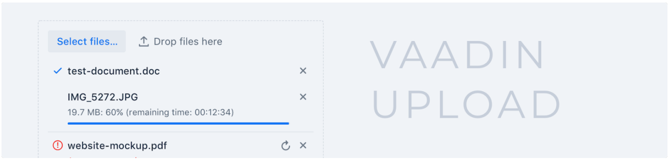
Usage:

```
TimePicker timePicker = new TimePicker();
```

4.1.12. Upload

The Upload Field allows the user to upload files. The field supports multiple formats, simultaneous upload of multiple files, drag and drop, a progress bar, and internationalization.

You can configure the allowed file formats, maximum upload size, HTTP upload request, and more.



Usage:

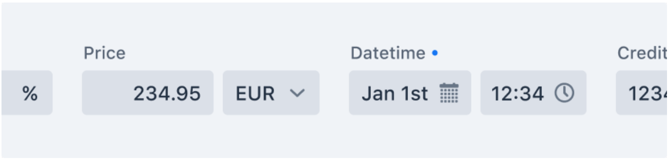
```
MemoryBuffer buffer = new MemoryBuffer();
Upload upload = new Upload(buffer);

upload.addSucceededListener(event -> {
    Component component =
        createComponent(event.getMIMEType(),
            event.getFileName(),
            buffer.getInputStream());
    showOutput(event.getFileName(),
        component, output);
});
```

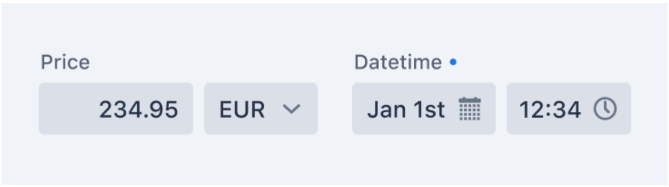
4.1.13. Custom Field

The Custom Field allows you to wrap multiple input fields into a single component.

The label of each field displays above the field, and error messages displays below it.

To use Custom Field you need to create your own component class that extends `CustomField` and implements `generateModelValue()` and the abstract field, `setPresentationValue(Object)`.





Usage:

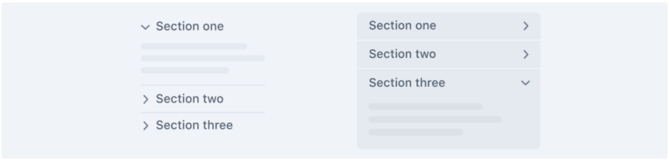
```java
public static class CustomDateTimePicker
    extends CustomField<LocalDateTime> {

    private final DatePicker datePicker =
        new DatePicker();
    private final TimePicker timePicker =
        new TimePicker();

    CustomDateTimePicker() {
        setLabel("Start datetime");
        add(datePicker, timePicker);
    }

    @Override
    protected LocalDateTime generateModelValue()
    {
        final LocalDate date =
            datePicker.getValue();
        final LocalTime time =
            timePicker.getValue();
        return date != null && time != null ?
            LocalDateTime.of(date, time) :
            null;
    }

    @Override
    protected void setPresentationValue(
        LocalDateTime newPresentationValue) {
        datePicker.setValue(newPresentationValue
            != null ?
            newPresentationValue.toLocalDate() :
            null);
        timePicker.setValue(newPresentationValue
            != null ?
            newPresentationValue.toLocalTime() :
            null);
    }
}
```

4.2. Visualization and Interaction

4.2.1. Accordion

The Accordion component contains a set of vertically stacked panels that the user can expand and collapse to reveal and hide the content associated with each panel.

The Accordion component reduces clutter and allows the user to find what they are looking for with ease. It is useful whenever you need to display content detail that can be delineated into distinct sections.



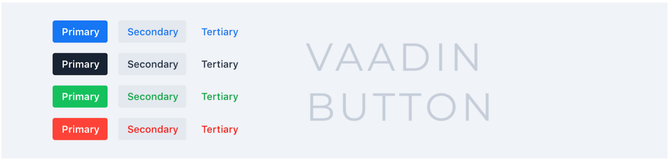
Usage:

```
Accordion accordion = new Accordion();

VerticalLayout personalInformationLayout =
    new VerticalLayout();
personalInformationLayout.add(
    new TextField("Name"),
    new TextField("Phone"),
    new TextField("Email")
);
accordion.add("Personal Information",
    personalInformationLayout);

VerticalLayout billingAddressLayout =
    new VerticalLayout();
billingAddressLayout.add(
    new TextField("Address"),
    new TextField("City"),
    new TextField("State"),
    new TextField("Zip Code")
);
accordion.add("Billing Address",
    billingAddressLayout);

VerticalLayout paymenLayout =
    new VerticalLayout();
paymenLayout.add(
    new Span("Not yet implemented")
);
AccordionPanel billingAddressPanel =
    accordion.add("Payment", paymenLayout);
billingAddressPanel.setEnabled(false);
```

4.2.2. Button

The Button component displays a button. In addition to button text, the component allows you to display icons or images in the button.

Configuration options include defining keyboard shortcuts, a custom tab index to ensure keyboard accessibility, automatically disabling the button after click, and more



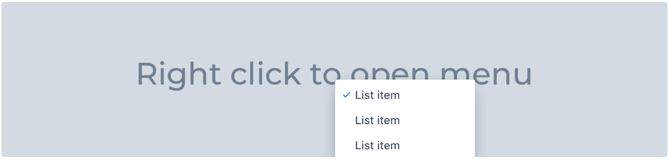
Usage:

```
Button button = new Button("Vaadin button");
button.addClickListener(this::showButtonClickedMessage);
```

4.2.3. Context Menu

The Context Menu component allows you to create a context menu that is activated when the user right clicks on a desktop or performs a touch-and-hold action on a mobile device.

You can define the list of items in the context menu, create nested menus, create checkable menu Items, attach event handlers to each item, and more.



Usage:

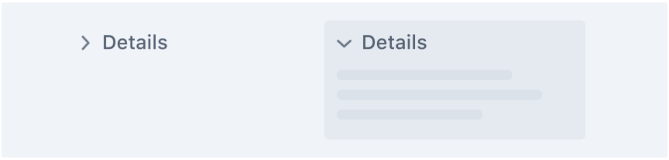
```
ContextMenu contextMenu = new ContextMenu();

Component target = createTargetComponent();
contextMenu.setTarget(target);

Label message = new Label("-");

contextMenu.addItem("First menu item",
    e -> message.setText("Clicked on " +
        "the first item"));

contextMenu.addItem("Second menu item",
    e -> message.setText("Clicked on " +
        "the second item"));

// The created MenuItem component can be saved
// for later use
MenuItem item = contextMenu.addItem("Disabled " +
        "menu item",
    e -> message.setText("This cannot happen"));
item.setEnabled(false);
```

4.2.4. Details

The Details component consists of a single expandable panel that opens to reveal the details when the user selects the down arrow control.

You can define the content heading and detail text, and the user can toggle the content open and closed to suit.

The Details component is useful when you want a page to appear less cluttered. It is similar to the Accordion component, but offers only a single panel.



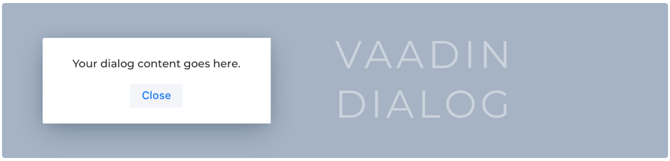
Usage:

```
Details component =
    new Details("Expandable Details",
    new Text("Toggle using mouse, Enter " +
        "and Space keys."));
```

4.2.5. Dialog

The Dialog component allows you to configure a pop-up dialog boc that can contain text, input fields, buttons, nested components, and more.

The dialog is modal in that it prevents the user interacting with the underlying page until the dialog is closed. You can configure the dialog to close using the Escape key or an external click, set the focus on an internal input element, and more.



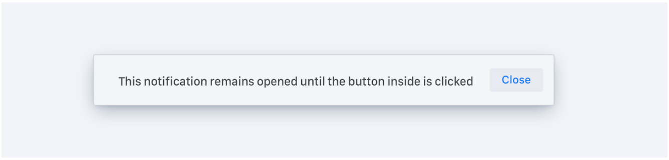
Usage:

```
Dialog dialog = new Dialog();
dialog.add(new Label("Close me with the " +
    "esc-key or an outside click"));

dialog.setWidth("400px");
dialog.setHeight("150px");

button.addClickListener(event -> dialog.open());
```

4.2.6. Notification

The Notification component allows you to display a custom notification. The component supports plain text, HTML content, buttons, other interactive content, and more. .

You can configure the notification position, the time period it remains visible, and whether it is dismissed automatically after a timeout or programmatically.


This notification remains opened until the button inside is clicked

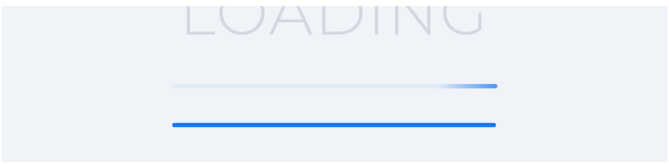
Usage:

```
Notification notification = new Notification(
        "This notification has text content", 3000);
button.addClickListener(event -> notification.open());
```

4.2.7. Progress Bar

The Progress Bar component allows you to display a bar that visually shows the progress of an operation.

You can configure minimum and maximum values, progression increments, and more.



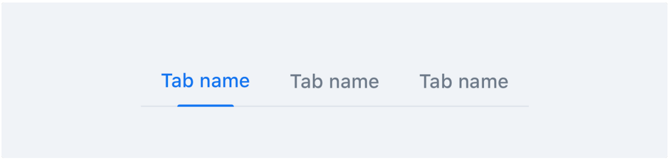
Usage:

```java
ProgressBar progressBar = new ProgressBar();
progressBar.setValue(0.345);
```

4.2.8. Tabs

The Tabs component allows you to display tabbed content.

You can use plain text, HTML, or icons as tab headers, set the tab orientation to display either horizontally or vertically, and define the available space allocated to each tab. The component supports scrolling and adds a scroll bar automatically when necessary.

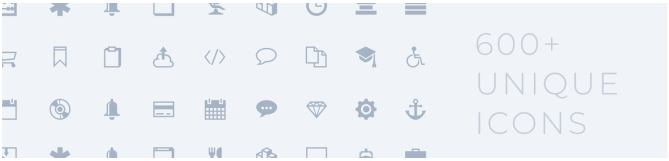


Usage:

```
Tab tab1 = new Tab("Tab one");
Tab tab2 = new Tab("Tab two");
Tab tab3 = new Tab("Tab three");
Tabs tabs = new Tabs(tab1, tab2, tab3);
```

4.2.9. Icons

Vaadin Icons is a collection of 600+ unique icons designed for web applications. You can download and install the package to use in your components.



Usage:

```
Icon edit = new Icon(VaadinIcon.EDIT);
Icon close = VaadinIcon.CLOSE.create();
```

4.3. Data Components

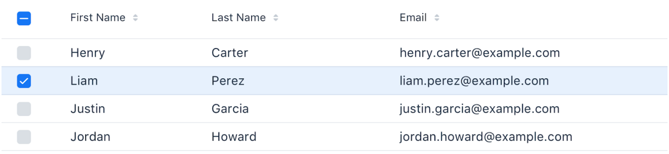
4.3.1. Grid

The Grid component allows you to display tabular data, set out in rows and columns. The component highly flexible and simple to use.

Features include:

Header and footer: You can use plain text, formatted text, or other components.

-
- **Sorting**: Column sorting is built in. Users can click the column header to sort the data, and shift click to enable secondary sorting criteria.
- **Smooth scrolling**: Users can scroll scrolled vertically and horizontally, and freeze left columns to keep them in view when scrolling horizontally.
- **Expandable rows**: You can configure the grid to allow users to expand and collapse rows.
- **More.**



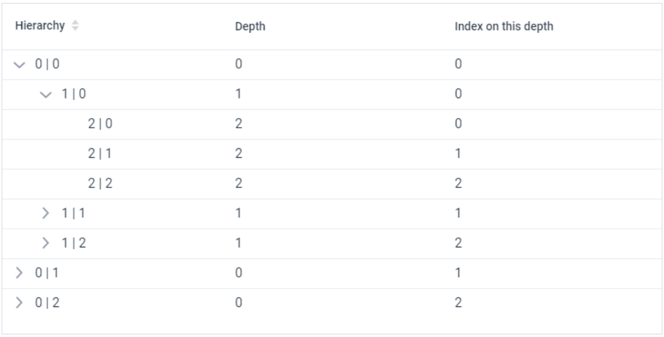
	First Name	Last Name	Email
	Henry	Carter	henry.carter@example.com
✓	Liam	Perez	liam.perez@example.com
	Justin	Garcia	justin.garcia@example.com
	Jordan	Howard	jordan.howard@example.com

Usage:

```java
List<Person> personList = new ArrayList<>();

DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("MM/dd/yyyy");
personList.add(new Person(100, "Lucas", "Kane",
    68,
    new Address("12080", "Washington"),
    "127-942-237"));
personList.add(new Person(101, "Peter",
    "Buchanan", 38,
    new Address("93849", "New York"),
    "201-793-488"));
personList.add(new Person(102, "Samuel",
    "Lee", 53,
    new Address("86829", "New York"),
    "043-713-538"));
personList.add(new Person(103, "Anton",
    "Ross", 37,
    new Address("63521", "New York"),
    "150-813-6462"));
personList.add(new Person(104, "Aaron",
    "Atkinson", 18,
    new Address("25415", "Washington"),
    "321-679-8544"));
personList.add(new Person(105, "Jack",
    "Woodward", 28,
    new Address("95632", "New York"),
    "187-338-588"));

Grid<Person> grid = new Grid<>(Person.class);
grid.setItems(personList);

grid.removeColumnByKey("id");

// The Grid<>(Person.class) sorts the properties and
// in order to reorder the properties we use the
// 'setColumns' method.
grid.setColumns("firstName", "lastName", "age",
    "address", "phoneNumber");
```

4.3.2. Tree Grid

The Tree Grid component extends the Grid component to enable a hierarchical structure for tabular data. The component includes all the features available in the Grid component.



Hierarchy	Depth	Index on this depth
0 | 0	0	0
1 | 0	1	0
2 | 0	2	0
2 | 1	2	1
2 | 2	2	2
1 | 1	1	1
1 | 2	1	2
0 | 1	0	1
0 | 2	0	2

Usage:

```
TreeGrid<PersonWithLevel> grid =
    new TreeGrid<>();
grid.setItems(getRootItems(), item -> {
    if ((item.getLevel() = 0
        && item.getId() > 10)
        || item.getLevel() > 1) {
        return Collections.emptyList();
    }
    if (!childMap.containsKey(item)) {
        childMap.put(item, createSubItems(81,
            item.getLevel() + 1));
    }
    return childMap.get(item);
});
grid.addHierarchyColumn(Person::getfirstName)
    .setHeader("Hierarchy");
grid.addColumn(Person::getAge).setHeader("Age");

grid.addExpandListener(event ->
    message.setValue(
    String.format("Expanded %s item(s)",
        event.getItems().size())
        + "\n" + message.getValue()));
grid.addCollapseListener(event ->
    message.setValue(
    String.format("Collapsed %s item(s)",
        event.getItems().size())
        + "\n" + message.getValue()));
```

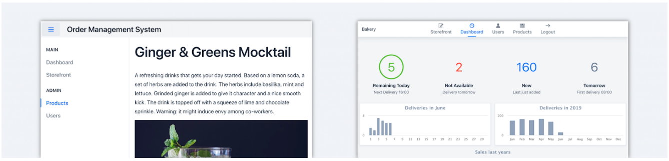
4.4. Layouts

4.4.1. App Layout

The App Layout component provides a quick and easy way to display a typical application layout structure. You can set a logo, navigation menus, page content, and more.

The App Layout component includes multiple, flexible configuration options and subparts typically found in applications. For example, using simple configuration options, you can choose to use only a horizontal or vertical navigation bar, or use both bars, plus an additional header bar. You can set specific behavior for each element.

The component is fully responsive and the elements intuitively adjust and modify, depending on the user's screen size and device type. For example, on small mobile screens, side menus collapse and open with animation, and top menus are repositioned below the main content.



Ginger & Greens Mocktail

A refreshing drinks that gets your day started. Based on a lemon soda, a set of herbs are added to the drink. The herbs include basilika, mint and lettuce. Grinded ginger is added to give it character and a nice smooth kick. The drink is topped off with a squeeze of lime and chocolate sprinkle. Warning: it might induce envy among co-workers.

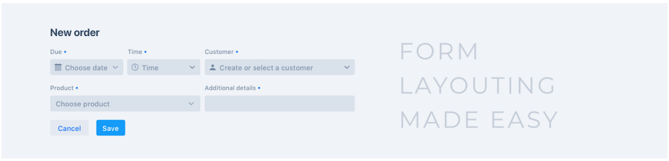
Usage:

```java
public class MainView extends AppLayout {
    public MainView() {
        setPrimarySection(AppLayout.Section.DRAWER);
        Image img = new Image("https://i.imgur" +
            ".com/GPpnszs.png", "Vaadin Logo");
        img.setHeight("44px");
        addToNavbar(new DrawerToggle(), img);
        Tabs tabs = new Tabs(new Tab("Home"),
            new Tab("About"));
        tabs.setOrientation(Tabs.Orientation.VERTICAL);
        addToDrawer(tabs);
    }
}
```

4.4.2. Form Layout

The Form Layout component is designed to layout form fields.

The component is fully responsive and the elements intuitively adjust and modify, depending on the user's screen size and device type.



Usage:

```
FormLayout nameLayout = new FormLayout();

TextField titleField = new TextField();
titleField.setLabel("Title");
titleField.setPlaceholder("Sir");
TextField firstNameField = new TextField();
firstNameField.setLabel("First name");
firstNameField.setPlaceholder("John");
TextField lastNameField = new TextField();
lastNameField.setLabel("Last name");
lastNameField.setPlaceholder("Doe");

nameLayout.add(titleField, firstNameField,
    lastNameField);

nameLayout.setResponsiveSteps(
    new ResponsiveStep("0", 1),
    new ResponsiveStep("21em", 2),
    new ResponsiveStep("22em", 3));
```

4.4.3. Ordered Layout

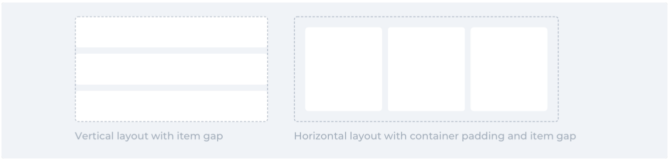
There are two ordered layout components:

- **Vertical Layout**: Align subcomponents vertically.

- **Horizontal Layout**: Aligns subcomponents horizontally.

Both components provide a parent layout container to which you can add subcomponents. The subcomponents are positioned in the order in which they are added. The size of the Horizontal Layout component is determined by the size of the subcomponents, and that of the Vertical Layout component is 100% wide, by default.

You can configure the border, padding, margins, spacing, and expansion ratio, to make the result more visually appealing.


Vertical layout with item gap
Horizontal layout with container padding and item gap

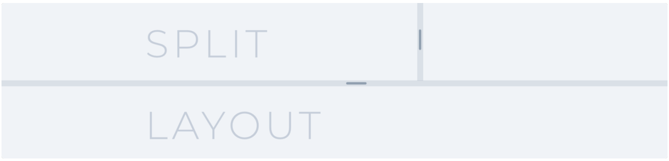
Usage:

```
// padding and spacing is on by default
VerticalLayout layout = new VerticalLayout();
layout.getStyle().set("border", "1px solid #9E9E9E");

Component component1 = createComponent(1, "#78909C");
Component component2 = createComponent(2, "#546E7A");
Component component3 = createComponent(3, "#37474F");

layout.add(component1, component2, component3);

// shorthand methods for changing the component
// theme variants
layout.setPadding(false);
layout.setMargin(true);
// just a demonstration of the API,
// by default the spacing is on
layout.setSpacing(true);
```

4.4.4. Split Layout

The Split Layout component allows you to partition a layout into two areas that the user can resize by dragging the splitter. The initial splitter position is determined by the size of the subcomponents.

You can set size limits, define custom styling and more. You can also create complex layouts by nesting several Split Layout components.



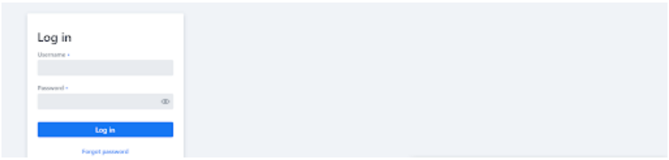
Usage:

```
SplitLayout layout = new SplitLayout(
        new Label("First content component"),
        new Label("Second content component"));
```

4.4.5. Login

Vaadin provides a number of login components, including the Login Form, Login Overlay, and Login i18n components

All login components support password managers, internationalization (i18n), password recovery, validation and error messaging, login event listeners, and more.



Usage:

```
LoginForm component = new LoginForm();
component.addLoginListener(e -> {
    boolean isAuthenticated = authenticate(e);
    if (isAuthenticated) {
        navigateToMainPage();
    } else {
        component.setError(true);
    }
});
```

4.5. Installing the Components

The `vaadin-core` module includes all open-source components, such as TextField, Button and Grid. The `vaadin` module extends this set to include all officially-supported components in Vaadin, like Vaadin Charts.

Example: Declaring all Vaadin components in pom.xml.

```
<dependencies>
    <!-- other dependencies -->

    <!-- component dependency -->
    <dependency>
        <groupId>com.vaadin</groupId>
        <artifactId>vaadin</artifactId>
        <version>${vaadin.version}</version>
    </dependency>
</dependencies>
```

As an alternative to using the single dependency, you can declare individual components as dependencies.

Example: Adding the Button component in your pom.xml using Maven.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-bom</artifactId>
            <version>
                ${vaadin.platform.version}
            </version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>


<dependencies>
    <!-- other dependencies -->
    <!-- component dependency -->
    <dependency>
        <groupId>com.vaadin</groupId>
        <artifactId>vaadin-button-flow</artifactId>
    </dependency>
</dependencies>
```

4.6. Vaadin Component Directory

In addition to built-in components, you can find hundreds of additional prebuilt Java and Web Components contributed by the Vaadin community in https://vaadin.com/directory.

5. Grid

The `Grid` component allows you to display and edit tabular data, set out in rows and columns.

The component is included in this documentation as a good example of a complex component, which is highly flexible yet simple to use, and offers a lot of functionality out of the box.

`Grid` features include:

- **Header and footer**: In addition to plain text, the header and footer can contain components. Allowing components makes it easy to implement additional functionality, such as filtering.
- **Sorting**: Column sorting is built in. Users can click the column header to sort the data, and shift click to enable secondary sorting criteria.
- **Scrolling**: The data area can be scrolled both vertically and horizontally. You can freeze the left columns to keep them in view when scrolling horizontally.
- **Lazy loading**: The data is loaded lazily from the server: only visible data is actually loaded. This provides an excellent user experience, even for low bandwidth devices, such as mobile phones.

5.1. Binding to Data

By default, `Grid` is bound to a `List` of items. You can use the `setItems()` method to set the items.

Example: Showing a list of beans in a `Grid`.

```
// Have some data
List<Person> people = Arrays.asList(
        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(Person::getName).setHeader("Name");
grid.addColumn(person -> Integer.toString(
               person.getYearOfBirth()))
        .setHeader("Year of birth");

layout.add(grid);
```

Behind the scenes `Grid` uses the `DataProvider` interface to communicate with the backend. The `setItems` method is a shorthand to create a `ListDataProvider`. For a large amount of data or other advanced use cases you should probably use the `DataProvider` interface and lazy loading. See Data Providers for more.

5.2. Handling Selection Changes

The `Grid` doesn't implement the `HasValue` interface directly. Other selection components do typically implement this interface. For this reason, selection handling in the `Grid` is different from typical selection components. The `Grid` supports three selection options: single selection, multiple selection, and no selection. Each option is defined by a specific selection model.

For basic switching between selection models, you can use the `setSelectionMode(SelectionMode)` method. Possible options are `SINGLE` (default), `MULTI`, or `NONE`.

To access the selection API or to use Grid as an input field with `Binder`, you can use `asSingleSelect()` or `asMultiSelect()`, depending on the currently defined selection mode. Both the `SingleSelect` and `MultiSelect` interfaces implement the `HasValue` interface. In the `MultiSelect` interface the value type is a `Set` of the item type.

Example: Using the `HasValue` interface with single and multi-select mode.

```
Grid<Person> grid = new Grid<>();

grid.setSelectionMode(SelectionMode.SINGLE);
SingleSelect<Grid<Person>, Person> personSelect =
        grid.asSingleSelect();
// personSelect can now be used with Binder or
// HasValue interface
personSelect.addValueChangeListener(e -> {
    Person selectedPerson = e.getValue();
});

grid.setSelectionMode(SelectionMode.MULTI);
MultiSelect<Grid<Person>, Person> multiSelect =
        grid.asMultiSelect();
multiSelect.addValueChangeListener(e -> {
    Set<Person> selectedPersons = e.getValue();
});
```

Alternatively you can use a grid-specific selection API. To get the selected value or values in any selection model, you can use a `SelectionListener`, with the provided generic `SelectionEvent`, to get the selected value or values.

Example: Using `addSelectionListener` to get all selected items.

```java
Grid<Person> grid = new Grid<>();

// switch to multiselect mode
grid.setSelectionMode(SelectionMode.MULTI);

grid.addSelectionListener(event -> {
   Set<Person> selected = event.getAllSelectedItems();
   message.setText(selected.size() + " items selected");
});
```

NOTE The listener is attached to the selection model and not the grid. It stops getting events when the selection mode is changed.

You can use the `select(T)` method to programmatically select values. In multi-selection mode, this adds the given item to the selection.

Example: Using the `select(T)` method.

```java
// in single-select, only one item is selected
grid.select(defaultItem);

// switch to multi select, clears selection
grid.setSelectionMode(SelectionMode.MULTI);
// Select items 2-4
people.subList(2, 3).forEach(grid::select);
```

You can get the current selection from the `Grid` using the `getSelectedItems()` method. The returned `Set` contains one item in single-selection mode, or several items in multi-selection mode.

WARNING If you change the grid's selection mode, it clears the selection and fires a selection event. To keep the previous selection, reset the selection afterwards, using the `select()` method.

WARNING

If you change the grid's items with either `setItems()` or the used `DataProvider`, it clears the selection and fires a selection event. To retain the previous selection, reset the selection afterwards, using the `select()` method.

5.2.1. Selection Models

You can access the used selection model using the `getSelectionModel()` method. The return type is the `GridSelectionModel` that has a generic selection model API, but you can cast that to the specific selection model type, typically either `SingleSelectionModel` or `MultiSelectionModel`.

You can also get the selection model using the `setSelectionMode(SelectionMode)` method.

Example: Using the `setSelectionMode(SelectionMode)` method to get the selection model.

```
// the default selection model
GridSingleSelectionModel<Person> defaultModel =
    (GridSingleSelectionModel<Person>) grid
        .getSelectionModel();

// Use multi-selection mode
GridMultiSelectionModel<Person> selectionModel =
    (GridMultiSelectionModel<Person>) grid
        .setSelectionMode(SelectionMode.MULTI);
```

Single-selection Model

Obtaining a reference to the `SingleSelectionModel` allows you access to a fine-grained API for the single-selection use case.

You can use the `addSingleSelect(SingleSelectionListener)` method to access `SingleSelectionEvent` that includes additional convenience methods and API options.

In single-selection mode, it is possible to control whether the empty (null) selection is allowed. This is enabled by default.

Example: Disallowing empty (null) selection using the `setDeselectAllowed()` method.

```
// preselect value
grid.select(defaultItem);

GridSingleSelectionModel<Person> singleSelect =
    (GridSingleSelectionModel<Person>) grid
        .getSelectionModel();

// disallow empty selection
singleSelect.setDeselectAllowed(false);
```

5.2.2. Multi-selection Model

In multi-selection mode, a user can select multiple items by selecting checkboxes in the left column.

Obtaining a reference to the `MultiSelectionModel` allows you access to a fine-grained API for the multi-selection use case.

You can use the `addMultiSelectionListener(MultiSelectionListener)` method to access `MultiSelectionEvent` that includes additional convenience methods and API options.

Example: Using the `addMultiSelectionListener` method to access selection changes.

```
// Grid in multi-selection mode
Grid<Person> grid = new Grid<>();
grid.setItems(people);
GridMultiSelectionModel<Person> selectionModel =
    (GridMultiSelectionModel<Person>) grid
        .setSelectionMode(SelectionMode.MULTI);

selectionModel.selectAll();

selectionModel.addMultiSelectionListener(event -> {
    message.setText(String.format(
            "%s items added, %s removed.",
            event.getAddedSelection().size(),
            event.getRemovedSelection().size()));

    // Allow deleting only if there's any selected
    deleteSelected.setEnabled(
            event.getNewSelection().isEmpty());
});
```

5.3. Handling Item-click Events

It is possible to handle item-click or double-click events, in addition to handling selection events. These can be used with selection events or on their own.

Example: Disabling the selection mode using `SelectionMode.NONE`, but still getting item-click events.

```
grid.setSelectionMode(SelectionMode.NONE);
grid.addItemClickListener(event -> System.out
        .println(("Clicked Item: " + event.getItem())));
```

- The clicked item, together with other information about click, is available via the event.

- Selection events are no longer available, and no visual selection is displayed when a row is clicked.

It is possible to get separate selection and click events.

Example: Using `Grid` in multi-selection mode with an added click (or double-click) listener.

```
grid.setSelectionMode(SelectionMode.MULTI);
grid.addItemDoubleClickListener(event ->
        copy(grid.getSelectedItems()));
```

- In the example code, we call a local `copy` method with the currently selected items when user double clicks a row.

5.4. Configuring Columns

The `addColumn()` method allows you to add columns to the `Grid`.

The column configuration is defined in `Grid.Column` objects that are returned by the `addColumn` method. The `getColumns()` method returns a list of currently configured columns.

The setter methods in `Column` have fluent-API functionality, making it easy to chain configuration calls for columns.

Example: Chaining column configuration calls.

```
Column<Person> nameColumn = grid
    .addColumn(Person::getName)
    .setHeader("Name")
    .setFlexGrow(0)
    .setWidth("100px")
    .setResizable(false);
```

5.4.1. Column Keys

You can set an identifier key for a column using the `setKey()` method. This allows you to retrieve the column from the grid at any time.

Example: Using the `setKey` method to set an identifier key for a column.

```
nameColumn.setKey("name");
grid.getColumnByKey("name").setWidth("100px");
```

5.4.2. Automatically Adding Columns

You can configure `Grid` to automatically add columns for every property in a bean, by passing the class of the bean type to the grid’s constructor. The property names are set as the column keys, and you can use them to further configure the columns.

Example: Automatically adding columns by passing the bean-type class to the constructor.

```
Grid<Person> grid = new Grid<>(Person.class);
grid.getColumnByKey("yearOfBirth").setFrozen(true);
```

- This constructor only adds columns for the direct properties of the bean type

- The values are displayed as strings.

You can add columns for nested properties by using the dot notation with the `setColumn(String)` method.

Example: Adding a column for `postalCode`. Assumes `Person` has a reference to an `Address` object that has a `postalCode` property.

```
grid.addColumn("address.postalCode");
```

- The column’s key is "address.postalCode" and its header is "Postal Code".
- To use these `String` properties in `addColumn`, you need to use the `Grid` constructor that takes a bean-class parameter.

Defining and Ordering Automatically-Added Columns

You can define which columns display, and the order in which they disaply, in the grid, using the `setColumns` method.

Example: Defining columns and their order using the `setColumns` method.

```
Grid<Person> grid = new Grid<>(Person.class);
grid.setColumns("name", "age", "address.postalCode");
```

TIP
You can also use the `setColumns` method to reorder the columns you already have.

NOTE When calling `setColumns`, all columns that are currently present in the grid are removed, and only those passed as parameters are added.

To add custom columns before the auto-generated columns, use the `addColumns` method instead. You can avoid creating the auto-generated columns using the `Grid(Class, boolean)` constructor.

Example: Adding custom columns.

```
Grid<Person> grid = new Grid<>(Person.class, false);
grid.addColumn(person -> person.getName().split(" ")[0])
    .setHeader("First name");
grid.addColumns("age", "address.postalCode");
```

NOTE An `IllegalArgumentException` is thrown if you attempt to add columns that are already present the grid.

Sortable Automatic Columns

By default, all property-based columns are sortable, if the property type implements `Comparable`.

Many data types, such as `String`, `Number`, primitive types and `Date`/`LocalDate`/`LocalDateTime` are `Comparable`, and therefore also sortable, by default.

To make the column of a non-comparable property type sortable, you need to define a custom `Comparator`. See Column Sorting for more.

You can disable sorting for a specific column, using the `setSortable` method.

Example: Disabling sorting on the `address.postalCode` column.

```
grid.getColumnByKey("address.postalCode")
        .setSortable(false);
```

You can also define a list of columns as sortable using the `setSortableColumns` method. This makes all other columns unsortable.

Example: Setting defined columns as sortable.

```
// All columns except "name" and "yearOfBirth"
// will be not sortable
grid.setSortableColumns("name", "yearOfBirth");
```

5.4.3. Column Headers and Footers

By default, columns do not have a header or footer. These need to be set explicitly using the `setHeader` and `setFooter` methods. Both methods have two overloads: one accepts a plain text string and the other a `TemplateRenderer`.

Examples: Setting headers and footers.

```
// Sets a simple text header
nameColumn.setHeader("Name");
// Sets a header using Html component,
// in this case simply bolding the caption "Name"
nameColumn.setHeader(new Html("<b>Name</b>"));

// Similarly for the footer
nameColumn.setFooter("Name");
nameColumn.setFooter(new Html("<b>Name</b>"));
```

See [Using Template Renderers] for more.

5.4.4. Column Reordering

Column reordering is not enabled by default. You can use the `setColumnReorderingAllowed()` method to allow drag and drop column reordering.

Example: Enabling column reordering.

```
grid.setColumnReorderingAllowed(true);
```

5.4.5. Hiding Columns

Columns can be hidden by calling the `setVisible()` method in `Column`.

NOTE A hidden column still sends the data required for its rendering to the client side. Best practice is to remove (or not add) columns, if the data is not needed on the client side. This reduces the amount of data sent and lessens the load on the client.

5.4.6. Removing Columns

You can remove a single column using the `removeColumn(Column)` and `removeColumnByKey(String)` methods. You can also remove all currently configured columns using the `removeAllColumns()` method.

5.4.7. Setting Column Widths

By default, columns do not have a defined width. They resize automatically based on the data displayed.

You can set the column width:

- Relatively, using flex grow ratios, by using the `setFlexGrow()` method, or
- Explicitly, using a CSS string value with `setWidth()` (with flex grow set to `0`).

You can also enable user column resizing using the `setResizable()` method. The column is resized by dragging the column separator.

5.4.8. Setting Frozen Columns

You can freeze a number of columns using the `setFrozen()` method. This ensures that the set number of columns on the left remain static (and visible) when the user scrolls horizontally.

When columns are frozen, user reordering is limited to only among other frozen columns.

Example: Setting a column as frozen.

```
nameColumn.setFrozen(true);
```

5.4.9. Grouping Columns

You can group multiple columns together by adding them in the `HeaderRow` of the grid.

When you retrieve the `HeaderRow`, using the `prependHeaderRow` or `appendHeaderRow` methods, you can then group the columns using the `join` method. In addition, you can use the `setText` and `setComponent` methods on the join result to set the text or component for the joined columns.

Example: Grouping columns

```
// Create a header row
HeaderRow topRow = grid.prependHeaderRow();

// group two columns under the same label
topRow.join(nameColumn, ageColumn)
        .setComponent(new Label("Basic Information"));

// group the other two columns in the same header row
topRow.join(streetColumn, postalCodeColumn)
        .setComponent(new Label("Address Information"));
```

5.5. Using Renderers in Columns

You can configure columns to use a renderer to show the data in the cells.

Conceptually, there are three types of renderer:

1. **Basic renderer**: Renders basic values, such as dates and numbers.
2. **Template renderer**: Renders content using HTML markup and Polymer data-binding syntax.
3. **Component renderer**: Renders content using arbitrary components.

5.5.1. Using Basic Renderers

There are several basic renderers that you can use to configure grid columns.

LocalDateRenderer

Use `LocalDateRenderer` to render `LocalDate` objects in the cells.

Example: Using `LocalDateRenderer` with the `addColumn` method.

```
grid.addColumn(new LocalDateRenderer<>(
        Item::getEstimatedDeliveryDate,
        DateTimeFormatter.ofLocalizedDate(
                FormatStyle.MEDIUM)))
    .setHeader("Estimated delivery date");
```

`LocalDateRenderer` works with a `DateTimeFormatter` or a String format to properly render `LocalDate` objects.

Example: Using a String format to render the `LocalDate` object.

```
grid.addColumn(new LocalDateRenderer<>(
        Item::getEstimatedDeliveryDate,
        "dd/MM/yyyy"))
    .setHeader("Estimated delivery date");
```

LocalDateTimeRenderer

Use `LocalDateTimeRenderer` to render `LocalDateTime` objects in the cells.

Example: Using `LocalDateTimeRenderer` with the `addColumn` method.

```
grid.addColumn(new LocalDateTimeRenderer<>(
        Item::getPurchaseDate,
        DateTimeFormatter.ofLocalizedDateTime(
                FormatStyle.SHORT,
                FormatStyle.MEDIUM)))
    .setHeader("Purchase date and time");
```

`LocalDateTimeRenderer` also works with `DateTimeFormatter` (with separate style for date and time) or a String format to properly render `LocalDateTime` objects.

Example: Using a String format to render the `LocalDateTime` object.

```
grid.addColumn(new LocalDateTimeRenderer<>(
        Item::getPurchaseDate,
        "dd/MM HH:mm:ss")
).setHeader("Purchase date and time");
```

NumberRenderer

Use `NumberRenderer` to render any type of Number in the cells. It is especially useful for rendering floating-point values.

Example: Using `NumberRenderer` with the `addColumn` method.

```
grid.addColumn(new NumberRenderer<>(Item::getPrice,
        NumberFormat.getCurrencyInstance())
).setHeader("Price");
```

It is possible to setup the `NumberRenderer` with a String format, and an optional null representation.

Example: Using a String format to render a price.

```
grid.addColumn(new NumberRenderer<>(
        Item::getPrice, "$ %(,.2f",
        Locale.US, "$ 0.00")
).setHeader("Price");
```

NativeButtonRenderer

Use `NativeButtonRenderer` to create a clickable button in the cells. It creates a native `<button>` on the client side. Click and tap (for touch devices) events are handled on the server side.

Example: Using `NativeButtonRenderer` with the `addColumn` method.

```
grid.addColumn(
    new NativeButtonRenderer<>("Remove item",
       clickedItem -> {
           // remove the item
    })
);
```

You can configure a custom label for each item.

Example: Configuring `NativeButtonRenderer` to use a custom label.

```
grid.addColumn(new NativeButtonRenderer<>(
        item -> "Remove " + item,
        clickedItem -> {
            // remove the item
        })
);
```

5.5.2. Using Template renderers

Providing a `TemplateRenderer` for a column allows you to define the content of cells using HTML markup, and to use Polymer notations for data binding and event handling.

Example: Using `TemplateRenderer` to bold the names of the persons.

```
Grid<Person> grid = new Grid<>();
grid.setItems(people);

grid.addColumn(TemplateRenderer
        .<Person>of("<b>[[item.name]]</b>")
        .withProperty("name", Person::getName)
).setHeader("Name");
```

- The template string is passed for the static `TemplateRenderer.of()` method.
- Every property in the template needs to be defined in the `withProperty()` method.
- `[[item.name]]` is Polymer syntax for binding properties for a list of items. See the Polymer 3 documentation[10] for more.

Creating Custom Properties

You can use a `TemplateRenderer` to create and display new properties (i.e. properties the item did not originally contain).

Example: Using `TemplateRenderer` to compute the approximate age of each person and add it in a new column. Age is the current year less the birth year.

```
grid.addColumn(TemplateRenderer
        .<Person>of("[[item.age]] years old")
        .withProperty("age",
                person -> Year.now().getValue()
                        - person.getYearOfBirth())
).setHeader("Age");
```

Binding Beans

If an object contains a bean property that has sub properties, it is only necessary to make the bean accessible by calling the `withProperty()` method. The sub properties become accessible automatically.

Example: Using the `withProperty()` method to access numerous sub properties. Assumes `Person` has a field for the `Address` bean, which has `street`, `number` and `postalCode` fields with corresponding getter and setter methods.

```
grid.addColumn(TemplateRenderer.<Person>of(
        "<div>[[item.address.street]], number " +
        "[[item.address.number]]<br>" +
        "<small>[[item.address.postalCode]]</small>" +
        "</div>")
        .withProperty("address", Person::getAddress))
    .setHeader("Address");
```

Handling Events

You can define event handlers for the elements in your template, and hook them to server-side code, by calling the `withEventHandler()` method on your `TemplateRenderer`. This is useful for editing items in the grid.

Example: Using the `withEventHandler()` method to map defined method names to server-side code. The snippet adds a new column with two buttons: one to edit a property of the item and one to remove the item. Both buttons define a method to call for `on-click` events.

```
grid.addColumn(TemplateRenderer.<Person>of(
     "<button on-click='handleUpdate'>Update</button>" +
     "<button on-click='handleRemove'>Remove</button>")
    .withEventHandler("handleUpdate", person -> {
        person.setName(person.getName() + " Updated");
        grid.getDataProvider().refreshItem(person);
    }).withEventHandler("handleRemove", person -> {
        ListDataProvider<Person> dataProvider =
            (ListDataProvider<Person>) grid
                .getDataProvider();
        dataProvider.getItems().remove(person);
        dataProvider.refreshAll();
    })).setHeader("Actions");
```

- When the server-side data used by the grid is edited, the grid's `DataProvider` is refreshed by calling the `refreshItem()` method. This ensures the changes show up in the element.
- When an item is removed, the `refreshAll()` method call ensures that all the data is updated.
- You need to use Polymer notations for event handlers. `on-click` (with a dash) is Polymer syntax for the native `onclick`.
- `TemplateRenderer` has a fluent API, so you can chain the commands, like `TemplateRenderer.of().withProperty().withProperty().withEventHandler()…`

5.5.3. Using Component Renderers

You can use any component in the grid cells by providing a `ComponentRenderer` for a column.

To define how the component will be generated for each item, you need to pass a `Function` for the `ComponentRenderer`.

Example: Adding a column that contains a different icon, depending on the person's gender.

```
Grid<Person> grid = new Grid<>();
grid.setItems(people);

grid.addColumn(new ComponentRenderer<>(person -> {
    if (person.getGender() == Gender.MALE) {
        return new Icon(VaadinIcon.MALE);
    } else {
        return new Icon(VaadinIcon.FEMALE);
    }
})).setHeader("Gender");
```

It is also possible to provide a separate `Supplier` to create the component, and a `Consumer` to configure it for each item.

Example: Using `ComponentRenderer` with a `Consumer`.

```
SerializableBiConsumer<Div, Person> consumer =
        (div, person) -> div.setText(person.getName());
grid.addColumn(
        new ComponentRenderer<>(Div::new, consumer))
    .setHeader("Name");
```

If the component is the same for every item, you only need to provide the `Supplier`.

Example: Using `ComponentRenderer` with a `Supplier`.

```
grid.addColumn(
    new ComponentRenderer<>(
            () -> new Icon(VaadinIcon.ARROW_LEFT)));
```

You can create complex content for the grid cells by using the component APIs.

Example: Using `ComponentRenderer` to create complex content that listens for events and wraps multiple components in layouts.

```
grid.addColumn(new ComponentRenderer<>(person -> {

    // text field for entering a new name for the person
    TextField name = new TextField("Name");
    name.setValue(person.getName());

    // button for saving the name to backend
    Button update = new Button("Update", event -> {
        person.setName(name.getValue());
        grid.getDataProvider().refreshItem(person);
    });

    // button that removes the item
    Button remove = new Button("Remove", event -> {
        ListDataProvider<Person> dataProvider =
            (ListDataProvider<Person>) grid
                .getDataProvider();
        dataProvider.getItems().remove(person);
        dataProvider.refreshAll();
    });

    // layouts for placing the text field on top
    // of the buttons
    HorizontalLayout buttons =
            new HorizontalLayout(update, remove);
    return new VerticalLayout(name, buttons);
})).setHeader("Actions");
```

- Editing grid items requires refreshing the grid’s `DataProvider`. The reasoning is the same as for Handling Events above.

See Data Providers for more.

5.6. Enabling Expanding Rows

The `Grid` supports expanding rows that reveal more detail about the items. The additional information is hidden, unless the user choses to reveal it, keeping the grid appearance clean and simple, while simultaneously allowing detailed explanations.

You can enable expanding rows using the `setItemDetailsRenderer()` method, which allows either a `TemplateRenderer` or a `ComponentRenderer` to define how the details are rendered.

Example: Using the `setItemDetailsRenderer` method with a `ComponentRenderer`.

```
grid.setItemDetailsRenderer(
    new ComponentRenderer<>(person -> {
        VerticalLayout layout = new VerticalLayout();
        layout.add(new Label("Address: " +
                person.getAddress().getStreet() + " " +
                person.getAddress().getNumber()));
        layout.add(new Label("Year of birth: " +
                person.getYearOfBirth()));
        return layout;
}));
```

By default, the row's detail opens by clicking the row. Clicking the row again, or clicking another row (to open its detail), automatically closes the first row's detail. You can disable this behavior by calling the `grid.setDetailsVisibleOnClick(false)` method. You can show and hide item details programmatically using the `setDetailsVisible()` method, and test whether an item's detail is visible using the `isDetailsVisible()` method.

NOTE
By default, items are selected by clicking them. If you want clicking to only show the item details without selection, you need to use the `grid.setSelectionMode(SelectionMode.NONE)` method.

5.7. Column Sorting

By default, this is how column sorting in the grid works:

- The first click on the column header sorts the column.
- The second click reverses the sort order.
- The third click resets the column to its unsorted state.

If multi-sorting is enabled, the user can sort by multiple columns. The first click sorts the first column. Subsequent clicks on second and more sortable column headers, add secondary and more sort criteria.

5.7.1. Defining Column Sorting

The difference between in-memory and backend sorting is key to understanding the sorting mechanism:

- **In-memory sorting** is sorting that is applied by the framework to items fetched from the backend, before returning them to the client.
- **Backend sorting** is applied by providing a list of `QuerySortOrder` objects to your `DataProvider`, that typically passes the sort hints to the backend code, and in some cases all the way to database queries. See Data Providers for more.

The sorting mechanism is flexible. You can configure in-memory and backend sorting together or separately.

The sections that follow detail options you can use to set up sorting for your grid.

Using a Sort Property Name

By using a sort property, you can override or customise the property or multiple properties that are used for sorting the column. This option includes both in-memory and backend sorting. The property is defined at the time of column construction and uses a sort property name.

You can use the `addColumn` method to set a sort property to be used for backend sorting when the column is added to the grid.

Example: Using the `addColumn` method to set a column sort property.

```
grid.addColumn(Person::getAge, "age").setHeader("Age");
```

- The `Age` column uses the values returned by the `Person::getAge` method to do in-memory sorting.
- The column uses the `age` string to build a `QuerySortOrder` that is sent to the `DataProvider` to do the backend sorting.

You can also define multiple properties.

Example: Using the `addColumn` method to set multiple column sort properties.

```
grid.addColumn(person -> person.getName() + " " +
        person.getLastName(), "name", "lastName"
).setHeader("Name");
```

- With multiple properties, the `QuerySortOrder` objects are created in the order they are declared.

You can also use use properties created for your `TemplateRenderer`.

Example: Using the `addColumn` method with `TemplateRenderer` to set column sort properties.

```
grid.addColumn(TemplateRenderer.<Person> of(
        "<div>[[item.name]]<br>" +
        "<small>[[item.email]]</small></div>")
        .withProperty("name", Person::getName)
        .withProperty("email", Person::getEmail),
    "name", "email")
    .setHeader("Person");
```

- For in-memory sorting to work correctly, the values returned by the `ValueProviders` in the `TemplateRenderer` (`Person::getName` and `Person::getEmail` in this example) should implement `Comparable`.
- The names of the sort properties must match the names of the properties in the template (set via `withProperty`).

Using a Comparator

This option is for in-memory sorting only, and uses a custom comparator.

If you need custom logic to compare items for sorting, or if your underlying data is not `Comparable`, you can set a `Comparator` for your column.

Example: Using the `setComparator` method to configure a comparator for a column.

```
grid.addColumn(Person::getName)
    .setComparator((person1, person2) ->
        person1.getName()
            .compareToIgnoreCase(person2.getName()))
    .setHeader("Name");
```

Setting Backend Sort Properties

This option is for backend sorting only, and uses a sort property name. It is similar to Using a Sort Property Name, but excludes in-memory sorting.

You can use the `setSortProperty` method to set strings describing backend properties to be used when sorting the column.

Example: Using the `setSortProperty` method to define sorting.

```
grid.addColumn(Person::getName)
        .setSortProperty("name", "email")
        .setHeader("Person");
```

- Unlike using the sorting properties in the `addColumn` method directly, calling `setSortProperty` does not configure any in-memory sorting.
- A `SortOrderProvider` is created automatically when the sort properties are set.

Setting a SortOrderProvider

This option is for backend sorting and uses a `SortOrderProvider`.

If you need fine-grained control over how `QuerySortOrder` objects are created and sent to the `DataProvider`, you can define a `SortOrderProvider`.

Example: Defining a `SortOrderProvider` for backend sorting.

```
grid.addColumn(Person::getName)
    .setSortOrderProvider(direction -> Arrays
        .asList(new QuerySortOrder("name", direction),
                new QuerySortOrder("email", direction))
        .stream())
    .setHeader("Person");
```

5.7.2. Enabling and Disabling Column Sorting

When a column is `sortable`, it displays the sorter element in the column header.

You can use the `setSortable` method to toggle the sorter element on an off.

Example: Using the `setSortable` method to disable sorting.

```
column.setSortable(false);
```

Setting a column as not `sortable` does not delete a `Comparator`, sort property, or `SortOrderProvider` that was previously set. You can toggle the `sortable` flag on and off, without reconfiguration.

To check if a column is currently `sortable`, you can use the `isSortable` method.

Example: Checking if a column is sortable.

```
column.isSortable();
```

5.7.3. Enabling Multi-sorting

To allow users to sort by more than one column at the same time, you can use the `setMultiSort` method to enable multi-sorting at the grid level.

Example: Using the `setMultiSort` method to enable multi-sorting.

```
grid.setMultiSort(true);
```

5.7.4. Receiving Sort Events

You can add a `SortListener` to the grid to receive general sort events. Every time sorting of the grid is changed, an event is fired. You can access the `DataCommunicator` to receive the sorting details.

Example: Using the `addSortListener` method to add a `SortListener`.

```
grid.addSortListener(event -> {
    String currentSortOrder = grid.getDataCommunicator()
            .getBackEndSorting().stream()
            .map(querySortOrder -> String.format(
                    "{sort property: %s, direction: %s}",
                    querySortOrder.getSorted(),
                    querySortOrder.getDirection()))
            .collect(Collectors.joining(", "));
    System.out.println(String.format(
            "Current sort order: %s. User-clicked: %s.",
            currentSortOrder, event.isFromClient()));
});
```

5.8. Styling the Grid

Styling the `Grid` component (or any Vaadin component) requires some Web Component and shadow-DOM knowledge. Styling depends on the components position in the DOM:

- If the component is in the shadow DOM, you can apply styling within the component or using variables.
- If the component is in the "normal" DOM (not in the shadow DOM), normal CSS styling applies.

In addition, the `Grid` supports the `theme` attribute that allows you to easily customize component styling.

Example: `Celebrity` grid used in styling examples below.

```
Grid<Celebrity> grid = new Grid<>();
grid.setItems(Celebrity.getPeople());
grid.addClassName("styled");
grid.addColumn(new ComponentRenderer<>(person -> {
    TextField textField = new TextField();
    textField.setValue(person.getName());
    textField.addClassName("style-" +
            person.getGender());
    textField.addValueChangeListener(
        event -> person.setName(event.getValue()));
    return textField;
})).setHeader("Name");

grid.addColumn(new ComponentRenderer<>(person -> {
    DatePicker datePicker = new DatePicker();
    datePicker.setValue(person.getDob());
    datePicker.addValueChangeListener(event -> {
        person.setDob(event.getValue());
    });
    datePicker.addClassName("style-" +
            person.getGender());
    return datePicker;
})).setHeader("DOB");

grid.addColumn(new ComponentRenderer<>(person -> {
    Image image = new Image(person.getImgUrl(),
            person.getName());
    return image;
})).setHeader("Image");
```

5.8.1. Styling with the Theme Property

The default Lumo theme includes different variations that you can use to style the grid. You can provide one or more variations.

Example: Using the `addThemeNames` method to define theme variations for the grid.

```
grid.addThemeNames("no-border", "no-row-borders",
        "row-stripes");
```

5.8.2. Styling with CSS

You can use normal CSS styling for the content in the grid cells. While the `Grid` component itself is in the shadow DOM, the actual values (cell contents) are in slots and therefore in the light DOM.

Example: Setting the maximum size for images in the grid.

```
vaadin-grid vaadin-grid-cell-content img {
    max-height: 4em;
}
```

- `vaadin-grid-cell-content` is in the light DOM, and the selector `vaadin-grid vaadin-grid-cell-content` points to the grid’s cells.

You can also use a class to apply styles to a specific component instance.

Example: Applying rounded borders and centering images in a Grid with "styled" class name.

```
vaadin-grid.styled vaadin-grid-cell-content img {
    border-radius: 2em;
    margin-left: 50%;
    transform: translate(-50%);
}
```

5.8.3. Styling by Overriding Component Styles

You can use custom styles to style the grid itself. This is achieved by overriding the default grid styling.

Example: Overriding component styles with custom styles.

```
<dom-module id="custom-grid" theme-for="vaadin-grid">
  <template>
    <style>
      :host(.styled) #table {
        border-radius: 20px;
        box-shadow: 0 0 5px rgba(81, 203, 238, 1);
        border: 1px solid rgba(81, 203, 238, 1);
      }
      :host(.styled) #header {
        border: none;
        border-bottom: 1px solid rgba(81, 203, 238, 1);
      }
      :host(.styled) #header tr {
        text-align: center;
        text-shadow: 0 0 3px rgba(81, 203, 238, 1);
        text-transform: uppercase;
      }
    </style>
  </template>
</dom-module>
```

- This sets custom styles for a `vaadin-grid` with a "styled" class. Grid’s without this class remain as normal.
- `theme-for="vaadin-grid"` indicates that it is overriding `vaadin-grid -components` styling.
- `:host(.styled)` is a selector for `vaadin-grid` that has "styled" as a class. Outside the shadow DOM this is `vaadin-grid.styled`, but because the shadow DOM is boxed in its own DOM, it is selected with `:host([selector])`.

5.8.4. Styling with CSS Variables

Although the shadow DOM is boxed and usually cannot be altered from the outside, you can use CSS variables to pass information to the shadow DOM. CSS variables pass through all levels of the DOM (light and shadow), and once a variable is set, it is available everywhere in that DOM.

CSS variables only work with components that support them, such as `Grid`.

The following example takes you through the process of styling the grid with text fields of different colors, depending on the user's gender.

1. Introduce CSS variable usage for the `TextField` component.

```
<dom-module id="custom-text-field"
        theme-for="vaadin-text-field">
  <template>
    <style>
      .vaadin-text-field-container[part="input-field"] {
          background-color: var(--custom-text-field-bg,
                var(--lumo-contrast-10pct));
      }
    </style>
  </template>
</dom-module>
```

- This overrides `vaadin-text-field` styles.
- The only change is the introduction of the `--custom-text-field-bg` variable.

   1. Change the variable, based on the person's gender.

```
.styled .style-female {
    --custom-text-field-bg: #ff99cc;
}
.styled .style-male {
    --custom-text-field-bg: #99ccff;
}
```

- After this change, any text field used with `.styled .style-female/male` will have the specified background color.
- This also applies to composite components that have internal text fields.

----

[10] https://polymer-library.polymer-project.org/3.0/api/elements/dom-repeat

6. Binding Data to Components

6.1. Binding Data to Forms

In many applications users provide structured data by completing fields in forms. This data is typically represented in code as an instance of a business object (JavaBean), for example a Person in an HR application.

The `Binder` class allows you to define how the values in a business object are bound to fields in the UI.

`Binder` reads the values in the business object and converts them from the format expected by the business object to the format expected by the field, and *vice versa*.

`Binder` can only bind components that implement the `HasValue` interface, for example `TextField` and `ComboBox`.

It is also possible to validate user input and present the validation status to the user in different ways.

6.1.1. How to Bind Form Data

The following steps include everything needed to load, edit and save values for a form. Java 8 method references are used.

To bind data to a form:

1. Create a `Binder` and bind the input fields.

NOTE There can only be one `Binder` instance for each form. You should use it for all fields in the form.

```java
Binder<Person> binder = new Binder<>(Person.class);

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);
```

2. Use the `Binder` to:

    a. Load values from a `person` into the field.

    b. Allow the user to edit the values.

    c. 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 (ValidationException e) {
            notifyValidationException(e);
        }
});

// Updates the fields again with the
// previously saved values
Button resetButton = new Button("Reset",
        event -> binder.readBean(person));
```

- Every time `writeBean` is called, the data is validated and then copied from the UI to the business object.
- If the data is invalid, a `ValidationException` that includes all errors in the data, is thrown. This is the reason `writeBean` is in a try/catch block.

It is also possible to use a Lambda expression, instead of a method reference.

```
// With lambda expressions
binder.bind(titleField,
        person -> person.getTitle(),
        (person, title) -> {
            person.setTitle(title);
            logger.info("setTitle: {}", title);
        });
```

6.1.2. Binding Read-only Data

To bind a component to read-only data, you can use a `null` value for the `setter`.

Example: Using a `null` value setter.

```
TextField fullName = new TextField();
binder.forField(fullName)
        .bind(Person::getFullName, null);
```

To bind components that do not implement the `HasValue` interface to read-only data, you can use the `ReadOnlyHasValue` helper class.

Example: Using the `ReadOnlyHasValue` helper class.

```
Label fullNameLabel = new Label();
ReadOnlyHasValue<String> fullName =
        new ReadOnlyHasValue<>(
            text -> fullNameLabel.setText(text));
binder.forField(fullName)
        .bind(Person::getFullName, null);
```

6.2. Validating and Converting User Input

`Binder` supports:

- Validating user input, and
- Converting value types from types used in business objects to types used in bound UI components, and *vice versa*.

These concepts go hand in hand, because validation can be based on a converted value, and the ability to convert a value is a kind of validation in itself.

Vaadin includes several validators and converters that you can implement.

6.2.1. Validating User Input

It is typical for applications to restrict the kind of value the user is allowed to enter into certain fields.

Defining Validators

`Binder` allows you to define validators for each bound field. By default, validators run whenever the user changes the field value. The validation status is also checked when writing to the bean.

You should define the field validator between the `forField` and `bind` code lines when creating the binding.

Example: Defining a validator using a `Validator` instance or an inline 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,
        "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);
```

- `Binder.forField` works like a builder: the `forField` call starts the process, it is followed by various configuration calls for the field, and `bind` is the final method of the configuration.
- `asRequired` is used for mandatory fields:
    - A visual "required" indicator displays.
    - If the user leaves the field empty, an error message displays.

Customizing Validation Error Messages

You can customize the way error messages display by defining a `ValidationStatusHandler` or configuring the `Label` for each binding. The label is used to show the status of the field. The label can be used for validation errors, as well as confirmation and helper messages.

Example: Configuring validation messages for email and minimum length validation.

```
Label emailStatus = new Label();
emailStatus.getStyle().set("color", "Red");
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,
        "Name must contain at least three characters")
    // Define how the validation status is displayed
    .withValidationStatusHandler(status -> {
        nameStatus.setText(status
                .getMessage().orElse(""));
        nameStatus.setVisible(status.isError());
    })
    // Finalize the binding
    .bind(Person::getName, Person::setName);
```

- The `withStatusLabel(Label label)` method sets the given label to show an error message if the validation fails.

As an alternative to using labels, you can set a custom validation status handler, using the `withValidationStatusHandler` method. This allows you to customize how the binder displays error messages and is more flexible than using the status label approach.

Adding Multiple Validators

You can add multiple validators for the same binding.

Example: Defining two validators: first, for the email input, and second, 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);
```

Triggering Revalidation

The validation of one field can depend on the value of another field. You can achieve this by saving the binding to a local variable and triggering revalidation when the other field fires a value-change event.

Example: Storing a binding for later revalidation.

```
Binder<Trip> binder = new Binder<>(Trip.class);
DatePicker departing = new DatePicker();
departing.setLabel("Departing");
DatePicker returning = new DatePicker();
returning.setLabel("Returning");

// Store return date binding so we can
// revalidate it later
Binder.Binding<Trip, LocalDate> returningBinding =
    binder
       .forField(returning).withValidator(
               returnDate -> !returnDate
                       .isBefore(departing.getValue()),
               "Cannot return before departing")
       .bind(Trip::getReturnDate, Trip::setReturnDate);

// Revalidate return date when departure date changes
departing.addValueChangeListener(
        event -> returningBinding.validate());
```

6.2.2. Converting User Input

You can bind application data to a UI field component, even if the types do not match.

Examples where this is useful include an application-specific type for a postal code that the user enters in a `TextField`, or requesting the user enter only integers in a `TextField`, or selecting enumeration values in a `Checkbox` field.

Defining Converters

Like validators, each binding can have one or more converters, with an optional error message.

You can define converters using callbacks (typically lambda expressions), method references, or by implementing the

`Converter` interface.

Examples: Defining converters.

```
TextField yearOfBirthField =
    new TextField("Year of birth");

binder.forField(yearOfBirthField)
    .withConverter(
        new StringToIntegerConverter("Not a number"))
    .bind(Person::getYearOfBirth,
        Person::setYearOfBirth);

// Checkbox for marital status
Checkbox marriedField = new Checkbox("Married");

binder.forField(marriedField).withConverter(
  m -> m ? MaritalStatus.MARRIED : MaritalStatus.SINGLE,
  MaritalStatus.MARRIED::equals)
.bind(Person::getMaritalStatus,
    Person::setMaritalStatus);
```

Adding Multiple Converters

You can add multiple converters (and validators) for each binding.

Each validator or converter is used in the order defined in the class. The value is passed along until:

- A final converted value is stored in the business object, or
- The first validation error or impossible conversion is encountered.

Example: Validator and converter sequence.

```
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);
```

When updating UI components, values from the business object are passed through each converter in reverse order (without validation).

NOTE

Although it is possible to use a converter as a validator, best practice is to use a validator to check the contents of a field, and a converter to modify the value. This improves code clarity and avoids excessive boilerplate code.

Conversion Error Messages

You can define a custom error message to be used if a conversion throws an unchecked exception.

When using callbacks, you should provide one converter in each direction. If the callback used for converting the user-provided value throws an unchecked exception, the field is marked as invalid, and the exception message is used as the validation error message. Java runtime exception messages are typically written for developers, and may not be suitable for end users.

Example: Defining a custom conversion error message.

```
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);
```

Implementing the Converter Interface

There are two methods to implement in the `Converter` interface:

- `convertToModel` receives a value that originates from the user.
    - The method returns a `Result` that either contains a converted value or a conversion error message.
- `convertToPresentation` receives a value that originates from the business object.
    - This method returns the converted value directly. It is assumed that the business object only contains valid values.

Example: Implementing a String to Integer Converter.

```java
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("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 to find the `Locale` to be used for the conversion.

6.3. Loading From and Saving To Business Objects

Once your bindings are set up, you are ready to fill the bound

UI components with data from your business objects.

Changes can be written to business objects automatically or manually.

6.3.1. Reading and Writing Automatically

Writing to business objects automatically when the user makes changes in the UI is ususally the most convenient option.

You can bind the values directly to an instance, by allowing `Binder` to automatically save values from the fields.

Example: Automatically saving field values.

```
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);
    }
});
```

- The `validate()` call ensures that bean-level validators are checked when saving automatically.

WARNING

When you use the `setBean` method, the business object instance updates whenever the user changes the value of a bound field. If another part of the application simultaneously uses the same instance, that part could display changes before the user saves. You can prevent this by using a copy of the edited object, or by manually writing to only update the object when the user saves.

6.3.2. Reading Manually

You can use the `readBean` method to manually read values from a business object instance into the UI components.

Example: Using the `readBean` method.

```
Person person = new Person("John Doe", 1957);

binder.readBean(person);
```

- This example assumes that `binder` has been configured with a `TextField` bound to the name property.
- The value "John Doe" displays in the field.

6.3.3. Validating and Writing Manually

To prevent displaying multiple errors to the user, validation errors only display after the user has edited each field and submitted (loaded) the form.

You can explicitly validate the form or attempt to save the values to a business object, even if the user has not edited a field.

Example: Explicitly validating a form.

```
// This will make all current validation errors visible
BinderValidationStatus<Person> status =
        binder.validate();

if (status.hasErrors()) {
   notifyValidationErrors(status.getValidationErrors());
}
```

Writing the field values to a business object fails if any of the bound fields contain an invalid value. You can deal with invalid values in a number of different ways:

Example: Handling a checked exception.

```
try {
    binder.writeBean(person);
    MyBackend.updatePersonInDatabase(person);
} catch (ValidationException e) {
    notifyValidationErrors(e.getValidationErrors());
}
```

Example: Checking a return value.

```
boolean saved = binder.writeBeanIfValid(person);
if (saved) {
    MyBackend.updatePersonInDatabase(person);
} else {
    notifyValidationErrors(binder.validate()
            .getValidationErrors());
}
```

Example: Adding bean-level validators.

```
binder.withValidator(
       p -> p.getYearOfMarriage() > p.getYearOfBirth(),
       "Marriage year must be bigger than birth year.");
```

- The `withValidator(Validator)` method runs on the bound bean after update of the values of the bound fields.
- Bean-level validators also run as part of `writeBean(Object)`, `writeBeanIfValid(Object)` and `validate(Object)`, if the content passes all field-level validators.

NOTE

For bean-level validators, the bean must be updated before the validator runs. If a bean-level validator fails in `writeBean(Object)` or `writeBeanIfValid(Object)`, the bean reverts to the state it was in before returning from the method. Remember to check your `getters/setters` to ensure there are no unwanted side effects.

6.3.4. Tracking Binding Status

`Binder` keeps track of which bindings have been updated by the user and which bindings are in an invalid state. It fires an event when there are status changes. You can use this event to appropriately enable and disable the form buttons, depending on the current status of the form.

Example: Enabling the save and reset buttons when changes are detected.

```
binder.addStatusChangeListener(event -> {
    boolean isValid = event.getBinder().isValid();
    boolean hasChanges = event.getBinder().hasChanges();

    saveButton.setEnabled(hasChanges && isValid);
    resetButton.setEnabled(hasChanges);
});
```

6.4. Binding Beans to Forms

Business objects are typically implemented as JavaBeans in an application. `Binder` supports binding the properties of a business object to UI components in your forms.

6.4.1. Manual Data Binding

You can use reflection based on bean property names to bind values. This reduces the amount of code needed when binding to fields in the bean.

Examples: Binding using reflection based on bean property names.

```
Binder<Person> binder = new Binder<>(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

Be cautious when using strings to identify properties. A typo in the string, or a subsequent changes to the setter and getter method names, will result in a runtime exception.

6.4.2. Automatic Data Binding

The `bindInstanceFields` method facilitates automatic data binding.

UI fields are typically defined as members of a UI Java class. This allows you to access the fields easily using the different methods made available by the class. In this scenario, binding the fields is also simple, because when you pass the object to the UI class, the `bindInstanceFields` method matches the fields of the object to the properties of the related business object, based on their names.

Example: Using the `bindInstanceFields` method to bind all fields in a UI class.

```java
public class MyForm extends VerticalLayout {
    private TextField firstName =
            new TextField("First name");
    private TextField lastName =
            new TextField("Last name");
    private ComboBox<Gender> gender =
            new ComboBox<>("Gender");

    public MyForm() {
        Binder<Person> binder =
                new Binder<>(Person.class);
        binder.bindInstanceFields(this);
    }
}
```

- This binds the `firstName` text field to the "firstName" property in the item, `lastName` text field to the “lastName” property, and the `gender` combo box to the “gender” property.

Without this method, it would be necessary to bind each field separately.

Example: Binding each field separately.

```
binder.forField(firstName)
    .bind(Person::getFirstName, Person::setFirstName);
binder.forField(lastName)
    .bind(Person::getLastName, Person::setLastName);
binder.forField(gender)
    .bind(Person::getGender, Person::setGender);
```

Specifying Property Names

The `bindInstanceFields` method processes all Java member fields with a type that extends `HasValue` (such as `TextField`) that can be mapped to a property name.

If the field name does not match the corresponding property name in the business object, you can use the `@PropertyId` annotation to specify the property name.

Example: Using the `@PropertyId` annotation to specify the "sex" property for the `gender` field.

```
@PropertyId("sex")
private ComboBox<Gender> gender =
        new ComboBox<>("Gender");
```

Configuring Converters and Validators

When using the automatic `bindInstanceFields` method to bind fields, all converters and validators must be configured beforehand using a special `forMemberField` configurator. It works similar to the `forField` method, but it requires no explicit call to a bind method. If the `bindInstanceFields` method finds incompatible property-field pairs, it throws an `IllegalStateException`.

Alternatively, you can bind properties that need validators manually and then bind all remaining fields using the `bindInstanceFields` method. This method skips the properties that have already been bound manually.

Example: Manually specifying `StringToIntegerConverter` before calling the `bindInstanceFields` method.

```
TextField yearOfBirthField =
        new TextField("Year of birth");

binder.forField(yearOfBirthField)
  .withConverter(
    new StringToIntegerConverter("Must enter a number"))
  .bind(Person::getYearOfBirth, Person::setYearOfBirth);

binder.bindInstanceFields(this);
```

If you use JSR-303 validators, you should use `BeanValidationBinder` that picks validators automatically when using `bindInstanceFields`.

6.4.3. Using JSR 303 Bean Validation

You can use `BeanValidationBinder` if you prefer to use JSR 303 Bean Validation annotations such as `Max`, `Min`, `Size`, etc.

`BeanValidationBinder` extends `Binder` (and therefore has the same API), but its implementation automatically adds validators based on JSR 303 constraints.

To use Bean Validation annotations, you need a JSR 303 implementation like Hibernate Validator available in your classpath. If your environment, such as Java EE container, does not provide the implementation, you can use the following dependency in Maven:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.4.1.Final</version>
</dependency>
```

Defining Constraints for Properties

Example: Using JSR 303 Bean Validation annotations with `BeanValidationBinder`

```
public class Person {
    @Max(2000)
    private int yearOfBirth;

    // Non-standard constraint provided by
    // Hibernate Validator
    @NotEmpty
    private String name;

    // + other fields, constructors, setters and getters
}

BeanValidationBinder<Person> binder =
        new BeanValidationBinder<>(Person.class);

binder.bind(nameField, "name");
binder.forField(yearOfBirthField)
    .withConverter(
        new StringToIntegerConverter("Enter a number"))
    .bind("yearOfBirth");
```

Constraints defined for properties in the bean, work in the same way as if configured programmatically when the binding is created. For example, the following code snippets have the same result:

Example: Declarative Bean Validation annotation.

```
public class Person {
    @Max(value = 2000, message =
     "Year of Birth must be less than or equal to 2000")
    private int yearOfBirth;
```

Example: Programmatic validation using Binder specific API.

```
binder.forField(yearOfBirthField)
  .withValidator(
    yearOfBirth -> yearOfBirth <= 2000,
    "Year of Birth must be less than or equal to 2000")
  .bind(Person::getYearOfBirth, Person::setYearOfBirth);
```

NOTE As an alternative to defining constraint annotations for specific properties, you can define constraints on the bean level, but Vaadin's `BeanValidationBinder` does not currently support them. It simply ignores all JSR 303 validations that are not assigned directly to properties.

Automatically Marking Form Fields as Required

Some built-in validators in the bean validation API suggest that a value is required in input field. `BeanValidationBinder` automatically enables the visual "required" indicator using the `HasValue.setRequiredIndicatorVisible(true)` method for properties annotated with such validators. By default, `@NotNull`, `@NotEmpty` and `@Size` (if `min()` value is greater than 0) configures the field as required. You can change this behavior using the `BeanValidationBinder.setRequiredConfigurator` method.

Example: Overriding the default `@Size` behavior.

```
binder.setRequiredConfigurator(
        RequiredFieldConfigurator.NOT_EMPTY
            .chain(RequiredFieldConfigurator.NOT_NULL));
```

6.5. Showing a List of Data with Data Providers

Many applications present the user with a list of items from which they can select one or more items to work on. Example lists include inventory records to survey, messages requiring a response, or blog drafts to edit or publish.

A listing component is a component that:

- Displays one or several properties from a list of items,
- Allows the user to inspect the data and mark items as selected, and
- Optionally, allows the user to edit items directly in the component.

There are a number of officially-supported listing components, such as `ListBox` and `ComboBox`. Each component has its own API to configure exactly how the data is represented and manipulated.

All list components have a `setItems` method to define which items display. They also have the `DataProvider` interface for more fine-grained control of the displayed data.

6.5.1. Changing How Items Display

By default, most components use the `toString()` method to display items. If this is not suitable, you can change the behavior by configuring the component. Components are configured with one or more callbacks that define how to display the items.

Example: A `ComboBox` component that lists status items and uses the `Status.getLabel()` method to represent each status. There is also a `Grid` with two columns, Name and Year of birth. .

```
ComboBox<Status> comboBox = new ComboBox<>();
comboBox.setItemLabelGenerator(Status::getLabel);

Grid<Person> grid = new Grid<>();
grid.addColumn(Person::getName).setHeader("Name");
grid.addColumn(person -> Integer.toString(
               person.getYearOfBirth()))
        .setHeader("Year of birth");
```

6.5.2. Displaying In-memory Data

The easiest way to pass data to listing components is to use the `setItems` method. It accepts a collection, an array, or a stream of items.

Example: Passing data values 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)
);
```

Sorting In-memory Data

Listing components that allow the user to control the item-display order, such as `Grid`, are automatically also capable of sorting data by any property, provided the property type implements `Comparable`.

You can also define a custom `Comparator` if you need to customize the way a specific column is sorted. The comparator can be based on either the item instances or on the displayed property values.

Example: Defining a custom comparator.

```
grid.addColumn(Person::getName)
        .setHeader("Name")
        // Override default natural sorting
        .setComparator(Comparator.comparing(person ->
                person.getName().toLowerCase()));
```

NOTE This kind of sorting is only supported for in-memory data. See Sorting Lazy-loaded Data for how to sort data loaded from a backend service.

6.5.3. Lazy Loading Data from a Backend Service

When fetching data from a backend service, it is often more efficient to only load the items that currently display. For example, when loading all available data uses excessive memory or slows down page load.

NOTE Regardless of how you 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.

Assume you have a prebuilt backend service that fetches items from a database or a REST service.

Example: Prebuilt `PersonService`.

```java
public interface PersonService {
    List<Person> fetchPersons(int offset, int limit);
    int getPersonCount();
}
```

To use this service with a listing component, you can create a data provider that defines two callbacks using the `fromCallbacks` method.

Example: Data provider with callbacks that fetch specified items and the number of items available.

- The first callback loads specific items.
- The second callback finds out how many items are currently available.
- Information about the items to fetch is made available in a `Query` object that is passed to both callbacks
- Information about the items to fetch includes `offset`, `limit`, and additional details.

```
DataProvider<Person, Void> dataProvider =
    DataProvider.fromCallbacks(
        // 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
```

- The results of the first and second callbacks must be symmetric, so that fetching all available items using the first callback returns the number of items indicated by the second callback.
- If you impose any restrictions in the first callback, you must add the same restrictions for the second callback.
- The second `DataProvider` type parameter defines how the provider can be filtered. In the example, the filter type is `Void`, meaning filtering in not supported. See Filtering Lazy-loaded Data below for more.

Sorting Lazy-loaded Data

It is not practical to order items based on a `Comparator` when the items are loaded on demand, because this requires all items to be loaded and inspected.

Every backend has a defined way of ordering fetched items. Generally, ordering is based on a list of property names and whether it should be ascending or descending.

Example: `PersonService` interface with descending ordering based on a property name.

```
public interface PersonService {
    List<Person> fetchPersons(
    int offset,
    int limit,
    List<PersonSort> sortOrders);
    int getPersonCount();

    PersonSort createSort(
            String propertyName,
            boolean descending);
}
```

When using this service interface, you can enhance the data source by converting the provided sorting options into a format expected by the service.

Sorting options set in the component are available using the `query.getSortOrders()` method.

Example: Using the `query.getSortOrders()` method in a component.

```
DataProvider<Person, Void> dataProvider =
  DataProvider.fromCallbacks(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()
);
```

It is also necessary to configure the `Grid` to know which property name to include in the query when the user wants to sort by a specific column. When a data source does lazy loading, `Grid` and similar listing components, only allow the user to sort by columns if a sort property name is provided.

Example: Configuring a property name in `Grid` to be used for sort queries.

```
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)
        .setHeader("Name")
        .setSortProperty("name");

// Will not be sortable since no sorting info is given
grid.addColumn(Person::getYearOfBirth)
        .setHeader("Year of birth");
```

In some cases, providing a single property name is not enough. For example, if the backend sorts by multiple properties for one column in the UI, or if the backend sort order needs to be inverted when compared to the sort order defined by the user. In these cases, you need to define a callback that generates suitable `SortOrder` values for the given column.

Example: Generating a `SortOrder` by last name and then first name.

```
grid.addColumn(person ->
        person.getName() + " " + person.getLastName())
    .setHeader("Name")
    .setSortOrderProvider(
        // Sort according to last name, then first name
        direction -> Stream.of(
           new QuerySortOrder("lastName", direction),
           new QuerySortOrder("firstName", direction)));
```

Filtering Lazy-loaded Data

Different backends support filtering in different ways: some offer no filtering support, some allow filtering by a single value (of a specific type), and some support complex filtering options.

The following examples use the `ComboBox` component to demonstrate filtering in various scenarios.

Filtering by a Single String

A `DataProvider<Person, String>` accepts a single string to filter by in the query. How the data provider uses this value depends on the implementation. It could, for example, look for all Persons with a name beginning with the provided string.

Listing components that allow the user to control how displayed data is filtered, all use a specific filter type. For `ComboBox`, the filter is the string the user enters in the search field. This means that you can only use `ComboBox` with a data provider with a String filtering type.

Example: `DepartmentService` backend service.

```
public interface DepartmentService {
    List<Department> fetch(int offset, int limit,
            String filterText);
    int getCount(String filterText);
}
```

Example: `DataProvider` that uses the `DepartmentService` interface service methods to fill a `ComboBox` component with data.

```
DataProvider<Department, String>
createDepartmentDataProvider(DepartmentService service)
{
   return DataProvider.fromFilteringCallbacks(query -> {
       // getFilter returns Optional<String>
       String filter = query.getFilter().orElse(null);
       return service.fetch(query.getOffset(),
               query.getLimit(), filter).stream();
   }, query -> {
       String filter = query.getFilter().orElse(null);
       return service.getCount(filter);
   });
}
```

Example: Using the `DataProvider`.

```
DataProvider<Department, String> dataProvider =
        createDepartmentDataProvider(service);
ComboBox<Department> departmentComboBox =
        new ComboBox<>();
departmentComboBox.setDataProvider(dataProvider);
```

Filtering Based on Another Component

In this scenario, filtering is based on the value of a different component than the combo box component you are working on. For example, you are defining a combo box to select an employee that is filtered by the value of a combo box for selecting a department. The employee combo box should also allow filtering by text entered by the user.

Example: Backend `EmployeeService`.

```
public interface EmployeeService {
    List<Employee> fetch(int offset, int limit,
                         EmployeeFilter filter);
    int getCount(EmployeeFilter filter);
}
public class EmployeeFilter {
    private String filterText;
    private Department department;

    public EmployeeFilter(String filterText,
                          Department department) {
        this.filterText = filterText;
        this.department = department;
    }

    public String getFilterText() {
        return filterText;
    }

    public void setFilterText(String filterText) {
        this.filterText = filterText;
    }

    public Department getDepartment() {
        return department;
    }

    public void setDepartment(Department department) {
        this.department = department;
    }
}
```

Because there are two different types of filters - one for the input text and one for the selected department - you can no longer use `DataProvider<Employee, String>` directly. To overcome this, you can create a data provider wrapper that allows you to set the filter value to include in the query programmatically.

Example: Using the `withConfigurableFilter` method to create a `ConfigurableFilterDataProvider<Employee,`

`String, Department>`.

```
ConfigurableFilterDataProvider<Employee, String,
Department> getDataProvider(EmployeeService service) {
  DataProvider<Employee, EmployeeFilter> dataProvider =
  DataProvider.fromFilteringCallbacks(query -> {
      // getFilter returns Optional<String>
      EmployeeFilter filter = query.getFilter()
              .orElse(null);
      return service.fetch(query.getOffset(),
              query.getLimit(), filter).stream();
    }, query -> {
        EmployeeFilter filter = query.getFilter()
                .orElse(null);
        return service.getCount(filter);
    });

  ConfigurableFilterDataProvider<Employee, String,
  Department> configurableFilterDataProvider =
      dataProvider.withConfigurableFilter(
         (String filterText, Department department) ->
            new EmployeeFilter(filterText, department));

  return configurableFilterDataProvider;
}
```

Example: Using the DataProvider:

```
ConfigurableFilterDataProvider<Employee, String,
Department> employeeDataProvider =
        getDataProvider(service);
ComboBox<Employee> employeeComboBox = new ComboBox<>();
employeeComboBox.setDataProvider(employeeDataProvider);
```

Example: Manually setting the department when it changes by calling the `setFilter` method.

```
departmentComboBox.addValueChangeListener(event -> {
    employeeDataProvider.setFilter(event.getValue());
    employeeDataProvider.refreshAll();
});
```

Flexible Filtering Using a Predicate Parameter

You can a predicate parameter in your service methods to implement flexible filtering.

Example: Backend `PersonService`.

```
public interface PersonService {
    List<Person> fetch(int offset, int limit,
            Optional<Predicate<Person>> predicate);
    int getCount(Optional<Predicate<Person>> predicate);
}
```

While it is still possible to use the `fromFilteringCallbacks` method to create a `DataProvider<Person, String>` directly, the example below is a far cleaner coding solution.

Example: Creating a `DataProvider<Person, Predicate<Employee>>` and converting it into a `DataProvider<Person, String>` using the `withConvertedFilter` method.

```
DataProvider<Person, String> getDataProvider(
        PersonService service) {
    DataProvider<Person, Predicate<Person>>
      predicateDataProvider =
        DataProvider.fromFilteringCallbacks(
          query -> service.fetch(query.getOffset(),
                query.getLimit(),
                query.getFilter()).stream(),
          query -> service.getCount(query.getFilter()));

    DataProvider<Person, String> dataProvider =
      predicateDataProvider.withConvertedFilter(
        text -> (person -> person.getName()
                .startsWith(text)));

    return dataProvider;
}
```

- The `withConvertedFilter` method allows you to use a data provider that filters by another type.
- The example filters a series of people by name. When users input text, it is not used directly to select data items from the existing objects. A lambda produces a predicate (another lambda) that filters the people by name.

Example: Using the DataProvider.

```
DataProvider<Person, String> dataProvider =
        getDataProvider(service);
ComboBox<Person> comboBox = new ComboBox<>();
comboBox.setDataProvider(dataProvider);
```

Filtering in the Grid Component

You can use the `withConfigurableFilter` method on a data provider to create a data provider wrapper that allows you to configure the filter that is passed through the query.

All components that use the same data provider refresh their data when a new filter is set.

Example: Using the `withConfigurableFilter` method to create a data provider wrapper.

```
DataProvider<Employee, String> employeeProvider =
        getEmployeeProvider();

ConfigurableFilterDataProvider<Employee, Void, String>
    wrapper = employeeProvider.withConfigurableFilter();

Grid<Employee> grid = new Grid<>();
grid.setDataProvider(wrapper);
grid.addColumn(Employee::getName).setHeader("Name");

searchField.addValueChangeListener(event -> {
    String filter = event.getValue();
    if (filter.trim().isEmpty()) {
        // null disables filtering
        filter = null;
    }

    wrapper.setFilter(filter);
});
```

- The filter type of the `wrapper` instance is `Void`. This means that the data provider does not support further filtering through the query. It is therefore not possible to use this data provider with a combo box.

Refreshing Data from a Backend Service

`DataProvider` has two methods, `refreshAll` and `refreshItems`, that you can use to ensure that backend changes reflect in all parts of you application.

Whether refreshing is required depends on your implementation and environment. Spring Data, for example, gives new instances with every request, and changes to the repository make old instances of the same object "stale". In cases similar to this, you should inform interested components by calling `dataProvider.refreshItem(newInstance)`. This works out of the box, if your beans have equals and hashCode implementations that check if the objects represent the same data. Since this is not always the case, when using `CallbackDataProvider` you can give it a `ValueProvider` that will provide a stable ID for the data objects. This is usually a method reference, for example `Person::getId`.

Example: `PersonService` interface with an update method that returns a new instance of the item. *Other functionality is omitted.*

```
public interface PersonService {
    Person save(Person person);
}
```

Example: Data provider to update a person’s name and save it to the backend.

```
DataProvider<Person, String> allPersonsWithId =
    new CallbackDataProvider<>(
        fetchCallback, sizeCallback, Person::getId);

Grid<Person> persons = new Grid<>();
persons.setDataProvider(allPersonsWithId);
persons.addColumn(Person::getName).setHeader("Name");

Button modifyPersonButton = new Button("", event -> {
    Person personToChange = service.fetchById(128);
    personToChange.setName("Changed person");
    Person newInstance = service.save(personToChange);
    allPersonsWithId.refreshItem(newInstance);
});
```

6.5.4. Using a ListDataProvider for Advanced In-memory Data Handling

As an alternative to assigning the items in a collection directly, you can create a `ListDataProvider` that contains the items a component should use.

Multiple components can share a single list data provider to display the same data. You can also configure the instance to filter out some items or display items in a specific order.

For components like `Grid` that can be separately configured to sort data in a specific way, sorting configured in the data provider is only used as a fallback. The fallback is used if no sorting is defined in the component, or if the order between items is considered equal by the component's sorting definition. Components update automatically when you change sorting in the data provider.

Example: Defining differing sort orders in the `ListDataProvider` and components.

```
ListDataProvider<Person> dataProvider =
        DataProvider.ofCollection(persons);

dataProvider.setSortOrder(Person::getName,
        SortDirection.ASCENDING);

Grid<Person> grid = new Grid<>(Person.class);
// The grid shows the persons sorted by name
grid.setDataProvider(dataProvider);

// Makes the combo box show persons in descending order
button.addClickListener(event -> {
    dataProvider.setSortOrder(Person::getName,
            SortDirection.DESCENDING);
});
```

Filtering In-memory Data

You can configure the list data provider to always apply a specific filter to limit which items display, or to filter by data that is not included in the displayed item caption.

Example: Defining a `ListDataProvider` with a filter.

```
ListDataProvider<Person> dataProvider =
        DataProvider.ofCollection(persons);

ComboBox<Person> comboBox = new ComboBox<>();
comboBox.setDataProvider(dataProvider);

departmentSelect.addValueChangeListener(event -> {
    Department selectedDepartment = event.getValue();
    if (selectedDepartment != null) {
        dataProvider.setFilterByValue(
                Person::getDepartment,
                selectedDepartment);
    } else {
        dataProvider.clearFilters();
    }
});
```

- The selected department in the `departmentSelect` component is used to dynamically change the persons displayed in the combo box.
- In addition to `setFilterByValue`, it is also possible to set a filter based on a predicate that tests each item or the value of some specific property in the item.
- Multiple filters can be stacked using `addFilter` methods instead of `setFilter`.

Notifying the Data Provider About Item Changes

The listing component does not automatically know about changes to the list of items or the individual items. For changes to reflect in the component, you need to notify the list data provider when items are changed, added or removed.

`DataProvider` has two methods for this purpose, `refreshAll` and `refreshItems`.

Example: Using the `refreshAll` and `refreshItems` methods to update the data provider.

```
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.refreshItem(personToChange);
        });
```

6.6. Creating a Component that Has a Value

To work with `Binder`, a component must implement the `HasValue` interface.

`HasValue` defines:

- Methods to access the value itself,

  An event when the value changes,

- Helpers to deal with empty values,
-
- `ReadOnly` mode,
- Required indicator.

6.6.1. Helper classes

You can use the following helper classes as a base class for custom components that display, and allow the user to change, a value:

- `AbstractField` is the most basic, but also the most flexible, base class. There are many details to take care of, but it supports complex use cases.
- `AbstractCompositeField` is similar to `AbstractField`, except it uses `Composite` instead of `Component` as the base class. It is suitable when the value input component is made up of several individual components.
- `AbstractSinglePropertyField` is suitable when the the value is based on a single-element property of the component's only element. This base class simplifies a common use case found in many Web Components that are similar in design to the native `<input>` element.

6.6.2. Using a Single-element Property as the Value

Many components are based on Web Components that have a property that contains the component's value. The property name is typically `value`, and it fires a `value-changed` event when changed.

When the property type is a string, number or boolean, all you need to do is to extend `AbstractSinglePropertyField` and call its constructor with the name of the property, the default value, and whether null values are allowed.

The `paper-slider` Web Component is a compliant example. It has an integer property named `value`, displays the slider at the 0 position if no value is set, and does not support showing no value at all.

Example: `PaperSlider` component that extends `AbstractSinglePropertyField` and works perfectly with `Binder`.

```java
@Tag("paper-slider")
@NpmPackage(value = "@polymer/paper-slider",
            version = "3.0.1")
@JsModule("@polymer/paper-slider/paper-slider.js")
public class PaperSlider
        extends AbstractSinglePropertyField<PaperSlider,
            Integer> {
    public PaperSlider() {
        super("value", 0, false);
    }
}
```

- The type parameters of `AbstractSinglePropertyField` are:
    - The type of the `getSource()` method in fired value-change events (`PaperSlider`).
    - The value type (`Integer`).
- The default value of `0` is automatically used by the `clear()` and `isEmpty()` methods: `clear()` sets the field value to the default value, and `isEmpty()` returns `true` if the field value is the default value.

NOTE Vaadin uses Polymer 3[11]. This version provides the best compatibility for integrating third-party Web Components.

Converting Property Values

With some Web Components, there is a Java type that is more suitable than the type of the element property.

It is possible to configure `AbstractSinglePropertyField` to apply a converter when changing, reading, or writing the value to the element property.

For example, the `value` property of `<input type="date">` is an ISO 8601 formatted string (YYYY-MM-DD). You can convert this into a `DatePicker` component for selecting a `LocalDate`.

Example: `DatePicker` component that allows the selection of a `LocalDate`. It extends `AbstractSinglePropertyField` and provides a callback to convert from `LocalDate` to `String`, and a callback in the opposite direction.

```
@Tag("input")
public class DatePicker
        extends AbstractSinglePropertyField<DatePicker,
            LocalDate> {

    public DatePicker() {
        super("value", null, String.class,
                LocalDate::parse,
                LocalDate::toString);

        getElement().setAttribute("type", "date");

        setSynchronizedEvent("change");
    }

    @Override
    protected boolean hasValidValue() {
        return isValidDateString(getElement()
                .getProperty("value"));
    }
}
```

- In this scenario, the convention of listening for an event named `<propertyName>-changed` is inappropriate. Instead, the `setSynchronizedEvent("change")` call overrides the default configuration, and listens for the change event in the browser.
- Overriding the `hasValidValue` method validates the element value before it is passed to the `LocalDate.parse` method that is defined in the constructor. In this way, invalid values are ignored, instead of causing exceptions.

6.6.3. Combining Multiple Properties Into One Value

`AbstractSinglePropertyField` only works with Web Components that have the value in a single-element property. However, the value of a component is often a composition of multiple-element properties that may belong to the same element or multiple elements. In this type of case, the best solution is often to extend `AbstractField`.

When you extend `AbstractField`, there are two different value representations to handle:

- **Presentation value**: The value displayed to the user in the browser, for example as element properties.
- **Model value**: The value available through the `getValue()` method.

Both values need to be kept in sync, except when the value is in the process of changing, or when the element properties are in an invalid state that cannot, or should not, be represented through `getValue()`.

To demonstrate, we build a `simple-date-picker` Web Component that has separate integer properties for the selected date: `year`, `month` and `dayOfMonth`. For each property there is a corresponding event when the user makes a change: `year-changed`, `month-changed` and `day-of-month-changed`.

Start by implementing a `SimpleDatePicker` component that extends `AbstractField` and passes the default value to its constructor.

```
@Tag("simple-date-picker")
public class SimpleDatePicker
    extends AbstractField<SimpleDatePicker, LocalDate> {

    public SimpleDatePicker() {
        super(null);
    }
}
```

NOTE The type parameters are the same as for `AbstractSinglePropertyField`: the `getSource()` type for the value-change event and the value type.

When you call `setValue(T value)` with a new value, `AbstractField` invokes the `setPresentationValue(T value)` method with the new value.

We will implement the `setPresentationValue(T value)` method so that the component updates the element properties to match the values set.

```
@Override
protected void setPresentationValue(LocalDate value) {
    Element element = getElement();

    if (value == null) {
        element.removeProperty("year");
        element.removeProperty("month");
        element.removeProperty("dayOfMonth");
    } else {
        element.setProperty("year", value.getYear());
        element.setProperty("month",
                value.getMonthValue());
        element.setProperty("dayOfMonth",
                value.getDayOfMonth());
    }
}
```

To handle value changes from the user’s browser, the component must listen to appropriate internal events and pass a new value to the `setModelValue(T value, boolean fromClient)` method. `AbstractField` will then check if the provided value has actually changed, and if it has, it fires a value-change event to all listeners.

We will update the constructor to define each of the element properties as synchronized, and add the same property-change listener to each of them.

```
public SimpleDatePicker() {
    super(null);

    setupProperty("year", "year-changed");
    setupProperty("month", "month-changed");
    setupProperty("dayOfMonth", "dayOfMonth-changed");
}

private void setupProperty(String name, String event) {
    Element element = getElement();

    element.synchronizeProperty(name, event);
    element.addPropertyChangeListener(name,
            this::propertyUpdated);
}
```

TIP

By default, `AbstractField` uses `Objects.equals` to determine whether a new value is the same as the previous value. If the `equals` method of the value type is not appropriate, you can override the `valueEquals` method to implement your own comparison logic.

WARNING

`AbstractField` should only be used with immutable-value instances. No value-change event is fired if the original `getValue()` instance is modified and passed to `setModelValue` or `setValue`.

The final step is to implement the property-change listener to create a new `LocalDate` based on the element property values, and pass it to `setModelValue`.

```
private void propertyUpdated(
        PropertyChangeEvent event) {
    Element element = getElement();

    int year = element.getProperty("year", -1);
    int month = element.getProperty("month", -1);
    int dayOfMonth = element.getProperty(
            "dayOfMonth", -1);

    if (year != -1 && month != -1 && dayOfMonth != -1) {
        LocalDate value = LocalDate.of(
                year, month, dayOfMonth);
        setModelValue(value, event.isUserOriginated());
    }
}
```

- If any of the properties are not filled in, `setModelValue` is not called. This means that `getValue()` returns the same value it returned previously.
- The component can call `setModelValue` from inside its `setPresentationValue` implementation. In this case, the value of the component is set to the value passed to `setModelValue`, which is used instead of the original value. This is useful to transform provided values, for example to make all strings uppercase.

If you have a percentage field that can only be 0-100%, for example, you can use:

```
@Override
protected void setPresentationValue(Integer value) {
        if (value < 0) value = 0;
        if (value > 100) value = 100;

        getElement().setProperty("value", false);
}
```

If the value set from the server is 138, for example, the following code sets the value at 100 on the client, but the internal server value remains 138. You can change the internal server value using :

```
@Override
protected void setPresentationValue(Integer value) {
        if (value < 0) value = 0;
        if (value > 100) value = 100;

        getElement().setProperty("value", value);
        setModelValue(value, false);
}
```

- Calling `setModelValue` from the `setPresentationValue` implementation does not fire a value-change event.
- If `setModelValue` is called multiple times, the value of the last invocation is used, and it is not necessary to worry about causing infinite loops.

6.6.4. Creating Fields from Other Fields

`AbstractCompositeField` makes it possible to create a field component that has a value based on the value of one or more internal fields.

To demonstrate, we build an employee selector field that allows the user to first select a department from a combo box, and then select an employee from the selected department in a second combo box. The component itself is a `Composite`, based on a `HorizontalLayout` that contains the two `ComboBox` components, displayed side by side.

TIP — Another use case for `AbstractCompositeField` is to create a field component that is based directly on another field, while converting the value from that field.

The class declaration is a mix of `Composite` and `AbstractField`.

1. The first type parameter defines the `Composite` content type, the second is for the value-change event `getSource()` type, and the third is the `getValue()` type of the field.
2. We also initialize instance fields for each `ComboBox`.

```
public class EmployeeField extends
        AbstractCompositeField<HorizontalLayout,
            EmployeeField, Employee> {
    private ComboBox<Department> departmentSelect =
            new ComboBox<>("Department");
    private ComboBox<Employee> employeeSelect =
            new ComboBox<>("Employee");
}
```

In the constructor:

1. Configure `departmentSelect` value changes to update the items in `employeeSelect`.
2. The employee selected in `employeeSelect` is set as the field’s value.
3. Both combo boxes are added to the horizontal layout.

```
public EmployeeField() {
    super(null);

    departmentSelect.setItems(
            EmployeeService.getDepartments());

    departmentSelect.addValueChangeListener(event -> {
        Department department = event.getValue();

        employeeSelect.setItems(EmployeeService
                .getEmployees(department));
        employeeSelect.setEnabled(department != null);
    });

    employeeSelect.addValueChangeListener(event ->
            setModelValue(event.getValue(), true));

    getContent().add(departmentSelect, employeeSelect);
}
```

As a next step, implement `setPresentationValue` to update the combo boxes according to a provided employee.

```
@Override
protected void setPresentationValue(Employee employee) {
    if (employee == null) {
        departmentSelect.clear();
    } else {
        departmentSelect.setValue(
                employee.getDepartment());
        employeeSelect.setValue(employee);
    }
}
```

Now we're going to change how the required indicator is shown for the field.

The default implementation assumes the component's root element reacts to a property named `required`, which works nicely for Web Components that mimic the API of `<input>`.

In our case, we want to show the required indicator for the employee combo box.

```
@Override
public void setRequiredIndicatorVisible(
        boolean required) {
   employeeSelect.setRequiredIndicatorVisible(required);
}

@Override
public boolean isRequiredIndicatorVisible() {
    return employeeSelect.isRequiredIndicatorVisible();
}
```

The last thing left is to implement `readonly` handling to mark both combo boxes as read only. The default implementation is similar to how required indicators are handled, except that it uses the `readonly` property instead.

```
@Override
public void setReadOnly(boolean readOnly) {
    departmentSelect.setReadOnly(readOnly);
    employeeSelect.setReadOnly(readOnly);
}

@Override
public boolean isReadOnly() {
    return employeeSelect.isReadOnly();
}
```

----

[11] https://polymer-library.polymer-project.org/3.0/docs/about_30

7. Routing and Navigation

Vaadin provides the `Router` class to structure the navigation of your web application into logical parts.

The router takes care of serving content when the user navigates within an application. It includes support for nested routes, access to URL parameters and more.

7.1. Using the @Route Annotation

You can use the `@Route` annotation to define any component as a route target for a given URL fragment.

Example: Defining the `HelloWorld` component as the default route target (empty route) for your application.

```java
@Route("")
public class HelloWorld extends Div {
    public HelloWorld() {
        setText("Hello world");
    }
}
```

Example: Defining the `SomePathComponent` component as the target for the specific route, `some/path`.

```java
@Route("some/path")
public class SomePathComponent extends Div {
    public SomePathComponent() {
        setText("Hello @Route!");
    }
}
```

- Assuming your app is running from the root context, when the user navigates to `http://example.com/some/path`, either by clicking a link in the application or entering the address in the address bar, the `SomePathComponent` component is shown on the page.

NOTE: If you omit the `@Route` annotation parameter, the route target is derived from the class name. For example, MyEditor becomes "myeditor", PersonView becomes "person" and MainView becomes "".

7.2. Navigation Lifecycle

The navigation lifecycle is made up of a number of events that are fired when a user navigates in an application from one state or view to another.

The events are fired to listeners added to the `UI` instance and to attached components that implement related *observer* interfaces.

7.2.1. BeforeLeaveEvent

`BeforeLeaveEvent` is the first event fired during navigation.

The event allows the navigation to be postponed, canceled, or changed to a different destination.

This event is delivered to any component instance implementing `BeforeLeaveObserver` that is attached to the UI before the navigation starts.

It is also possible to register a standalone listener for this event using the `addBeforeLeaveListener(BeforeLeaveListener)` method in `UI`.

A typical use case for this event is to ask the user whether they want to save any unsaved changes before navigating to another part of the application.

Postpone method

`BeforeLeaveEvent` includes the `postpone` method that can be used to postpone the current navigational transition until a specific condition is met.

Example: The client requests the user's confirmation before leaving the page:

```
public class SignupForm extends Div
        implements BeforeLeaveObserver {
    @Override
    public void beforeLeave(BeforeLeaveEvent event) {
        if (this.hasChanges()) {
            ContinueNavigationAction action =
                    event.postpone();
            ConfirmDialog.build("Are you sure you want"+
                    " to leave this page?")
                    .ifAccept(action::proceed)
                    .show();
        }
    }

    private boolean hasChanges() {
        // no-op implementation
        return true;
    }
}
```

Postponing interrupts the process of notifying observers and listeners. When the transition resumes, the remaining observers (those after the observer that initiated the postpone) are called.

Example:

- Assume the current page has 3 observers, `a`, `b` and `c`, which are notified in the same order.
- If `b` calls `postpone`, the call to `c` (and the rest of the transition process), is deferred.
  - If the transition is not resumed, `c` is never notified of the event and the transition never finishes.
  - If `b` executes `ContinueNavigationAction` to resume the transition, it continues from the point of interruption: `a` and `b` are not called again, but `c` is notified.

NOTE Only one navigation event may be postponed at a time. Starting a new navigation transition, while a previous one is in a postponed state, makes the postponed state obsolete, and executing `ContinueNavigationAction` has no effect.

7.2.2. BeforeEnterEvent

`BeforeEnterEvent` is the second event fired during navigation.

The event allows you to change the navigation to go to a destination that is different from the original.

This event is typically used to react to special situations, for example if there is no data to show, or if the user does not have appropriate permissions.

This event is delivered to any component instance implementing `BeforeEnterObserver` that is attached to the UI after navigation completes.

The event is fired:

- Only after a `postpone` (called during a `BeforeLeaveEvent`) has been continued.
- Before detaching and attaching components to make the UI match the location being navigated to.

It is also possible to register a standalone listener for this event using the `addBeforeEnterListener(BeforeEnterListener)` method in `UI`.

Rerouting

Both `BeforeLeaveEvent` and `BeforeEnterEvent` can be used to reroute dynamically.

Rerouting is typically used when there is a need to show completely different information in a particular state.

When the `reroute` method is called:

- The event is not fired by any further listeners or observers.
- The method triggers a new navigation phase, based on the new navigation target, and events are fired based on this instead.

Example: Rerouting when entering a `BlogList` with no results.

```java
@Route("no-items")
public class NoItemsView extends Div {
    public NoItemsView() {
        setText("No items found.");
    }
}

@Route("blog")
public class BlogList extends Div
        implements BeforeEnterObserver {
    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        // implementation omitted
        Object record = getItem();

        if (record == null) {
            event.rerouteTo(NoItemsView.class);
        }
    }

    private Object getItem() {
        // no-op implementation
        return null;
    }
}
```

NOTE There are several `rerouteTo` overload methods that can be used for different use cases.

Forward

The `forwardTo` method reroutes navigation and updates the browser URL.

Forwarding can be used during BeforeEnter and BeforeLeave lifecycle states to dynamically redirect to a different URL.

Calling `forwardTo` for the event stops propagation of the event to other listeners that have not yet been called. Instead, the method triggers a new navigation phase, based on the new navigation target, and fires new lifecycle events for the new forward navigation target.

Example: Forwarding when viewing `BlogList` without the required permissions.

```java
@Route("no-permission")
public class NoPermission extends Div {
    public NoPermission() {
        setText("No permission.");
    }
}

@Route("blog-post")
public class BlogPost extends Div
        implements BeforeEnterObserver {
    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        if (!hasPermission()) {
            event.forwardTo(NoPermission.class);
        }
    }

    private boolean hasPermission() {
        // no-op implementation
        return false;
    }
}
```

NOTE `forwardTo` has several overloads that serve different use cases.

7.2.3. AfterNavigationEvent

`AfterNavigationEvent` is the third and last event fired during navigation.

This event is typically used to update various parts of the UI after the actual navigation is complete. Examples include adjusting the content of a breadcrumb component and visually marking the active menu item as active.

The event is fired:

- After `BeforeEnterEvent`, and
- After updating which components are attached to the UI.

At this point, the current navigation state is actually shown to the user, and further reroutes and similar changes are no longer possible.

The event is delivered to any component instance implementing `AfterNavigationObserver` that is attached after completing the navigation.

It is also possible to register a standalone listener for this event using the `addAfterNavigationListener(AfterNavigationListener)` method in UI.

Example: Marking the active navigation element as active.

```java
public class SideMenu extends Div
        implements AfterNavigationObserver {
    Anchor blog = new Anchor("blog", "Blog");

    @Override
    public void afterNavigation(
          AfterNavigationEvent event) {
        boolean active = event.getLocation()
                .getFirstSegment()
                .equals(blog.getHref());
        blog.getElement()
                .getClassList()
                .set("active", active);
    }
}
```

7.3. Router Layouts and Nested Router Targets

7.3.1. RouterLayout Interface

All parent layouts of a navigation target component must implement the `RouterLayout` interface.

You can define a parent layout using the `Route.layout()` method.

Example: Render `CompanyComponent` inside `MainLayout`:

```java
@Tag("div")
@Route(value = "company", layout = MainLayout.class)
public class CompanyComponent extends Component {
}
```

NOTE

When using the `@Route("path")` annotation to define a route, the component by default renders in the `<body>` tag on the page. This is because the element returned by `HasElement.getElement()` is attached to the `<body>` tag.

Multiple Router Target Components

Where multiple router target components use the same parent layout, the parent layout instances remain the same when the user navigates between the child components.

See Updating Page Title on Navigation for more.

Multiple Parent Layouts

Use the `@ParentLayout` annotation to define a parent layout for components in the routing hierarchy.

You can create a parent layout for a parent layout, where necessary.

Example: `MainLayout` used for everything and `MenuBar` reused for views:

```
public class MainLayout extends Div
        implements RouterLayout {
}

@ParentLayout(MainLayout.class)
public class MenuBar extends Div
        implements RouterLayout {
    public MenuBar() {
        addMenuElement(TutorialView.class, "Tutorial");
        addMenuElement(IconsView.class, "Icons");
    }
    private void addMenuElement(
            Class<? extends Component> navigationTarget,
            String name) {
        // implementation omitted
    }
}

@Route(value = "tutorial", layout = MenuBar.class)
public class TutorialView extends Div {
}

@Route(value = "icons", layout = MenuBar.class)
public class IconsView extends Div {
}
```

- `MainLayout` encapsulates `MenuBar`, which in turn encapsulates `TutorialView` or `IconsView` depending on where the user has navigated to.

7.3.2. ParentLayout Route Control

A parent layout can supplement the navigation route by adding to the route location.

This is done by annotating the parent layout with `@RoutePrefix("prefix_to_add")`

Example: `PathComponent` receives the `some/path` route.

```
@Route(value = "path", layout = SomeParent.class)
public class PathComponent extends Div {
    // Implementation omitted
}

@RoutePrefix("some")
public class SomeParent extends Div
        implements RouterLayout {
    // Implementation omitted
}
```

Absolute Routes

You can use same parent component in many parts, without using a `@RoutePrefix` from the parent chain, or by only using it in defined parts.

This is done by adding `absolute = true` to either the `@Route` or `@RoutePrefix` annotations.

Example: Building a `MyContent` class to add "something" to multiple places in the `SomeParent` layout, without adding the route prefix to the navigation path:

```
@Route(value = "content", layout = SomeParent.class,
       absolute = true)
public class MyContent extends Div {
    // Implementation omitted
}
```

- Even though the full path would typically be `some/content`, we actually get only `content` because it has been defined as `absolute`.

Example: Defining `absolute` in the middle of the chain.

```
@RoutePrefix(value = "framework", absolute = true)
@ParentLayout(SomeParent.class)
public class FrameworkSite extends Div
        implements RouterLayout {
    // Implementation omitted
}

@Route(value = "tutorial", layout = FrameworkSite.class)
public class Tutorials extends Div {
    // Implementation omitted
}
```

- The bound route is `framework/tutorial` even though the full chain is `some/framework/tutorial`.
- If a parent layout defines a `@RoutePrefix`, the "default" child could have its route defined as `@Route("")` and be mapped to the parent layout route. For example, in the case of `Tutorials` with route `""` it would be mapped as `framework/`.

7.4. Routing and URL Parameters

7.4.1. URL Parameters for Navigation Targets

A navigation target that supports parameters passed through the URL should:

- Implement the `HasUrlParameter` interface, and
- Define the parameter type using generics.

`HasUrlParameter` defines the `setParameter` method that is called by the `Router`, based on values extracted from the URL. This method will always be invoked before a navigation target is activated.

Example: Defining a navigation target that takes a string parameter and produces a greeting string from it, which the target then sets as its own text content on navigation:

```
@Route(value = "greet")
public class GreetingComponent extends Div
        implements HasUrlParameter<String> {

    @Override
    public void setParameter(BeforeEvent event,
                             String parameter) {
        setText(String.format("Hello, %s!", parameter));
    }
}
```

- On startup, the navigation target is automatically configured for every `greet/<anything>` path, except where a separate navigation target with the exact `@Route` is configured to match `greet/<some specific path>`.

NOTE An exact navigation target always takes precedence when resolving the URL.

7.4.2. Optional URL parameters

URL parameters can be annotated as optional using `@OptionalParameter`.

Example: Defining the route to match both `greet` and `greet/<anything>`:

```
@Route("greet")
public class OptionalGreeting extends Div
        implements HasUrlParameter<String> {

    @Override
    public void setParameter(BeforeEvent event,
            @OptionalParameter String parameter) {
        if (parameter == null) {
            setText("Welcome anonymous.");
        } else {
            setText(String.format("Welcome %s.",
                    parameter));
        }
    }
}
```

NOTE A more specific route always takes precedence over a parameterised route.

7.4.3. Wildcard URL parameters

Where more parameters are needed, the URL parameter can also be annotated with `@WildcardParameter`.

Example: Defining the route to match `greet` and anything after it, for instance `greet/one/five/three`:

```java
@Route("greet")
public class WildcardGreeting extends Div
        implements HasUrlParameter<String> {

    @Override
    public void setParameter(BeforeEvent event,
            @WildcardParameter String parameter) {
        if (parameter.isEmpty()) {
            setText("Welcome anonymous.");
        } else {
            setText(String.format(
                    "Handling parameter %s.",
                    parameter));
        }
    }
}
```

NOTE The wildcard parameter will never be `null`.

NOTE More specific routes always take precedence over wildcard routes.

7.4.4. Query parameters

It is possible to get any query parameters contained in a URL, for example `?name1=value1&name2=value2`.

Use the `getQueryParameters()` method of the `Location` class to access query parameters. You can obtain the `Location` class through the `BeforeEvent` parameter of the `setParameter` method.

NOTE A `Location` object represents a relative URL made up of path segments and query parameters, but without the hostname, e.g. `new Location("foo/bar/baz?name1=value1")`.

```
@Override
public void setParameter(BeforeEvent event,
        @OptionalParameter String parameter) {

    Location location = event.getLocation();
    QueryParameters queryParameters = location
            .getQueryParameters();

    Map<String, List<String>> parametersMap =
            queryParameters.getParameters();
}
```

NOTE: `getQueryParameters()` supports multiple values associated with the same key, for example `https://example.com/?one=1&two=2&one=3` will result in the corresponding map `{"one" : [1, 3], "two": [2]}`.

7.5. URL Generation

`Router` exposes methods to get the navigation URL for registered navigation targets.

7.5.1. Standard Navigation Targets

For standard navigation targets, the request is a simple call for `Router.getUrl(Class target)`

Example: Returned URL is resolved to `path`

```java
@Route("path")
public class PathComponent extends Div {
    public PathComponent() {
        setText("Hello @Route!");
    }
}

public class Menu extends Div {
    public Menu() {
        String route = UI.getCurrent().getRouter()
                .getUrl(PathComponent.class);
        Anchor link = new Anchor(route, "Path");
        add(link);
    }
}
```

If parent layouts add path parts, it is not always simple to generate the path by hand.

7.5.2. Navigation Target with Parameters

For navigation targets with required parameters, the parameter is given to the resolver and the returning string contains the parameter.

Example: Returning string contains `Router.getUrl(Class target, T parameter)`.

```java
@Route(value = "greet")
public class GreetingComponent extends Div
        implements HasUrlParameter<String> {

    @Override
    public void setParameter(BeforeEvent event,
            String parameter) {
        setText(String.format("Hello, %s!", parameter));
    }
}

public class ParameterMenu extends Div {
    public ParameterMenu() {
        String route = UI.getCurrent().getRouter()
                .getUrl(GreetingComponent.class,
                        "anonymous");
        Anchor link = new Anchor(route, "Greeting");
        add(link);
    }
}
```

7.6. Navigating Between Routes

7.6.1. Using the RouterLink Component

You can use the `RouterLink` component to create links pointing to route targets in your application.

Navigation with `RouterLink` fetches the content of the new component without reloading the page. The page is updated in place.

Example: Using `RouterLink` for navigation targets, with and without URL parameters.

```
void routerLink() {
    Div menu = new Div();
    menu.add(new RouterLink("Home", HomeView.class));
    menu.add(new RouterLink("Greeting",
            GreetingComponent.class, "default"));
}
```

Example: GreetingComponent with URL parameter.

```
@Route(value = "greet")
public class GreetingComponent extends Div
        implements HasUrlParameter<String> {

    @Override
    public void setParameter(BeforeEvent event,
            String parameter) {
        setText(String.format("Hello, %s!", parameter));
    }
}
```

7.6.2. Using Standard href Links

It is also possible to navigate with standard `<a href="company">` type links.

Standard links result in a page reload, but you can enable navigation without page reload by adding the `router-link` attribute, for example `<a router-link href="company">Go to the company page</a>`.

7.6.3. Server-side Navigation

You can trigger navigation from the server side using `UI.navigate(String)`, where the string parameter is the location to navigate to.

You can also use `UI.navigate(Class<? extends Component> navigationTarget)` or `navigate(Class<? extends C> navigationTarget, T parameter)`. This avoids having to generate the route string manually and triggers the browser location update and the addition of a new history state entry.

Example: Navigation to the `company` route target when clicking a button:

```
NativeButton button = new NativeButton(
        "Navigate to company");
button.addClickListener(e ->
     button.getUI().ifPresent(ui ->
          ui.navigate("company"))
);
```

NOTE `Router` links work even if the session has expired. We recommend that you use them instead of handling navigation on the server side.

7.7. Preserving the State on Refresh

When a URL is entered in the browser, Vaadin's routing subsystem resolves it into a view component by inspecting `@Route` class annotations. When a matching class is found, a new instance is created by default. This also happens when the user refreshes the page in the same browser tab.

Occasionally, you may want to keep the state of the view between these refreshes. For example, if the view contains many data entry components, and the user is likely to refresh the page (intentionally or unintentionally) before the data is persisted in the backend. By preserving the view, you ensure the entries are not lost and provide a better UX. Another use case is supporting browser tab-specific "sessions" as an alternative to the standard cookie-based session.

The `@PreserveOnRefresh` annotation instructs Vaadin to re-use the view component of a route, whenever the route is reloaded in the same browser tab. The routed component instance is then the same server-side object that was created in the first request, with all of its state (member fields, subcomponent hierarchy, and so on) preserved.

7.7.1. Preserving the State of a Component

To make a single-view component preserve its content on refresh, simply add the `@PreserveOnRefresh` annotation to the class.

Example: Adding the `@PreserveOnRefresh` annotation to the `PreservedView` class.

```java
@Route("myview")
@PreserveOnRefresh
public class PreservedView extends VerticalLayout {

    public PreservedView() {
        add(new TextField("Content will be preserved"));
        // ...
    }
}
```

If the view component has a router layout (via the `layout` parameter of the `@Route` annotation), the router layout is also preserved on refresh. As an alternative, you can add the `@PreserveOnRefresh` annotation to a class that implements `RouterLayout`.

Example: Adding the `@PreserveOnRefresh` annotation to an implementation of RouterLayout.

```
@PreserveOnRefresh
public class PreservedLayout extends FlexLayout
        implements RouterLayout {

    public PreservedLayout() {
        // ...
    }
}
```

- The PreservedLayout instance itself, as well as any view laid out inside it, is preserved on refresh.

Any elements that are not direct children of the view component, such as notifications and dialogs, are also preserved. This means that if your @PreserveOnRefresh annotated-view class opens a dialog, in which the user makes edits and then refreshes, the dialog remains visible in its edited state.

7.7.2. Preconditions and Limitations

Using the @PreserveOnRefresh annotation has the following conditions/limitations:

- The annotation must be placed in a component class that is a route target (typically annotated with @Route) or on a component that implements RouterLayout.
- The annotation does not support partial preserving. You cannot preserve only some components on the route chain. If the annotation is present on any component in the chain, the entire chain is preserved.
- The component is persisted only when reloaded in the same browser tab (the window.name client-side property is

used to identify the tab), and only if the URL stays the same (visiting another route or changing a URL parameter discards the component state permanently).

- Vaadin 10 and later does not preserve the `UI` instance between refreshes. The view is detached from its previous `UI` and then attached to a fresh `UI` instance on refresh.
- The `AttachEvent` and `DetachEvent` events are also generated when a preserved component is moved to a new `UI`. This means, for instance, that your view component should expect multiple calls to `onAttach` and listeners registered through `addAttachListener` during its lifetime.

7.8. Router Exception Handling

`Router` provides special support for navigation target exceptions. When an unhandled exception is thrown during navigation, the user is shown an error view.

Exception targets generally work in the same way as regular navigation targets, except they typically do not have a specific `@Route` because they are shown for arbitrary URLs.

7.8.1. Error Resolving

Errors in navigation are resolved to a target that is based on the exception type thrown during navigation.

At startup, all classes implementing the `HasErrorParameter<T extends Exception>` interface are collected for use as exception targets during navigation. Example classes include `RouteNotFoundError` for `NotFoundException`.

Example: RouteNotFoundError defines the default target for the NotFoundException that is shown when there is no target for the given URL.

```
@Tag(Tag.DIV)
public class RouteNotFoundError extends Component
       implements HasErrorParameter<NotFoundException> {

    @Override
    public int setErrorParameter(BeforeEnterEvent event,
          ErrorParameter<NotFoundException> parameter) {
        getElement().setText("Could not navigate to '"
                    + event.getLocation().getPath()
                    + "'");
        return HttpServletResponse.SC_NOT_FOUND;
    }
}
```

- This returns a 404 HTTP response and displays the setText to the user.

The exception matching order is first by exception cause and then by exception super type.

The 404 RouteNotFoundError (for NotFoundException), and 500 InternalServerError (for java.lang.Exception) are implemented by default.

7.8.2. Custom Exception Handlers

You can override the default exception handlers by extending them.

Example: Custom route not found handler that uses a custom application layout

```
@ParentLayout(MainLayout.class)
public class CustomNotFoundTarget
        extends RouteNotFoundError {

    @Override
    public int setErrorParameter(BeforeEnterEvent event,
          ErrorParameter<NotFoundException> parameter) {
        getElement().setText(
                "My custom not found class!");
        return HttpServletResponse.SC_NOT_FOUND;
    }
}
```

Note:

- Only extending instances are allowed.
- Exception targets may define `ParentLayouts`. `BeforeNavigationEvent` and `AfterNavigationEvent` are still sent, as in the case of normal navigation.
- One exception may only have one exception handler.

Advanced Exception Handling Example

The following example assumes an application `Dashboard` that collects and shows widgets to users. Only authenticated users are allowed to see protected widgets.

If the collection instantiates a `ProtectedWidget` in error, the widget itself will check authentication on creation and throw an `AccessDeniedException`.

The unhandled exception propagates during navigation and is handled by the `AccessDeniedExceptionHandler` that keeps the `MainLayout` with its menu bar, but displays information that an exception has occurred.

```
@Route(value = "dashboard", layout = MainLayout.class)
@Tag(Tag.DIV)
public class Dashboard extends Component {
    public Dashboard() {
        init();
    }

    private void init() {
        getWidgets().forEach(this::addWidget);
    }

    public void addWidget(Widget widget) {
        // Implementation omitted
    }

    private Stream<Widget> getWidgets() {
        // Implementation omitted, gets faulty state
        // widget
        return Stream.of(new ProtectedWidget());
    }
}

public class ProtectedWidget extends Widget {
    public ProtectedWidget() {
        if (!AccessHandler.getInstance()
                .isAuthenticated()) {
            throw new AccessDeniedException(
                    "Unauthorized widget access");
        }
        // Implementation omitted
    }
}

@Tag(Tag.DIV)
public abstract class Widget extends Component {
    public boolean isProtected() {
        // Implementation omitted
        return true;
    }
}

@Tag(Tag.DIV)
@ParentLayout(MainLayout.class)
public class AccessDeniedExceptionHandler
     extends Component
```

```
      implements HasErrorParameter<AccessDeniedException>
{

    @Override
    public int setErrorParameter(BeforeEnterEvent event,
            ErrorParameter<AccessDeniedException>
                    parameter) {
        getElement().setText(
            "Tried to navigate to a view without "
            + "correct access rights");
        return HttpServletResponse.SC_FORBIDDEN;
    }
}
```

7.8.3. Rerouting to an Error View

It is possible to reroute from the `BeforeEnterEvent` and `BeforeLeaveEvent` to an error view registered for an exception.

You can use one of the `rerouteToError` method overloads. All you need to add is the exception class to target and a custom error message, where necessary.

Example: Reroute to error view

```java
public class AuthenticationHandler
        implements BeforeEnterObserver {
    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        Class<?> target = event.getNavigationTarget();
        if (!currentUserMayEnter(target)) {
            event.rerouteToError(
                    AccessDeniedException.class);
        }
    }

    private boolean currentUserMayEnter(
            Class<?> target) {
        // implementation omitted
        return false;
    }
}
```

If the rerouting method catches an exception, you can use the `rerouteToError(Exception, String)` method to set a custom message.

Example: Blog sample error view with a custom message

```java
@Tag(Tag.DIV)
public class BlogPost extends Component
        implements HasUrlParameter<Long> {

    @Override
    public void setParameter(BeforeEvent event,
            Long parameter) {
        removeAll();

        Optional<BlogRecord> record =
                getRecord(parameter);

        if (!record.isPresent()) {
            event.rerouteToError(
                    IllegalArgumentException.class,
                    getTranslation("blog.post.not.found",
                         event.getLocation().getPath()));
        } else {
            displayRecord(record.get());
        }
    }

    private void removeAll() {
        // NO-OP
    }

    private void displayRecord(BlogRecord record) {
        // NO-OP
    }

    public Optional<BlogRecord> getRecord(Long id) {
        // Implementation omitted
        return Optional.empty();
    }
}

@Tag(Tag.DIV)
public class FaultyBlogPostHandler extends Component
  implements HasErrorParameter<IllegalArgumentException>
{

    @Override
    public int setErrorParameter(BeforeEnterEvent event,
            ErrorParameter<IllegalArgumentException>
                    parameter) {
        Label message = new Label(
                parameter.getCustomMessage());
        getElement().appendChild(message.getElement());

        return HttpServletResponse.SC_NOT_FOUND;
    }
}
```

7.9. Getting Registered Routes

To retrieve all registered `Routes`, use:

```
Router router = UI.getCurrent().getRouter();
List<RouteData> routes = router.getRoutes();
```

- The `RouteData` object contains all the relevant information about the defined route, such as the URL, parameters, and parent layout.

7.9.1. Getting Registered Routes by Parent Layout

To retrieve all the routes defined by parent layout, use:

```
Router router = UI.getCurrent().getRouter();
Map<Class<? extends RouterLayout>, List<RouteData>>
        routesByParent = router.getRoutesByParent();
List<RouteData> myRoutes =
        routesByParent.get(MyParentLayout.class);
```

7.10. Updating the Page Title During Navigation

There are two ways to update the page title during navigation:

1. Use the `@PageTitle` annotation.
2. Implement `HasDynamicTitle`.

These approaches are mutually exclusive: using both in the same class will result in a runtime exception at startup.

7.10.1. Using @PageTitle Annotation

The simplest way to update the Page Title[12] is to use the `@PageTitle` annotation on your `Component` class:

Example: Using `@PageTitle` to update the page title:

```
@PageTitle("home")
class HomeView extends Div {

    public HomeView() {
        setText("This is the home view");
    }
}
```

NOTE The `@PageTitle` annotation is read only from the actual navigation target: super classes and parent views are not considered.

7.10.2. Setting the Page Title Dynamically

As an alternative, you can implement the `HasDynamicTitle` interface. This approach allows you to change the title from Java at runtime:

Example: Implementing `HasDynamicTitle` to update the page title.

```
@Route(value = "blog")
class BlogPost extends Component
        implements HasDynamicTitle,
             HasUrlParameter<Long> {
    private String title = "";

    @Override
    public String getPageTitle() {
        return title;
    }

    @Override
    public void setParameter(BeforeEvent event,
            @OptionalParameter Long parameter) {
        if (parameter != null) {
            title = "Blog Post #" + parameter;
        } else {
            title = "Blog Home";
        }
    }
}
```

7.11. Registering Routes Dynamically

In addition to registering routes using the `@Route` annotation, you can add and remove routes dynamically during runtime. This is useful when a route should be added or removed based on changed business data or application configuration at startup, for example.

The `RouteConfiguration` class allows you to configure routes to a specific scope. You can configure routes to:

- All users using the **application** scope, or
- Only certain active users using the **session** scope.

You can access the scope using the `forSessionScope` and `forApplicationScope` static methods. All components with an `@Route` annotation are added to the application scope.

7.11.1. Configuring User-specific Routes

You can add and remove routes for certain users, for example based on their access rights.

Example: Adding two views for currently active users.

```
RouteConfiguration.forSessionScope().setRoute("admin",
        AdminView.class);

// parent layouts can be given as a vargargs parameter
RouteConfiguration.forSessionScope().setRoute("home",
        HomeView.class, MainLayout.class);
```

A route set for the user can override a route with the same path from the application scope. This means that any statically registered `@Route` can be overridden for a specific user, if necessary.

The routes in the session scope are only accessible for the current user for as long as the session is valid. When the session is invalidated by the user logging out, the session-scoped routes are no longer available automatically. It is not necessary to specifically remove these routes.

When removing routes, you need to define precisely which route to remove.

Examples:

- Removing a navigation target (`AdminView.class`) with all possible route aliases registered to it.

```
RouteConfiguration configuration = RouteConfiguration
        .forSessionScope();
// No view AdminView will be available
configuration.removeRoute(AdminView.class);
```

- Removing a path (`"admin"`) which only removes the target mapped to it.

```
// No path "admin" will be available
configuration.removeRoute("admin");
```

- Removing a single navigation target registered to a path (`"users"` from `UsersView.class`, without removing target with parameter, like `"users/{id}"`).

```
// Remove the "/users" path but keep e.g.
// "/users/123"
configuration.removeRoute("users", UsersView.class);
```

For more related information, see:

- Adding Route Aliases for Dynamic Routes (below).
- Routing and URL Parameters (multiple navigation targets on a single path).

NOTE Removing a route in the session scope that had previously overridden a route in the application scope, makes the application-scoped route accessible once again.

NOTE When dynamically registering a route, any annotations on the classes **are ignored**, except when the method used contains `Annotated`, for example `setAnnotatedRoute`. See Dynamic Registration of @Route Annotated Classes below for more.

7.11.2. Adding Routes on Application Startup

You can register routes during application startup using the `ServiceInitLister`.

Example: Using `ServiceInitLister` to register a route during deployement.

```
public class ApplicationServiceInitListener
        implements VaadinServiceInitListener {

    @Override
    public void serviceInit(ServiceInitEvent event) {
        // add view only during development time
        if (!event.getSource()
                .getDeploymentConfiguration()
                .isProductionMode()) {
            RouteConfiguration configuration =
               RouteConfiguration.forApplicationScope();

            configuration.setRoute("crud",
               DBCrudView.class);
        }
    }
}
```

See VaadinServiceInitListener[13] for more.

7.11.3. Getting Registered Routes and Listening for Changes

When routes are registered dynamically, you may need to update UI components, like navigation menus, based on the added or removed routes.

You can retrieve the registered routes using the `getAvailableRoutes()` method from the registry. To be notified of route changes, you can register a listener using the `addRoutesChangeListener` method.

NOTE You should use the session registry to monitor changes, because it contains all the routes that are available for the current user.

Example: Getting available routes and registering a routes change listener.

```
RouteConfiguration configuration = RouteConfiguration
        .forSessionScope();
// add all currently available views
configuration.getAvailableRoutes()
        .forEach(menu::addMenuItem);

// add and remove menu items when routes are added and
// removed
configuration.addRoutesChangeListener(event -> {
    // ignoring any route alias changes
    event.getAddedRoutes().stream()
            .filter(route -> route instanceof RouteData)
            .forEach(menu::addMenuItem);
    event.getRemovedRoutes().stream()
            .filter(route -> route instanceof RouteData)
            .forEach(menu::removeMenuItem);
});
```

7.11.4. Adding Route Aliases for Dynamic Routes

When adding dynamic routes, the first path for which a navigation target is added is marked as the main path. The main path is returned by the `getUrl` methods in a `RouteConfiguration`. Any additional registered path is seen as a route alias.

Example: Adding multiple routes as navigation targets in a `RouteConfiguration`.

```
RouteConfiguration configuration =
        RouteConfiguration.forSessionScope();
configuration.setRoute("main", MyRoute.class);
configuration.setRoute("info", MyRoute.class);
configuration.setRoute("version", MyRoute.class);
```

In this scenario, the `configuration.getUrl(MyRoute.class)` method returns `main`.

Example: Static class definition equivalent of the above route registration example.

```
@Route("main")
@RouteAlias("info")
@RouteAlias("version")
private class MyRoute extends Div {
}
```

If the `"main"` path is removed and an alias path remains available for use, the main path is updated to the first alias path found in the registry.

WARNING

Be cautious when adding or removing routes from the `ApplicationRouteRegistry`, because this impacts every user of the system.

7.11.5. Dynamic Registration of @Route Annotated Classes

If you want to map all routes in the same way using the `@Route` annotation, you can configure the routes statically, but postpone registration until runtime.

To skip static registration on servlet initialization, add the `registerAtStartup = false` parameter to the `@Route` annotation. This prevents the route being registered on startup to the application-scoped registry. It also makes it easier to use existing parent chains and paths that are modified from the parent.

Example: Using the `registerAtStartup` parameter to postpone route registration.

```
@Route(value = "quarterly-report",
       layout = MainLayout.class,
       registerAtStartup = false)
@RouteAlias(value = "qr", layout = MainLayout.class)
public class ReportView extends VerticalLayout
        implements HasUrlParameter<String> {
    // implementation omitted
}

// register the above view during runtime
if (getCurrentUser().hasAccessToReporting()) {
    RouteConfiguration.forSessionScope()
            .setAnnotatedRoute(ReportView.class);
}
```

7.11.6. Example: Adding a New View on User Login

This example demonstrates how to add a new view on user login. There are two types of users: admin users and normal users. After login, we show a different view, depending on the user's access rights.

The demo application contains:

- The `LoginPage` class that defines a statically registered route, `""`. This route is mapped to the login used for user authentication.

```
@Route("")
public class LoginPage extends Div {

    private TextField login;
    private PasswordField password;

    public LoginPage() {
        login = new TextField("Login");
        password = new PasswordField("Password");

        Button submit = new Button("Submit",
                this::handleLogin);

        add(login, password, submit);
    }

    private void handleLogin(
            ClickEvent<Button> buttonClickEvent) {
    }
}
```

- The `MainLayout` class that contains a menu.

```
public class MainLayout extends Div
        implements RouterLayout {
    public MainLayout() {
        // Implementation omitted, but could contain
        // a menu.
    }
}
```

- The `InfoView` class that defines the `"info"` route. This route is not statically registered, because it has the `registerAtStartup = false` parameter.

```java
@Route(value = "info", layout = MainLayout.class,
       registerAtStartup = false)
public class InfoView extends Div {
    public InfoView() {
        add(new Span("This page contains info about "
                + "the application"));
    }
}
```

After login, we want to add a new route depending on the access rights of the user. There are two available targets:

- `AdminView` class.

  ```java
  public class AdminView extends Div {
  }
  ```

- `UserView` class.

  ```java
  public class UserView extends Div {
  }
  ```

In the `LoginPage` class, we handle adding to only the user session as follows:

```
private void handleLogin(
        ClickEvent<Button> buttonClickEvent) {
    // Validation of credentials is skipped

    RouteConfiguration configuration =
            RouteConfiguration.forSessionScope();

    if ("admin".equals(login.getValue())) {
        configuration.setRoute("", AdminView.class,
                MainLayout.class);
    } else if ("user".equals(login.getValue())) {
        configuration.setRoute("", UserView.class,
                MainLayout.class);
    }

    configuration.setAnnotatedRoute(InfoView.class);

    UI.getCurrent().getPage().reload();
}
```

- A new target for the path `""` is added to the session-scoped route registry. The new target overrides the application-scoped path `""` for the user.
- The `InfoView` class is added using the `layout` setup, configured using the `@Route` annotation. It is registered to the path `"info"` with the same `MainLayout` as the parent layout.

NOTE | Other users on other sessions still get Login for the `""` path and cannot access `"info"`.

----

[12] https://developer.mozilla.org/en-US/docs/Web/API/Document/title
[13] https://vaadin.com/docs/flow/advanced/tutorial-service-init-listener.html

8. Browser Features and Events

8.1. Browser Window Resize Events

The Page class allows you to register a listener for events that affect the web page and the browser window in which the Vaadin UI resides. The Page instance corresponding to a given UI is accessed by the getPage() method of the UI.

Example: Accessing the browser window size and adding a size-change listener.

```
Page page = someUI.getPage();
page.addBrowserWindowResizeListener(
        event -> Notification.show("Window width="
                + event.getWidth()
                + ", height=" + event.getHeight()));
```

8.2. Executing JavaScript in the Browser

You can use server-side Java to execute simple JavaScript snippets in the browser. You can also pass parameters to the executed script as variables named $0, $1 and so on. Vaadin automatically serializes and escapes the parameter values.

Example: Executing JavaScript in the browser and passing parameters.

```java
public static void logElementSize(String name,
        Element element) {
    Page page = UI.getCurrent().getPage();

    page.executeJs(
            "console.log($0 + ' size:', "
            + "$1.offsetWidth, $1.offsetHeight)",
            name, element);
}
```

The supported parameter types are `String`, `Boolean`, `Integer`, `Double`, `JsonValue` and `Element`.

The script is executed after the DOM tree has been updated based on server-side changes. The parameter value is `null` for a parameter of type Element that is not attached after the update (according to the server-side component structure)

NOTE The script is executed asynchronously, so you cannot directly pass values back to the server. Instead, the returned `PendingJavaScriptResult` instance can be used to add a callback that is run when the result is available.

Vaadin provides a ready-made solution (without custom JavaScript execution) to listen to browser window-resize events for the from the server side. See Browser Window Resize Events for more.

9. Embedding Vaadin Applications

9.1. Introduction to Embedding Applications

Embeddeding applications is an alternative to writing monolithic frontends for your applications.

Embedded applications are also know as micro frontends[14]. They are isolated and self-contained pieces of code that can be maintained by different teams, using different frameworks. A simple example is an embedded calendar in a web page: the calendar functionality is isolated and has no relation to the logic of the main application.

Embedding an application is similar to adding a client-side widget to a page, except the embedded application has back-end logic and is a real application in its own right.

9.1.1. Embedding Applications in Vaadin

Overview

Vaadin allows you to embed applications using web components and provides the `WebComponentExporter` class for this purpose.

These are the basic steps to creating an embedded application:

- Create the application that will be embedded and export it:

    - Write and declare a server-side component in a specific way, using a custom element tag name.
    - Create an exporter for the component by extending the `WebComponentExporter` class.
- Embed and use the application in your host (embedding) page.
    - Make your page aware of the application by adding an element with a matching custom tag.

The embedded application behaves like a standard Vaadin component, regardless of any other content on the page, except that certain features are not available. See Embedded Application Limitations for more.

Creating an Embedded Application

To create an embedded application, you need to export a component as an embeddable application:

- Create the component (`MyComponent`) that will be used as the embedded application. You can create a new component or use an existing one. This component (application) has no relation to the host application.
- Create an exporter for the component, by extending the `WebComponentExporter<MyComponent>` class.
    - Implement its constructor providing the tag name that you will use to identify it on the host application.
    - Configure properties in the constructor using the `addProperty` method.
    - Implement the `configureInstance` method, if you need additional initialization of the exported component (for example, add a listener to the original

component).

- Deploy your embeddable application.

See Creating an Embedded Vaadin Application Tutorial for a detailed example.

Importing an Embeddable Application

To embed the exported application in a page:

- Add `webcomponent-loader.js` polyfill script to your page, for example `<script type="text/javascript"src="YOUR_EMBEDDED_APPLICATION_URI/VAADIN/build/webcomponentsjs/webcomponents-loader.js"></script>`
  - `YOUR_EMBEDDED_APPLICATION_URI` is the URI at which you deploy your exported application. This depends on how and where you deploy the application.
  - While the example above uses the polyfill provided with Vaadin, you can use any CDN (such as unpkg.com).
- Import the web component URL resource of the embedded application, for example `<script type='module' src='YOUR_EMBEDDED_APPLICATION_URI/web-component/my-component.js'></script>`.
  - The application is imported using the path `"web-component/my-component.js"`, where `"web-component"` is the base path for embeddable applications, and `"my-component.js"` is the *`custom-tag-name`*`.js`.
  - As before, `YOUR_EMBEDDED_APPLICATION_URI` is the URI at which you deploy your exported application.

- Use the embedded web component in your HTML code using the tag name you assigned to the embedded application, for example `<my-component></my-component>``.
  - The tag name, `"my-component"`, is used to identify the embedded application.
  - The element `my-component` is used in your HTML page content. This can be a static HTML file or content generated by any framework, for example a plain servlet, JSP, and more.

For more on embedded applications, see:

- Embedded Application Properties
- Theming Embedded Applications
- Configuring Push Notifications in Embedded Applications
- Securing Embedded Applications
- Creating an Embedded Application Tutorial
- Importing Embedded Applications in Compatibility and Production Mode
- Embedded Application Limitations

9.2. Embedded Application Properties

In this section we cover:

- How to define web component properties for embedded Vaadin applications.
- How to handle property changes on the server side.

  How to fire custom events on the client side.

See Creating an Embedded Application Tutorial for a detailed example of how to create an embedded Vaadin application.

9.2.1. Defining Web Component Properties

The `WebComponentExporter` class defines the properties that are exposed by the `WebComponent` public API.

Calling `WebComponentDefinition#addProperty` defines a property and adds it to the public API of the web component. The supported types are `Integer`, `Double`, `Boolean`, `String`, and `JsonValue`.

Example: Using the `addProperty` method to define web component properties.

```
public class PersonExporter
        extends WebComponentExporter<PersonComponent> {
    private PropertyConfiguration<PersonComponent,
              Boolean> isAdultProperty;

    public PersonExporter() {
        super("person-display");
        addProperty("name", "John Doe")
                .onChange(PersonComponent::setName);
        addProperty("age", 0)
                .onChange(PersonComponent::setAge);

        isAdultProperty = addProperty("is-adult",
                false);
    }
```

- This example defines three properties: `name`, `age`, and `is-adult`. (The `is-adult` property is used in an example below in Updating Properties on the Client Side)

  The `name` property type is a `String` and the `age` property type is an `Integer`.

-
- The default values serve a dual purpose: they define the property type and set the default value. If no default value is provided, you need to define the type explicitly by calling `definition.addProperty(String, Class<?extends Serializable>)`.

Property Event Attributes

Adding a property exposes a fluent API that you can use to configure the property.

Properties have two event attributes:

- `.onChange(…)`:
    - Registers a callback that is called when the value of the property changes on the client side.
    - Accepts the parameter, `SerializableBiConsumer<C, P>`, where `C` is the type of the `Component` being exported and `P` is the type of the property. The component's associated setter method is a conventional choice.
- `.readOnly()`.
    - Sets the property to read-only mode: the value of the property cannot be changed on the client side.

`addProperty` Method Return Type

The `addProperty` method returns a `PropertyConfiguration<C, P>` object that provides the fluent API for configuring the property.

If you need to refer to the property later, you can use the received `PropertyDefinition` to identify the property in question.

9.2.2. Updating Properties on the Client Side

In this section we cover how the host environment (sever) communicates with the client, and explain how to update client-side property values from the server.

To update client-side property values, you need:

- A reference to the web component that contains the exported component, and
- A reference to the instance of the exported component itself.

You can implement the abstract `configureInstance` method to update properties and fire client-side events.

The `configureInstance` method receives references to `WebComponent<PersonComponent>` and `PersonComponent`, where `PersonComponent` is the exported component. `WebComponent` is used to communicate with the client-side.

Example: Updating the `is-adult` boolean property when the `age` property changes in the `PersonComponent` instance.

```
    @Override
    protected void configureInstance(
            WebComponent<PersonComponent> webComponent,
            PersonComponent component) {
        component.setAdultAge(18); // initialization

        component.addAgeChangedListener(event -> {
            webComponent.setProperty(isAdultProperty,
                    component.isAdult());
        });
```

- The `WebComponent#setProperty()` method updates the property identified by the `PropertyConfiguration` to the new value.

NOTE The `configureInstance` method can also be used to do further initialization on the `component` instance.

Now that the `is-adult` property is configured to update on the client side, the next step is to access and leverage this property.

Example: Embedding the `person-display` component in a web page and updating `<span id='designator'>`.

```
<person-display id="person" age=15></person-display>
<span id="designator">is a child</span>

<script>
    function updateDesignator() {
        var personComponent = document
                .querySelector("#person");
        if (personComponent["is-adult"]) {
            document.querySelector("#designator")
                    .innerText = "is an adult!";
        } else {
            setTimeout(updateDesignator, 1000);
        }
    }

    updateDesignator();
</script>
```

- The script checks periodically whether or not the person has reached adulthood, and updates `<span id="designator">` when this occurs.

9.2.3. Firing Custom Events on the Client Side

A `WebComponent` instance can also be used to fire custom events on the client side.

You can use the `webComponent#fireEvent()` method to fire events for given parameters.

Example: Using the `webComponent#fireEvent()` method to fire the `"retirement-age-reached"` event.

```
        component.addAgeChangedListener(event -> {
            if (event.getAge() > 65) {
                webComponent.fireEvent(
                        "retirement-age-reached");
            }
        });
    }
}
```

- This example uses custom logic and a custom event: if a person's age reaches 66 or more, an event of type `"retirement-age-reached"` is fired on the client-side.

The `fireEvent()` method has three variants:

- `fireEvent(String)`.
- `fireEvent(String, JsonValue)`.
- `fireEvent(String, JsonValue, EventOptions)`.

The parameters are:

- `String`: The name or `type` of the event.
- `JsonValue`: A custom JSON object set as the value of the `detail` key in the client-side event.
- `EventOptions`: To configure the `bubbles`, `cancelable`, and `composed` event options.

See CustomEvent[15] in the MDN documentation for more information about these parameters.

The final step is to update the `<span>` tag with the event results.

Example: updating `<span id="designator">` with the `"retirement-age-reached"` event result.

```
<person-display id="person" age=15></person-display>
<span id="designator">is a child</span>

<script>
    var personComponent = document
            .querySelector("#person");

    personComponent.addEventListener(
            "retirement-age-reached", function(event) {
        document.querySelector("#designator")
                .innerText = "is allowed to retire!";
    });
</script>
```

9.3. Theming Embedded Applications

Theming of embedded applications works in exactly the same way as for any other Vaadin component.

By default, embedded Vaadin applications use the `Lumo` theme (if it is found in the classpath), or no theme at all.

See Theming Overview for more.

9.3.1. Assigning a Theme

You can specify a theme, for example the `Material` theme, for your embedded Vaadin application using the `@Theme` annotation.

Example: Using the `@Theme` annotation to apply the `Material` theme to the `MyExporter` embedded application.

```
@Theme(Material.class)
public class MyExporter
        extends WebComponentExporter<Div> {
```

The annotation `@Theme(Material.class)` applies the `Material` theme to the embedded application and it works just like in regular Vaadin applications.

It’s enough to specify a theme in only one of the exporters of your application. If there is no theme declaration then the `Lumo` theme is used when available in the classpath as mentioned above.

9.3.2. Using Multiple Themes

It is not possible to use more than one theme in a single embedded application. Themes are detected during build time (or startup in compatibility mode’s development mode) and an exception is thrown if different themes are found.

If you need to use multiple themes, create multiple embedded applications (one per theme) and split the functionality accordingly. Each embedded application uses its own theme and the main application embeds several themed applications instead of one.

9.4. Securing Embedded Applications

You can prevent the host (embedding) application accessing the embedded application using the properties of the embedded application.

WARNING

Keep in mind that the embedded application is instantiated, regardless of of whether access is restricted or not. The reason is that property values are only checked when the server-side listener detects an update on the client side. Always avoid including sensitive data in the embedded web component’s constructor.

Example: Setting a property from the host (embedding) page and checking it inside an embedded application.

```
<!doctype html>

<head>
  <link rel="import" href="web-component/my-comp.js">
  <script type="text/javascript">
        function login(){
            // request token for the current user
            // somehow
            var token =
                "d9f6a737-b2b8-46a7-a834-209c8b214969";
            var comp = document.querySelector(
                "#embedded-web-component");
            comp.token = token;
        }
  </script>
</head>

<body>
  <p>
    Web components implemented using server side Java
  </p>

  <button onclick="login()">Login</button>

  <my-comp id="embedded-web-component"></my-comp>

</body>
```

- The `my-comp` element is embedded in a static page.

- The `token` property is set from a JavaScript function that retrieves it within the `login` function (that is invoked on the login button click).

Example: Associated web component and its exporter class.

```java
public class EmbeddedComponentExporter
       extends WebComponentExporter<EmbeddedComponent> {

    public EmbeddedComponentExporter() {
        super("my-comp");

        addProperty("token", "")
                .onChange(this::authorize);
    }

    @Override
    protected void configureInstance(
           WebComponent<EmbeddedComponent> webComponent,
           EmbeddedComponent component) {
    }

    private void authorize(EmbeddedComponent component,
                           String token) {
        // check the token
        if (isValidToken(token)) {
            component.init();
        }
    }

    private boolean isValidToken(String token) {
        return true;
    }

}

public class EmbeddedComponent extends Div {

    public EmbeddedComponent() {
        // Don't retrieve any sensitive data here
        // without granted access (via security token)
    }

    public void init() {
        // Initialize your secured component here
    }
}
```

- The embedded web component is instantiated before the exporter instance receives the token value. For this reason, avoid retrieving or initializing components with sensitive date in the constructor. Do your initialization only after the valid token value is received.

NOTE If you cannot prevent the initialization of the web component in the constructor, you can wrap it in a container, like `Div`. Create a `Div` subclass (instead of using `EmbeddedComponent`), and add an `EmbeddedComponent` instance into the `Div` subclass when the token is validated.

NOTE If you use a Dependency Injection (DI) framework, keep in mind that `EmbeddedComponentExporter` is instantiated directly without DI, and you may not use this instance to inject anything from DI context. As a workaround, you can create a a wrapper component that is instantiated within the DI context, and then use the wrapper instance to access the DI context.

9.5. Creating an Embedded Vaadin Application Tutorial

In this section, we demonstrate how to create a basic embedded application: that is, an application that is capable of being embedded in another page.

NOTE The technology you use to create the host page is irrelevant: you can use Vaadin or non-Vaadin server-side Java technologies, like JSP, Thymeleaf, or servlet, or even a basic static HTML page.

From a overview perspective, to create an embedded application you need to create:

- A separate Vaadin application. This is achieved by declaring the web component exporter as a separate web component, and
- A `VaadinServlet` to handle requests to your web component. The servlet can be declared in a non-Vaadin application, or deployed separately as a standalone WAR file.

Our scenario uses:

- A single custom servlet to handle the main application logic.
  - The servlet displays primarily static content.
  - The content differs depending on whether a user is logged in or not.
- A Vaadin server-side web component to implement a login form.

NOTE `VaadinServlet` forms part of the application, but its mapping differs from the main servlet mapping.

1. Create the `MainAppServlet` servlet class.

```java
@WebServlet(urlPatterns = { "/*" })
public class MainAppServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req,
            HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType(
                "text/html;charset=UTF-8");

        Object authToken = req.getSession()
                .getAttribute("auth_token");
        boolean isAuthenticated = authToken != null;
        try (PrintWriter out = response.getWriter()) {
            out.println("<!DOCTYPE html>");
            out.println("<html><head>");
            out.println(
                      "<meta http-equiv='Content-Type'
"
                    + "content='text/html; "
                    + "charset=UTF-8'>");
            out.println(
                      "<script type='text/javascript'
"
                    + "src='log-in.js'></script>");
            if (!isAuthenticated) {
                out.println(
                          "<script
type='text/javascript' "
                        +
"src='/vaadin/VAADIN/build/webcomponentsjs/"
                        + "webcomponents-
loader.js'></script>");
                out.println(
                          "<script type='module' src="
                        + "'/vaadin/web-component/"
                        + "login-form.js'></script>");
            }
            out.println("<body>");
            if (isAuthenticated) {
                out.println("<h1>Welcome "
                        + UserService.getInstance()
                            .getName(authToken)
                        + "</h1>");
            } else {
                out.println(
                        "<login-form
userlbl='Username' "
                      + "pwdlbl='Password'>"
                      + "</login-form>");
            }
            out.println("</body>");
            out.println("</html>");
        }
    }
}
```

2. Create the `UserService` class. This class contains the authentication logic and is shared between the `MainAppServlet` and the web component class (created in 6. below). You can use any interface and implementation you like. This example is a custom stub implementation and is provided as a reference.

   ```
   public final class UserService {

       private static final UserService INSTANCE =
               new UserService();

       private UserService(){
       }

       public static UserService getInstance() {
           return INSTANCE;
       }

       public String getName(Object authToken) {
           return "Joe";
       }

       public Optional<Object> authenticate(String user,
               String passwd) {
           if ("admin".equals(user) &&
                   "admin".equals(passwd)) {
               return Optional.of(new Object());
           } else {
               return Optional.empty();
           }
       }
   }
   ```

3. Analyze the HTML content generated by `MainAppServlet`. It is fairly simple:

   - This line loads the web component polyfill:

```
<script type='text/javascript'
        src=
'/vaadin/VAADIN/build/webcomponentsjs/webcomponents
-loader.js'>
</script>
```

- This line loads the web component:

```
<script type='module'
        src='/vaadin/web-component/login-form.js'>
</script>
```

  - Both script `src` attributes start with `/vaadin/`. This is the URI to use to map the Vaadin servlet (see 4. below).
  - The second part the link URI,`/web-component/login-form.js`, is the standard URI to use to import the web component. It consists of the hard-coded `web-component` part, followed by `login-form.js`, which is the web component file. The web component file is generated by Vaadin, based on the configuration set in the exporter.

- The name of the web component in our example must be `"login-form"`. This name must be used in both the `super` constructor of the exporter (see `LoginFormExporter` in 6. below) and the HTML code where the web component is inserted. In our example this is right under the `<body>` tag:

```
<login-form userlbl='Username' pwdlbl='Password'>
</login-form>
```

  - The `"login-form"` web component has two properties, `userlbl` and `pwdlbl`. These values are passed from the HTML to a web component instance.

NOTE If embedding applications is targeted towards very specific browsers, the polyfill is not needed. For example Chrome and Firefox do not need the polyfill while Edge does.

1. Register the `VaadinServlet`.

```
@WebServlet(urlPatterns = {"/vaadin/*", "/frontend/*"
})
public class WebComponentVaadinServlet
        extends VaadinServlet {
}
```

- As mentioned above, the `/vaadin/\*` mapping allows the `VaadinServlet` to handle web component requests. You can use any URI, but be sure to use the same URI in the mapping and in the import declaration.
- Our example also uses `/frontend/*` mapping for the servlet because we need to handle WebJar resource URIs: we use various Vaadin components in the server-side web component and this requires a `frontend` URI handler.

NOTE If you deploy your web component exporter(s) as a standalone WAR application, an explicit servlet registration is unnecessary. A servlet instance is registered automatically with the `"/*"` mapping.

2. Create the `LoginForm` component class.

```
public class LoginForm extends Div {
    private TextField userName = new TextField();
    private PasswordField password =
            new PasswordField();
    private Div errorMsg = new Div();
    private String userLabel;
    private String pwdLabel;
    private FormLayout layout = new FormLayout();
    private List<SerializableRunnable> loginListeners
=
            new CopyOnWriteArrayList<>();

    public LoginForm() {
        updateForm();

        add(layout);

       Button login = new Button("Login",
                event -> login());
        add(login, errorMsg);
    }

     public void setUserNameLabel(
            String userNameLabelString) {
        userLabel = userNameLabelString;
        updateForm();
    }

    public void setPasswordLabel(String pwd) {
        pwdLabel = pwd;
        updateForm();
    }

    public void updateForm() {
        layout.removeAll();

        layout.addFormItem(userName, userLabel);
        layout.addFormItem(password, pwdLabel);
    }

    public void addLoginListener(
            SerializableRunnable loginListener) {
        loginListeners.add(loginListener);
    }

    private void login() {
        Optional<Object> authToken = UserService
                .getInstance()
                .authenticate(userName.getValue(),
                    password.getValue());
        if (authToken.isPresent()) {
            VaadinRequest.getCurrent()
                    .getWrappedSession()
                    .setAttribute("auth_token",
                            authToken.get());
            fireLoginEvent();
        } else {
            errorMsg.setText("Authentication failure"
);
        }
    }

    private void fireLoginEvent() {
        loginListeners.forEach(
                SerializableRunnable::run);
    }
}
```

- The example uses several Vaadin components: `FormLayout`, `TextField`, `PasswordField` and `Button`.
- The code takes care of authentication and sets an authentication token in the `HttpSession`, which makes it available while the session is live.
- Because the main application servlet uses the same `HttpSession` instance, it changes behavior and redirects authenticated users to the main servlet that now shows content specific to authenticated users. There are various ways to do this:
  - Execute JavaScript directly from your Java code and set the location to `"/"`: `getUI().get().getPage().executeJs("window.location.href='/'");`.
  - Use a solution similar to this example: design the component code so that its logic is isolated and it does not need to know anything about the embedding context. This method allows you to

completely decouple the embedded component logic from the application that uses it. In this example, the `addLoginListener` method allows you to register a listener which is called in the `fireLoginEvent` method.

3. The final step is to export the `LoginForm` component as an embeddable web component using the web component exporter.

```
public class LoginFormExporter
        extends WebComponentExporter<LoginForm> {
    public LoginFormExporter() {
        super("login-form");
        addProperty("userlbl", "")
                .onChange(LoginForm::setUserNameLabel
);
        addProperty("pwdlbl", "")
                .onChange(LoginForm::setPasswordLabel
);
    }

    @Override
    protected void configureInstance(
            WebComponent<LoginForm> webComponent,
            LoginForm form) {
        form.addLoginListener(() ->
                webComponent.fireEvent("logged-in"));
    }
}
```

- The exporter defines its tag name as `"login-form"` by calling the super constructor `super("login-form");`.
- The `addProperty` method defines the component properties (`userlbl='Username' and `pwdlbl='Password'`) to receive values from the HTML element to the web component instance. In this example we declare the labels for user name field and password field via HTML, instead of hard-coding them

in the LoginForm component class.

- LoginFormExporter class implements the abstract method, configureInstance, which registers a login listener.
- The login listener fires a client-side "logged-in" event, using the webcomponent.fireEvent() method. The main application needs to handle this event somehow.
- The custom event is handled by the JavaScript file declared via the line <script type='text/javascript' src='log-in.js'></script> in MainAppServlet. This is the log-in.js file content:

```
var editor = document.querySelector("login-form");
editor.addEventListener("logged-in",
function(event) {
    window.location.href='/';
});
```

- The embedding servlet uses the API provided by LoginForm via a custom event and adds an event listener for the event. The listener simply redirects the page to the "/" location.

9.6. Embedding Applications in Compatibility and Production Mode

9.6.1. Compatibility Mode

When embedding applications in compatibility mode, the following differences to development mode need to be taken into account:

- The `"webcomponents-loader.js"` polyfill library is required to import web components using HTML import.
- The web component URL resource is imported via HTML `import`.

  Example: Importing a web component in compatibility mode.

  ```
  <script type='text/javascript' src=
  '/frontend/bower_components/webcomponentsjs/webcompone
  nts-loader.js'></script>
  <link rel='import' href='/vaadin/web-component/login-
  form.html'>
  ```

The rest of the embedding process is the same.

9.6.2. Production Mode

Embedding applications in production mode is similar to development mode in that it requires these two steps:

1. Package the application for production in the normal way.
2. Import the packaged application onto the target page.

However, in production mode the second step differs slightly from in development mode. The reason is that the `"webcomponents-loader.js"` polyfill library is located in a different folder, which depends on the user's browser. This library provides support for browsers that do not have native support for `rel="import` on a `link` element.

To avoid using the boilerplate line in production mode, you should use a `script` tag, instead of a `link` element for the web component.

Example Using the `<script>` tag for a web component in production mode.

```
<script type='text/javascript'
        src='/vaadin/web-component/login-form.html'>
</script>
```

In production mode, the generated `login-form.html` content is simply JavaScript code that adds a proper polyfill library together with the required imports.

9.7. Configuring Push in Embedded Applications

You can configure and enable Push in your embedded applications.

There are two ways to configure Push:

- Use the `@Push` annotation[16] in your `WebComponentExporter` class.

  Example: Using the `@Push` annotation in the `PushComponentExporter` class.

  ```
  @Push
  public class PushComponentExporter
          extends WebComponentExporter<Div> {
  ```

- Declare Push on the servlet level, by defining them in the servlet configuration[17].

NOTE The `@Push` annotation declaration has the same limitation as the @Theme` annotation: it is only possible to configure Push for one exporter. Declaring different `@Push` annotations for different exporter classes will result in an exception during startup.

See Server Push Configuration[18] for more about configuring Push.

9.8. Embedded Application Limitations

Some Vaadin features are not available in embedded applications.

Limitations in embedded applications include:

- **Navigation and routing**: Both features are not available for embedded applications.
  - There is no point annotating your classes with the`@Route` annotation, because it is not possible to navigate to the route target.
  - You can also not use the router link, whether via the `RouterLink` class or in a custom way.
- **Theming**: You can only specify one `@Theme` annotation. See Theming Embedded Applications for more.
- **Push**: You can only use one `@Push` annotation. See Configuring Push Notifications in Embedded Applications for more.
- **CORS headers**: Cross-Origin Resource Sharing (CORS) headers[19] are not defined automatically. If the Vaadin servlet providing the embeddable application is outside of the servlet container that provides the page in which it is

embedded, these headers need to be provided.

The responses from the Vaadin servlet should contain appropriate CORS headers. You can add these by:

- Configuring the servlet container (see the documentation on adding HTTP headers for responses for your specific container), or
- Packaging the embeddable application with a custom `VaadinServlet`.

  Example: Custom `VaadinServlet` that adds CORS headers

```
@WebServlet(urlPatterns = { "/*" }, asyncSupported
= true)
public class CustomServlet extends VaadinServlet {

    @Override
    public void service(ServletRequest req,
ServletResponse res) throws ServletException,
IOException {
        setAccessControlHeaders(
(HttpServletResponse) res);
        super.service(req, res);
    }

    private void setAccessControlHeaders
(HttpServletResponse resp) {
        resp.setHeader("Access-Control-Allow-
Origin", "http://localhost:80");
        resp.setHeader("Access-Control-Allow-
Methods", "*");
        resp.setHeader("Access-Control-Allow-
Headers", "Content-Type");
        resp.setHeader("Access-Control-Allow-
Credentials", "true");
    }
}
```

- This example assumes that the embedding (host) site is served from the same host mapped to port 80 (be it a servlet container or a custom Python HTTP server). Our servlet container with our Vaadin servlet is bound to, for example, 8080.

----

[14] https://micro-frontends.org/
[15] https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent
[16] https://vaadin.com/docs/flow/advanced/tutorial-push-configuration.html#push.configuration.annotation
[17] https://vaadin.com/docs/flow/advanced/tutorial-push-configuration.html#push.configuration.servlet
[18] https://vaadin.com/docs/flow/advanced/tutorial-push-configuration.html
[19] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

10. Theming and styling applications

10.1. Application Theming Basics

Theming ensures that an application has a consistent and professional look and feel throughout.

The content in this section focuses on theming a single application, that is, specifying styles that apply to one application.

NOTE If you want to share styles across multiple applications, the best practice is to create a dedicated theming module that can be shared as a dependency by multiple applications.

10.1.1. Application Theming Approaches

Most developers use a prebuilt Vaadin theme, such as Lumo[20] or Material[21], as the starting point to customize styling in their application.

NOTE The Material theme is fully compliant with Google's Material Design guidelines[22].

When starting with a prebuilt theme, you can follow either of the following approaches:

1. Use the default Lumo theme and only add styling to customize the shadow DOM of specific components.
   - Specifically, this means you:

     Do not use the `@Theme` annotation. The Lumo theme

    - is used by default (as long as it exists in your classpath). For example, when a `vaadin-button` component is rendered, the corresponding Lumo file is loaded.
    - Import only component-specific theme modules to style the shadow DOM of the components you want to customize.
  - **Advantages**: This is the fastest and simplest approach.

NOTE	Lumo only loads the styles/themes required for the components that you use.

2. Use a custom theme class.
  - Specifically, this means you:
    - Use the `@Theme` annotation to specify your custom theme class, for example `MyTheme.class`.
    - In your custom theme class, you define imports for global files and the path from which to load the component styles. The imports are replacements for Lumo implementations for Vaadin components.
  - **Advantages**:
    - You can select and customize only the Lumo styles that need to load for each component. This reduces the number of styles loaded and gives you more control.
    - You can achieve better encapsulation of the custom component stying.
  - **Disadvantages**: It is necessary to create a file for every Vaadin component that you want the custom theme class to support.

10.1.2. Theming File Types and Storage

You can theme your application with CSS. However, to theme Polymer Web Components, you need to instruct Vaadin to import the styles using the @CssImport annotation.

NOTE In addition to the `@CssImport` annotation, you can use the classic `@StyleSheet` annotation for files that do not need to be imported as ES6 modules. These files are loaded dynamically using the `<style>` tag (instead of being added to the application bundle). However, styles loaded in this way cannot target the internals of Polymer Web Components.

We recommend that you name your application-wide styles, `shared-styles.css`, and store them in the `/frontend/styles` folder (full path: `/frontend/styles/shared-styles.css`).

Example: Using the @CssImport` annotation to import `shared-styles.html`.

```
@CssImport("./styles/shared-styles.css")
@Viewport("width=device-width, minimum-scale=1.0,
initial-scale=1.0, user-scalable=yes")
public class MainLayout extends Div
         implements RouterLayout {
}
```

`shared-styles.css` should typically contain all global and view-specific theming for your application. See Theming Web Components for more about the expected contents.

For custom components, you can add component-specific styling to the `template` block in the component's `.js` file. See Theming Crash Course for an example of this.

10.2. Integrating a Custom Component Theme

You can integrate a custom component theme to be used with the wrapped Vaadin components. To do this, you need to create a theme class that tells Vaadin how to translate the base un-themed component HTML import into your themed version.

The most important methods are:

- `getBaseUrl`: This should return the part of the import that is used to determine if it is an import that can be changed into a theme import. For Vaadin components this is `/src/`.
- `getThemeUrl`: This should return what the base URL part should be changed into to get the correct theme import. For Vaadin components this is `/theme/[themeName]`.

Example: Overriding the `getBaseUrl` and `getThemeUrl` methods in the `MyTheme` class.

```java
@JsModule("@vaadin/vaadin-lumo-styles/color.js")
public class MyTheme implements AbstractTheme {
    @Override
    public String getBaseUrl() {
        return "/src/";
    }

    @Override
    public String getThemeUrl() {
        return "/theme/myTheme/";
    }
}
```

If you need more control, you can use the `getHeaderInlineContents()` method that returns a collection of HTML that will be inlined to the `BootstrapPage` body.

Example: Using the `getHeaderInlineContents()` method to add a custom style that includes the correct typography styles.

```
@Override
public List<String> getHeaderInlineContents() {
    return Collections.singletonList("<custom-style>\n"
      + "<style include=\"lumo-color lumo-typography\">"
      + "</style>\n"
      + "</custom-style>");
}
```

Your can also support theme variants. The Lumo theme, for example, supports both light and dark variants.

To add support for variants:

1. Override the `getHtmlAttributes` method in your custom theme class.

   Example: Overriding the `getHtmlAttributes` method.

   ```
   @Override
   public Map<String, String> getHtmlAttributes(
           String variant) {
       if ("dark".equals(variant)) {
           // the <body> element will have the "theme"
           // attribute set to "dark" when the dark
   variant
           // is used
           return Collections.singletonMap("theme", "dark
   ");
       }
       return Collections.emptyMap();
   }
   ```

2. Create themed `.js` files for all the Vaadin elements in the `/frontend/theme/myTheme/` folder.

   Example: Modifying the `vaadin-button` component by adding custom rules to the original Lumo rules in `frontend/theme/myTheme/vaadin-button.js`.

   ```
   // import the non-themed component
   import '@vaadin/vaadin-button/src/vaadin-button.js';
   // optional: reuse lumo styles for button
   import '@vaadin/vaadin-button/theme/lumo/vaadin-
   button-styles.js';

   import { html } from
           '@polymer/polymer/lib/utils/html-tag.js';

   // set your custom CSS rules for button.
   // Use an unique id for the dom-module.
   const $_documentContainer = html`
     <dom-module id="my-lumo-button"
                 theme-for="vaadin-button">
       <template>
         <style>
           :host {
             border-radius: 0;
           }
         </style>
       </template>
     </dom-module>`;

   document.head.appendChild($_documentContainer.content)
   ;
   ```

NOTE

You need to provide theme files for all components used in your application. Omitted component display without any styles. This is because all Vaadin components are imported using the `/src/` path, and it is the responsibility of the theme class to replace the `getBaseUrl()` pattern with an appropriate path to the themed elements (as returned in the `getThemeUrl()`).

10.2.1. Creating a Custom Component Theme

Theming for the Vaadin components is built using `Vaadin.ThemableMixin`. See the vaadin-themable-mixin documentation[23] to learn more.

10.3. Theming Web Components

This section describes how to theme applications that include, or are built with, Web Components (also know as Polymer templates).

When styling Polymer templates, it is important to distinguish global and scoped styles:

- **Global styles** apply to all elements globally. A CSS selector targets any and all elements. This was the case generally before the introduction of Web Components.
- **Scoped styles** are isolated from global styles and apply to only the shadow DOM of a specific Web Component (Polymer template).

10.3.1. Defining Global Styles

Global styles are styles defined in the document scope, that is, styles that target the document body and regular DOM contents (including application views). Global styles exclude styles that target shadow DOM content, for example the internals of a Vaadin Polymer template or other Web Component.

To define global styles:

1. Include your styles in a CSS file.

   Example: `shared-styles.css` typically stored in the `frontend/styles/` folder.

   ```
   html {
     font-size: 1em;
   }
   ```

2. Configure your application to import the CSS file using the `@CssImport` annotation.

   Example: Using the `@CssImport` annotation to import `shared-styles.css`.

   ```
   @Route(value = "")
   @CssImport("./styles/shared-styles.css")
   public class MyApplication extends Div {
   }
   ```

Vaadin wraps the styles appropriately and instructs Polymer to enable correct cross-browser scoping.

10.3.2. Styling Polymer Templates

Since Polymer templates are Web Components, their content is in the shadow DOM. By design, the shadow DOM defines a local-style scope that is isolated from global styles.

You can add component-specific scoped styles directly in the `<style>` tag in the template getter..

Example: Adding component-specific styles in `my-view.js`.

```
import { PolymerElement } from
        '@polymer/polymer/polymer-element.js';
import { html } from
        '@polymer/polymer/lib/utils/html-tag.js';

class MyView extends PolymerElement {
  static get template() {
    return html`
      <style>
        :host {
          /* Styles for the <my-view>
             hosting element */
          display: block;
        }

        .my-view-title {
          font-weight: bold;
          border-bottom: 1px solid gray;
        }
      </style>
      <div class="my-view-title">My view title</div>
    `;
  }

  static get is() {
    return 'my-view';
  }
}
customElements.define(MyView.is, MyView);
```

10.3.3. Using Custom CSS Properties

You can use custom CSS properties to share common style values, such as sizes and colors, among different parts of your application.

Custom CSS properties use the double dash (`--`) syntax, for example `--main-color: black;`, and allow you to assign and reference CSS variables.

Custom CSS property values are inherited and are able to penetrate the shadow DOM. You can use them to style component elements within the shadow DOM. Polymer templates can reuse custom CSS property values defined by global styles, and also override them.

Example: Defining a global custom CSS property in `shared-styles.css` for a HTML element.

```css
html {
  /* Example global custom CSS property definition */
  --my-theme-color: brown;
}
```

You can reference your custom CSS properties using the `var(--my-property)` function.

Example: Using the `var()` function in `my-view.css` .

```css
.my-view-title {
  /* Example referencing custom CSS property */
  color: var(--my-theme-color);
}
```

10.3.4. Using Style Modules

Style modules allow you to use the same style sheet for multiple Polymer templates and global styles.

Example: `common-styles.css` style module.

```css
/* Example style module */
.my-outline-style {
  outline: 1px solid green;
}
```

To import a style module into your application, you need to provide a unique name for the `id` attribute in the `@CssImport` annotation.

Example: Importing `common-styles.css` using the `@CssImport` annotation.

```
@Route(value = "")
@CssImport(value = "./styles/common-styles.css",
           id = "common-styles")
public class MyApplication extends Div {
}
```

You can now reuse the style module when importing other style files using the `include` attribute.

Example: `specific-styles.css`

frontend/styles/specific-styles.css

```
/* Example style */
.my-border-style {
  border: 2px solid grey;
}
```

Example: Using the `include` attribute to include the `common-styles` style module when importing `specific-styles.css`.

```
@Route(value = "")
@CssImport(value = "./styles/specific-styles.css",
           include = "common-styles")
public class MyApplication extends Div {
}
```

If you need to style your templates directly in the client modules, you can define and import modules using JavaScript.

Example: Importing the `common-styles.css` style module in `common-styles.js`.

```
import styles from './common-styles.css'
const $_documentContainer = document
        .createElement('template');
$_documentContainer.innerHTML = `
  <dom-module id="common-styles">
    <template><style>${styles}</style></template>
  </dom-module>`;
document.head.appendChild($_documentContainer.content);
```

Example: Using the `include` attribute to include the `common-styles` style module in `my-view.js`.

```
import { PolymerElement } from
        '@polymer/polymer/polymer-element.js';
import { html } from
        '@polymer/polymer/lib/utils/html-tag.js';
import '../styles/common-styles.js'

class MyView extends PolymerElement {
  static get template() {
    return html`
      <style include="common-styles">
        .my-border-style {
          border: 2px solid grey;
        }
      </style>

      <div class="my-view-title">My view title</div>
    `;
  }
  static get is() {
    return 'my-view';
  }
}
customElements.define(MyView.is, MyView);
```

NOTE You can include multiple style modules into a single scope using a space-separated list of style module ids, for example `<style include="shared-styles other-shared-styles"></style>`.

You can also include style modules in global styles.

Example: Using the `@CssImport` annotation to include the `common-styles` style module while importing `shared-styles.css`.

MyApplication.java

```
@Route(value = "")
@CssImport(value = "./styles/shared-styles.css",
           include = "common-styles")
public class MyApplication extends Div {
}
```

10.4. Theming Overview

Vaadin uses themes to separate the appearance of an application from its logic.

Themes allow you to customize the look and feel of your application and components. You can use Cascading Style Sheets (CSS)[24] files in combination with the `@CssImport` Java annotation to style any content.

In this section we cover:

- Using Themes.
- Theming Basics.
- Theming Web Components.

- Integrating a Custom Component Theme.
- Theming Overlay Components.

NOTE This content does not cover CSS basics. There are many external resources you can use for this purpose.

NOTE If your application includes Polymer templates[25], you need to use style modules for theming.

10.5. Using Component Themes

A theme class automatically handles two things:

- It tells Vaadin what theme to use for the Vaadin components and where the files can be found.
- It specifies a set of shared styles, like fonts etc., that are loaded to the initial page by Vaadin.

10.5.1. Theme Resolving Order

Vaadin applies the following logic, in the following resolving order, to determine which theme to use in your application:

1. If the `@Theme` annotation is found at the root navigation level, the theme set in the annotation is used.
2. If the `@NoTheme` annotation is found at the root navigation level, theming is disabled.
3. If the `com.vaadin.flow.theme.lumo.Lumo` class (from the `vaadin-lumo-theme` project) is available in the classpath, the Lumo[26] theme is used.

When a match is found, resolving stops. If none of the conditions are met, no theme is used. This means that projects that use the `vaadin-core` dependency, use the Lumo theme by default, unless they declare an `@Theme` or `@NoTheme`.

10.5.2. Applying a Theme

To apply a theme, add the `@Theme` annotation to your root-navigation-level class, or to the `RouterLayout` component defined in its `@Route` annotation.

NOTE Your application can only have one `@Theme` annotation. Vaadin does not support different `@Theme` values for each route in your application.

The following examples include various theming scenarios.

Example: Specifying the `Lumo.class` in the `@Theme` annotation.

```
@Route(value = "")
@Theme(value = Lumo.class)
public class LumoApplication extends Div {
}
```

Example: Not using the `@Theme` annotation (because `Lumo.class` is in the classpath).

```
@Route(value = "")
public class DefaultLumoApplication extends Div {
}
```

Example: Using the `@JsModule` annotation to import custom component styles and specifying a custom theme in the

@Theme annotation.

```
@JsModule("@vaadin/vaadin-lumo-styles/color.js")
public class MyTheme implements AbstractTheme {
    @Override
    public String getBaseUrl() {
        return "/src/";
    }

    @Override
    public String getThemeUrl() {
        return "/theme/myTheme/";
    }
}


@Route(value = "")
@Theme(MyTheme.class)
public class MaterialApplication extends Div {
}
```

Example: Specifying a custom theme in the @Theme annotation in a RouterLayout.

```
@Theme(MyTheme.class)
public class MainLayout extends Div
        implements RouterLayout {
}

@Route(value = "", layout = MainLayout.class)
public class HomeView extends Div {
}

@Route(value = "blog", layout = MainLayout.class)
public class BlogPost extends Div {
}
```

Example: Using the @NoTheme annotation to disable theming.

```
@Route(value = "")
@NoTheme
public class UnThemedApplication extends Div {
}
```

NOTE If the `@Theme` annotation is not on a `@Route` component or a top-level `RouterLayout`, an exception is thrown on startup.

10.5.3. Using Theme Variants

A variant is a special string value that can be used as the `theme` attribute value of any custom element. When the corresponding theme is enabled, this values change the visual appearance of the component.

Example: HTML representation of a variant.

```
<vaadin-button theme="contrast primary">
    Themed button
</vaadin-button>
```

You can apply multiple variants to the same element by separating them with white spaces.

Themes can have different types of variants:

- **Global** variants apply globally throughout the application.
- **Component** variants apply only to specific components.

The Lumo and Material themes come with two global variants: light (default) and dark.

10.5.4. Using and Customizing Vaadin Themes

Vaadin provides two ready-made component themes: *Lumo* (main theme for all Vaadin components) and *Material*. Both themes give you with a full set of building blocks to build a modern-looking web application that work well on desktop and mobile. Both themes are a part of the `vaadin-core` dependency. To use the ready-made Vaadin themes:

- **Lumo**: Either explicitly declare it, using `@Theme(Lumo.class)`, on the navigation target, or omit it completely, because the default behavior is to use it, if it is in the classpath.
- **Material**: Explicitly declare it, using `@Theme(Material.class)`, on the navigation target.

Both themes provide customization points for Vaadin components. These allow you to fine tune component appearance and UX. You can customize using CSS custom properties. See the the **Customization** section of the Lumo[27] and Material[28] documentation for more.

Theming for Vaadin components is built using `Vaadin.ThemableMixin`. See the vaadin-themable-mixin[29] documentation for more.

Defining Global Theme Variants

You can set a global theme variant by defining it on the `@Theme` annotation.

Example: Setting the `large` global theme variant for the `MyTheme.class`.

```
@Route(value = "")
@Theme(value = MyTheme.class, variant = "large")
public class LargeThemedApplication extends Div {
}
```

Theme variants are not used by the Lumo or Material themes, by default. You can set the dark variant for either theme by defining it in the `@Theme` annotation.

Example: Setting the dark variant for the Lumo theme.

```
@Route(value = "")
@Theme(value = Lumo.class, variant = Lumo.DARK)
public class DarkApplication extends Div {
}
```

Example: Setting the dark variant for the Material theme.

```
@Route(value = "")
@Theme(value = Material.class, variant = Material.DARK)
public class DarkMaterialApplication extends Div {
}
```

Defining Component Theme Variants

Variants are also available for individual components. Each theme provides a predefined set of variants that you can use. There are different variants for different components, and some components have no variants.

Available component variants are applied using the `Element` API to set the variant as the `theme` attribute.

Variants are converted to their equivalent HTML value. For example, the `ButtonVariant.LUMO_PRIMARY.getVariantName()` method is used to convert a button variant to HTML. After conversion, the HTML representation is added as the `theme` attribute value.

All components that implement the `HasTheme` interface have an `addThemeVariants` method and can use the API.

The following three examples all achieve the same result. They demonstrate different ways to add `contrast` and `primary` Lumo theme variants to the `theme` attribute value of the button component:

- **Example**: Using the `addThemeVariants` method to add theme variants for the `Button` component.

  ```
  Button button = new Button("Themed button");
          button.addThemeVariants(ButtonVariant
  .LUMO_PRIMARY,
                  ButtonVariant.LUMO_CONTRAST);
  ```

- **Example**: Using the `getThemeNames().addAll` method to add an array of theme variants for the `Button` component.

  ```
  Button button = new Button("Themed button");
  button.getThemeNames().addAll(
          Arrays.asList("contrast", "primary"));
  ```

- **Example**: Adding variants to the `theme` attribute of the `Button` component by manipulating the `theme` attribute.

```
Button button = new Button("Themed button");
String themeAttributeName = "theme";
String oldValue = button.getElement()
        .getAttribute(themeAttributeName);
String variantsToAdd = "contrast primary";
button.getElement().setAttribute(themeAttributeName,
        oldValue == null || oldValue.isEmpty() ?
            variantsToAdd
            : ' ' + variantsToAdd);
```

- This example provides more flexibility. It allows you to manipulate the value of the `theme` attribute directly. This is useful when adding non-standard theme variants to a component.

NOTE Component theme variants only work if the corresponding theme is set. If a different theme or no theme is set, variants in the `theme` attribute of the have no effect on the look and feel of the component.

Using Vaadin Theme Presets

The Lumo theme include a compact preset that defines values for sizing and spacing properties. The preset reduces the visual space required by components and allows you to fit more content on the screen. You can use the `@JsModule` annotation to import the compact present.

Example: Using the `@JsModule` annotation to import the compact preset on a `RouterLayout`.

```
@JsModule("@vaadin/vaadin-lumo-styles/presets/compact.js
")
@Theme(Lumo.class)
public class CompactMainLayout extends Div
        implements RouterLayout {
}
```

10.6. Theming Overlay Components

The `<vaadin-overlay>` component allows you to create an overlay. This component an essential part of many components, for example, `dialog`, `notification`, `combo-box`, `date-picker`, `time-picker`, `select` and `context-menu`. These components are made up of two components:

- **Main** component, for example `<vaadin-dialog>`, that is not visible (`display:none`) on the page.
- **Overlay** component, for example `<vaadin-dialog-overlay>`, that is visible on the page.

10.6.1. Styling an Overlay

The `<vaadin-overlay>` component contains three stylable parts:

- `backdrop`: The optional modality curtain that covers the whole viewport.
- `overlay`: The container to position, size, and align the content. It is typically also a scrolling container.
- `content`: The content area inside the scrolling container (overlay). You can apply padding in this part to affect the size of the scrolled content.

The `theme` attribute is the only attribute that is copied from the main component to the overlay component. This allows you to style individual overlays. The `theme` attribute is the only exception: all other attributes (for example, the class selector) are not copied from the main component to the overlay component.

To style an overlay component, you need to create a style CSS module, target the stylable parts, instruct Vaadin to import it as a module targeting the specific component, and set the `theme` attribute to the main component in your view.

Example: Creating the `my-overlay-theme.css` style CSS module.

```
:host([theme~="custom-theme-variant"]) [part~="overlay"]
{
}
```

Example: Using the `@CssImport` annotation to import `my-overlay-theme.css` into `MyApplication`.

```
@Route(value = "")
@CssImport(value = "./styles/my-overlay-theme.css",
           themeFor = "vaadin-*-overlay")
public class MyApplication extends Div {
}
```

Example: Using the `setAttribute` method to set the `theme` attribute in `MyView`.

```
public class MyView extends VerticalLayout {
    public MyView() {
        Dialog dialog = new Dialog();
        dialog.getElement().setAttribute("theme",
                "custom-theme-variant");
    }
}
```

If you want to be more specific and target the overlay of `vaadin-dialog`, do not use wildcards in the `themeFor` attribute, rather target the `vaadin-dialog-overlay` element directly.

Example: Creating the `my-dialog-overlay-theme.css` style CSS module.

```
[part="backdrop"] {
}
[part="overlay"] {
}
[part="content"] {
}
```

Example: Using the `@CssImport` annotation to import `my-dialog-overlay-theme.css` into `MyApplicationWithDialog`.

MyApplicationWithDialog.java

```
@Route(value = "")
@CssImport(value="./styles/my-dialog-overlay-theme.css",
           themeFor = "vaadin-dialog-overlay")
public class MyApplicationWithDialog extends Div {
}
```

NOTE — `ThemableMixin` does not guarantee the order in which the style modules are applied. It is important to declare CSS rules whose specificity is greater than the Lumo properties that are being overridden.

10.7. Migrating Theming Files from Polymer 2 to Polymer 3

The most significant change in Polymer 3.0 is that you need to use ES6 modules, instead of HTML imports (that are no longer used).

10.7.1. Steps to migrate `.html` theming files to `.js` and `.css`

The instructions below assume the following Polymer 2 example in a Vaadin version 13 application:

```
<link rel="import" href=
"../bower_components/polymer/lib/elements/custom-
style.html">

<dom-module id="my-app-layout-theme" theme-for="vaadin-
app-layout">
  <template>
    <style>
      :host {
        background-color: var(--lumo-shade-5pct)
!important;
      }
      [part="content"] {
        height: 100%;
      }
    </style>
  </template>
</dom-module>

<custom-style>
  <style>
    .v-system-error {
        color: red;
    }
  </style>
</custom-style>
```

To convert your files to Polymer 3 and make them them usable in a Vaadin version 14 application.

1. Generate as many `.css` files as HTML blocks in your `.html` file

   frontend/styles/my-app-layout-theme.css

   ```
   :host {
     background-color: var(--lumo-shade-5pct) !important;
   }
   [part="content"] {
     height: 100%;
   }
   ```

   frontend/styles/my-custom-styles.css

   ```
   .v-system-error {
       color: red;
   }
   ```

2. Instruct Vaadin to import these CSS files using the `CssImport` annotation.

   ```
   @Route(value = "")
   // will be imported as a <dom-module> tag for theming
   components
   @CssImport(value = "./styles/my-app-layout-theme.css",
   themeFor = "vaadin-app-layout")
   // will be imported as a <custom-style> tag
   @CssImport(value = "./styles/my-custom-styles.css")
   public class MyApplication extends Div {
   }
   ```

----

[20] https://vaadin.com/themes/lumo
[21] https://vaadin.com/themes/material
[22] https://material.io/

[23] https://github.com/vaadin/vaadin-themable-mixin/wiki
[24] https://www.w3.org/Style/CSS/
[25] https://vaadin.com/docs/flow/polymer-templates/tutorial-template-basic.html
[26] https://vaadin.com/themes/lumo
[27] https://vaadin.com/themes/lumo
[28] https://vaadin.com/themes/material
[29] https://github.com/vaadin/vaadin-themable-mixin/wiki

11. Spring integration

11.1. Using Vaadin with Spring Boot

The Vaadin Spring[30] add-on allows you to use Vaadin with Spring Boot[31].

Spring Boot[32] speeds up the development process and provides a fast and efficient development environment by emphasising . It is the easiest way to use the Spring framework.

NOTE See Using Vaadin with Spring MVC to learn how to use Vaadin in more traditional Spring MVC[33] web application, without Spring Boot.

The easiest way to create an application with Spring Boot and Vaadin is to start with a template application created by https://vaadin.com/start or https://start.spring.io/, but you can also add required dependencies manually to your project.

11.1.1. Adding Dependencies

Like many other tech stacks on Spring Boot, Vaadin provides a starter dependency that provide all essential modules and autoconfiguration. Only the `vaadin-spring-boot-starter` dependency is needed, but it is suggested to also declare the vaadin-bom if you need additional Vaadin dependencies. For production builds, it is also suggested to add the vaadin-maven-plugin, that generates the optimized JavaScript packages.

Example Vaadin Spring Boot dependencies in `pom.xml`.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-bom</artifactId>
            <!-- declare the latest Vaadin version
                 as a property or directly here -->
            <version>${vaadin.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>com.vaadin</groupId>
        <artifactId>
            vaadin-spring-boot-starter
        </artifactId>
        <version>${vaadin.version}</version>
    </dependency>
</dependencies>
<build>
    <plugins>
        <!-- The Spring Boot Maven plugin for easy
             execution from CLI and packaging -->
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>
                spring-boot-maven-plugin
            </artifactId>
        </plugin>

        <!--
            Takes care of synchronizing java
            dependencies and imports in package.json and
            main.js files. It also creates
            webpack.config.js if does not exist yet.
        -->
        <plugin>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-maven-plugin</artifactId>
            <version>${vaadin.version}</version>
            <executions>
                <execution>
                    <goals>
                        <goal>prepare-frontend</goal>
                        <goal>build-frontend</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

- The `vaadin-bom` dependency in the `dependencyManagement` section declares the versions of modules in current Vaadin release.

11.1.2. Running Spring Boot applications

Spring Boot applications are executed via traditional main method. If Vaadin Spring dependency is on your classpath, Spring Boot automatically starts a web server and configures Vaadin with Spring. If you created your project via vaadin.com/start or start.spring.io, an application class with the main method is already available for you.

Example: `Application` class.

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

- The `@SpringBootApplication` annotation enables Spring Boot under the hood. This includes Spring configuration, component scanning and auto-configuration.

TIP Follow the instructions on Spring Boot documentation[34], if you want to deploy your Spring Boot application as a traditional WAR file.

11.1.3. Vaadin Spring Boot examples

Vaadin Spring Examples[35] is an example application that showcases basic usage of Vaadin and Spring Boot. You can use it to test the concepts and features covered in this documentation.

11.2. Using Vaadin with Spring MVC

In this section we cover how to use Vaadin with Spring MVC [36]. Spring MVC is the original Spring web framework built on the Servlet API.

NOTE See Using Vaadin with Spring Boot to use Vaadin with Spring Boot[37].

11.2.1. Registering the Vaadin Serlet

To use Vaadin in your Spring web application you need to register the Vaadin `SpringServlet` as a dispatcher servlet.

Example: Registering the `SpringServlet` as a dispatcher servlet.

```
public abstract class ExampleWebAppInitializer
        implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext)
            throws ServletException {
        AnnotationConfigWebApplicationContext context =
            new AnnotationConfigWebApplicationContext();
        registerConfiguration(context);
        servletContext.addListener(
                new ContextLoaderListener(context));

        ServletRegistration.Dynamic registration =
            servletContext.addServlet("dispatcher",
                new SpringServlet(context, true));
        registration.setLoadOnStartup(1);
        registration.addMapping("/*");
    }

    private void registerConfiguration(
        AnnotationConfigWebApplicationContext context) {
        // register your configuration classes here
    }
}
```

11.2.2. Registering Vaadin Scopes

To use Vaadin Spring scopes you need to register the `VaadinScopesConfig` configuration class. As an alternative, you can add the `@EnableVaadin` annotation to your configuration class to import `VaadinScopesConfig`.

The Vaadin Spring add-on[38] provides the `VaadinMVCWebAppInitializer` class that is an abstract subclass of the `WebApplicationInitializer` class. You can extend this class and provide your configuration classes by implementing the `getConfigurationClasses()` method.

Example: Extending `VaadinMVCWebAppInitializer` and implementing the `getConfigurationClasses()` method.

```
public class SampleWebAppInitializer
        extends VaadinMVCWebAppInitializer {

    @Override
    protected Collection<Class<?>>
            getConfigurationClasses() {
        return Collections.singletonList(
                SampleConfiguration.class);
    }
}

@Configuration
@ComponentScan
public class SampleConfiguration {
}
```

- This registers `VaadinScopesConfig` and `VaadinServletConfiguration` automatically.

11.2.3. Handling URLs

To handle URLs, you need at least one Vaadin component, annotated with `@Route`. See Handling URLs for an `@Route` annotation example.

11.2.4. Declaring Dependencies

To use your Spring web application, you need to declare dependencies in your `pom.xml` file to `vaadin-bom` and `spring-web` as follows:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-bom</artifactId>
            <version>${vaadin.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>com.vaadin</groupId>
        <artifactId>vaadin-spring</artifactId>
    </dependency>

    <!-- Spring -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>5.0.2.RELEASE</version>
    </dependency>
```

11.3. Using Routing with Spring

Routing works the same way with Spring as it does in plain Vaadin applications. See Routing and Navigation for more.

This content applies to using Vaadin with both Spring Boot and Spring MVC.

11.3.1. Defining Routes

To handle the default route, all you need to do is define a component with the `@Route("")` annotation.

Example: Defining `RootComponent` as the default root target using the `@Route` annotation.

```
@Route("")
public class RootComponent extends Div {
    public RootComponent(){
        setText("Default path");
    }
}
```

You can also define all other possible routes in the same way as in standard Vaadin applications. See Defining Routes with @Route for more about using the `Router` class.

11.3.2. Using Dependency Injection and Spring Autowiring

The only difference between using the router in a standard application and a Spring application, is that in Spring you can use dependency injection in components annotated with `@Route`. These components are instantiated by Spring and become Spring-initialized beans. In particular, this means you can autowire other Spring-managed beans.

Example: Using autowiring in a component annotated with `@Route`.

```java
@Route("")
public class RootComponent extends Div {

    public RootComponent(@Autowired DataBean dataBean) {
        setText(dataBean.getMessage());
    }
}

public interface DataBean {
    String getMessage();
}

@Component
public class DataBeanImpl implements DataBean {

    public String getMessage(){
        return "message";
    }
}
```

11.3.3. Routing in Spring Boot and WAR Applications

There is a difference between running an application as a Spring Boot application and as a WAR application deployed to a Web server.

In WAR applications, all `@Route` annotations are discovered automatically, due to the Servlet 3.0 specification. With Spring Boot applications, this is, by design, not the case. See Spring Boot #321[39] for more.

The Vaadin Spring add-on[40] implements scanning for router classes in Spring Boot applications. This is also true for other Vaadin types that need to be discovered and registered at startup. However, scanning only occurs inside the Spring Boot application class package, that is the package in which the `@SpringBootApplication` class resides. If your application contains route classes in packages that are not scanned by default, you have two options: move them to the package (or subpackage) in which the `@SpringBootApplication` application class resides, or explicitly specify the packages that should be scanned. You can specify packages to scan using the `value` parameter in the `@EnableVaadin` annotation.

NOTE You do not need to use the `@EnableVaadin` annotation at all with Spring Boot. This annotation is intended to be used with Spring MVC to enable the Vaadin configuration. In Spring Boot, auto-configuration is available which makes it work out of the box. The only reason to use `@EnableVaadin` is to specify the packages to scan with Spring MVC.

11.4. Vaadin Spring Scopes

11.4.1. Contexts and Scopes in Spring

- **Contexts** in Spring are services that manage the lifecycle of objects and handle their injection. Generally, a context refers to a situation in which an instance is used with a unique identity. These objects are essentially "singletons" in the context.
- **Scopes** are narrower than contexts. While conventional singletons are application-wide, objects managed by a Spring container are "singletons" in a narrower scope. Examples include a user session, a particular UI instance associated with the session, or even a single request. A scope defines the lifecycle of the object, that is its creation, use, and destruction.

11.4.2. Using Spring Scopes

In most programming languages, a variable name refers to a unique object within the scope of the variable. In Spring, an object has a unique identity within a scope, but instead of identifying the object by its variable name, it is identified by its type (object class) and qualifiers, if any.

In addition to standard Spring scopes, the Vaadin Spring add-on[41] introduces two additional scopes: `VaadinSessionScope` and `UIScope`.

- The `@VaadinSessionScope` annotation manages the Spring beans during the Vaadin session lifecycle. It ensures that the same bean instance is used during the whole Vaadin session.

  Example: Using the `@VaadinSessionScope` annotation.

```
@Route("")
public class MainLayout extends Div {
    public MainLayout(@Autowired SessionService bean)
{
        setText(bean.getText());
    }
}

@Route("editor")
public class Editor extends Div {
    public Editor(@Autowired SessionService bean) {
        setText(bean.getText());
    }
}

@Component
@VaadinSessionScope
public class SessionService {
    private String uid = UUID.randomUUID().toString();

    public String getText(){
        return "session " + uid;
    }
}
```

  - Provided you access the application from the same Vaadin session, the same instance of `SessionService` is used. This is because it is session scoped.
  - If you open the root target in one tab and the `editor` target in another, the text in both is the same. This happens because the session is the same, even though the tabs (and `UI` instances) are different.

- The `@UIScope` annotation manages the Spring beans during the `UI` lifecycle.

  Example: Using the `@UIScope` annotation.

```
@Route("")
public class MainLayout extends Div {
    public MainLayout(@Autowired UIService bean) {
        setText(bean.getText());
    }
}

@Route("editor")
public class Editor extends Div {
    public Editor(@Autowired UIService bean) {
        setText(bean.getText());
    }
}

@Component
@UIScope
public class UIService {
    private String uid = UUID.randomUUID().toString();

    public String getText() {
        return "ui " + uid;
    }
}
```

- The `UIService` is now the same while in the same `UI`, because it is UI scoped. When using `@UIScope`, the same bean instance is used inside the same `UI` instance.
- If you open the root target in one tab and the `editor` target in another, the text in each is different, because the `UI` instances are different. However, if you navigate to the `Editor` instance via the router (or a `UI` instance which delegates navigation to the router), the text is the same.
- **Example**: Navigating to the `editor` target.

```
public void edit() {
    getUI().get().navigate("editor");
}
```

See Application Lifecycle > Loading a UI[42] and User Session[43] for more.

11.5. Vaadin Spring Configuration

You can use many properties to configure your Vaadin application. See, for example, the `com.vaadin.server.DeploymentConfiguration` and `com.vaadin.server.Constants` classes for the numerous property names. In addition to these properties, you can also set Spring properties as system properties. Spring configuration properties have the same names, but are prefixed with `vaadin.`.

11.5.1. Special configuration parameters

`blacklisted-packages` is a comma separated string that can be used to blacklist packages from getting scanned in v14 (npm mode) projects. As the default set of packages doesn't cover everything that we shouldn't be interested in.

application.properties

```
vaadin.blacklisted-
packages=org/bouncycastle,com/my/db/package
```

`whitelisted-packages` is a comma separated string that can be used to specify the only packages that need to be scanned for UI components and views. In order to improve the performance during development, it's recommended to set this property especially in big applications. Note that `com/vaadin/flow/component` is implicitly included and is always scanned.

application.properties

```
vaadin.whitelisted-
packages=com/foo/myapp/ui,com/foo/components
```

NOTE You should use either `whitelisted-packages` or `blacklisted-packages`. In case both of them have values, `blacklisted-packages` will be ignored.

11.5.2. Using Spring Boot Properties

You can set properties for Spring Boot in your `application.properties` file.

Example: Setting Spring URL mapping in `application.properties`.

```
vaadin.urlMapping=/my_mapping/*
```

- By default, URL mapping is `/*`.

NOTE An additional servlet, such as `/my_mapping/*`, is required to handle the frontend resources for non-root servlets. The servlet can be defined in your application class. See this `Application` class[44] for a example.

11.5.3. Configuring Spring MVC Applications

If you use Spring MVC, and therefore the `VaadinMVCWebAppInitializer` subclass, you need to populate your configuration properties yourself.

Example: Setting configuration properties in a Spring MVC application.

```
@Configuration
@ComponentScan
@PropertySource("classpath:application.properties")
public class MyConfiguration {

}
```

- The `application.properties` file is still used, but you can use any name and any property source.

11.6. Getting Started with Spring and Vaadin

A tutorial application which showcases the basic usage of a Vaadin & Spring Boot Application is available at https://github.com/vaadin/flow-spring-tutorial. You can use this application example to test the different concepts and features presented in the documentation.

For starting a new project with Spring and Vaadin, you can clone the Project Base for Vaadin and Spring[45] repository. It is a project template with the necessary configuration and dependencies included for starting building you own application. This starter will be soon available for download from https://vaadin.com/start, where you can customize the naming (package, project) for it.

----

[30] https://vaadin.com/directory/component/vaadin-spring/
[31] https://spring.io/projects/spring-boot
[32] https://spring.io/projects/spring-boot
[33] https://docs.spring.io/spring/docs/current/spring-framework-

reference/web.html
[34] https://docs.spring.io/spring-boot/docs/current/reference/html/howto-traditional-deployment.html
[35] https://github.com/vaadin/flow-spring-examples
[36] https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html
[37] https://spring.io/projects/spring-boot
[38] https://vaadin.com/directory/component/vaadin-spring/overview
[39] https://github.com/spring-projects/spring-boot/issues/321
[40] https://vaadin.com/directory/component/vaadin-spring/overview
[41] https://vaadin.com/directory/component/vaadin-spring/overview
[42] https://vaadin.com/docs/flow/advanced/tutorial-application-lifecycle.html#application.lifecycle.ui
[43] https://vaadin.com/docs/flow/advanced/tutorial-application-lifecycle.html#application.lifecycle.session
[44] https://raw.githubusercontent.com/vaadin/flow-and-components-documentation/master/tutorial-servlet-spring-boot/src/main/java/org/vaadin/tutorial/spring/Application.java
[45] https://github.com/vaadin/flow-spring-tutorial

12. CDI integration

12.1. Using Vaadin with CDI

The Vaadin CDI[46] add-on allows you to use Vaadin with CDI [47].

12.1.1. Adding Dependencies

The `vaadin-cdi` add-on should be packaged in your appplication and deployed to an application server that is compliant with Java EE 7 (or newer).

If you are a `vaadin-platform` user, add the following dependencies in your `pom.xml`.

Example CDI dependencies in `pom.xml`.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-bom</artifactId>
            <version>${vaadin.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
         <groupId>com.vaadin</groupId>
         <artifactId>vaadin-cdi</artifactId>
    </dependency>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>7.0</version>
        <scope>provided</scope>
    </dependency>
```

12.1.2. Vaadin Version Compatibility

The version for `vaadin-cdi` is managed by `vaadin-bom`. For Vaadin 14, the CDI add-on version is 11.0.

You need the CDI API version 1.2 and a provided implementation. In practice, because other Java EE/Jakarta EE features are used together with CDI, the simplest way to ensure compatibility is to use a Java EE 7 (or newer) container.

12.1.3. Configuring the Vaadin CDI Add-on

There are no specific CDI configuration options.

An instance of the CDI-enabled Vaadin servlet, `com.vaadin.cdi.CdiVaadinServlet`, is deployed automatically, provided you do not setup a Vaadin servlet in your `web.xml`or use the `@WebServlet` annotation. You can also customize `CdiVaadinServlet` to suit your setup.

NOTE	See Changing Vaadin Behavior with Runtime Configuration [48] for more about about Vaadin servlet configuration.

12.2. Getting Started with CDI and Vaadin Tutorial

Vaadin CDI tutorial[49] is a prebuilt example application that showcases basic usage of a Vaadin and CDI. You can use it to test the concepts and features covered in this documentation.

To start a new project with CDI and Vaadin, select the **CDI and Java EE** technology stack at https://vaadin.com/start and follow the instructions in the webpage. This project template includes the necessary configuration and dependencies to start building you own application.

12.3. Using CDI Beans in Instantiated Components

When using Vaadin CDI[50], most objects instantiated by framework become managed beans. The framework uses the CDI `BeanManager` to get references to beans. This means they are fully-fledged CDI contextual instances.

The add-on looks up the CDI bean by type (component class) and `@Any`.

If the type is not found as a CDI bean - for example, because it is ambiguous or does not have a no-arg public constructor - instantiation falls back to the default Vaadin behavior, and the component is instantiated as a POJO. Dependency injection is performed after instantiation and injects still work, but other CDI features do not. The reason is that the instantiated component is not a contextual instance.

NOTE Methods annotated with `@PreDestroy` in `Dependent` beans instantiated by the framework are not run.

12.3.1. Using Router Components

All route targets, router layouts, and exception targets become managed beans when the add-on is used. The components look and behave the same as without the add-on, but CDI features are available.

Example: Using the `@Inject` annotation on a basic route target.

```
@Route
public class MainView extends VerticalLayout {
    @Inject
    public MainView(Greeter greeter) {
        add(new Span(greeter.sayHello()));
    }
}
```

NOTE Vaadin scans for router components on startup and is unaware of CDI beans. Using producers or the `@Typed` annotation causes issues with this kind of bean.

12.3.2. Using Components Injected into Polymer Templates

Components injected into Polymer template classes[51] using the `@id` annotation become managed beans when the add-on is used.

Example: Using the `@Id` annotation to inject the `DependentLabel` component into `TestTemplate` class.

```
public class TestTemplate
        extends PolymerTemplate<TemplateModel> {
    @Id
    private DependentLabel label;
}
```

Example: `DependentLabel` class.

```
@Dependent
@Tag("dependent-label")
public class DependentLabel extends Label {
    @Inject
    private Greeter greeter;

    @PostConstruct
    private void init() {
        setText(greeter.sayHello());
    }
}
```

Example: `TestTemplate.html` Polymer template.

```
import {PolymerElement,html} from
        '@polymer/polymer/polymer-element.js';

class TestTemplate extends Polymer.Element {
    static get template() {
        return html`
            <div>
                <dependent-label id="label"/>
            </div>`;
    }

    static get is() { return 'test-template' }
}
customElements.define(TestTemplate.is, TestTemplate);
```

IMPORTANT The managed bean injected into the template should not exist before the template is instantiated. If it does exist at this time, it may not bind to its element, and this may result in an incorrect component tree.

12.3.3. Using a Custom UI

It is not necessary to define a custom `UI` subclass for your application, but it is possible to define one using the corresponding servlet parameter, if needed.

The custom `UI` subclass is instantiated by Vaadin as a POJO (not as a managed bean), but it is still possible to achieve dependency injection. Use `BeanManager` in your overridden `UI.init` method, for example `BeanProvider.injectFields(this)` (in Deltaspike).

12.4. Vaadin CDI Contexts

In addition to standard CDI contexts, the Vaadin CDI[52] add-on introduces new contexts.

Vaadin CDI contexts are conceptually similar to Vaadin Spring scopes.

12.4.1. Normal Scopes

In CDI, most scopes are normal scopes. This means that most calls to managed beans are delegated by a client proxy to the active instance. The active instance is provided by the context.

The Vaadin CDI[53] add-on introduces the `@VaadinServiceScoped`, `@VaadinSessionScoped`, `@NormalUIScoped`, `@NormalRouteScoped` normal scopes..

NOTE The Vaadin component hierarchy does not work properly with CDI client proxies. As a precaution, the `vaadin-cdi` add-on does not deploy if managed beans are found.

12.4.2. Pseudo Scopes

Any scope that is not a normal scope is called a pseudo scope. The standard `@Dependent` and `@Singleton` are pseudo scopes.

The Vaadin add-on additionally introduces the `@UIScoped` and `@RouteScoped` pseudo scopes.

Injection of a pseudo-scoped bean creates a direct reference to the object, but there are some limitations when not using proxies:

- Circular referencing, for example injecting A to B and B to A, does not work.

- Injecting into a larger scope binds the instance from the currently active smaller scope, and ignores changes in the smaller scope. For example, a `@UIScoped` bean after being injected into a session scope will point to the same instance (even its `UI` is closed) regardless of current `UI`.

Using Push

Vaadin contexts are usable inside the `UI.access` method with any push transport.

Certain default contexts from CDI, such as `RequestScoped` or `SessionScoped`, can be problematic. HttpServletRequest can't be resolved from a WebSocket connection in CDI and that is needed for HTTP request, session, and conversation contexts. You should, therefore, use `WEBSOCKET_XHR` (the default), or `LONG_POLLING` transport mode, to avoid losing the standard contexts.

Background-thread contexts that depend on HTTP requests are not active, regardless of push.

See Asynchronous Updates[54] for more about using push.

`@VaadinServiceScoped` Context

The `@VaadinServiceScoped` context manages the beans during the Vaadin service lifecycle. The lifecycle of the service is the same as the lifecycle of its Vaadin servlet. See Vaadin Servlet and Service[55] for more about the Vaadin service.

For beans that are automatically picked up by `VaadinService`, you need to use the `@VaadinServiceEnabled` annotation, together with the

`@VaadinServiceScoped` annotation. See Vaadin Service Interfaces as CDI Beans for more.

`@VaadinSessionScoped` Context

The `@VaadinSessionScoped` context manages the beans during Vaadin session lifecycle. This means that the same bean instance is used within the whole Vaadin session.

See User Session[56] for more.

Example: Using the `@VaadinSessionScoped` annotation on route targets.

```
@Route("")
public class MainLayout extends Div {
    @Inject
    public MainLayout(SessionService bean){
        setText(bean.getText());
    }
}

@Route("editor")
public class Editor extends Div {
    @Inject
    public Editor(SessionService bean){
        setText(bean.getText());
    }
}

@VaadinSessionScoped
public class SessionService {
    private String uid = UUID.randomUUID().toString();

    public String getText(){
        return "session " + uid;
    }
}
```

- Because it is session scoped, the same instance of `SessionService` is used if the application is accessed from the same Vaadin session.
- If you open the root target in one tab and the `editor` target in another, the text in both is the same. This is because the session is the same even though the tabs (and `UI` instances) are different.

12.4.3. `@UIScoped` and `@NormalUIScoped` Contexts

The `@UIScoped` and `@NormalUIScoped` contexts manage the beans during the `UI` lifecycle. Use `@UIScoped` for components and `@NormalUIScoped` for other beans.

See Loading a UI[57] for more about the `UI` lifecycle.

Example: Using the `@NormalUIScoped` annotation on route targets.

```java
@Route("")
public class MainLayout extends Div {
    @Inject
    public MainLayout(UIService bean){
        setText(bean.getText());
    }
}

@Route("editor")
public class Editor extends Div {
    @Inject
    public Editor(UIService bean){
        setText(bean.getText());
    }
}

@NormalUIScoped
public class UIService {
    private String uid = UUID.randomUUID().toString();

    public String getText(){
        return "ui " + uid;
    }
}
```

- Because it is UI scoped, the same `UIService` is used while in the same `UI`.
- If you open the root target in one tab and the `"editor"` target in another, the text is different because the `UI` instances are different.
- If you navigate to the `editor` instance via the router (or the `UI` instance which delegates navigation to the router) the text is the same.

  Example: Navigating to the `"editor"` target.

  ```java
  public void edit() {
      getUI().get().navigate("editor");
  }
  ```

- In the same `UI` instance, the same bean instance is used with both `@UIScoped` and `@NormalUIScoped`.

12.4.4. `@RouteScoped` and `@NormalRouteScoped` Contexts

The `@RouteScoped` and `@NormalRouteScoped` manage the beans during the `Route` lifecycle. Use `@RouteScoped` for components and `@NormalRouteScoped` for other beans.

Together with the `@RouteScopeOwner` annotation, both `@RouteScoped` and `@NormalRouteScoped` can be used to bind beans to router components (`@Route`, `RouteLayout`, `HasErrorParameter`). While the owner remains in the route chain, all beans owned by it remain in the scope.

See Defining Routes With @Route and Router Layouts and Nested Router Targets for more about route targets, route layouts, and the route chain.

Example: Using the `@NormalRouteScoped` annotation on route targets.

```
@Route("")
@RoutePrefix("parent")
public class ParentView extends Div
        implements RouterLayout {
    @Inject
    public ParentView(
            @RouteScopeOwner(ParentView.class)
            RouteService routeService) {
        setText(routeService.getText());
    }
}

@Route(value = "child-a", layout = ParentView.class)
public class ChildAView extends Div {
    @Inject
    public ChildAView(
            @RouteScopeOwner(ParentView.class)
            RouteService routeService) {
        setText(routeService.getText());
    }
}

@Route(value = "child-b", layout = ParentView.class)
public class ChildBView extends Div {
    @Inject
    public ChildBView(
            @RouteScopeOwner(ParentView.class)
            RouteService routeService) {
        setText(routeService.getText());
    }
}

@NormalRouteScoped
@RouteScopeOwner(ParentView.class)
public class RouteService {
    private String uid = UUID.randomUUID().toString();

    public String getText() {
        return "ui " + uid;
    }
}
```

- `ParentView`, `ChildAView`, and `ChildBView` (paths: `/parent`,

`/parent/child-a`, and `/parent/child-b`) use the same `RouteService` instance while you navigate between them. After navigating away from `ParentView`, the `RouteService` is also destroyed.

- Even though `@RouteScopeOwner` is redundant because it is a CDI qualifier, you need to define it on both the bean and on the injection point.

Route components can also be `@RouteScoped`. In this case, `@RouteScopeOwner` should point to a parent layout. If you omit it, the owner itself becomes the class.

Example: Using the `@RouteScoped` annotation on an `@Route` component.

```
@Route("scoped")
@RouteScoped
public class ScopedView extends Div {
    private void onMessage(
            @Observes(notifyObserver = IF_EXISTS)
            MessageEvent message) {
        setText(message.getText());
    }
}
```

- The message is delivered to the `ScopedView` instance that was already navigated to. If on another view, there is no instance of this bean and the message is not delivered to it.

12.5. Observable Vaadin Events

The Vaadin CDI[58] add-on publishes many Vaadin events to CDI.

It is not necessary to register a listener, using only an observer is sufficient to handle these events.

Events published to CDI include:

- `ServiceInitEvent` See VaadinServiceInitListener[59] for more.
- `PollEvent`.
- `BeforeEnterEvent`. See Navigation Lifecycle for more.
- `BeforeLeaveEvent`. See Navigation Lifecycle for more.
- `AfterNavigationEvent`. See Navigation Lifecycle for more.
- `UIInitEvent`. See UIInitListener[60] for more.
- `SessionInitEvent`. See Handling Session Initialization and Destruction[61] for more.
- `SessionDestroyEvent`. See Handling Session Initialization and Destruction[62] for more.
- `ServiceDestroyEvent`.

WARNING Whether or not `ServiceDestroyEvent` works with CDI during application shutdown depends on each specific implementation.

Example: Using the `@Observes` annotation to listen `ServiceInitEvent`.

```
public class BootstrapCustomizer {

    private void onServiceInit(@Observes
            ServiceInitEvent serviceInitEvent) {
        serviceInitEvent.addBootstrapListener(
                this::modifyBootstrapPage);
    }

    private void modifyBootstrapPage(
            BootstrapPageResponse response) {
        response.getDocument().body().append(
                "<p>By CDI add-on</p>");
    }
}
```

12.6. Vaadin Service Interfaces as CDI Beans

Some Vaadin service interfaces can be implemented as CDI beans. If you do this, the service interface becomes a managed bean with CDI features, and there is no need to register the implementation in Vaadin.

The Vaadin CDI[63] add-on references the following interfaces:

- `I18NProvider`.
- `Instantiator`.
- `SystemMessagesProvider`.
- `ErrorHandler`.

To ensure that the beans are recognized, they should be qualified by the `@VaadinServiceEnabled` annotation.

Example: Using the `@VaadinServiceEnabled` annotation to qualify `TestSystemMessagesProvider`.

```
@VaadinServiceEnabled
@VaadinServiceScoped
public class TestSystemMessagesProvider
        implements SystemMessagesProvider {

    @Override
    public SystemMessages getSystemMessages(
            SystemMessagesInfo systemMessagesInfo) {
        CustomizedSystemMessages messages =
                new CustomizedSystemMessages();
        messages.setInternalErrorMessage(
                "Sorry, something went wrong :(");
        return messages;
    }
}
```

- The purpose of the `@VaadinServiceScoped` context is to define a context with the lifespan of the Vaadin service. It is not mandatory for this kind of bean, but is recommended because other Vaadin contexts can be problematic. For example there is no guarantee that an active Vaadin session or UI context exists when the add-on looks up any of these beans. It is safe to use standard CDI `@Dependent` and `@ApplicationScoped` contexts.

12.7. Getting Started with CDI and Vaadin

A tutorial application which showcases the basic usage of a Vaadin CDI Application is available at https://github.com/vaadin/flow-cdi-tutorial. You can use this application example to test the different concepts and features presented in the documentation.

For starting a new project with CDI and Vaadin, you can get a project base for Vaadin and CDI from [vaadin.com/start[64]. It is a project template with the necessary configuration and dependencies included for starting building you own application. This starter is also available for cloning from Github[65].

----

[46] https://vaadin.com/directory/component/vaadin-cdi/
[47] https://tools.jboss.org/features/cdi.html
[48] https://vaadin.com/docs/flow/advanced/tutorial-flow-runtime-configuration.html
[49] https://github.com/vaadin/flow-cdi-tutorial
[50] https://vaadin.com/directory/component/vaadin-cdi/
[51] https://vaadin.com/docs/flow/polymer-templates/tutorial-template-basic.html
[52] https://vaadin.com/directory/component/vaadin-cdi/
[53] https://vaadin.com/directory/component/vaadin-cdi/
[54] https://vaadin.com/docs/flow/advanced/tutorial-push-access.html#asynchronous-updates
[55] https://vaadin.com/docs/flow/advanced/tutorial-application-lifecycle.html#vaadin-servlet-and-service
[56] https://vaadin.com/docs/flow/advanced/tutorial-application-lifecycle.html#user-session
[57] https://vaadin.com/docs/flow/advanced/tutorial-application-lifecycle.html#loading-a-ui
[58] https://vaadin.com/directory/component/vaadin-cdi/
[59] https://vaadin.com/docs/flow/advanced/tutorial-service-init-listener.html
[60] https://vaadin.com/docs/flow/advanced/tutorial-ui-init-listener.html
[61] https://vaadin.com/docs/flow/advanced/tutorial-application-lifecycle.html#handling-session-initialization-and-destruction
[62] https://vaadin.com/docs/flow/advanced/tutorial-application-lifecycle.html#handling-session-initialization-and-destruction
[63] https://vaadin.com/directory/component/vaadin-cdi/
[64] https://vaadin.com/start/latest
[65] https://github.com/vaadin/cdi

13. Progressive Web Applications (PWA)

13.1. Introduction

Progressive Web Applications (PWAs) combine new technologies with established best practices. They allow you to create reliable, accessible, and engaging experiences that give users a native-like experience, with a user friendly opt-in installation flow.

NOTE This documentation covers how to create PWAs with Vaadin. For detailed generic PWA information, see the Vaadin PWA Handbook[66].

13.1.1. Basic PWA Concepts

All PWAs have the following common basic features that enable native-app-like behavior:

- **Web App Manifest**: This provides information about an application, for example its name, theme, icon, and description. These details are needed to make an installable version of web application.
- **Service Worker**: This is a type of web worker. Essentially, it is a a JavaScript file that:
  - Runs separately from the main browser thread.
  - Intercepts network requests.
  - Caches and retrieves resources from the cache.
  - Delivers Push messages.

The ability to intercept network requests, makes it possible to serve files directly from cache and create a full application experience, even when no network is available.

13.1.2. Application Installation Requirements

To support installation on devices, the following additional features are required. These depend on the device and browser used:

- **Icons**: Different icon sizes are needed to support different devices. To enhance the experience, splash screen images are also required.
- **Offline support**: The service worker must be able respond to serve the client if a network is not available.
- **Header information**: The application must include browser and/or device-specific theming and icon data in the header. This is in addition to the manifest file,
- **Installation prompts**: Special handling information is required for Google Chrome.

NOTE Many new browser features, including those required for PWAs, require HTTPS. Even if your PWA currently works without HTTPS in some environments (for example, Android), this is likely to change and it is probable that PWAs that do not support HTTPs will malfunction in the future.

13.2. Creating PWAs with Vaadin

The Vaadin server automatically serves the needed resources for a PWA, when you use the `@PWA` annotation in the root layout of your application.

Example: Using the `@PWA` annotation with the `@Route` annotation to automatically serve PWA resources.

```
@PWA(name = "My Progressive Web Application",
     shortName = "MyPWA")
@Route("")
public class MyPWA extends Div {
    public MyPWA() {
        setText("Welcome to my PWA");
    }
}
```

- Vaadin server automatically serves the web manifest, service worker, icons, offline page, and installation prompt, and adds the necessary additions to the application headers.
- The `shortName` parameter should not exceed 12 characters. See PWA Web App Manifest for a list of `@PWA` annotation parameters you can use.

NOTE

You can only have one `@PWA` annotation per application. The annotation must be placed in the application's parent layout, or in a view annotated with `@Route`.

13.3. PWA Application Icons

PWAs need at least three icons: a favicon for the browser page, a device icon for the for the installed application, and an icon used on the splash screen of the installed application.

13.3.1. Using a Custom Icon

Vaadin uses and serves default PWA icons automatically, but you can use a custom icon.

To use a custom icon image:

1. Create an icon image named `icon.png`. The icon must be in PNG format.
2. Add the image to your `src/main/webapp/icons/` folder.

Vaadin automatically scans for an image named `icon.png` in the **`/icons`** folder in the `webapp` resources folder. It uses this image to create appropriately-sized images for different devices. If no icon is found, the default image is used as a fallback.

To ensure that all resized images are attractive, use an image of at least 512 x 512 pixels. This is large enough to only be downscaled, as upscaling can cause pixelation.

13.3.2. Overriding Generated Icons

All generated images are named using the `icon-`*`[width]x[height]`*`.png` notation, for example `icon-1125x2436.png`.

To override any generated image:

1. Create an image of the size you want to override and name in using the notation mentioned above. For example, `icon-1125x2436.png` for a custom hi-res splash screen image for iOS devices.
2. Add the image to your `src/main/webapp/icons/` folder.

13.3.3. Renaming Icons

You can change the default icon path to a custom path, using the `iconPath` parameter in the `@PWA` annotation.

Example: Defining a custom path using the `iconPath` parameter in the `@PWA` annotation.

```
@PWA(name = "My Progressive Web Application",
     shortName = "MyPWA",
     iconPath = "img/icons/logo.png")
```

- Icon images will now be:
  - Named using the value in the `iconPath*` parameter. For example the 512 x 512 pixel image would be `img/icons/logo-512x512.png`.
  - Stored in the `src/main/webapp/img/icons/` folder.

13.4. PWA Web App Manifest

When the `@PWA` annotation is found, Vaadin automatically generates a web app manifest file, named `manifest.webmanifest`.

Here is a list of properties in the file that you can customize. With the exception of `scope`, all properties can be set in the `@PWA` annotation.

- `name`: The name of the application. Set this property in the `name` parameter in the `@PWA` annotation.
- `short_name`: The short name of the application. This should not exceed 12 characters. It is used on the device home screen, where there is a limited amount of space. Set this property in the `shortName` parameter in the `@PWA`

annotation.

- `description`: The description of the application. The default value is an empty string. Set this property in the `description` parameter in the `@PWA` annotation.
- `display`: Defines the preferred display mode for the application. The default value is `standalone`. Set this property in the `display` parameter in the `@PWA` annotation.
- `background_color`: The background color of the application. The default value is `#f2f2f2` (gray). Set this property in the `backgroundColor` parameter in the `@PWA` annotation.
- `theme_color`: The theme color of application. The default value is `#ffffff` (white). Set this property in the `backgroundColor` parameter in the `@PWA` annotation.
- `scope`: Defines the navigation scope of the website's context. This restricts the web pages that can be viewed while the manifest is applied. The value is set to the context path of application. You cannot change this property in the `@PWA` annotation.
- `start_url`: The start URL that is navigated to when the application is launched from the installed app (home screen). The default value is an empty string `""` that points to the default route target for the application (marked with `@Route("")`). Set this property in the `startPath` parameter in the `@PWA` annotation.
- `icons`: Automatically created from icon resources.

NOTE For more information about these properties, see Web App Manifest[67] in the MDN web docs.

13.4.1. Renaming the Manifest

You can change the default name (`manifest.webmanifest`) of the web app manifest, using the `manifestPath` parameter in the `@PWA` annotation.

Example: Setting the `manifestPath` parameter in the `@PWA` annotation.

```
@PWA(name = "My Progressive Web Application",
     shortName = "MyPWA",
     manifestPath = "manifest.json")
```

13.4.2. Overriding the Generated Manifest

You can override the generated manifest file with a custom manifest.

To override the generated web app manifest file:

1. Create a custom manifest file and name it to match the file name set in the `manifestPath` parameter in the `@PWA` annotation, for example `manifest.webmanifest`.
2. Add the file to your `src/main/webapp/` folder.

13.5. PWA Service Worker

When the `@PWA` annotation is found, Vaadin automatically generates a simple service worker during application startup.

The generated service worker:

- Caches offline resources, including the offline fallback page, icons, and custom (user-defined) offline resources.

- Handles simple offline navigation by serving the offline page.

NOTE The service worker can only respond to full navigation events, like refresh or direct navigation to a URL.

The service worker uses Google Workbox[68] to cache resources. The necessary Workbox files are stored in the `VAADIN/resources/workbox/` folder.

13.5.1. Defining Custom Cache Resources

You can define custom resources to be cached automatically by the service worker, using the `offlineResources` parameter in the `@PWA` annotation.

Example: Defining `styles/offline.css`, `img/offline.jpg` and `js/jquery.js` as cacheable offline resources.

```
@PWA(name = "My Progressive Web Application",
     shortName = "MyPWA",
     offlineResources = { "styles/offline.css",
         "js/jquery.js", "img/offline.jpg" })
```

13.5.2. Overriding the Generated Service Worker

You can override the generated service worker with a custom service worker.

To override the generated service worker file:

1. Create a custom service worker file and name it `sw.js`.

2. Add the file to your `src/main/webapp/` folder.

NOTE To ensure that your custom service worker deals with offline support and resource caching properly, you can copy the default service worker from browser resources and use it as a template.

13.6. PWA Offline Page

Vaadin automatically generates and serves an offline page. This is a simple page that:

- Includes the application name and icon.
- Communicates to the user that the application is offline, because there is no network connection.

13.6.1. Creating a Custom Offline Page

To override the default offline page:

1. Create a file named `offline.html`.
2. Add the file to your `src/main/webapp/` folder.

You can change the name of the offline page file using the `offlinePath` parameter in the `@PWA` annotation.

The offline page can only use resources found in the cache. By default, only the offline page, manifest, and icons are cached. If your page needs additional resources (such as CSS, images, Web Components), you can define them using the `offlineResources` parameter in the `@PWA` annotation. See Defining Custom Cache Resources for more.

----

[66] https://vaadin.com/pwa
[67] https://developer.mozilla.org/en-US/docs/Web/Manifest
[68] https://developers.google.com/web/tools/workbox/

14. Manipulating DOM with Element API

14.1. Element Properties and Attributes

The Element API contains methods to update and query parts of an element.

You can use the Element API to change property and attribute values for server-side elements.

NOTE By default, values updated in the browser are not sent to the server. See Retrieving User Input for how to transfer data to the server.

14.1.1. About Attributes

Attributes are used mainly for the initial configuration of elements.

Attribute values are always stored as strings.

Example: Setting attributes for the `nameField` element.

```
Element nameField = ElementFactory.createInput();
nameField.setAttribute("id", "nameField");
nameField.setAttribute("placeholder", "John Doe");
nameField.setAttribute("autofocus", "");
```

Example: The same example as above expressed as HTML.

```
<input id="nameField" placeholder="John Doe" autofocus>
```

You can also retrieve and manipulate attributes after they have been set.

Example: Retrieving and changing attributes in the `nameField` element.

```
// "John Doe"
String placeholder = nameField
        .getAttribute("placeholder");

// true
nameField.hasAttribute("autofocus");

nameField.removeAttribute("autofocus");

// ["id", "placeholder"]
nameField.getAttributeNames().toArray();
```

14.1.2. About Properties

Properties are used mainly to dynamically change the settings of an element after it has been initialized.

Any JavaScript value can be used as a property value in the browser.

You can use different variations of the `setProperty` method to set a property value as a `String`, `boolean`, `double` or `JsonValue`.

Example: Setting a property value as a `double`.

```
Element element = ElementFactory.createInput();
element.setProperty("value", "42.2");
```

Similarly, you can use different variations of the `getProperty` method to retrieve the value of a property as a `String`, `boolean`, `double` or `JsonValue`.

If you retrieve the value of a property as a different type to that as which it was set, JavaScript type coercion rules are used to convert the value. For example, a property set as a non-empty `String` results as `true` if fetched as a `boolean`.

Example: Converting retrieved value types.

```
// true, since any non-empty string is
// true in JavaScript
boolean helloBoolean =
        element.getProperty("value", true);

// 42, string is parsed to a JS number and
// truncated to an int
int helloInt = element.getProperty("value", 0);
```

14.1.3. Using Attributes Vs. Properties

Be cautious when using attributes and properties:

- In many cases it is possible to use either an attribute or property with the same name for the same effect, and both work fine.
- However, in certain cases:
  - Only one or the other works, or
  - The attribute is considered only when the element is initialized, and the property is effective after initialization.

You should always check the specific documentation for the element you're using to find out whether a feature should be configured using a property or an attribute.

14.1.4. Using the textContent Property

You can set an element's `textContent` property using the `setText` method. This removes all children of the element and replaces them with a single text node with the given value.

The `ElementFactory` interface provides helpers that you can use to create an element with a given text content.

Example: Using the `createSpan` and `createDiv` helper methods with the `setText` method.

```
// <div>Hello world</div>
Element element = ElementFactory
        .createDiv("Hello world");

// <div>Hello world<span></span></div>
element.appendChild(ElementFactory.createSpan());

// <div>Replacement text</div>
element.setText("Replacement text");
```

To retrieve the text of an element, you can use the:

- `getText` method to return the text in the element itself. Text in child elements is ignored.
- `getTextRecursively` method to return the text of the entire element tree, by recursively concatenating the text from all child elements.

Example: Using the `getText` and `getTextRecursively` methods.

```
element.setText("Welcome back ");

Element name = ElementFactory
        .createStrong("Rudolph Reindeer");
// <div>Welcome back <strong>Rudolph
// Reindeer</strong></div>
element.appendChild(name);

// will return "Welcome back Rudolph Reindeer"
element.getTextRecursively();
// will return "Welcome back "
element.getText();
```

14.2. Listening to User Events Using the Element API

The Element API provides the `addEventListener` method that you can use to listen to any browser event.

Example: Using the `addEventListener` method to create a `click` event.

```
Element helloButton = ElementFactory
        .createButton("Say hello");
helloButton.addEventListener("click", e -> {
    Element response = ElementFactory
            .createDiv("Hello!");
    getElement().appendChild(response);
});
getElement().appendChild(helloButton);
```

- Clicking the "Say hello" button in the browser sends the event to the server for processing, and a new `<div>Hello!</div>` element is added to the DOM.

14.2.1. Accessing Data from Events

You can get more information about the element or user interaction by defining the required event data on the `DomListenerRegistration` returned by the `addEventListener` method.

Example: Using the `addEventData` method to define the required event data.

```
helloButton.addEventListener("click", this::handleClick)
    .addEventData("event.shiftKey")
    .addEventData("element.offsetWidth");

private void handleClick(DomEvent event) {
    JsonObject eventData = event.getEventData();
    boolean shiftKey = eventData
            .getBoolean("event.shiftKey");
    double width = eventData
            .getNumber("element.offsetWidth");

    String text = "Shift " + (shiftKey ? "down" : "up");
    text += " on button whose width is " + width + "px";

    Element response = ElementFactory.createDiv(text);
    getElement().appendChild(response);
}
```

- The requested event data values are sent as a JSON object from the client.
- You can retrieve the event data using the `event.getEventData()` method in the listener.
- Make sure that you use the:
  - Correct getter based on the data type.
  - Same keys that were provided as parameters in the `addEventData` method.

TIP The `filter` and `debounce` settings in the `@DomEvent` annotation can also be set through the `DomListenerRegistration` object. See Using Events with Components[69] for more.

14.3. Remote Procedure Calls

Remote Procedure Calls (RPCs) are a way to execute procedures or subroutines in a different address space, typically on another machine.

Vaadin Flow handles server-client communication by allowing RPC calls from the server to the client, and *vice versa*.

14.3.1. Calling Client-side Methods from the Server

You can execute client-side methods from the server by accessing the `Element` API.

callJsFunction Method

The `callJsFunction` method allows you to execute a client-side component function from the server side. The method accepts two parameters: the name of the function to call, and the arguments to pass to the function.

The arguments passed to the function must be of a type supported by the communication mechanism. The supported types are `String`, `Boolean`, `Integer`, `Double`, `JsonValue`, `Element`, and `Component`.

Example: Using the `callJsFunction` method to execute the clearSelection function.

```
public void clearSelection() {
    getElement().callJsFunction("clearSelection");
}

public void setExpanded(Component component) {
    getElement().callJsFunction("expand",
            component.getElement());
}
```

executeJs Method

You can also use the executeJs method to execute JavaScript asynchronously from the server side. You can use this method in addition to the callJsFunction method.

The executeJs method accepts two parameters: the JavaScript expression to invoke, and the parameters to pass to the expression. Note that the given parameters are available as variables named $0, $1, and so on.

The arguments passed to the expression must be of a type supported by the communication mechanism. The supported types are String, Integer, Double, Boolean and Element.

Example: Using the executeJs method.

```
public void complete() {
    getElement().executeJs("this.complete($0)", true);
}
```

It is also possible to call the executeJs method to access methods and fields of a Web Component.

Return values

The return value from the JavaScript function called using `callJsFunction` or the value from a `return` statement in an `executeJs` expression can be accessed by adding a listener to the `PendingJavaScriptResult` instance returned from either method.

Example: Check if the browser supports Constructable Stylesheets.

```
public void checkConstructableStylesheets() {
    getElement().executeJs(
            "return 'adoptedStyleSheets' in document")
            .then(Boolean.class, supported -> {
                if (supported) {
                    System.out.println(
                            "Feature is supported");
                } else {
                    System.out.println(
                            "Feature is not supported");
                }
            });
}
```

TIP If the return value is a JavaScript `Promise`, then a return value will be sent to the server only when the `Promise` is resolved.

14.3.2. Calling Server-side Methods from the Client

You can call a server-side method from the client side using either the `@EventHandler` or `@ClientCallable` annotation.

@EventHandler Annotation

The `@EventHandler` annotation allows you to register a server-side method as an event handler. It publishes the annotated method and allows it to be invoked from the client side as a template event handler. See Handling User Events in a PolymerTemplate[70] for more.

@ClientCallable annotation

The `@ClientCallable` annotation allows you to invoke a server-side method from the client side. It marks a template method as a method that can be called from the client side using the `this.$server.serverMethodName(args)` notation.

You can use it anywhere in your client-side Polymer class implementation, and can pass your own arguments in the method. Note that the types should match the method declaration on the server side.

Example: Using `this.$server.clickHandler()` to mark a template method.

```
this.$server.clickHandler()
```

Example: Using the `@ClientCallable` annotation on the server side.

```
    @ClientCallable
    public void clickHandler() {
        // do your server side action here
    }
```

IMPORTANT

Property changes, DOM events, event-handler methods (methods annotated with `@EventHandler`) and client-delegate methods (methods annotated with `@ClientCallable`) are blocked for disabled components.

14.4. Retrieving User Input Using the Element API

In this section we demonstrate how to use the Element API to retrieve user input. Our example adds a text input field that allows the user to enter their name.

1. Create a text input element.

   Example: Creating a `textInput` element with a `placeholder` attribute.

   ```
   Element textInput = ElementFactory.createInput();
   textInput.setAttribute("placeholder",
           "Please enter your name");
   ```

2. Transfer the value to the server, by asking the client to update the server-side input element every time the value changes in the browser.

   Example: Using the `synchronizeProperty` method to update the value of the text input element.

   ```
   textInput.synchronizeProperty("value", "change");
   ```

   - Configures Flow to synchronize the `value` property to the server-side when a `change` event occurs.

NOTE

As an alternative, you can use the `addEventData` method to transfer the value from the input to the server. See Using the Element API to Listen to User Events for more.

3. Retrieve the synchronized properties using the `Element.getProperty` API.

   Example: Using the `textInput.getProperty("value")` method to retrieve the property value.

```
button.addEventListener("click", e -> {
    String responseText = "Hello " +
            textInput.getProperty("value");
    Element response = ElementFactory
            .createDiv(responseText);
    getElement().appendChild(response);
});
```

NOTE

The `value` property of the `TextInput` element returns `null` if the property was not previously set and the user has not typed text into the field.

14.5. Dynamic Styling Using the Element API

You can use the Element API to style elements using dynamic class names or inline styles.

The Element API includes two methods that facilitate styling,

- `getClassList()`: Gets the set of CSS class names used for the element.

  `getStyle()`: Gets the style instance to manage element

inline styles.
-

14.5.1. Using classLists and classNames

You can use the `getClassList` method to get a collection of CSS class names used for the element. The returned set can be modified to add or remove class names.

Example: CSS style rules.

```
.blue {
  background: blue;
  color: white;
}
```

Example: Using the `getClassList()` method to dynamically modify the class names of an element.

```
button.setText("Change to blue");
button.addEventListener("click",
    e -> button.getClassList().add("blue"));
```

Example: Using the `getClassList` method to add and remove classes.

```
element.getClassList().add("error");
element.getClassList().add("critical");
element.getClassList().remove("primary");

// will return "error critical".
element.getProperty("className");
```

- The `element.getProperty("className")` method gets a set of all classes as a concatenated string.

You cannot modify `classList` or `className` properties directly using the `setProperty` methods.

You can set and get an element's `class` attribute using:

- `element.setAttribute("class", "foo bar");`: This clears any previously set `classList` property.
- `element.getAttribute('class')`: This returns the contents of the `classList` property as a single concatenated string.

14.5.2. Using the Style Object

You can set and remove inline styles for an element using the `Style` object returned by the `element.getStyle()` method. Style property names can be formatted in Camel case, for example `backgroundColor`, or Kebab case, for example `background-color`.

Example: Using the `getStyle()` method for dynamic inline styling.

```
Element input = ElementFactory.createInput();
button.setText("Change to the entered value");
button.addEventListener("click",
    e -> button.getStyle().set("background",
            input.getProperty("value")));
```

Example: Setting and removing style objects using the `element.getStyle()` method.

```
element.getStyle().set("color", "red");
//camelCase
element.getStyle().set("fontWeight", "bold");
//kebab-case
element.getStyle().set("font-weight", "bold");

//camelCase
element.getStyle().remove("backgroundColor");
//kebab-case
element.getStyle().remove("background-color");

element.getStyle().has("cursor");
```

14.6. Using the Shadow Root in Server-side Elements

The Element API supports adding a shadow root to element types that support this. This allows you to create server-side Web Components.

You can use the `element.attachShadow()` method to add a shadow root.

Example: Using the `element.attachShadow` method to add a shadow root node.

```
Element element = new Element("custom-element");
ShadowRoot shadowRoot = element.attachShadow();
```

Note:

- A `ShadowRoot` is not an actual element. Its purpose is to support handling of child elements and getting the host element that contains the shadow root.
- Elements added to a `ShadowRoot` parent are only visible if

the `ShadowRoot` contains a `<slot></slot>` element. See Server-side components in Polymer 2 templates[71] for more.

To ensure that new elements are encapsulated in the shadow tree of the hosting element, you should add all new elements to the `ShadowRoot` element.

Example: Adding an element to the `ShadowRoot`.

```
@Tag("my-label")
public class MyLabel extends Component {

    public MyLabel() {
        ShadowRoot shadowRoot = getElement()
                .attachShadow();
        Label textLabel = new Label("In the shadow");
        shadowRoot.appendChild(textLabel.getElement());
    }
}
```

14.6.1. Elements That Do Not Support a Shadow Root

The DOM specification[72] defines a list of elements that can't host a shadow tree. Typical reasons for this include:

- The browser already hosts its own internal shadow DOM for the element, for example. `<textarea>` and `<input>`.
- It doesn't make sense for the element to host a shadow DOM, for example `<img>`.

----

[69] https://vaadin.com/docs/flow/creating-components/tutorial-component-events.html
[70] https://vaadin.com/docs/flow/polymer-templates/tutorial-template-event-handlers.html

[71] https://vaadin.com/docs/flow/polymer-templates/tutorial-template-components-in-slot.html
[72] https://dom.spec.whatwg.org/#dom-element-attachshadow

15. Creating Components

15.1. Creating Components Overview

There are many ways to create components in Vaadin Flow .

In this part of the documentation, we demonstrate the numerous ways to create components:

- Creating a Simple Component Using the Element API
- Creating a Component with Multiple Elements
- Creating a Component Using Existing Components
- Creating a Component Container
- Extending Components

NOTE You can also create components using Polymer templates. See Creating a Simple Component Using the Template API[73] for more.

We also cover cover these topics that apply generally when creating components:

- Using API Helpers to Define Component Properties
- Using Events with Components
- Using Component Lifecycle Callbacks
- Implementing Vaadin Mixin Interfaces

15.2. Creating a Simple Component Using the Element API

In this section, we demonstrate how to create a simple component using the `Element` API and a single DOM element.

Example: Creating a `TextField` component based on an `<input>` element.

```
@Tag("input")
public class TextField extends Component {

    public TextField(String value) {
        getElement().setProperty("value",value);
    }
}
```

- The root element is:
  - Created automatically (by the `Component` class) based on the `@Tag` annotation.
  - Accessed using the `getElement()` method.
  - Used to set the initial value of the field.

TIP You can use predefined constants in the `@Tag` annotation. For example, the `@Tag("input")` annotation is equivalent to `@Tag(Tag.INPUT)`. There are constants for most, but not all, tag names.

15.2.1. Adding an API

To make the component easier to use, you can add an API to get and set the value.

Example: Adding an API using the `@Synchronize` annotation.

```
@Synchronize("change")
public String getValue() {
    return getElement().getProperty("value");
}
public void setValue(String value) {
    getElement().setProperty("value", value);
}
```

- Adding the `@Synchronize` annotation to the getter ensures that the browser sends property changes to the server.
- The annotation defines the name of the DOM event that triggers synchronization, in this case a `change` event.
- Changes to the input element cause the updated `value` property (deduced from the getter name) to be sent to the server.

TIP The `@Synchronize` annotation can specify multiple events and override the name of the property, if necessary.

NOTE The `@Synchronize` annotation only maps events that originate from the root element, or are bubbled to the root element. For example, if you have an `<input>` element inside a `<div>` element, `@Synchronize` only maps events from the `<div>` element.

See Using API Helpers to Define Component Properties for an alternative, and simpler, way to address properties and attributes.

15.2.2. Overriding Default Disabled Behavior

The `setEnabled` method is available for all components that implement the `HasEnabled` interface.

NOTE The `setEnabled` method is also available for all components that implement the `HasValue`, `HasComponents` or `Focusable` interfaces.

By default, disabling a component adds a `disabled` property to the client element. You can modify this by overriding the `Component:onEnabledStateChanged(boolean)` method.

Example: Overriding the default disabled behavior to ensure items are updated in a component requiring a custom disabled marking.

```
@Override
public void onEnabledStateChanged(boolean enabled) {
    setDisabled(!enabled);
    refreshButtons();
}
```

15.3. Creating a Component with Multiple Elements

In this section we demonstrate how to create a component using the `Element` API and multiple DOM elements. We create a `TextField` component that supports a label.

Example: DOM structure of the component.

```
<div>
    <label></label>
    <input>
</div>
```

Example: TextField component with <input> and <label> elements

```
@Tag("div")
public class TextField extends Component {

    Element labelElement = new Element("label");
    Element inputElement = new Element("input");

    public TextField() {
        inputElement
               .synchronizeProperty("value", "change");
        getElement()
               .appendChild(labelElement, inputElement);
    }
}
```

- The DOM structure is set up by marking the root element as a <div> in the @Tag annotation.
- The label and input elements are appended to the root element.
- Value synchronization is set up using the input element.

See Creating a Simple Component Using the Element API for an alternative way to synchronize.

15.3.1. Adding an API

To make the component easier to use, you can add an API to set the input value and label text.

Example: Adding an API to get and set values for the input and label elements.

```
public String getLabel() {
    return labelElement.getText();
}

public String getValue() {
    return inputElement.getProperty("value");
}

public void setLabel(String label) {
    labelElement.setText(label);
}

public void setValue(String value) {
  inputElement.setProperty("value", value);
}
```

15.4. Using API Helpers to Define Component Properties

The `PropertyDescriptor` interface (and associated `PropertyDescriptors` helper class) simplifies managing attributes and properties in a component.

You can use `PropertyDescriptors` to define a property name and default value in a single place, and then use the descriptor from the setter and getter methods.

Example: Using the `PropertyDescriptors.propertyWithDefault` method to define the default property value.

```java
@Tag("input")
public class TextField extends Component {
    private static PropertyDescriptor<String, String>
        VALUE = PropertyDescriptors
                .propertyWithDefault("value", "");

    public String getValue() {
        return get(VALUE);
    }
    public void setValue(String value) {
        set(VALUE, value);
    }
}
```

For your component API for a given property, for example the `value` of an input field, to function correctly:

- The getter and setter should use the same property or attribute.
- The default value should be handled correctly.
- The getter return value should be either:
  - The type used by the setter, for example `String` for an input `value`, or
  - An optional version of the type used by the setter, that is `Optional<String>` if the property is not mandatory.

`PropertyDescriptors` automatically take the above into consideration.

15.4.1. PropertyDescriptor Interface

`PropertyDescriptor` instances are created using the helper methods available in the `PropertyDescriptors` class.

Different helper methods, depending on how you want your component to work, are available:

- `PropertyDescriptors.propertyWithDefault` maps to an element property with a given default value.
- `PropertyDescriptors.attributeWithDefault` maps to an element attribute with a given default value.
- `PropertyDescriptors.optionalAttributeWithDefault` maps to an element attribute with a given default value, but returns an empty `Optional` when the default value is set.

Example: Using `PropertyDescriptors.optionalAttributeWithDefault` method for a non-mandatory `placeholder` in a `TextField`.

```
@Tag("input")
public class TextField extends Component {
  private static PropertyDescriptor<String,
    Optional<String>> PLACEHOLDER = PropertyDescriptors
       .optionalAttributeWithDefault("placeholder", "");

  public Optional<String> getPlaceholder() {
    return get(PLACEHOLDER);
  }
  public void setPlaceholder(String placeholder) {
      set(PLACEHOLDER, placeholder);
  }
}
```

NOTE The default value used in all `PropertyDescriptors` methods should match the value in the browser when the attribute or property is NOT set. Otherwise, when the user sets the value to the default value, the value will not be correctly sent to the browser.

15.5. Creating a Component Using Existing Components

In this section we demonstrate how to create a `Composite` component using existing components.

We create a `TextField` component by combining existing `Div`, `Label` and `Input` HTML components into this hierarchy:

- `Div`
  - `Label`
  - `Input`

NOTE Creating the component based on a `Composite` is the best practice in these circumstances. While it is possible to create a new component by extending the `Div` HTML component, this is not advisable, because it unnecessarily exposes `Div` API methods, such as `add(Component)`, to the user.

Example: Creating a `TextField` component by extending `Composite<Div>`.

```
public class TextField extends Composite<Div> {

    private Label label;
    private Input input;

    public TextField(String labelText, String value) {
        label = new Label();
        label.setText(labelText);
        input = new Input();
        input.setValue(value);

        getContent().add(label, input);
    }
}
```

- The `Composite` automatically creates the root component (specified using generics (`Composite<Div>`)).
- The root component is available through the `getContent()` method.
- In the constructor, it is only necessary to create the child components and add them to the root `Div`.
- The value is set using the `setValue` method in the `Input` component.

15.5.1. Adding an API

To make the component easier to use, you can add an API to get and set the value and label text, by delegating to the `Input` and `Label` components.

Example: Adding an API to get and set the value and label.

```
public String getValue() {
    return input.getValue();
}
public void setValue(String value) {
    input.setValue(value);
}

public String getLabel() {
    return label.getText();
}
public void setLabel(String labelText) {
    label.setText(labelText);
}
```

- The public API only exposes the defined methods, and a few generic methods defined in the `Component` interface.

TIP Neither using a `Component` (instead of an `Element`), nor using a `Composite`, introduces additional overhead in the browser.

15.6. Extending Components

You can create a new component by extending any existing component.

For most components, there is a client-side component and a corresponding server-side component:

- **Client-side component**: Contains the HTML, CSS, and JavaScript, and defines a set of properties that determine the behavior of the component on the client side.
- **Server-side component**: Contains Java code that allows for changing of the client-side properties, and manages the component behavior on the server side.

You can extend a component on either the server or client side. Note that these are alternative approaches that are mutually exclusive.

In this section we demonstrate the two different approaches to achieve the **same changes** to the prebuilt text field component.

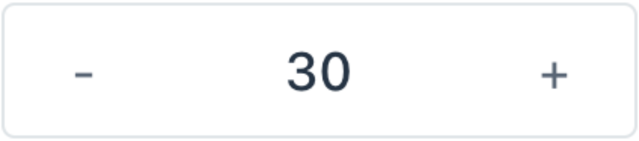
15.6.1. Extending a Component Using the Server-side Approach

Extending a server-side component is useful when you want to add new functionality (as opposed to visual aspects) to an existing component. Suitable examples include automatically processing data, adding default validators, and combining multiple simple components into a field that manages complex data.

TIP

If your component contains a lot of logic that could easily be done on the client side, consider implementing it as a Web Component and creating a wrapper for it. This approach may offer a better user experience and result in less load on the server.

In this example, we create a `NumericField` component by extending the `TextField` component. The new component contains a default number that the user can change using **+** and **-** controls.



Example: Creating a `NumericField` component by extending the `TextField` component.

```
public class NumericField extends TextField {

    private Button substractBtn;
    private Button addBtn;

    private static final int DEFAULT_VALUE = 0;
    private static final int DEFAULT_INCREMENT = 1;

    private int numericValue;
    private int incrementValue;
    private int decrementValue;

    public NumericField() {
        this(DEFAULT_VALUE, DEFAULT_INCREMENT,
                -DEFAULT_INCREMENT);
    }

    public NumericField(int value, int incrementValue,
                        int decrementValue) {
        setNumericValue(value);
        this.incrementValue = incrementValue;
        this.decrementValue = decrementValue;

        setPattern("-?[0-9]*");
        setPreventInvalidInput(true);

        addChangeListener(event -> {
            String text = event.getSource().getValue();
            if (StringUtils.isNumeric(text)) {
                setNumericValue(Integer.parseInt(text));
            } else {
                setNumericValue(DEFAULT_VALUE);
            }
        });

        substractBtn = new Button("-", event -> {
            setNumericValue(numericValue +
                    decrementValue);
        });

        addBtn = new Button("+", event -> {
            setNumericValue(numericValue +
                    incrementValue);
        });

        styleBtns();

        addToPrefix(substractBtn);
        addToSuffix(addBtn);
    }

    private void styleBtns() {
        // Note: The same as addThemeVariants
        substractBtn.getElement()
                .setAttribute("theme", "icon");
        addBtn.getElement()
                .setAttribute("theme", "icon");
    }
    public void setNumericValue(int value) {
        numericValue = value;
        setValue(value + "");
    }

    // getters and setters
}
```

NOTE

As an alternative, you can extend the `Composite` class that has a minimal API. This hides methods available in the more extensive API that is exposed when your custom components extends an implementation of `Component`.

NOTE

The `Element` API contains methods to update and query various parts of the element, such as the attributes. Every component has a `getElement()` method that allows you to to access it. See Creating a Component Using Multiple Elements for more.

It may also be necessary to add CSS styles for the new component.

Example: Creating `vaadin-numeric-field-theme.html` to customize the appearance of the `vaadin-text-field` component.

```
const documentContainer = document
        .createElement('template');

documentContainer.innerHTML =
`<dom-module id="vaadin-numeric-field-theme"
             theme-for="vaadin-text-field">
    <template>
        <style>
            :host(vaadin-text-field)
                    [part="input-field"] {
                background-color: #FFFFFF;
                border: solid 1px #E0E5E8;
                box-sizing: border-box;
            }

            :host(vaadin-text-field) [part="value"]{
                --_lumo-text-field-overflow-mask-image:
                        none;
                text-align:center;
            }

            :host(vaadin-text-field)
                    [part="input-field"]
                    ::slotted(vaadin-button) {
               background-color: transparent !important;
            }
        </style>
    </template>
</dom-module>`;

document.head.appendChild(documentContainer.content);
```

NOTE Remember to import the new styles in the view in which the component is used, by adding `@JsModule("./styles/vaadin-numeric-field-theme.js")`.

See Integrating Your Own Component Theme for more.

15.6.2. Extending a Component Using the Client-side Approach

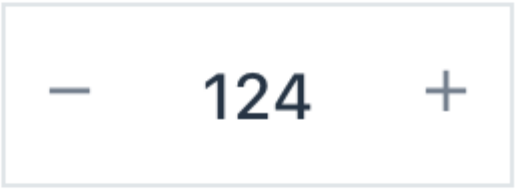
Vaadin client-side components are based on Polymer 3[74] that supports extending existing components. You can use the `extends` property to extend existing Polymer elements.

There are five ways to inherit a template from another Polymer element:

1. Inheriting a base class template without modifying it.
2. Overriding a base class template in a child class.
3. Modifying a copy of a superclass template.
4. Extending a base class template in a child class.
5. Providing template-extension points in a base class for content from a child class.

Extending by Modifying a Copy of a Superclass Template

In this example, we demonstrate how to create a new component by modifying a copy of a superclass template. We build a `NumberFieldElement` by extending `Vaadin.TextFieldElement`. The new component contains a default number that the user can change using **+** and **-** controls.



It is important to remember that when a component template is extended, the properties and methods of the parent template become available to the child template.

NOTE

By default, a child component uses the template of the parent component, unless the child component provides its own template by overriding the static getter method `template`. The parent's template is accessed using `super.template`.

Next, specify the element from which the child component inherits. In this case we specify that `NumberFieldElement` inherits (including the properties and methods) from `Vaadin.TextFieldElement`:

```
import {html} from
   '@polymer/polymer/lib/utils/html-tag.js';
import {TextFieldElement} from
   '@vaadin/vaadin-text-field/src/vaadin-text-field.js';

let memoizedTemplate;

class NumberFieldElement extends TextFieldElement {

    static get template() {
        if (!memoizedTemplate) {
            const superTemplate = super.template
                    .cloneNode(true);
            const inputField = superTemplate.content
                .querySelector('[part="input-field"]');
            const prefixSlot = superTemplate.content
            .querySelector('[name="prefix"]');
            const decreaseButton = html`<div
                part="decrease-button"
                on-click="_decreaseValue"></div>`;
            const increaseButton = html`<div
                part="increase-button"
                on-click="_increaseValue"></div>`;
            inputField.insertBefore(
                decreaseButton.content, prefixSlot);
            inputField.appendChild(
                increaseButton.content);
            memoizedTemplate = html`<style>
                 [part="decrease-button"]::before {
                   content: "-";
                 }

                 [part="increase-button"]::before {
                   content: "+";
                 }
               </style>
               ${superTemplate}`;
        }
        return memoizedTemplate;
    }

    static get is() {
        return 'vaadin-number-field';
    }

    static get properties() {
        return {
            decrementValue: {
              type: Number,
              value: -1,
              reflectToAttribue: true,
              observer: '_decrementChanged'
            },
            incrementValue: {
              type: Number,
              value: 1,
              reflectToAttribue: true,
              observer: '_incrementChanged'
            }

            // Note: the value is stored in the
            // TF's value property.
        };
    }

    _decreaseValue() {
        this.__add(this.decrementValue);
    }

    _increaseValue() {
        this.__add(this.incrementValue);
    }

    __add(value) {
        this.value = parseInt(this.value, 10) + value;
        this.dispatchEvent(
            new CustomEvent('change', {bubbles: true}));
    }

    _valueChanged(newVal, oldVal) {
        this.value = this.focusElement.value;
        super._valueChanged(this.value, oldVal);
    }

    /* ... */
}
```

To modify the template we override the `template` static getter. Note that the expression `${super.template}` inserts the base class template into the newly constructed template. The newly constructed template is memoized for further invocations of `template`.

See Inherit a template from another Polymer element[75] in the Polymer documentation for more.

15.7. Using Events with Components

Your component class can provide an event-handling API that uses the event bus provided by the `Component` base class.

The event bus supports:

- Event classes that extend `ComponentEvent`, and
- Listeners of the type `ComponentEventListener<EventType>`.

15.7.1. Defining an Event

To use the event bus, your event should extend `ComponentEvent`. The base type is parameterized with the type of the component firing the event. This means that the `getSource()` method automatically returns the correct component type.

The second constructor parameter determines whether the event is triggered by a DOM event in the browser or through the component's server-side API.

Example: Creating an event by extending `ComponentEvent`.

```
public class ChangeEvent
        extends ComponentEvent<TextField> {
    public ChangeEvent(TextField source,
                            boolean fromClient) {
        super(source, fromClient);
    }
}
```

15.7.2. Defining an Event Listener

Event listeners are of the generic `ComponentEventListener<EventType>` type, so it is not necessary to create a separate interface for your event type.

In addition, the method for adding a listener returns a handle that can be used to remove the listener, so it is unnecessary to implement a separate method to remove an event listener.

Example: Using the `addChangeListener` method to add an event listener.

```
@Tag("input")
public class TextField extends Component {
    public Registration addChangeListener(
         ComponentEventListener<ChangeEvent> listener) {
        return addListener(ChangeEvent.class, listener);
    }

    // Other component methods omitted
}
```

Example: Adding and removing event listeners.

```
TextField textField = new TextField();
Registration registration = textField
        .addChangeListener(e ->
                System.out.println("Event fired"));

// In some other part of the code
registration.remove();
```

15.7.3. Firing Events from the Server

You can fire an event on the server by creating the event instance and passing it to the `fireEvent` method. Use `false` as the second constructor parameter to indicate that the event does not come from the client.

Example: Using the `fireEvent` method set to `false` to fire an event from the server.

```java
@Tag("input")
public class TextField extends Component {

    public void setValue(String value) {
        getElement().setAttribute("value", value);
        fireEvent(new ChangeEvent(this, false));
    }

    // Other component methods omitted
}
```

15.7.4. Firing Events From the Client

You can connect a component event to a DOM event that is fired by the element in the browser.

To do this, use the `@DomEvent` annotation in your event class to specify the name of the DOM event to listen to. Vaadin Flow automatically adds a DOM event listener to the element when a component event listener is present.

Example: Using the `@DomEvent` annotation to connect `TextField` component to a DOM event.

```java
@DomEvent("change")
public class ChangeEvent
        extends ComponentEvent<TextField> {
    public ChangeEvent(TextField source,
                       boolean fromClient) {
        super(source, fromClient);
    }
}
```

Adding Event Data

An event can include additional information, for example the mouse button used for a click event.

The @DomEvent annotation supports additional constructor parameters. You can use the @EventData annotation to define which data to send from the browser.

Example: Using the @EventData annotation to define additional click-event data.

```
@DomEvent("click")
public class ClickEvent
        extends ComponentEvent<NativeButton> {
    private final int button;

    public ClickEvent(NativeButton source,
            boolean fromClient,
            @EventData("event.button") int button) {
        super(source, fromClient);
        this.button = button;
    }

    public int getButton() {
        return button;
    }
}
```

- The @EventData definition runs as JavaScript in the browser.
- The DOM event is available as event and the element to which the listener was added is available as element.
- See DomListenerRegistration.addEventData[76] in the Javadoc for more about how event data is collected and sent to the server.

TIP See Event[77] in the MDN web docs for an overview of standard DOM events and properties.

Filtering Events

Instead of sending all DOM events to the server, you can filter events by defining a `filter` in the `@DomEvent` annotation. The filter is typically based on things related to the event.

Example: Defining a `filter` in the `@DomEvent` annotation.

```
@DomEvent(value = "keypress",
          filter = "event.key == 'Enter'")
public class EnterPressEvent
        extends ComponentEvent<TextField> {
    public EnterPressEvent(TextField source,
                           boolean fromClient) {
        super(source, fromClient);
    }
}
```

- The `filter` definition runs as JavaScript in the browser.
- The DOM event is available as `event` and the element to which the listener was added is available as `element`.
- See `DomListenerRegistration.setFilter`[78] in the Javadoc for more about how the filter is used.

Limiting Event Frequency

Certain kinds of events are fired very frequently when the user interacts with the component. For example, text input events fired while the user types.

You can configure the rate at which events are sent to the server by defining different `debounce` settings in the `@DomEvent` annotation. Debouncing always requires a `timeout` (in milliseconds) and a burst `phase`, which determines when events are sent to the server. There are three burst phase options:

- `LEADING` phase: An event is sent at the beginning of a burst, but subsequent events are only sent after one timeout period has passed, without any new events. This is useful for things like button clicks to prevent accidental double submissions.
- `INTERMEDIATE` phase: Periodical events are sent while a burst is ongoing. Subsequent events are delayed until one timeout period since the last event has passed. This is useful for things like text input, if you want to react continuously while the user types.
- `TRAILING` phase: This phase is triggered at the end of a burst after the timeout period has passed without any further events. This is useful for things like text input if you want to react only when the user stops typing.

Example: Configuring an `input` event to be sent to the server half a second after the user's last input.

```
@DomEvent(value = "input",
          debounce = @DebounceSettings(
              timeout = 250,
              phases = DebouncePhase.TRAILING))
public class InputEvent
        extends ComponentEvent<TextField> {
    private String value;

    public InputEvent(TextField source,
            boolean fromClient,
            @EventData("element.value") String value) {
        super(source, fromClient);
        this.value = value;
    }

    public String getValue() {
        return value;
    }
}
```

You can configure active events for several phases at the same time.

Example: Configuring an event for both the `LEADING` phase (immediately when a burst starts) and the `INTERMEDIATE` phase (while the burst is ongoing).

```
@DomEvent(value = "input",
          debounce = @DebounceSettings(
              timeout = 500,
              phases = {DebouncePhase.LEADING,
                        DebouncePhase.INTERMEDIATE }))
public class ContinuousInputEvent
        extends ComponentEvent<TextField> {
    private String value;

    public ContinuousInputEvent(TextField source,
            boolean fromClient,
            @EventData("element.value") String value) {
        super(source, fromClient);
        this.value = value;
    }

    public String getValue() {
        return value;
    }
}
```

- See `DomListenerRegistration.debounce`[79] in the Javadoc for more about debouncing events.

NOTE If you configure a `filter` and a `debounce` rate, only events that pass the filter are considered when determining whether a burst has ended.

15.8. Creating a Component Container

In this section we demonstrate how to create a `Component` container.

A component container is a component to which you can add other components. A container is typically created through a generic public API.

Example: Simple component container.

```
@Tag("div")
public class MyComponentContainer extends Component
        implements HasComponents {
}
```

- The `HasComponents` interface provides `add(Component…)` and `remove(Component…)` methods to handle attaching component elements to the `MyComponentContainer` root element (`<div>` in this example).

15.8.1. Implementing a Custom Add Method

If necessary, you can implement your own add method. For example if you need a different kind of API or have a complex internal element hierarchy.

Example: Implementing a custom method to add components to a container.

```
@Tag("div")
public class MyComponentContainer extends Component {

    public void add(Component child) {
        getElement().appendChild(child.getElement());
    }
}
```

15.8.2. Attaching Child Components to the DOM

When a child component is added to a container component, the container must attach the child to the DOM. This is the only absolute requirement for a container component.

In the previous example, the child element attaches to the root element (like `HasComponents` does). As an alternative, you can wrap each child in a wrapper element or use a more complex element hierarchy, where necessary.

Example: Wrapping a child component in an element wrapper.

```
@Tag("div")
public class MyComponentContainer extends Component {

    public void add(Component child) {
        Element childWrapper = ElementFactory
                .createDiv();
        childWrapper.appendChild(child.getElement());
        getElement().appendChild(childWrapper);
    }
}
```

15.8.3. Using Component Hierarchy Methods

Component hierarchy methods, such as `getChildren` and `getParent`, work automatically for container components, because they are implemented based on the element hierarchy. Theses methods also work if you add wrapper elements in between.

You can add a similar method to remove components.

Example: Using the `removeFromParent` method to detach a component.

```
public void remove(Component child) {
    Element wrapper = child.getElement().getParent();
    wrapper.removeFromParent();
}
```

NOTE You cannot assume that a component will always be removed when using the `remove` method. A child element can be detached manually through the `Element` API (for example, using the `Element.removeFromParent()` method) or by adding it to another component (for example, the `Element.appendChild` method moves the element from the old parent, if it is still attached).

TIP If you need to know when a child component is removed, add a detach listener to it using the `Component.addDetachListener()` method.

15.8.4. Enabling and Disabling Container Components

When you set a container component as disabled, all child components are automatically also set as disabled, and updates from the client to the server are blocked.

Components that implement the `HasEnabled` interface are updated accordingly to reflect the disabled state in the UI (which usually means setting the `disabled` attribute).

If your container includes elements or components that do not implement the `HasEnabled` interface, you can still **visually** update them to reflect the disabled state in the UI, by overriding the `onEnabledStateChanged` method.

Example: Setting a component as `disabled` by overriding he `onEnabledStateChanged` method.

```
@Override
public void onEnabledStateChanged(boolean enabled) {
    super.onEnabledStateChanged(enabled);
    if (enabled) {
        childElement.removeAttribute("disabled");
    } else {
        childElement.setAttribute("disabled", true);
    }
}
```

- You only need to override the `onEnabledStateChanged` method to update the visual aspect of the element. When the container is disabled, communication from the client to the server is blocked, regardless of whether or not you override the method.
- It is important to call `super.onEnabledStateChanged(enabled)` when overriding, because this is common logic and relevant to all components regarding the enabled state.
- The `onEnabledStateChanged` method is called every time the enabled state changes, whether by direct calls to `setEnabled`, by calling `setEnabled` on a parent container, or by attaching or detaching the component to a disabled container.

See Component Enabled State[80] for more.

15.9. Using Component Lifecycle Callbacks

If the content of a component depends on resources that are not available during the construction of the component, you can postpone content creation until the component attaches to the `UI`, by overriding the `onAttach()` method (provided by the `Component` class).

Example: Overriding the `onAttach` method.

```java
@Tag("div")
public class UserNameLabel extends Component {

  @Override
  protected void onAttach(AttachEvent attachEvent) {
    // user name can be stored to session after login
    String userName = (String) attachEvent.getSession()
            .getAttribute("username");
    getElement().setText("Hello " + userName +
            ", welcome back!");
  }
}
```

The `onAttach` method is invoked when the `Component` has attached to the `UI`. Its counterpart, the `onDetach` method, is invoked right before the component detaches from the `UI`. These are good times to reserve and release resources used by the component.

Example: Overriding the `onAttach` and `onDetach` methods.

```
@Tag("div")
public class ShoppingCartSummaryLabel
        extends Component {

  private final Consumer<EventObject> eventHandler =
        this::onCartSummaryUpdate;

  @Override
  protected void onAttach(AttachEvent attachEvent) {
    ShopEventBus eventBus = attachEvent.getSession()
            .getAttribute(ShopEventBus.class);
    // registering to event bus for updates
    // from other components
    eventBus.register(eventHandler);
  }

  @Override
  protected void onDetach(DetachEvent detachEvent) {
    ShopEventBus eventBus = detachEvent.getSession()
    .getAttribute(ShopEventBus.class);
    // after detaching don't need any updates
    eventBus.unregister(eventHandler);
  }

  private void onCartSummaryUpdate(EventObject event) {
    // update cart summary ...
  }
}

interface ShopEventBus {
  void register(Consumer<EventObject> eventHandler);

  void unregister(Consumer<EventObject> eventHandler);
}
```

- Using methods available in `attachEvent` and `detachEvent` to get the UI or session is more convenient than using the `getUI()` method in `Component`, because these methods return values directly. The `getUI()` method returns an `Optional<UI>`, because a component is not always attached.

- The default implementations of the `onAttach` and `onDetach` methods are empty, so you don't need to call `super.onAttach()` or `super.onDetach()` from your overridden methods. However, when extending other component implementations you may need to do this.

> TIP
>
> To find out when another component gets attached or detached, you can use the `Component.addAttachListener` and `Component.addDetachListener` methods. The corresponding events are fired after the `onAttach` and `onDetach` methods are invoked. The `getUI()` method for the component will return the UI instance during both events.

15.10. Using Vaadin Mixin Interfaces

A mixin refers to a defined amount of functionality that can be added to a class. Traditionally, Java did not support this kind of multiple inheritance, but since Java 8 interfaces can also include default methods, which allows them to work as mixins.

Vaadin Flow uses the mixin concept to provide common APIs and default behavior for sets of functionalities found in most Web Components.

The most important predefined mixins are provided by the `HasSize`, `HasComponents` and `HasStyle` interfaces. You can use these interfaces to add typical functions to your Java components.

15.10.1. HasSize Interface

If your component implements the `HasSize` interface, you can set the size of the component using the `setWidth(String)` and `setHeight(String)` methods.

Methods available in the `HasSize` interface:

- `void setWidth(String width)`
- `String getWidth()`
- `void setHeight(String height)`
- `String getHeight()`
- `void setSizeFull()`
- `void setSizeUndefined()`

15.10.2. HasComponents Interface

If your component implements the `HasComponents` interface, you can add and remove child components to and from it.

Methods available in the `HasComponents` interface:

- `void add(Component… components)`
- `void remove(Component… components)`
- `void removeAll()`

15.10.3. HasStyle Interface

Components that implement the `HasStyle` interface can have a class attribute and support inline styles.

Methods available in the `HasStyle` interface:

- `void addClassName(String className)`
- `boolean removeClassName(String className)`
- `void setClassName(String className)`
- `String getClassName()`
- `ClassList getClassNames()`
- `void setClassName(String className, boolean set)`
- `boolean hasClassName(String className)`
- `Style getStyle()`
- `void addClassNames(String… classNames)`
- `void removeClassNames(String… classNames)``

15.10.4. Using Mixin Interfaces

Example: Creating a custom `Tooltip` component that implements the `HasComponents` and `HasStyle` interfaces.

```
public class Tooltip extends Component
        implements HasComponents, HasStyle {

}
```

```
class Tooltip extends PolymerElement {
    static get template() {
        return html`
            <div part="content" theme="dark">
                <slot></slot>
            </div>`;
    }
}
```

- A component that `HasComponents` needs to extend from a tag that supports having child components. The `slot` tag is used in Web Components to define where child components should be put.

When you implement the `HasComponents` interface, adding child components to the parent component is allowed automatically.

Example: Adding new `H5` and `Paragraph` child components to the `Tooltip` parent component.

```
Tooltip tooltip = new Tooltip();

tooltip.add(new H5("Tooltip"));
tooltip.add(new Paragraph("I am a paragraph"));
```

15.10.5. Other Useful Mixin Interfaces

Vaadin Flow provides many additional useful mixin interfaces:

- `HasEnabled`: Generic interface for components and other UI objects that can be enabled or disabled.
- `HasElement`: Marker interface for any class that is based on an `Element`.
- `HasDataProvider<T>`: Generic interface for listing components that use a data provider to display data.
- `HasValidation`: Generic interface that supports input validation.
- `HasItems`: Mixin interface for components that display a collection of items.
- `HasOrderedComponents`: Generic interface that supports ordered child components, with an index for the layout.
- `HasText`: Generic interface that supports text content.

- `Focusable<T>`: Interface that provides methods to gain and lose focus.

15.10.6. Advantages of Using Mixin Interfaces

Using Vaadin mixins is a best practice because their code and functionality has been throughly checked and tested by Vaadin.

Mixins also keep your code clean and simple. For example, compare setting component width:

- Without mixins: `getElement().getStyle().set("width", "300px")`.
- After implementing the `HasSize` interface: `setWidth("300px")`.

----

[73] https://vaadin.com/docs/flow/polymer-templates/tutorial-template-basic.html
[74] https://polymer-library.polymer-project.org/3.0/docs/about_30
[75] https://polymer-library.polymer-project.org/3.0/docs/devguide/dom-template#inherit
[76] https://vaadin.com/api/platform/com/vaadin/flow/dom/DomListenerRegistration.html
[77] https://developer.mozilla.org/en-US/docs/Web/API/Event
[78] https://vaadin.com/api/platform/com/vaadin/flow/dom/DomListenerRegistration.html
[79] https://vaadin.com/api/platform/com/vaadin/flow/dom/DomListenerRegistration.html
[80] https://vaadin.com/docs/flow/components/tutorial-enabled-state.html

16. Integrating Web Components

16.1. What are Web Components?

- Web Components are a collection of web standards allowing you to create new HTML tags with custom name, reusability and full encapsulation on styles & markup. The standards are promised to be ubiquitous in modern browsers.
- The term "Web Components" might seem unfamiliar at first but in fact, you are already using web components**(*)** when you develop. HTML elements like input, select, or textarea are all browser native web components. In short, native elements have followed the concept of the web components for a long time now.

(*) Not to be confused with the uppercase initials - Web Components

16.1.1. Specifications

Web Components consist of four main standards [1] which can be used independently or all together:

- **Custom Elements**: A set of APIs to define new HTML elements and their functionalities.
- **Shadow DOM**: A set of APIs to provide encapsulation of the element's styles markup and functions so that your web component may remain "hidden"* and separated from the rest of the DOM.
- **HTML Template**: The <template> element allows you to input fragments of HTML, which remain inert at page load, but can be instantiated at runtime.

- **HTML Imports** **: A mechanism allows you to import and reuse a piece or the whole custom component from an external source.

**everything in the shadow DOM can still be viewed and accessed.*

***HTML Imports is not standardized. It was a specification at the phase of the draft but received a heavy backlash from some modern browser vendors [2][3] and WebKit engineers [4].*

16.1.2. Vaadin and Web components

Vaadin both makes and maintains a set of Web Components as well as uses them to provide Java web developers API through Vaadin Flow[81].

To find out more how Vaadin utilizes Web Component:

- Vaadin Components[82], a modern set of Web Components for business applications.
- Using any Web Component in Java web applications with Vaadin Flow
- Vaadin Directory[83] - Listing of curated and rated third party Web Components.

16.1.3. External links and references

[1] https://github.com/w3c/webcomponents/

[2] https://hacks.mozilla.org/2014/12/mozilla-and-web-components/

[3] https://developer.mozilla.org/en-US/docs/Web/Web_Components/HTML_Imports

[4] https://webkit.org/status/#feature-html-imports

16.2. Integrating a Web Component

Web Components are a collection of web standards allowing you to create new HTML tags with custom name, reusability and full encapsulation on styles & markup. To understand more about what Web Components are, visit Introduction to Web Components.

To be able to use a web component from Flow you need two things: to load the HTML/JS/CSS files needed by the component and a Java API used to configure the component and to listen to events from it.

The client-side files for a web component, typically JS module files, are available using npm[84]. Flow 2.0 and above comes with npm support. It will automatically install and use npm packages and serve the static files to the browser.

16.2.1. Step 1. Integrating a JS module with Vaadin

While you can start from scratch and do all the things manually, it's easiest to start with the component project available at https://vaadin.com/start/lts/component. This will give you a project with Flow dependencies, npm import for the selected component and a stub Java class for your web component integration. It also contains a Maven profile which will handle all things needed to deploy it to Vaadin Directory.

As an example, if you create a starter project for https://github.com/PolymerElements/paper-slider, following annotations are attached to the server-side component:

```
@Tag("paper-slider")
@NpmPackage(value = "@polymer/paper-slider",
            version = "3.0.1")
@JsModule("@polymer/paper-slider/paper-slider.js")
```

The name of the HTML element is defined using `@Tag("paper-slider")` and the JS import for the component is defined using `@JsModule("@polymer/paper-slider/paper-slider.js")` and `@NpmPackage(value = "@polymer/paper-slider", version = "3.0.1")`.

If your component requires in-project front-end files, for example JavaScript modules, add them to the `src/main/resources/META-INF/resources/frontend` directory so that they are packaged in the component jar if you choose to make an add-on of your component. A local JavaScript module should be loaded with a `@JsModule` annotation as follows:

```
@JsModule("./my-local-module.js")
```

The `vaadin-maven-plugin` will automatically install the npm package in `node_modules` and import the JS module file into the document provided to the browser when running `mvn clean install`. Moreover, if the Jetty webserver is run from Maven (using `mvn jetty:run`), your project's source code is monitored for changes to these types of annotations. So any change to `@NpmPackage` or `@JsModule` annotations will trigger installation of the referenced packages and hot deployment of your app including the new JS module imports.

The main test/demo Java class

`src/test/java/…/DemoView.java`

```
@Route("")
public class DemoView extends VerticalLayout {
```

The project is set up in a slightly unconventional way so it can be a single-module Maven project. The test folder is used for a test/demo application in addition to actual test files. When you run

```
mvn jetty:run
```

in the project, it will deploy `DemoView` and show it at http://localhost:8080

Now you are set to create the Java API, for more details see Creating Java API for a Web Component

NOTE Some web components will not show any UI when they are just added as empty tags to the page. If the demo view is empty, inspect first the browser browser console to verify that all files were found (no 404s) and then check if the component is correctly configured.

NOTE While the project setup is easy to use for development/testing, it does not allow you to easily produce a demo war file for deployment. It’s usually better to create a separate project (or convert the project into a multi-module project) for this as the "demo" files included in the addon itself tend to be test UIs whereas a demo should be aimed at the end user.

NOTE If you want to make your component OSGi compatible refer to the Making a component add-on OSGi-compatible document.

IMPORTANT

If the project you are using is configured as a multi-module project (the base project is an older version or you have done manual conversion), the source monitoring will not work and changes to the component are not automatically reflected to your demo application!

16.2.2. Step 3. Deploying the Add-on to Vaadin Directory

When you are satisfied with the API, you can make the add-on available to the world by deploying it into Vaadin Directory. You can create the Directory compatible add-on package using

```
mvn clean install -Pdirectory
```

This creates a zip file in the `target` directory.

Go to https://vaadin.com/directory, log in or register, and upload this zip file. Be sure to write an overview for your add-on to let others know what you can do with it, what browsers it supports etc. Then publish it and others can take your add-on into use by copying the dependency information from the add-on page in the directory.

NOTE

The metadata used by Vaadin Directory is defined in `assembly/MANIFEST.MF`, based on the project's metadata. If you do changes to the project such as removing `<name></name>`, make sure you update the metadata.

16.3. Creating Java API for a Web Component

The component class you get when using the component starter (see Integrating a Web Component), e.g. `PaperSlider.java`, is only a stub which handles the imports. There are multiple ways to interact with a web component but the typical pattern is:

- Use properties on the element to define how it should behave
- Listen to events on the element to get notified of when the user does something
- Call functions on the element to perform specific tasks such as open a popup
- Add sub elements to define child contents

16.3.1. Setting and reading properties

You can typically find out what properties an element supports from its JavaScript docs. E.g., for paper-slider: https://www.webcomponents.org/element/@polymer/paper-slider/elements/paper-slider. The slider has a boolean property called `pin` which defines if "a pin with numeric value label is shown when the slider thumb is pressed". To create the corresponding Java setter-getter API, you can add:

```
public void setPin(boolean pin) {
    getElement().setProperty("pin", pin);
}
public boolean isPin() {
    return getElement().getProperty("pin", false);
}
```

The setter will now set the given property to the requested value and the getter will return the property value, or `false` as the default if the property has not been set (this should match the default of the web component property).

If you then update `DemoView`

```
public DemoView() {
    PaperSlider paperSlider = new PaperSlider();
    paperSlider.setPin(true);
    add(paperSlider);
}
```

you will see the pin appear when dragging the slider knob.

A drawback of writing the `getElement` methods directly like above is that you end up repeating the property name in the getter and the setter. To avoid repeating the property name you can use the `PropertyDescriptor` helper. `PropertyDescriptor` and the factory methods in `PropertyDescriptors` allow defining the `pin` property as a single static field in the component that can be referenced from the getter and the setter:

```
public class PaperSlider extends Component {

    private static final PropertyDescriptor<Boolean,
Boolean> pinProperty = PropertyDescriptors
.propertyWithDefault("pin", false);

    public void setPin(boolean pin) {
        pinProperty.set(this, pin);
    }

    public boolean isPin() {
        return pinProperty.get(this);
    }
}
```

The `pinProperty` descriptor here defines a property with the name `pin` and a default value of `false` (matches the web component) and both a setter and getter type of `Boolean` through generics (`<Boolean, Boolean>`). The setter and getter code then only invokes the descriptor with the component instance.

16.3.2. Synchronizing the Value

`paper-slider` is a component that allows the user to input a single value. This kind of component should implement the `HasValue` interface so it will automatically work as a field in forms with data binding.

The value should be synchronized automatically from the client to the server when the user changes it, as well from the server to the client when updated programmatically. Additionally a value change event should be emitted on the server whenever the value changes. In the common case where `getValue()` is based on a single element property, the `AbstractSinglePropertyField` base class takes care of everything related to the value.

```
public class PaperSlider extends
AbstractSinglePropertyField<PaperSlider, Integer> {

    public PaperSlider() {
        super("value", 0, false);
    }

}
```

The type parameters define the component type (`PaperSlider`) returned by `getSource()` in value change events and the value type (`Integer`). The constructor parameters define the name of the element property that contains the value (`"value"`), the default value to use if the property isn't set (`0`) and whether `setValue(null)` should be allowed or throw an exception (`false` means that `null` is not allowed).

NOTE For more advanced cases that still rely on only one element property, there's an alternative constructor for defining callbacks that convert between the low-level element property type and the high level `getValue()` type. For cases where the value cannot be derived based on a single element property, there's a more generic `AbstractField` base class.

You can test this for instance as follows in the demo class:

```
public DemoView() {
    PaperSlider paperSlider = new PaperSlider();
    paperSlider.setPin(true);
    paperSlider.addValueChangeListener(e -> {
        String message = "The value is now " + e.
getValue();
        if (e.isFromClient()) {
            message += " (set by the user)";
        }
        Notification.show(message, 3000, Position.MIDDLE
);
    });
    add(paperSlider);

    Button incrementButton = new Button("Increment using
setValue", e -> {
        paperSlider.setValue(paperSlider.getValue() + 5);
    });
    add(incrementButton);
}
```

NOTE

Some web components also update other properties that are not related to `HasValue`. Creating A Simple Component Using the Element API[85] describes how you can use the `@Synchronize` annotation to synchronize property values without automatically firing a value change event.

16.3.3. Listening to Events

All web elements emit a `click` event when the user clicks on them. To allow the user of your component to listen to the `click` event, you can extend `ComponentEvent` together with the `@DomEvent` and `@EventData` annotations:

```
@DomEvent("click")
public class ClickEvent extends ComponentEvent
<PaperSlider> {

    private int x,y;

    public ClickEvent(PaperSlider source, boolean
fromClient, @EventData("event.offsetX") int x,
@EventData("event.offsetY") int y) {
        super(source, fromClient);
        this.x = x;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}
```

Then use `ClickEvent` class as an argument when invoking `addListener` method on your `PaperSlider` component.

```
public Registration addClickListener
(ComponentEventListener<ClickEvent> listener) {
    return addListener(ClickEvent.class, listener);
}
```

The `addListener` method in the superclass will set up everything related to the event based on the annotations in the `ClickEvent` class that also need to be created.

The `ClickEvent` defined above uses `@DomEvent` to define the name of the DOM event to listen for (`click` in this case). Like all other events fired by a `Component`, it extends `ComponentEvent` which provides a typed `getSource()` method.

It uses two additional constructor parameters annotated with `@EventData` to get the click coordinates from the browser. The expression inside the `@EventData` is evaluated when the event is handled in the browser, and can access DOM event properties using the `event.` prefix (e.g. `event.offsetX`) and element properties using the `element.` prefix.

You can test the event integration in the demo, e.g., by adding to `DemoView.java`:

```
paperSlider.addClickListener(e -> {
    Notification.show("Clicked at " + e.getX() + "," + e
.getY(), 1000, Position.BOTTOM_START);
});
```

NOTE The two first parameters to a `ComponentEvent` constructor must be `PaperSlider source, boolean fromClient` and are filled automatically. All parameters following these two initial parameters must carry the `@EventData` annotation.

TIP The click event was used here for illustrative purposes. In a real use case, you should use the `ClickEvent` provided by Flow instead, which will also provide additional event details.

TIP As the event data expression is evaluated as JavaScript, you can control propagation behavior using, e.g., `@EventData("event.preventDefault()") String ignored`. This is a workaround for the lack of other API to control this behavior.

16.3.4. Calling Element Functions

In addition to properties and events, many elements offer methods which can be invoked for various reasons, e.g. `vaadin-board` has a `refresh()` method which is called whenever a change is made that the web component itself is not able to detect automatically. To call a function on an element, you can use the `callJsFunction` method in `Element`, e.g. to offer an API to the `increment` function on `paper-slider`, you could add to `PaperSlider.java`:

```
public void increment() {
    getElement().callJsFunction("increment");
}
```

You can test this by adding a call to `DemoView.java`:

```
Button incrementJSButton = new Button("Increment using
JS", e -> {
    paperSlider.increment();
});
add(incrementJSButton);
```

If you do this and add also the value change listener described earlier, you will see that you get a notification with the new value after clicking on the button. The notification also indicates that the user changed the value because `isFromClient` checks that the change originates from the browser (as opposed to from the server) but does not differentiate between the cases when a user event changed the value and when a JavaScript call changed it.

NOTE

This particular example is quite artificial as it is doing a server visit from a button click only to call a Javascript method on another element on client side. In practice you would either call `increment()` directly from client side, or from some other server-side business logic.

TIP

In addition to the method name, `callJsFunction` takes an arbitrary number of parameters of certain supported types. Supported types are at the time of writing `String`, `Boolean`, `Integer`, `Double`, the corresponding primitive types, `JsonValue`, `Element` and `Component` references. It also returns a server-side promise for the JavaScript function's return value. See the method's javadoc for more information.

16.3.5. Final Slider Integration Result

After doing the steps described above, you should end up with the following `PaperSlider` class:

```
@Tag("paper-slider")
@NpmPackage(value = "@polymer/paper-slider", version =
"3.0.1")
@JsModule("@polymer/paper-slider/paper-slider.js")
public class PaperSlider extends
AbstractSinglePropertyField<PaperSlider, Integer> {

    private static final PropertyDescriptor<Boolean,
Boolean> pinProperty = PropertyDescriptors
.propertyWithDefault("pin", false);

    public PaperSlider() {
        super("value", 0, false);
    }

    public void setPin(boolean pin) {
        pinProperty.set(this, pin);
    }

    public boolean isPin() {
        return pinProperty.get(this);
    }

    public Registration addClickListener
(ComponentEventListener<ClickEvent> listener) {
        return addListener(ClickEvent.class, listener);
    }

    public void increment() {
        getElement().callJsFunction("increment");
    }
}
```

This can now be further extended to support more configuration properties like `min` and `max`.

16.3.6. Add Sub Elements to Define Child Contents

Some web components can contain child elements. If the component is a layout type where you just want to add child components, it is enough to implement `HasComponents`. The

HasComponents interface provides default implementations for add(Component…), remove(Component…) and removeAll(). As an example, you could implement your own <div> wrapper as

```
@Tag(Tag.DIV)
public class Div extends Component implements
HasComponents {
}
```

You can then add and remove components using the provided methods, e.g.

```
Div root = new Div();
root.add(new Span("Hello"));
root.add(new Span("World"));
add(root);
```

If you do not want to provide a public add/remove API, you have two options: use the Element API or create a new Component for encapsulating the internal element behavior.

As an example, say you wanted to create a specialized Vaadin Button which can only show a VaadinIcon. Using the available VaadinIcon enum, which lists the icons in the set, you can do e.g

```
@Tag("vaadin-button")
@NpmPackage(value = "@vaadin/vaadin-button", version =
"2.1.5")
@JsModule("@vaadin/vaadin-button/vaadin-button.js")
public class IconButton extends Component {

    private VaadinIcon icon;

    public IconButton(VaadinIcon icon) {
        setIcon(icon);
    }

    public void setIcon(VaadinIcon icon) {
        this.icon = icon;

        Component iconComponent = icon.create();
        getElement().removeAllChildren();
        getElement().appendChild(iconComponent.
getElement());
    }

    public void addClickListener(
            ComponentEventListener<ClickEvent<IconButton
>> listener) {
        addListener(ClickEvent.class,
(ComponentEventListener) listener);
    }

    public VaadinIcon getIcon() {
        return icon;
    }
}
```

The relevant part here is in the `setIcon` method. As there happens to be a feature in `VaadinIcon` which creates a component for a given icon (the `create()` call), it is used to create the child element. What remains is then to attach the root element of the child component by calling `getElement().appendChild(iconComponent.getElement());`.

In case the `VaadinIcon.create()` method was not available, you would have to resort to either creating the component yourself or using the element API directly. If you use the element API, the `setIcon` method might look something like:

```
public void setIcon(VaadinIcon icon) {
    this.icon = icon;
    getElement().removeAllChildren();

    Element iconElement = new Element("iron-icon");
    iconElement.setAttribute("icon", "vaadin:" + icon
.name().toLowerCase().replace("_", "-"));
    getElement().appendChild(iconElement);
}
```

The first part is the same but in the second part, the element with the correct tag name `<iron-icon>` is created manually and the `icon` attribute is set to the correct value, defined in `vaadin-icons.html`, e.g. `icon="vaadin:check"` for `VaadinIcon.CHECK`. The element is then attached to the `<vaadin-button>` element, after removing any previous content. With this approach you must also ensure that the `vaadin-button.html` dependency is loaded, otherwise handled by the `Icon` component class:

```
@NpmPackage(value = "@vaadin/vaadin-button", version =
"2.1.5")
@JsModule("@vaadin/vaadin-button/vaadin-button.js")
@NpmPackage(value = "@vaadin/vaadin-icons", version =
"4.3.1")
@JsModule("@vaadin/vaadin-icons/vaadin-icons.js")
public class IconButton extends Component {
```

With either approach, you can test the icon button, e.g., as

```
IconButton iconButton = new IconButton(VaadinIcon.CHECK);
iconButton.addClickListener(e -> {
    int next = (iconButton.getIcon().ordinal() + 1) %
VaadinIcon.values().length;
    iconButton.setIcon(VaadinIcon.values()[next]);
});
add(iconButton);
```

This will show the `CHECK` icon and then change the icon on every click of the button.

NOTE You could extend `Button` directly instead of `Component` but then you would also inherit all the public API of `Button`.

16.4. Debugging a Web Component Integration

Not everything is smooth sailing and sometimes the component just refuses to work like you want it to. If the problem is on the Java side, you can use your standard IDE debugger to figure out what happens but when the problem is in the browser, it gets a bit trickier. Chrome Inspector is an invaluable tool when trying to figure out what goes wrong.

16.4.1. Is the element not configured as it should?

Check with the DOM inspector that the element contains the expected attributes (most of the time properties are synchronized to attributes and vice versa). If the property is not synchronized to an attribute, select the element in the inspector and write `$0.somePropertyName` in the console to check that the value is the expected one.

16.4.2. Is an event not sent to the server as you would expect?

Select the element, and write `monitorEvents($0,'event-name');` in the console. You will now see a log row if the event is triggered and will know you have the correct event name and that the web component actually fires the event. You can leave out `'event-name'` to log all events but be prepared to see a lot of `mousemove` events. You can also use this to see which properties are defined for the event so that you can know what to include in `@EventData`.

16.4.3. Do you need to debug the Javascript?

If you need to debug what the web component does, use the browser debugger to set breakpoints at suitable places. In more problematic cases, e.g., if the problem occurs on initial setup, you can add a `debugger;` statement to the web component code to make the browser always and automatically break at that point. To do that, you need to edit the web component included in your project. All the components used in this project will be downloaded by npm and located in the `node_modules` folder under the project root folder.

To debug the `increment()` function in `paper-slider` you can thus do:

1. Launch the project in dev mode so that any frontend file change is automatically used after the page reload
2. Find a `paper-slider` in the `node_modules` directory: `node_modules/@polymer/paper-slider`
3. Add a `debugger` statement to the `increment: function() {` function

4. Reload the page, click on the "Increment" button when the inspector window is open

TIP Disable the cache in the browser network tab to avoid getting old versions of the files you are debugging.

16.5. Creating Another type of Add-on

If you want to create an add-on which is not a UI component, e.g. a data provider, you can still use the same component starter described in Integrating a Web Component. Leave the default web component URL in the starter form, download the project and delete:

1. The `@NpmPackage` and `@JsModule` annotations
2. The UI component class

You now have a generic project which can be used for any add-on purposes and which supports Directory deployment using

```
mvn clean install -Pdirectory
```

16.6. Creating an In-project Web Component

To integrate existing, public web components it typically makes sense to do as described above in Integrating a Web component. If you want to create a UI component which is specific to the application project you are working on, you can integrate and develop it within your application project instead. Assuming you have an existing project, here the https://vaadin.com/start/lts/project-base project is used as an example, you need to

1. Create a Polymer 3 template for the component
2. Create a Java API for the component

16.6.1. Template

Create a `frontend/my-test-element/my-test-element.js` file with the following contents:

```
import {html, PolymerElement} from
'@polymer/polymer/polymer-element.js';

class MyTestElement extends PolymerElement {
  static get template() {
    return html`
      <h2>Hello</h2>
    `;
  }
}

window.customElements.define('my-test-element',
MyTestElement);
```

16.6.2. Java API

This works exactly as described in Creating Java API for a Web Component. The only difference is that static files are loaded from your project and you can modify them easily while creating the Java API.

As an example, for the generated `my-test-element`, you could do

```
@Tag("my-test-element")
@JsModule("my-test-element/my-test-element.js")
public class MyTest extends Component {

    public MyTest(String prop1) {
        getElement().setProperty("prop1", prop1);
    }
}
```

You can now use the component as e.g.

```
public class MainView extends VerticalLayout {
    public MainView() {
        add(new MyTest("World"));
    }
}
```

----

[81] https://vaadin.com/flow
[82] https://vaadin.com/components
[83] https://vaadin.com/directory
[84] https://www.npmjs.com/
[85] https://vaadin.com/docs/flow/creating-components/tutorial-component-basic.html

17. Packaging for Production

17.1. Taking your Application into Production

17.1.1. Simple steps for production mode build

To get your application prepared for production you want to create a production mode profile which brings in the production-mode dependency.

pom.xml

```
<profiles>
    <profile>
        <id>production</id>
        <properties>
            <vaadin.productionMode>
true</vaadin.productionMode>
        </properties>
        <dependencies>
            <dependency>
                <groupId>com.vaadin</groupId>
                <artifactId>flow-server-production-
mode</artifactId>
            </dependency>
        </dependencies>
    </profile>
</profiles>
```

If you only have the `prepare-frontend` goal in the `vaadin-maven-plugin` then you need to add the `build-frontend` goal to the plugin or define the whole plugin in the production profile:

pom.xml

```
<profiles>
    <profile>
        <id>production</id>
        <properties>
            <vaadin.productionMode>
true</vaadin.productionMode>
        </properties>
        <dependencies>
            <dependency>
                <groupId>com.vaadin</groupId>
                <artifactId>flow-server-production-
mode</artifactId>
            </dependency>
        </dependencies>
        <build>
            <plugins>
                <plugin>
                    <groupId>com.vaadin</groupId>
                    <artifactId>vaadin-maven-
plugin</artifactId>
                    <version>${vaadin.version}</version>
                    <executions>
                        <execution>
                            <goals>
                                <goal>prepare-
frontend</goal>
                                <goal>build-
frontend</goal>
                            </goals>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
    </profile>
</profiles>
```

The profile is recommended so that you don’t get any unexpected problems due to production settings when running in development mode.

NOTE For users who use `Vaadin Flow` only, you can use `flow-maven-plugin` as artifactId and match the version number with current Flow version

After this all that is needed is to run `mvn clean package -Pproduction`.

The simplest way to get a production ready setup is to get a project base from https://vaadin.com/start

NOTE If your environment(OS) is 64bit make sure to use JDK 64bit as well. There is a known issue when JDK 32bit is used on a 64bit environment.

NOTE For an old project running with bower see V13 documentation about the production mode.

17.1.2. What is transpilation and bundling

Transpilation in Flow means converting all ES6 JavaScript to ES5 JavaScript format for older browsers which for us is IE 11 and Safari 9.

NOTE IOS 10 has a known issue with `let bindings in for loops are incorrectly treated as function-scoped instead of block scoped`[86], in this case, all browsers running on it need the transpilation, too.

Minimisation is done to make the file smaller. When minifying the code of also often obscured making it harder to read.

Bundling is an optimisation where we merge multiple files to a collection so that the browser doesn't need to request so many files making loading faster.

17.2. Advanced production mode topics

17.2.1. Production mode issues

After adding the `flow-server-production-mode` dependency the application no longer starts.

One likely cause of this problem is that the `build-frontend` of the `flow-maven-plugin` was not executed, either because the plugin is missing from the `pom.xml` or it is missing configuration. To fix this simply add the `flow-maven-plugin` to your maven `build` block (make sure it's visible in your production mode profile), and enable the `build-frontend` goal.

17.2.2. Splitting the Webpack bundle into multiple chunks

This is an upcoming feature in Vaadin Platform 14. For information on customizing bundling when using compatibility mode, see the Vaadin 13 documentation on this topic[87].

17.2.3. Plugin goals and goal parameters

Here we describe the maven plugin goals and their usage.

prepare-frontend

The intention of the goal is to validate whether `node` and `npm` tools are installed. Node.js is needed to run npm for installing frontend dependencies and webpack which bundles the frontend files served to client. In case they are missing, an exception is thrown and the build process terminates.

In addition, it visits all resources used by the application and copies them under `node_modules` folder so they are available when `webpack` builds the frontend. It also creates or updates `package.json`, `webpack.config.json` and `webpack.generated.json` files.

Goal parameters

- **jarResourcePathsToCopy** (default: `META-INF/resources/frontend`): Comma separated values for the paths that should be analyzed in every project dependency jar and, if files suitable for copying present in those paths, those should be copied.
- **includes** (default: `**/*.js,**/*.css`): Comma separated wildcards for files and directories that should be copied. Default is only .js and .css files.
- **npmFolder** (default: `${project.basedir}`): The folder where `package.json` file is located. Default is project root folder.
- **webpackTemplate** (default: `webpack.config.js`): Copy the `webapp.config.js` from the specified URL if missing. Default is the template provided by this plugin. Set it to empty string to disable the feature.
- **webpackGeneratedTemplate** (default: `webpack.generated.js`): Copy the `webapp.config.js` from the specified URL if missing. Default is the template

provided by this plugin. Set it to empty string to disable the feature.

- **generatedFolder** (default: `${project.build.directory}/frontend/`): The folder where Flow will put generated files that will be used by Webpack.

build-frontend

This goal builds the frontend bundle. This is a complex process involving several steps:

- update `package.json` with all `@NpmPackage` annotation values found in the classpath and automatically install these dependencies.
- update the JavaScript files containing code for importing everything used in the application. These files are generated in the `target/frontend` folder, and will be used as entry point of the application.
- create `webpack.config.js` if not found, or updates it in case some project parameters have changed.
- generate JavaScript bundles, chunks and transpile to ES5 using `webpack` server. Target folder in case of `war` packaging is `target/${artifactId}-${version}/build` and in case of `jar` packaging is `target/classes/META-INF/resources/build`.

Goal parameters

- **npmFolder** (default: `${project.basedir}`: The folder where `package.json` file is located. Default is project root folder.

- **generatedFolder** (default: `${project.build.directory}/frontend/`): The folder where Flow will put generated files that will be used by Webpack.
- **frontendDirectory** (default: `${project.basedir}/frontend`): A directory with project’s frontend source files.
- **generateBundle** (default: `true`): Whether to generate a bundle from the project frontend sources or not.
- **runNpmInstall** (default: `true`): Whether to run `npm install` after updating dependencies.
- **generateEmbeddableWebComponents** (default: `true`): Whether to generate embeddable web components from WebComponentExporter inheritors.

17.3. How to Run and Deploy a Flow Application on Jetty

This document explains how to run and deploy a Vaadin Flow application on Jetty.

Jetty is an open-source project providing an HTTP server, HTTP client, and javax.servlet container.

Jetty applications can be deployed in 2 different ways:

1. Embedded Jetty
   - Jetty Maven Plugin
   - Programmatically
2. Standalone Jetty

- WAR
- Exploded directory
- Context File

17.3.1. Embedded Jetty

Jetty can be used during the development phase of an application to increase the productivity of developers.

Using Jetty has the advantage that it can be instantiated and used in a Java program.

"Don’t deploy your application in Jetty, deploy Jetty in your application!" by Jetty.

This application is Embedded Jetty.

Embedded Jetty can be used in Vaadin application in 2 different ways:

1. Jetty Maven Plugin
2. Programmatically (without Maven)

Jetty Maven Plugin

The Jetty Maven plugin is useful for rapid development and testing.

To be able to deploy and run applications with it, it is only needed to add the plugin inside the `pom.xml`:

pom.xml

```
<build>
    <plugins>
        <!-- Jetty plugin for easy testing without a
server -->
        <plugin>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-maven-plugin</artifactId>
            <version>9.4.15.v20190215</version>
            <configuration>
                <scanIntervalSeconds>
2</scanIntervalSeconds>
            </configuration>
        </plugin>
    </plugins>
</build>
```

To run the application after adding the plugin, it is necessary to be situated in the project's root directory where `pom.xml` is located and run the following command:

```
mvn jetty:run
```

It is possible to run the application on jetty using an exploded WAR file:

```
mvn jetty:run-exploded
```

Programmatically (without Maven)

Jetty can also be configured to run programmatically. This requires a manual configuration to make it work with Vaadin.

main.java

```
public final class ManualJetty {
    public static void main(String[] args) throws
Exception {
        Server server = new Server(8080);

        // Specifies the order in which the
configurations are scanned.
        Configuration.ClassList classlist =
Configuration.ClassList.setServerDefault(server);
        classlist.addAfter(
"org.eclipse.jetty.webapp.FragmentConfiguration",
"org.eclipse.jetty.plus.webapp.EnvConfiguration",
"org.eclipse.jetty.plus.webapp.PlusConfiguration");
        classlist.addBefore(
"org.eclipse.jetty.webapp.JettyWebXmlConfiguration",
"org.eclipse.jetty.annotations.AnnotationConfiguration");

        // Creation of a temporal directory.
        File tempDir = new File(System.getProperty(
"java.io.tmpdir"), "JettyTest");
        if (tempDir.exists()) {
            if (!tempDir.isDirectory()) {
                throw new RuntimeException("Not a
directory: " + tempDir);
            }
        } else if (!tempDir.mkdirs()) {
            throw new RuntimeException("Could not make: "
+ tempDir);
        }

        WebAppContext context = new WebAppContext();
        context.setInitParameter("productionMode", "
false");
        // Context path of the application.
        context.setContextPath("");
        // Exploded war or not.
        context.setExtractWAR(false);
        context.setTempDirectory(tempDir);

        // It pulls the respective config from the
VaadinServlet.
        context.addServlet(VaadinServlet.class, "/*");

        context.setAttribute(
"org.eclipse.jetty.server.webapp.ContainerIncludeJarPatte
rn", ".*");
```

```
        context.setParentLoaderPriority(true);
        server.setHandler(context);

        // This add jars to the jetty classpath in a
certain syntax and the pattern makes sure to load all of
them.
        List<Resource> resourceList = new ArrayList<>();
        for (String entry : ClassPathHelper
.getAllClassPathEntries()) {
            File file = new File(entry);
            if (entry.endsWith(".jar")) {
                resourceList.add(Resource.newResource(
"jar:" + file.toURI().toURL() + "!/"));
            } else {
                resourceList.add(Resource.newResource
(entry));
            }
        }

        // It adds the web application resources. Styles,
client-side components, ...
        resourceList.add(Resource.newResource(
"./src/main/webapp"));
        // The base resource is where jetty serves its
static content from.
        context.setBaseResource(new ResourceCollection
(resourceList.toArray(new Resource[0])));

        server.start();
        server.join();
    }
}
```

This programmatically configuration requires to add extra dependencies to the `pom.xml`.

pom.xml

```
<dependency>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-project</artifactId>
    <version>${project.version}</version>
</dependency>
<dependency>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-webapp</artifactId>
    <version>${jetty.version}</version>
</dependency>
```

NOTE Depending of Jetty’s configuration, it could require additional dependencies, such as: `jetty-annotations` ,`jetty-continuation`,`javax-websocket-server-impl`, ... For more information about jetty, please consult Jetty Documentation[88].

17.3.2. Standalone Jetty

When the application has to be deployed on a server, it is necessary to generate a WAR file or an exploded directory with the application in it.

It is possible to change the name of the WAR file and exploded directory specifying the `finalName`:

pom.xml

```
<build>
    <finalName>application</finalName>
    ...
</build>
```

Deploying by Copying WAR

The easiest way to deploy a web application on a Jetty server is probably by copying the WAR file into the `webapps` directory of Jetty.

The WAR file can be generated executing the following Maven goal:

```
mvn package -Pproduction
```

NOTE: The `production` Maven profile performs a number of tasks to make sure the application performs optimally in production. For more information about the process, please consult Taking your Application into Production.

After copying the WAR file into the `webapps` directory, Jetty can be started by navigating to Jetty's folder and running the command:

```
`java -jar start.jar`
```

Deploying by Copying exploded directory

An exploded directory is a folder containing the unzipped (exploded) contents and all the application files. It is actually an extracted WAR file.

`mvn package` creates the exploded directory before creating the WAR file.

NOTE: The WAR file and the exploded directory can be found with the same name in the `target` directory.

Deploying Using Context File

Jetty web server offers the possibility of deploying a web archive located anywhere in the file system by creating a context file for it.

jetty-app.xml

```
<?xml version="1.0"  encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD
Configure//EN"
  "http://www.eclipse.org/jetty/configure.dtd">
<Configure class="org.eclipse.jetty.webapp.WebAppContext
">
    <Set name="contextPath">/jetty</Set>
    <Set name="war">absolute/path/to/jetty-app.war</Set>
</Configure>
```

17.3.3. Spring Boot

When the Vaadin Flow application is using Spring Boot, it requires an additional configuration for several aspects of the application.

One example of this is `urlMapping`:

```
vaadin.urlMapping=/my_mapping/*
```

An additional Servlet is required to handle the frontend resources for non-root servlets, such as /my_mapping/*. The servlet can be defined in your application class, see here for an example[89].

For more information about Spring configuration, please consult the Vaadin Spring configuration guide.

----

[86] https://caniuse.com/#search=let
[87] https://vaadin.com/docs/v13/flow/production/tutorial-production-mode-customising.html
[88] https://wiki.eclipse.org/Jetty
[89] https://raw.githubusercontent.com/vaadin/flow-and-components-documentation/master/tutorial-servlet-spring-boot/src/main/java/org/vaadin/tutorial/spring/Application.java

18. OSGi Support

18.1. Vaadin OSGi Support

Vaadin applications can be deployed on an OSGi compatible servlet container.

An OSGi application typically consists of multiple bundles that can be deployed separately.

To deploy Vaadin applications as OSGi bundles, static resources must be published using the appropriate APIs.

The application is typically packaged as a JAR file, and needs to have a valid OSGi bundle manifest which can be created e.g. by the `bnd-maven-plugin` or Apache Felix `maven-bundle-plugin`. All the dependencies of the application should be available as OSGi bundles.

18.1.1. Minimal Vaadin Project For OSGi

Vaadin application for OSGi should be a valid bundle, i.e. it should be packaged as a `.jar` file, and it should have a proper OSGi manifest inside. The easiest way to convert regular maven-based Vaadin application into a valid OSGi bundle consists of five steps:

- Change packaging type to `jar` in your `pom.xml`:

```
<packaging>jar</packaging>
```

- Change the scope for all vaadin dependencies from default to `provided`, like this:

```
    <dependency>
        <groupId>com.vaadin</groupId>
        <artifactId>vaadin-core</artifactId>
        <scope>provided</scope>
    </dependency>
```

- Add OSGi-related dependencies to the project

```
    <groupId>com.vaadin</groupId>
        <artifactId>flow-osgi</artifactId>
        <version>${flow.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.osgi</groupId>
        <artifactId>osgi.core</artifactId>
        <version>6.0.0</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.osgi</groupId>
        <artifactId>osgi.annotation</artifactId>
        <version>6.0.1</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.osgi</groupId>
        <artifactId>osgi.cmpn</artifactId>
        <version>6.0.0</version>
        <scope>provided</scope>
    </dependency>
```

- Setup necessary plugins for building the project:

```
 <build>
    <plugins>
        <plugin>
            <groupId>biz.aQute.bnd</groupId>
            <artifactId>bnd-maven-plugin</artifactId>
            <version>3.3.0</version>
            <executions>
                <execution>
                    <goals>
                        <goal>bnd-process</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>3.0.2</version>
            <configuration>
                <archive>

<manifestFile>${project.build.outputDirectory}/META-
INF/MANIFEST.MF</manifestFile>
                </archive>
            </configuration>
        </plugin>
        ...
    </plugins>
</build>
```

- Add bundle script (`bnd.bnd`) into the project root folder:

```
Bundle-Name: ${project.name}
Bundle-Version: ${project.version}
Bundle-SymbolicName:
${project.groupId}.${project.artifactId}
Export-Package: com.example.osgi.myapplication
Import-Package: *
Vaadin-OSGi-Extender: true
```

NOTE The last line in the manifest tells Vaadin OSGi integration to scan all classes in the bundle and discover routes.

18.1.2. Publishing a Servlet With OSGi

It's a developer responsibility to register a `VaadinServlet` in the servlet container (inside OSGi container). There are many ways to do it. One way is to use HTTP Whiteboard specification.

```
@Component(immediate = true)
public class VaadinServletRegistration {

    private static class FixedVaadinServlet extends
VaadinServlet {
        @Override
        public void init(ServletConfig servletConfig)
throws ServletException {
            super.init(servletConfig);

            getService().setClassLoader(getClass()
.getClassLoader());
        }
    }

    @Activate
    void activate(BundleContext ctx) {
        Hashtable<String, Object> properties = new
Hashtable<>();
        properties.put(
                HttpWhiteboardConstants
.HTTP_WHITEBOARD_SERVLET_ASYNC_SUPPORTED,
                true);
        properties.put(HttpWhiteboardConstants
.HTTP_WHITEBOARD_SERVLET_PATTERN,
                "/*");
        ctx.registerService(Servlet.class, new
FixedVaadinServlet(),
                properties);
    }

}
```

NOTE

`FixedVaadinServlet` class is used here as a workaround for the Classloader bug[90]. Once it's fixed there will be no need in it.

NOTE When you have more than one bundle created by Vaadin, note that you should not have multiple □□□`VaadinServlet` registrations with the same servlet pattern. So, you should either use a unique pattern for each bundle or create `VaadinServlet` in only one bundle. In the latter case, keep in mind that for the other bundles to work, it is required that the bundle containing the servlet is active.

18.1.3. Publishing Static Resources With OSGi

If your project has resources which are supposed to be available as static web resources then you should register them. In case you are using standalone servlet container you are usually using a `webapp` folder which is configured to be a static web resources folder for the web server. But there is no any dedicated `webapp` folder for OSGi bundles. Instead you should register your resource via the way provided by Vaadin OSGi integration. To do that implement either `OsgiVaadinStaticResource` or `OsgiVaadinContributor` as an OSGi service. Here the resource packaged in the jar file with `/META-INF/resources/frontend/my-component.html` is registered to be available by URL `"http://localhost:8080/frontend/my-component.html"`:

```
@Component
public class MyComponentResource implements
OsgiVaadinStaticResource {

    public String getPath(){
        return "/META-INF/resources/frontend/my-
component.html";
    }

    public String getAlias(){
        return "/frontend/my-component.html";
    }

}
```

18.1.4. Classes discovering

Vaadin discovers a number of classes to delegate them some functionality. E.g. classes annotated with `@Route` annotation are used in the routing functionality (see Defining Routes with @Route). There are many other cases which requires classes discovering functionality (see also Router Exception Handling, Creating PWA with Flow). It doesn't happen out of the box in OSGi container for every bundle. To avoid scanning all classes in all bundles Vaadin uses `Vaadin-OSGi-Extender` manifest header as a marker for those bundles that needs to be scanned. So if you have a bundle which contains routes or other classes whose functionality relies on inheritance or annotation presence you should mark this bundle using `Vaadin-OSGi-Extender` manifest header (so normally every Vaadin application bundle should have this manifest header otherwise routes declared in this bundle won't be discovered):

```
....
Export-Package: com.example.osgi.myapplication
Import-Package: *
Vaadin-OSGi-Extender: true
....
```

18.1.5. Deployment to OSGi container.

In order to have your application running under OSGi container, you need to have Vaadin Flow bundles deployed, and then the application bundle can be deployed and started. Please note that there are many transitive dependencies which are also need to be deployed. Bundle won't be activated if all its dependencies are not deployed and activated (it might be that some OSGi containers may deploy transitive dependencies along with the bundle deployment). Here is a minimal list of required Vaadin Flow bundles:

- `flow-server-X.Y.Z.jar`
- `flow-client-X.Y.Z.jar`
- `flow-html-components-X.Y.Z.jar`
- `flow-data-X.Y.Z.jar`
- `flow-osgi-X.Y.Z.jar`

This is not a full list of all required bundles. The full list is too long and may vary due to transitive dependencies. Here are some of the required external dependencies (the versions are omitted):

- `jsoup`
- `gentyref-x.y.z.vaadin1.jar`
- `gwt-elemental-x.y.z.vaadin2.jar`

- `ph-css`
- ....

Please note that some of the dependencies are repackaged by Vaadin because original jars are not OSGi compatible (like `gwt-elemental`). Other dependencies require some OSGi features which needs to be deployed at runtime but they don't depend on them during compilation. This is the case with `ph-css` bundle. It depends on `ph-commons` (which should be deployed also of course) but the latter bundle requires `ServiceLoader` OSGi implementation. You will need to deploy the bundle which contains this implementation suitable for your OSGi container. Also Vaadin OSGi support uses OSGi Compendium API (which allows registering an OSGi service using declarative services annotations). If your OSGI container doesn't have it out of the box, you have to deploy an implementation bundle to support the Compendium API.

NOTE There exists an OSGi base starter project that is ready to use and it declares all bundles which needs to be deployed to the OSGi container as provided dependencies in the dedicated profile. Those bundles are copied into the specific folder using `maven-dependency-plugin` and auto-deployed from there. As a result all required bundles are deployed to the OSGi container. See https://github.com/vaadin/base-starter-flow-osgi.

In your project you will most likely want to use some ready-made Vaadin components like Vaadin Button. In this case you should deploy `vaadin-button-flow` bundle as a dependency. Please note that all Vaadin Flow components are OSGi compatible bundles but they depend on webjars with the client side web component resources which are not OSGi compatible unfortunately. See the next section about this topic.

18.1.6. Make webjar resource working in OSGi.

Normally every Flow component has a client side part which is distributed as a webjar. Webjars contain only web resources and they are not OSGi compatible. It means that webjar is not a bundle and cannot be deployed to an OSGi container. As a result you won't get Flow component working without additional setup. We suggest a solution for repackaging webjar resources into the application bundle. Here is the code snippet of the project configuration which we use to repackage the webjars:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <executions>
        <execution>
            <id>unpack-dependencies</id>
            <phase>generate-resources</phase>
            <goals>
                <goal>unpack-dependencies</goal>
            </goals>
            <configuration>
                <includes>**/webjars/**</includes>
            </configuration>
        </execution>
    </executions>
</plugin>
<plugin>
    <artifactId>maven-antrun-plugin</artifactId>
    <version>1.7</version>
    <executions>
        <execution>
            <id>copy-frontend</id>
            <phase>generate-resources</phase>
            <configuration>
                <tasks>
                    <mkdir
                        dir=
"${project.build.directory}/generated-
resources/frontend/bower_components"></mkdir>
                    <copy
```

```
                        todir=
"${project.build.directory}/generated-
resources/frontend/bower_components">
                        <fileset
                            dir=
"${project.build.directory}/dependency/META-
INF/resources/webjars/" />
                    </copy>
                </tasks>
            </configuration>
            <goals>
                <goal>run</goal>
            </goals>
        </execution>
    </executions>
</plugin>
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <version>3.0.0</version>
    <executions>
        <execution>
            <id>add-resource</id>
            <phase>generate-resources</phase>
            <goals>
                <goal>add-resource</goal>
            </goals>
            <configuration>
                <resources>
                    <resource>

<directory>${project.build.directory}/generated-
resources</directory>
                        <targetPath></targetPath>
                    </resource>
                </resources>
            </configuration>
        </execution>
    </executions>
</plugin>
```

This code snippet unpacks all dependencies and extracts `webjars` folder from them. Then it copies the resulting resources to the dedicated folder to create the appropriate structure for them and the folder is added as a resource folder. In the result the folder will be packaged in the jar archive and the resources will be available in the jar bundle starting from the archive root. It makes them automatically available as web resources.

> TIP
> This is done in our OSGi base starter project https://github.com/vaadin/base-starter-flow-osgi. You may check out the code.

18.2. Create OSGi compatible components

If you want to create an OSGi compatible component in a separate bundle then you should be aware about several aspects: * Making OSGi compatible jar * Whether you have classes which need to be discovered by Vaadin * Static resource registration

All those aspects are already shortly covered in the basic tutorial Vaadin OSGi Support since there are common parts, but we'll go through them here in more depth and in regards to component bundle creation.

In all simplicity, an OSGi compatible bundle is just a jar file with the proper manifest file.

18.2.1. Making OSGi compatible jar

Every OSGi compatible jar should have a proper manifest file which is located by the path `/META-INF/MANIFEST.MF`.

You can hardcode this file or use a maven plugin which generates the manifest for you from a template file. Here is the code snippet for your `pom.xml` which generates the `/META-INF/MANIFEST.MF` file and tells maven Jar plugin to use this manifest.

```
 <build>
    <plugins>
        <plugin>
            <groupId>biz.aQute.bnd</groupId>
            <artifactId>bnd-maven-plugin</artifactId>
            <version>3.3.0</version>
            <executions>
                <execution>
                    <goals>
                        <goal>bnd-process</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>3.0.2</version>
            <configuration>
                <archive>

<manifestFile>${project.build.outputDirectory}/META-
INF/MANIFEST.MF</manifestFile>
                </archive>
            </configuration>
        </plugin>
    </plugins>
</build>
```

It requires a `bnd.bnd` file which needs to be located in your project root folder:

```
Bundle-Name: ${project.name}
Bundle-Version: ${project.version}
Bundle-SymbolicName:
${project.groupId}.${project.artifactId}
Export-Package: com.example.mycomponent
Import-Package: *
```

18.2.2. Making Java classes being discovered by Flow

If you want to extend somehow a Vaadin application behavior in your bundle then you should mark your bundle using `Vaadin-OSGi-Extender` manifest header. E.g. you may want to provide a `HasErrorParameter` implementation class which handles your own exception type (or you are making an extension which has some routes). You bundle won't be scanned for such classes if there is no `Vaadin-OSGi-Extender` header. So you should include this header if there are extension classes:

```
...
Export-Package: com.example.mycomponent
Import-Package: *
Vaadin-OSGi-Extender: true
...
```

18.2.3. Static resource registration

Your component may requires some static files which should be available via HTTP. E.g. this component relies on JavaScript file:

```
@JavaScript("src/my-component.js")
public class MyCoponent extends Div {

}
```

You normally use standard locations for static resources since your jar should work also in non OSGi environment. So let’s assume your resource folder is `META-INF/resources`.

NOTE If you use maven then all resources are located inside `src/main/resources` folder. So the full path of the static resources directory is `src/main/resources/META-INF/resources`.

This resource should be registered via the way provided by Vaadin OSGi integration. We are using `OsgiVaadinStaticResource` service interface since there is only one resource. The resource path in the jar file is `/META-INF/resources/frontend/src/my-component.js` and it should be registered to be available by URI `/frontend/src/my-component.js` (so the full URL e.g. is `"http://localhost:8080/frontend/src/my-component.js"`):

```
@Component
public class MyComponentResource implements
OsgiVaadinStaticResource {

    public String getPath(){
        return "/META-INF/resources/frontend/src/my-
component.js";
    }

    public String getAlias(){
        return "/frontend/src/my-component.js";
    }

}
```

NOTE

Your component project will most likely use webjars (see Integrating a Web Component[91]). You should be aware of the fact that webjars are not OSGi compatible. So webjar archive is not an OSGi bundle and cannot be deployed to an OSGi container. So resources in webjar won't be available via HTTP out of the box for you. Here Vaadin OSGi Support we have a working suggestion for this issue via repackaging.

----

[90] https://github.com/vaadin/flow/issues/4367
[91] https://vaadin.com/docs/flow/web-components/integrating-a-web-component.html

19. Migrating from Vaadin 8 to Vaadin 10

19.1. Migrating from Vaadin 8 to Vaadin platform

NOTE This migration documentation has been revised for Vaadin 14.0 release. It is recommended to migrate to Vaadin 14 since it is the latest LTS and has more features available than versions 10-13.

Vaadin platform stands for Vaadin versions 10 and later, and is a continuum from Vaadin 8. While the core concepts, architecture and programming model stay the same, the platform is still a big leap forward:

In Vaadin platform the provided UI components are based on the Web Components standards. The components are written with the Polymer library, instead of GWT. Building a component based on the Web Components standards makes it possible to reuse it with any modern web framework, instead of just limiting it to Vaadin or GWT. Vaadin Flow is the Java framework in the platform, and it is a total rewrite of the Vaadin Framework. It makes all the components available for server side Java developers. It gives a much better toolset for building any reusable UI components, not just Web Components, and does not force you to use GWT for that.

Switching from Vaadin 8 to Vaadin platform could be considered as switching from a modern car to a flying car - the use cases are the same, the features are mostly the same, except the new technology brings more capabilities and potential for the future. Some features may have been removed to make way for new things.

19.1.1. Migration Strategies

There are different migration strategies for Vaadin 8 applications:

- Staying with Vaadin Eight - it is Great! And supported at least until 2022.
- Using the upcoming Multiplatform Runtime from the Prime subscription to run views or components from a V8 app inside Vaadin platform for
  - Migrating the application bit by bit
  - Extending the application with new parts implemented with Vaadin platform
- Transform an existing application to Vaadin platform
- Fresh start by redesigning an application for Vaadin platform

The latest LTS version for the platform is version 14 and it is recommended to use that instead of versions 10-13. The migration strategies are described in more detail in the next chapter.

The easiest and fastest way to understand what migrating to Vaadin platform means to you is to use our migration assessment service[92]. What's the most suitable strategy in your case and what's the needed effort? All of these questions get answered by our experts.

In addition, migration from V8 to Vaadin platform is illustrated in a simple example here. The application that is chosen as an example is Bookstore Starter and the whole migration story is described step-by-step.

19.1.2. The More Things Change The More They Stay The Same

Most of the migration documentation covers the features that have changed or been removed in Vaadin platform. But not everything has changed and at the core it is still the same product. The following concepts are not covered, since they have **stayed the same** between platform and 8:

- Using high level APIs to compose UIs with ready-made components in Java on the server side
- Stateful server-side architecture
- Automated client-server communication using request-response or server push
- Minimum server requirements: Java 8 and Servlet 3.0
- Minimum browsers supported: Internet Explorer 11 and evergreen browsers
- Data Binding API with `Binder` and `DataProvider`

A major part of any application, the API for binding data to components, was modernized in Vaadin 8 with the introduction of `Binder` and `DataProvider`. These same concepts are used in platform and the API is mostly the same.

In Vaadin platform, building UIs is still the same. By using ready made components to make beautiful apps and it is super easy by using high level type-safe Java APIs. But now the components have been redesigned to provide better end user experience, but at the same time the initial set of components is smaller than in V8.§ More components will be added later on. Creating your own components is a lot easier than before.

The following is the list of things that have changed, which does not list all the new features of Vaadin platform, but the differences with V8:

- Differences between Vaadin platform and Vaadin 8 Applications
- Routing and Navigation
- Components in Vaadin platform
  - Component Set
  - Basic Component Features
  - Layouts in Vaadin platform
- Themes and Theming Applications
- Add-ons, Integrations and Tools

19.2. Migration Strategies

For any existing Vaadin 8 or 7 application it might be desirable to extend the lifetime of the application and make it possible to add new features using Vaadin 14. There are several migration strategies to consider as the need for migration is always application and business specific.

19.2.1. Use Vaadin Eight - It Is Great!

You can stay with Vaadin 8. We are going to maintain it at least until 2022. That gives you the option to plan your migration longer term, and consider what is the needed lifetime for your application. You should look at the needs of your users and how those needs and the technologies used evolve during that time.

Internet Explorer 11 support is still a critical thing for many applications, and for IE11 Vaadin 8 might offer better performance than Vaadin platform since polyfills are needed with IE 11. During the guaranteed lifetime of Vaadin 8 the usage of IE11 should decrease significantly from what it is now.

During the remaining years of Vaadin 8 support, the quarterly Vaadin platform releases will bring more and more features that you can start using when you eventually decide to migrate to latest Vaadin LTS version (currently 14) or start building a new application with a fresh design.

19.2.2. Using the Multiplatform Runtime for Running V8 Application Inside V14

Multiplatform Runtime makes it possible to run views and components from your Vaadin 7 and 8 application inside Vaadin 14. This helps you migrate it to version 14 bit by bit, or just incrementally add V10 features while also including existing application features as-is.

Using the runtime, you will be able to get your V7 or V8 App running inside a V14 App after couple of quick steps. Then you can migrate the necessary parts while having a working application after each step.

Multiplatform runtime is part of the platform and has LTS support for using it with Vaadin 14. For more information about MPR, see the documentation for it[93].

19.2.3. Transform an Existing Application To Vaadin 14

Transforming a Vaadin 8 app to Vaadin 14 by reusing backend interaction, business logic, and as much as possible of UI logic, but updating the UI to use brand new components. When planning this migration, you should at a minimum read through all of this documentation to understand the differences and verify what features are available and what you need to redesign. One big benefit is that the Data APIs, `Binder` and `DataProvider`, work the same way in V14, so you won't have to rework your backend integration.

19.2.4. Fresh Start by Redesigning Application for Vaadin 14

Sometimes the best approach is to make a fresh start - this way you will be able to focus on using new V14 features to deliver value to your users, instead of carrying over legacy code. As mentioned in previous topics, there is still probably some opportunities to reuse existing solutions related to integration to your business logic and backend.

19.2.5. Migration Assessment Service

To help you understand what a migration to Vaadin 14 means to you, Vaadin offers an assessment service led by Vaadin experts. During the assessment, our experts will gain an understanding of your organization's objectives and concerns, and analyze your application code. Read more about the service[94].

You can also get us to do the migration for you.

19.3. Differences Between Vaadin 10+ and V8 Applications

19.3.1. The UI Class is Different and it is Optional

A subcass of `UI` is not needed anymore in Vaadin 10+ applications. It is still there, but it is not *the starting point* of the application anymore, and you don't need to define your custom `UI`. Adding the content to show is handled by `Router` which is introduced in the next chapter.

Most things you have previously been done by subclassing `UI` can be achieved in alternative ways that are based on composition instead of inheritance, or by configuring things through a router layout class that is used by all views in the application. These alternatives are covered in this documentation, but if some information is missing, please create an issue[95] so we may add the necessary information.

In Vaadin 8 and 7, the `UI` referenced a `<div>` element which was a child of the `<body>` in the DOM. In Vaadin platform, the `UI` is instead directly connected to to the `<body>` element.

The API in the `UI` class in Vaadin 10+ has gone through a clean-up and some API that is not meant for application developers to access has been moved to internal classes. For using API that is still in `UI`, the same `UI.getCurrent()` call can still be used to obtain a reference to the UI for the currently active request.

The `@PreserveOnRefresh` annotation in Vaadin 8 and 7 was added to the `UI` subclass. In Vaadin 10+, the `@PreserveOnRefresh` annotation should be added to the view class or router layout class. For details see preserving UI state on refresh.

While migrating, you still might want to create your own `UI`, and you can see an example for that in the next topic.

19.3.2. The Servlet Definition is Optional

Similarly as the `UI`, the servlet definition is optional in Vaadin 10+. And the reason is also the same, the new Routing API, which is introduced in the next chapter. By default the servlet is automatically mapped to the root context path. You can of course still configure the servlet yourself, and it happens the same way as previously:

```
@WebServlet(urlPatterns = "/*", name = "myservlet",
asyncSupported = true,
// Example on initialization parameter configuration
initParams = {
        @WebInitParam(name = "frontend.url.es6", value =
"http://mydomain.com/es6/"),
        @WebInitParam(name = "frontend.url.es5", value =
"http://mydomain.com/es5/") })
// The UI configuration is optional
@VaadinServletConfiguration(ui = MyUI.class,
productionMode = false)
public class MyServlet extends VaadinServlet {
}

// this is not necessary anymore, but might help you get
started with migration
public class MyUI extends UI {
    protected void init(VaadinRequest request) {
        // do initial steps here.
        // previously routing
    }
}
```

19.3.3. Single Step Bootstrap and No UIProvider

In Vaadin 8, the application bootstrap happened in two phases. The initial page response only contained code to obtain more data related to the browser, data which was not available in the initial request. Based on this, the correct UI would be created. This was a good option back then since mobile devices capabilities required completely own client engine and UI to deliver the best possible end user experience.

This is not necessary anymore, and in Vaadin 10+ the UI content is delivered on the first response and the application is bootstrapped without further network activity. This means that `UIProvider` has become obsolete.

19.3.4. Modifying the Bootstrap Page

If you had previously customized the initial response with `BootstrapListener`, this tutorial[96] show the new and simpler way of customizing the initial page response using `PageConfigurator` or specific annotations on the application's main layout.

`BoostrapListener` is still there and can be registered using a `VaadinServiceInitListener`, as shown by this tutorial[97].

19.3.5. Configuring Server Push

As a custom `UI` class is no longer needed, you can configure the used `PushMode` for your app in the main layout of your application. Please see the Server Push Configuration tutorial [98] for more info.

19.3.6. Loading Indicator Configuration

The default loading indicator is the same as in Vaadin 8 Valo Theme. If you want to customize it or disable it, you should see the loading indicator documentation[99].

19.3.7. Similarities and Package Naming

Some of the Java code in Vaadin 10+ is directly inherited from Vaadin 8. Even in these cases, the package names are however changed. This is because we want to make it possible to use Vaadin 8 components and views without classpath conflicts inside Vaadin 10+ using the multiplatform tool[100]. Thus the package names for Vaadin 10+ contain flow that separates them from the Vaadin 8 packages.

19.4. Routing and Navigation

Routing and navigation are core concepts for any web application or site. In Vaadin 10 and later this has been completely reinvented. In Vaadin 8 and 7 `Navigator` only supported single-level navigation, had limited support for parameters and did not support HTML5 History API until Vaadin 8.2.

This document only outlines the core concept and how it differs from the old `Navigator`. To get the best picture on the capabilities of the new `Router`, you should visit the Router documentation.

The `Router` API allows building robust and complex application structures with hierarchies, error handling and view access control by using lifecycle events.

Unlike in Vaadin 8 where route configuration was made for each UI instance separately, in Vaadin 14 the routes are configured declaratively on each navigation target and are the same for the entire application:

```
@Route(value = "company", layout = MainLayout.class)
public class CompanyView extends Composite<Div> {
     // Implementation omitted
}
```

Thus, when migrating from using `Navigator`, any `View` from Vaadin 8 can be migrated to Vaadin 14 by removing the registration from the now optional `UI` class and instead applying the `@Route` annotation the class. **Note** that there is no `View` interface in Vaadin 14, but instead the class must be a `Component`!

To receive an event when the user navigates to or from a view, make it implement one of `BeforeEnterObserver`, `BeforeLeaveObserver` or `AfterNavigationObserver` interfaces instead of the `enter` and `beforeLeave` methods from `View` in Vaadin 8. `BeforeLeaveEvent.postpone()` can be used to postpone or cancel navigation to achieve the same results as selectively calling `ViewBeforeLeaveEvent.navigate()` in Vaadin 8.

Instead of manually constructing URLs and using `ViewChangeEvent.getParameters()` to find parameter values, you can use `UI.navigate(NavigationTargetClass.class)` and have your view implement the `HasUrlParameter` interface.

The `ViewDisplay` concept has been replaced in Vaadin 14 with the `RouterLayout` interface, but now it is possible to have nested hierarchies, by using the `@ParentLayout` annotation to set one `RouterLayout` class as the parent of another.

There is no `ViewProvider` in Vaadin 14 as it is not needed.

With the HTML5 History API, it is possible to have deep-linking and use proper navigation state and parameters for the navigation targets. Optional parameters are also supported since Vaadin 10. However, it is no longer possible to use the fragment style (`#!`) navigation state from Vaadin 8 as the fragment is not intended to be used on the server side at all, but just be a browser feature for navigation inside a page.

It is very much recommended to take a look at the router documentation to get full understanding on how to structure your Vaadin 14 application.

19.5. Components in Vaadin platform

19.5.1. Component Set

While all the components have been rebuilt based on Web Components, there are some components that don't yet have a replacement with server side Java API. The ones that have, might have gone through some changes. Some older features might have been removed.

TIP Watch the Vaadin 10+: Intro[101] free training video to learn more about the Vaadin platform terminology and what Vaadin components are.

The following table lists the existing Vaadin 8 components and their direct replacements in the Vaadin platform. Note that the replacement component might not have 1-1 feature parity. If no replacement is yet available, current plans or options for replacement is mentioned.

NOTE The mentioned components are available starting from Vaadin 10 or from the specifically mentioned platform version.

Table 1. Comparison Matrix

V8	Vaadin platform	Details
AbsoluteLayout	-	Not planned. Similar functionality in Vaadin platform can be achieved using eg. `<div>`[102] elements and CSS positioning.
Accordion	Accordion (V13)	Demo[103]
Audio	-	Not planned. Use the native `<audio>` element.
Button	Button	Demo[104]
BrowserFrame	IFrame (V13)	
Checkbox	Checkbox	Demo[105]
CheckBoxGroup	CheckBoxGroup (V12)	Demo[106]
ColorPicker	-	Not planned. See vaadin.com/directory[107] for alternatives. `<input type="color">` is supported in some browsers.
ComboBox	ComboBox	Demo[108].

V8	Vaadin platform	Details
ContextMenu (official add-on)	ContextMenu (V12)	Demo[109]
CssLayout	Div & FlexLayout	More details later in this chapter
CustomComponent	Composite	Tutorial[110]
CustomField	AbstractField, AbstractCompositeField or AbstractSinglePropertyField	Tutorial
CustomLayout	HTML or PolymerTemplate	See notes below.
DateField	DatePicker	Demo[111]
DateTimeField	DatePicker (V10) & TimePicker (V12)	Planned for 2019. Possible by combining `DatePicker` and `TimePicker`. `<input type="datetime">` is supported in some browsers.

V8	Vaadin platform	Details
Embedded	-	Use `<object>` directly via `@Tag("object")` and `Element` API
FormLayout	FormLayout	Demo[112]
Grid	Grid	Demo[113], Tutorial
GridLayout	-	Not planned. See detailed notes about replacement alternatives below.
HorizontalLayout	HorizontalLayout	Demo, more details later in this chapter
HorizontalSplitPanel	SplitLayout	Demo[114]
Image	Image	-
InlineDateField	-	No inline option of DatePicker available nor planned
InlineDateTimeField	-	Not planned
Label	Text or Span	There is also a `Label` component based on the `<label>` element, and should therefore only be used for form field labels.
Link	Anchor	-
ListSelect	ListBox	Demo[115]

V8	Vaadin platform	Details
LoginForm	LoginForm or LoginOverlay (V13)	Demo[116]
MenuBar	MenuBar (V14)	Demo[117]
NativeButton	NativeButton	-
NativeSelect	Select (V13)	Demo[118]
Notification	Notification	Demo[119]
Panel	Planned H2/2019: VerticalLayout & HorizontalLayout	`setScrollable` API for VL & HL will be available in a minor release for 14 in 2019
PasswordField	PasswordField	Demo[120]
PopupView	-	Planned for 2019. Can be made by combining `Button` and `ContextMenu` (V12).
ProgressBar	ProgressBar	Demo[121]
RadioButtonGroup	RadioButtonGroup	Demo[122]
RichTextArea	RichTextEditor (V13)	Demo[123]

V8	Vaadin platform	Details
Slider	-	Not planned. There are Web Components available, check vaadin.com/directory[124]. You can also use DOM API and `<input type=”range”>`
TabSheet	Tabs	Demo[125]
TextArea	TextArea	Demo[126]
TextField	TextField	Demo[127]
Tree	-	Planned, no timeline yet. Go vote issue[128] if you need it.
TreeGrid	TreeGrid (V12)	Demo[129]
TwinColSelect	-	Not planned. Can be built as a composite using `ListBox` and `Button`.
Video	-	Not planned. Can directly use the native `<video>` element.
VerticalLayout	VerticalLayout	Demo, more details later in this chapter
VerticalSplitPanel	SplitLayout	Demo[130]
UI	UI	Not mandatory in 10+. Replaced with root layout and `PageConfigurator`.
Upload	Upload	Demo[131]

V8	Vaadin platform	Details
Window	Dialog	Demo[132] Note that there is only limited support due to missing eg. minimize / maximize feature.

For any missing components, you should first look for alternatives in vaadin.com/directory[133]. It shows both Vaadin platform add-ons with Java API and web components that can be integrated to Java.

For the components that are available in Vaadin platform, you can browse vaadin.com/components/browse[134] for features and examples.

19.5.2. Basic Component Features

The way components are structured has been renewed in Vaadin platform. While the basics stay the same, backwards compatibility has been discarded in favor of optimizing for current and future usage.

In Vaadin 8, there was a large and complex class hierarchy for components, and the `Component` interface already declared a large set of API that components were supposed to support. This meant that almost every time, the component had to extend at least `AbstractComponent` so that they would not need to implement all the methods from the interface. That would mean that there would be a lot of API in the actual component, some of which made no sense in all cases.

In Vaadin Flow the `Component` is an abstract class, with only the minimal set of API exposed. For the component implementations, it is up to them to pick up pieces of API as mixin interfaces that provide default implementations.

Component is Lightweight and it Maps to an Element

Every Vaadin Flow component always maps to one root element in the server-side DOM representation. A component can contain multiple components or elements inside it. The component is the high level API for application developers to compose UIs efficiently. The Element API is the low level API used to build components. The Element API makes it possible to modify the DOM easily from the server side.

If you look up the `Component` class in Vaadin Flow, you notice that there is no API even for setting the width or height of the component! For your own components, add the API by implementing the `HasSize` mixin interface, which has default implementations for e.g. `setWidth(String width)` and `setHeight(String height)`. So by adding two words of code you can achieve full sizing capabilities for your components. See the Creating A Simple Component Using the Element API[135] tutorial for more info.

All Components Don't Have Captions or Icons

In Vaadin 8 every component had a caption. The caption was usually shown next to the component, based on the parent layout's caption handling implementation. The caption could optionally be rendered with an icon. Some layouts didn't support showing captions and/or icons.

In Vaadin platform there is no universal caption concept anymore. Some components might have a similar feature, but that it is always component specific. Usually that API is `setLabel(String label)` instead of `setCaption`. Some layouts, such as `FormLayout`, also support showing a label text or component for each child component.

In other cases, you can create your own `Span` or `Text` component to contain the caption text and add it to the parent layout alongside the component.

Adding icons is possible, it is just HTML5 after all. But as with caption there is no universal support for that.

setEnabled(boolean enabled) is Still a Server Side Security Feature

In Vaadin 10+, the `setEnabled` method is specific to components marked with the `HasEnabled` mixin interface (which comes also with `HasValue`, `HasComponents`, and `Focusable`). When a component is disabled, by default, any property changes and DOM events coming from the client side are ignored. However, it is possible to whitelist some properties and events to be allowed if necessary.

The disabled state is automatically cascaded to child components it is up to the component to change the disabled UX to mark the component as "not-working" when it has been disabled. Changes from the client are still always blocked for disabled components even if the component isn’t implemented to appear disabled. All relevant Vaadin components change their looks when disabled.

Read the Component Enabled State[136] tutorial for more details.

setReadOnly(boolean readOnly) is Component Specific and Works Differently

In Flow the `setReadOnly(boolean readOnly)` method is specific to components accepting user input by implementing `HasValue`.

For a readonly component, changes from the client will not make the return value of `getValue()` to change nor fire any `ValueChangeEvent`. Most components will also update their visual status to indicate to the user that the value cannot be changed.

Tooltips are Component Specific

In Vaadin 8 the legacy framework made it possible to show a tooltip for any component if the user hovered the mouse on top of the component. In Vaadin platform there is no automatic way for this; it is a component specific feature and possible using CSS.

19.5.3. Layouts in Platform

In Vaadin 8 the layouting of components was managed by a `LayoutManager` on the client engine. This has its roots in a time when the differences between browsers were big, and the legacy Framework still supported Internet Explorer versions that worked by their own rules. Creating your own layouts was quite complex since it always required writing custom client side code with GWT.

In Vaadin platform, there is no more LayoutManager to do calculations in browser. All layouts are self-contained and mostly just rely on the HTML5 and CSS3 standards, which all modern browsers (as well as IE 11) support. Responsive layouts can be created now using the DOM API in Java on the server side.

As native browser features are used for rendering, layouts are rendered faster than in previous versions.

Core Layouts API and Creating Custom Layouts

In Vaadin platform you can create a custom layout with only server side Java code by using mixin-interfaces and the Element API. The mixin-interfaces are also the basis for the core layouts and replace a complex class hierarchy from Vaadin 8:

- `HasComponents` for simply adding components to the parent's root Element with:
  - `add(Component… component)`
  - `remove(Component… component)` & `removeAll()`
- `HasOrderedComponents` for accessing components based on index

All the core layouts except `FlexLayout` & `Div` are based on Web Components, but they still give a good example on how to create your own layouts if needed. For Element API usage, please see the Creating a Component Which Can Contain Other Components[137] tutorial.

Layout Click Listeners

There is currently no direct API exposed for this in the layouts. But if you want to, you can access the element and add a DOM event listener to it for click events. If this is a much requested API, we could make it a standard feature to the layouts. There is an enhancement issue[138] for this.

Available Layouts in Platform

HorizontalLayout & VerticalLayout

These layouts have made it easy to compose UIs. For Vaadin platform they are now based on fast native CSS rendering in browsers, instead of custom JavaScript calculations. This means that the API has been changed to match the underlying CSS concepts instead of custom names - this is also to highlight that it might not work exactly the same way as before:

- `setComponentAlignment` & `setDefaultComponentAlignement`
  - `HorizontalLayout`: `setVerticalComponentAlignment` and `setDefaultVerticalComponentAligment`
  - `VerticalLayout`: `setHorizontalComponentAlignment` and `setDefaultHorizontalComponentAligment`
  - These map to the `align-self` and `align-items` CSS property values.
- `setExpandRatio` is now `setFlexGrow`
- `expand()` sets `flex-grow` to 1
- `setMargin` is now `setPadding`
- Spacing and Padding are only available as on/off for all edges of the layout, instead of separately for top/right/bottom/left. Fine-grained control is available using CSS, e.g. `component.getElement().getStyle().set("padding-top", "20px")`

- Using `setSizeFull()`, `setHeight("100%")` or `setWidth("100%")` for any contained component will not have the same effect as before - **it will cause the component to get the full size of the parent layout, instead of full size of the slot**. Instead, leave the size undefined and `flex-grow` will take care of sizing the component.

For better understanding how to use the `setFlexGrow()` and `expand()` methods and how the *flex* layouts work, please see the Mozilla Foundation documentation on CSS flex[139].

FormLayout

`FormLayout` has been made responsive and it now supports multiple columns. Thus it also in some ways replaces the old `GridLayout`.

FlexLayout

This layout is a server side convenience API for using a `<div>` with `display: flex` and then setting the flexbox properties via Java. If you haven't already, you should introduce yourself to flexbox. It will allow you to easily build more responsive layouts.

Div AKA CssLayout

The most powerful layout of Vaadin 8 in terms of customizability is the `CssLayout`, which is just a `<div>` element in the browser. This is now also available, but it is now named to what is actually is - a `Div` element in the browser.

The `getCss` method from V8 is not available, but in Vaadin platform you can easily modify the element CSS from the server side for any component using `component.getElement().getStyle()`. This works with any layout, not only `Div`.

Replacing Existing Layouts

In addition to the options listed below, you should also see if directory[140] has add-ons available that can be used as a replacement.

AbsoluteLayout

`AbsoluteLayout` can be replaced with the `Div` component and then applying the CSS properties `position: absolute` and coordinates as top/right/bottom/left properties to the components added inside it using the Element API.

GridLayout

There is currently no direct replacement, but depending on your use case, you could replace the old `GridLayout` with either

- `Board` which is commercial and fully responsive
- `FormLayout` which now supports multiple columns
- `FlexLayout` which is powerful but requires mastering the flexbox concepts
- Nesting `HorizontalLayout` and `VerticalLayout` together
- Use `Div` together with the new CSS Grid functionality that is supported in most browsers

CustomLayout

For replacing `CustomLayout` you can just use a `Html` container component for static content. For dynamic content you can use `PolymerTemplate` with `@Id` bindings.

19.5.4. Migrating Your Own Components

One of the biggest improvements in Vaadin Flow compared to Vaadin 8 is making it possible to access and customize the DOM from server-side Java. This obsoletes many reasons for using GWT for creating components. It also means that existing custom components from V8 have to be rebuilt again. The server side API can be reused, but some changes may be needed since the class hierarchy has changed in Flow.

Simple components can be composed using existing components and the Element API. The creating components tutorials[141] have examples on this. For more complex components, with lots of client side logic or a complex DOM structure, it might be better to implement them as Web Components and provide a Java API to those.

19.6. Themes and Theming Applications

Themes define the look and feel of the Vaadin components. The built-in themes have been the base for the application specific theme. Vaadin 7 introduced the themes in Sass format and the parameterized Valo theme, which made it possible to customize the UI by tweaking the parameters.

19.6.1. New CSS Based Themes - No Sass

Since we introduced Sass as a helper to Vaadin Application theming, browsers have started to support CSS Custom Properties[142] (IE11 is polyfilled for production mode.), which brings the customizability gains from Sass to basic CSS, without the overhead of needing to compile the Sass to CSS.

Thus, Vaadin 14 itself isn't using Sass, but you can of course use it for your own application theming if you want to. You'll have to setup the sass-compiler workflow yourself, but there are Maven Plugins available for this.

None of the old themes are available for Vaadin 14. By customizing the new Vaadin 14 themes you can easily achieve the same look and feel your application you had before. The DOM is, however, different for the new components, so this is not a copy-paste step.

19.6.2. Theming in Vaadin 14

In Vaadin 14 the theming is built inside the Web Components. There is a different variant of the Web Component for each available theme. Themes cannot typically be mixed and matched, and for coherent user experience you should always use the same theme for all of the components.

The following example shows all the tricks for theming applications, covered in the next topics:

```
@Theme(Lumo.class) // the theme for Vaadin Components.
You can omit it for Lumo
@CssImport("./styles/shared-styles.css") // the
application specific styles
@Viewport("width=device-width, minimum-scale=1.0,
initial-scale=1.0, user-scalable=yes")
public class MainLayout extends Div implements
RouterLayout,
        AfterNavigationObserver, PageConfigurator {

}
```

@Theme Annotation

Vaadin needs to know which theme it should use for the components. Similarly to Vaadin 8 and previously, in Vaadin 14 this is done with a `@Theme` annotation. This should be applied on the root layout of the application. It can also be in an abstract class if you have multiple root layouts.

NOTE When no `@Theme` is used, the Lumo theme is used by default (if present in the classpath).

Instead of a magic string, you need to provide a `Class` reference to an theme class that extends `AbstractTheme`. There are two ready-made themes available for Vaadin 14: Lumo[143], which was introduced in Vaadin 10, and Material[144], which has been available since Vaadin 12. You should always use a theme, since the unthemed versions of the components are only meant as a baseline for creating a new theme from scratch!

The theme class will handle two things:

- It tells Vaadin what theme it should use for the Vaadin Components and where the files can be found

- It specifies a set of shared styles like fonts etc. that will be automatically loaded to the initial page by Vaadin for the theme.

Both Lumo and Material can be customized, see documentation for more information.

Application Theming

`@CssImport("./styles/shared-styles.css")` imports the application’s style file.

As the `@Theme` annotation only specifies the theme for the components, you should have a separate style module for the styling that only applies to the application. The recommended default is to have that inside the frontend folder, e.g. `/frontend/styles/shared-styles.css`.

You should always use this location for style files to

- have the file included in the bundle file and allow it to override default theme specific styles
- have the file automatically imported to templates when using Vaadin Designer to design UIs

You can also use and import a CSS file with the `@StyleSheet` annotation, but this is not recommended (staring from V14). Regular stylesheets are not processed by ShadyCSS, so you should avoid using custom properties or mixins in them if you want to support Internet Explorer 11.

When creating the UI by defining html templates in JavaScript modules, it makes sense to apply the theming that only applies to a specific template directly in the template and scope it to only affect that.

Specifying the Viewport and <body> element styles

In the previous Vaadin versions, the `@Viewport` styling was applied to the `UI`, but now it should be done for the root layout. Otherwise the usage has not changed from Vaadin 8:

```
@Viewport("width=device-width, minimum-scale=1.0,
initial-scale=1.0, user-scalable=yes")
```

Vaadin 14 maps the `UI` directly to the `<body>` element and gives you more fine grained control easily on what styles it gets with the `@BodyStyles` annotation. In Vaadin 14 the body has only the `margin: 0;` style applied, whereas in Vaadin 8 there are the following styles:

```
overflow: hidden;
margin: 0;
padding: 0;
border: 0;
```

19.7. Add-ons, Integrations and Tools

19.7.1. Maven Plugin

There is a Maven Plugin available for Vaadin 14. The plugin handles transpilation, minification and bundling of the front-end resources for the production version of the application. This is only necessary when you take the application into production, or want to test it with IE11.

By default there are no custom widgetsets or Sass themes that need compilation for development time in Vaadin 14. The plugin is thus not needed during development, except when testing with IE11.

19.7.2. Maven Archetypes

There are no archetypes available for Vaadin 14. However, you can find in vaadin.com/start[145] several Maven-based example applications and ready-made project bases for Vaadin 14.

19.7.3. Using Vaadin with Spring

Vaadin 14 has an integration for using it with Spring. The concept is mostly the same, but some features like the `@ViewScope` have been removed. Also there is currently no specific Spring Security support, although it can still be integrated manually. The Bakery App Starter[146] for Vaadin Flow and Spring shows an example of this.

19.7.4. CDI Support

There is a CDI Add-on[147] for easier CDI integration and to help using other Java EE features.

19.7.5. Vaadin Designer

In Vaadin 8, Designer was used to edit declarative files with a `.html` suffix. Despite the file format suffix, the declarative format was a generic XML syntax that mapped directly into a tree of Vaadin components on the server side. The XML was read by Vaadin at runtime on the server, and was never sent to the client. It's important to note that the syntax only allowed component declarations.

In Vaadin 14 the high level concept is the same. There are still "html template" files that can be edited with Designer to declaratively compose views. But as with Vaadin 14 in general, the underlying technology has completely changed from what it was in Vaadin 8. Starting from Vaadin 14, the templates are written as JavaScript modules[148], which is a part of the web standard and works natively in modern browsers. The modules define templates and are rendered by the browser, and allow using encapsulated CSS, HTML, Web Components, and JavaScript. Using this new format allows Designer to do a couple of new things. As one major improvement, any template can be rendered inside one another. Furthermore, because templates are themselves Web Components, custom components are now fully supported by the Designer as well. On the other hand, HTML as a syntax is flexible enough that Designer might not work with all templates created in other ways.

Some features of Designer are not available for all Vaadin versions. See the Release Notes[149] for an overview of the feature-level differences.

Migrating from Vaadin 8 Designs

As the underlying technology has been completely changed, Vaadin 8 designs are not compatible with Vaadin 14 applications. There are two paths to migration; either use the Multiplatform Runtime[150] (available through Prime or Enterprise subscription) to run a Vaadin 8 application inside a Vaadin 14 application, or migrate the HTML files manually. When migrating manually, the declarative component tree should be copied inside the `<template>` in a blank Vaadin 14 design, and then modified to fit the new element API's.

Version Support

The new Designer plugin will support editing both Vaadin 8 and Vaadin 14 designs. Whether you are working with

Vaadin 8 or Vaadin 14 designs, you should always update to the latest version of Designer to receive the latest bugfixes and enhancements.

19.7.6. Vaadin TestBench

Vaadin 14 provides access to the same TestBench features that are available for Vaadin 8 but the API has been tweaked in many places to correspond with the changes to the components/elements themselves, as well as the features they offer.

The ElementQuery operation `$` no longer has methods such as `caption()` as there is no generic "caption" concept in Vaadin 14. On the other hand, there is instead a generic `attribute(String key, String value)` method which can be used to find elements based on any HTML attribute.

The element classes have been moved to a sub package of the component, e.g. `com.vaadin.flow.component.textfield.testbench.TextFieldElement` instead of `com.vaadin.testbench.elements.TextFieldElement`.

If any API is missing, there are low level helper methods available such as `TestBenchElement.getProperty(String name)` and `TestBenchElement.callFunction(String name)` which makes it easy to interact with any web component with a public JavaScript API.

19.7.7. Vaadin Charts

Vaadin Charts 6 shares a lot of the Java API from Charts 4 even though the underlying technology has been changed. However, almost all of the styling related Java API has been replaced with an ability to style charts using CSS[151]. See list of breaking changes from Charts 4 to Charts 6[152].

19.7.8. Vaadin Board

Vaadin Board for Vaadin 14 contains the same API as Vaadin Board for Vaadin 8 but the API has been adapted to follow Vaadin 14 conventions, e.g. `Row` contains `add(Component… component)` instead of `addComponent(Component)` and `addComponents(Component…)`.

19.7.9. Vaadin Spreadsheet

Currently we don't have a version of Vaadin Spreadsheet for Vaadin 14.

19.8. Migration example - Bookstore Starter

This document shows an example migration from a Vaadin 8 app to Vaadin 14. The migration process is done step-by-step and can be seen through the history of its GitHub repository [153]. The idea is to keep the application compilable in order to be able to see the result of migrating steps.

19.8.1. Step 1 - Initial Vaadin Flow configuration

Maven

First of all, required maven dependency must be added to pom.xml. The Vaadin 8 dependencies, except `vaadin-themes`, are kept for now and will be eliminated after the whole application is migrated. The only Vaadin platform dependency is the following:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.vaadin</groupId>
      <artifactId>vaadin-bom</artifactId>
      <type>pom</type>
      <scope>import</scope>
      <version>${vaadin.version}</version>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
     <groupId>com.vaadin</groupId>
     <artifactId>vaadin-core</artifactId>
  </dependency>
</dependencies>
```

Starting from Vaadin 14, the `vaadin-maven-plugin` should to be configured for development time to make sure that all the needed resources are available for the development of the project. While it is not mandatory to have the plugin configured for all projects, it is needed for this migration so that necessary files are generated:

```xml
<build>
  <plugins>
    <plugin>
      <groupId>com.vaadin</groupId>
      <artifactId>vaadin-maven-plugin</artifactId>
      <version>${vaadin.version}</version>
      <executions>
        <execution>
          <goals>
            <goal>prepare-frontend</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

In this document, migrating custom components and extensions is not covered. So, the widgetset module that includes the following extensions is removed.

- `AttributeExtension`
- `ResetButtonForTextField`

UI class and Servlet configuration

- Both `UI` class and Servlet configuration are optional in Vaadin Flow. However, we can keep them to leverage them in some cases e.g. controlling user access.
- Since the components are different, the DOM structure has changed. The new components are based on web components and a new theme. Thus the theming is discussed later.
- In Vaadin Flow the locale is set automatically based on user preferred locale. So, `setLocale` can be removed.

  Best practice for setting page title is using `PageTitle`

- annotation on each view. So, `getPage().setTitle` is removed from `MyUI::init`.
- All packages names in Vaadin Flow start with `com.vaadin.flow`. One way to correct them is to remove all `import` statements starting by `com.vaadin` and reimport Vaadin Flow classes. For example some equivalent classes in Vaadin Flow are:
- `com.vaadin.ui.UI` → `com.vaadin.flow.component.UI`
- `com.vaadin.server.VaadinRequest` → `com.vaadin.flow.server.VaadinRequest`
- `com.vaadin.ui.TextField` → `com.vaadin.flow.component.textfield.TextField`
- `com.vaadin.ui.VerticalLayout` → `com.vaadin.flow.component.orderedlayout.VerticalLayout`

Test page

In order to verify that Vaadin Flow setup has been done correctly, a simple `HelloWorldPage` like the following can be added.

```
import com.vaadin.flow.component.html.Div;
import com.vaadin.flow.component.html.H1;
import com.vaadin.flow.router.PageTitle;
import com.vaadin.flow.router.Route;

@Route("")
@PageTitle("My")
public class HelloWorldPage extends Div {
    public HelloWorldPage() {
        this.add(new H1("Hello World!"));
    }
}
```

`@Route(“”)` shows that root path should be routed to this page. After running the application by `mvn jetty:run`, the “Hello World!” message can be seen in the browser by entering this address: http://127.0.0.1:8080[154].

19.8.2. Step 2 - Access Control and Login Screen

VaadinServletConfiguration and UI class

In Vaadin Flow, defining a Servlet class is optional. So, we don’t have to create an extended class of `VaadinServlet`, unless we need to change some configuration. Having a `UI` class is optional too and this class can be removed as well, because the `UI` class is created by the framework. However, we may have some tasks assigned to our `UI` class e.g. controlling access. In this example access control is moved to a more suitable place which is described in the following section.

Access Control

`BeforeEnter` event of `UI` class is a good place to control access and there is another event named `UIInit` in `VaadinService` class that is fired whenever a `UI` is created. In order to leverage these events, we can create a class extended from VaadinServiceInitListener[155] and add required code in `serviceInit` method. The result looks like the following piece of code:

```
public class BookstoreInitListener implements
VaadinServiceInitListener {
   @Override
   public void serviceInit(ServiceInitEvent initEvent) {
       initEvent.getSource().addUIInitListener
(uiInitEvent -> {
           uiInitEvent.getUI().addBeforeEnterListener
(enterEvent -> {
               // Controlling access can be done here.
           });
       });
   }
}
```

`MyUI` class had an instance of `BasicAccessControl` and other classes used it via its accessor; now after `MyUI` class is eliminated, there must be another provider for `AccessControl` implementation. The selected solution here is using a factory class (`AccessControlFactory`).

`CurrentUser` class is also needed to change because it is used in `BasicAccessControl` class. We need to apply new packages names of Flow that start with `com.vaadin.flow`. The same should be done in next steps of migration.

LoginScreen

This is the first `UI` screen migrated to 14. The following items describe what needs to be done in migration process:

- Instead of `CssLayout` another equivalent component must be used e.g. `FlexLayout` or a simple `Div`.
- Equivalent of `addComponent` method is `add` method.
- `setWidth` method in Flow has only one `String` parameter that includes both measurement unit and width as a number e.g. “15em” or “310px”.

- `Route` annotation determines the url associated with this screen.
- Predefined style changes to components in 14 are referred as "theme variants", and those change the `theme` attribute of the components instead of the `className`. So, `addStyleName(String)` can be replaced with `addThemeVariants(…)`. The available theme variants for components are showcased in the component demos. Changes in theming from V8 to Vaadin platform is described here.
- New `FormLayout` has a method named `addFormItem` takes a component as a parameter and in addition to adding it to the form, it adds a label beside the component as well.
- Instead of `Button::setClickShortcut` the API is now `Button::addClickShortcut;`.

Some other changes that have been done are not related to Vaadin framework migration process; however, it is a good idea to do such refactorings at the same time as migration.

19.8.3. Step 3 - Menu, MainScreen and AboutView

Menu

As explained before, instead of `CssLayout`, `FlexLayout` is used.

`Navigator` class is removed in Flow and this is one of many changes in routing and navigation since version 8. So, `navigator` field is removed from `Menu`. In `addView` method it can be seen that navigation is done by `RouterLink` component.

At this stage a pretty look is not aimed and it will be made nicer in later steps.

MainScreen

In Vaadin 8 version there is a `CssLayout` that acts as a view container and navigation between different views is done inside the `CssLayout`. In Vaadin Flow, parent layouts can be defined using a newly introduced `RouterLayout` interface. Since `MainScreen` is used as a layout for other views, it must implement `RouterLayout` interface.

AboutView

Layout of views can be specified in `Route` annotation like this `@Route(value = "About", layout = MainScreen.class)`. We don't need the `HelloWorldPage` anymore, so it is removed and since it's good to have a route to root path, `RouteAlias` annotation is used to add a secondary path for `AboutView`.

Another thing worth mentioning here is that in Vaadin platform, a component named `Icon` is added and can be created by calling `create` method of `VaadinIcon` enum.

Here[156] is the link to see the changes in step 3.

19.8.4. Step 4 - Product Grid

DataProvider

In Vaadin platform, when `DataProvider::fetch` method is overridden, `query.getOffset()` and `query.getLimit()`

must be used to fetch a specific chunk of data. If they are not used it shows that the returned data is incorrect and unexpected. To avoid such mistakes in implemented code, Vaadin platform throws an `IllegalStateException` to show us what is wrong. So, `ProductDataProvider::fetch` is fixed in order to use specified offset and limit. The data provider documentation for Vaadin platform can be found here[157].

ProductGrid

The following items briefly describe some of the changes in `ProductGrid`.

- There is no `HtmlRenderer` in Vaadin platform and it must be replaced by other renderers such as `TemplateRenderer` or `ComponentRenderer`. In this migration, `TemplateRenderer` is used. More info and guidance about all kinds of renderers can be found in "Using Renderers" section of Grid document. In `TemplateRenderer`, apart from HTML markup, Polymer data binding notation can also be used. In `ProductGrid`, there are three TemplateRenderers:
  - Price and StockCount columns leverage `TemplateRenderer` to align their text to right.
  - Availability column template uses a Vaadin component named `iron-icon` to show a circle colored based on availability value. In order to set different styles to the circle, three css classes with equivalent names to three values of availability (`Available`, `Coming` and `Discontinued`) are defined in a css file (grid.css). Also, the dependency of the grid on the css file is defined by adding `CssImport` annotation to `ProductGrid` class.
- `Grid.Column::setCaption` method is renamed to `setHeader`.

- `setFlexGrow` method is called for each column to set grow ratios of them.

SampleCrudView

This is the page that includes `ProductGrid` and `ProductForm` and since `ProductForm` is going to be migrated in next step, the parts of the code related to it are commented. Like in the other views, a `Route` annotation is added here with the "Inventory" value. Also, as this view is the main view of the project, the route to root path, the `RouteAlias` annotation, should be moved here. Other changes in `SampleCrudView` are the following items.

- `getElement().getThemeList()::add` is used to add a theme variant to a component. An improved API for this has been released in V12.
- In Vaadin 8, in order to get the parameters passed via the URL, `View` interface must be implemented and the `enter` method must be overridden. In Vaadin platform, there is an interface named `HasUrlParameter` that does the job. It is generic, so parameters are safely converted to the given types. More information about URL parameters can be found here[158].
- Instead of using `HorizontalLayout::setExpandRatio`, `HorizontalLayout::expand` method is used.

Here[159] is the link to see the changes in step four.

19.8.5. Step 5 - Product Form

Since after this step, all Java code is migrated to Vaadin platform, it is time to remove Vaadin 8 dependencies.

Besides, keeping both versions may cause some conflicts in their dependencies e.g. `jsoup`. So, `vaadin-server` and `vaadin-push` are removed from pom.xml. Other changes in this step are as follows.

ProductForm Design

The following items are some of the changes from Vaadin 8 to Vaadin platform in design files.

- In Vaadin 8, Vaadin Designer uses HTML markups to store designed views and they are stored in files with html extension. However, the tags that are used by Vaadin Designer are not standard HTML tags. So, these html files cannot be correctly shown and rendered by browsers. While in Vaadin platform, Polymer template is used to define views and Vaadin Designer also uses it to store designed views.
- Prefix of the Vaadin components names is changed from `v` to `vaadin`.
- For customizing the look and feel of the components using the provided theme variants, the variants are applied with the `theme` attribute, instead of the `style-name` (class name). E.g.

Vaadin 8 version:

```
<v-button style-name="primary" _id="save">Save</v-button>
```

Vaadin platform version:

```
<vaadin-button theme="primary" id="save">Save</vaadin-
button>
```

ProductForm Java Class

`ProductFormDesign` class is removed and its content is moved to `ProductForm` class. Actually, this is the recommended pattern in Vaadin platform and it is also supported by Vaadin Designer. In Vaadin 8, Vaadin Designer keeps two classes, a superclass for designer generated code and an inherited class for the code implemented by developer. The following items are some of the changes in `ProductForm`.

- `JsModule` and Tag annotations are the required annotations to connect `ProductForm` class to its design file, ProductFormDesign.html. And unlike Vaadin 8, reading the design file is done automatically and there is not need to call `Design.read`.
- `Id` annotation is used to connect fields to their equivalents in the associated polymer template.
- In `ComboBox`, `setEmptySelectionAllowed` method is renamed to `setAllowCustomValue`.

ErrorView

Router Exception Handling in Vaadin Flow is described here. Applications can have different views for catching different exceptions. For example, `ErrorView` catches `NotFoundException` that is thrown when something goes wrong while resolving navigation routes. And unlike Vaadin 8, there is no need to register `ErrorView` in a `navigator` or something like that. It is automatically detected and is used by Flow.

SampleCrudLogic

Apart from some cleaning, a small change that is worth mentioning is the change in how the URL of the browser is updated. In Vaadin 8, `page.setUriFragment` is called and the new URL must be constructed and passed as a parameter. While in Vaadin Flow, it is done in a more elegant way; `navigate` method of `UI` class is called and the view parameter is passed as a parameter to `navigate` method.

19.8.6. Step 6 - Production Mode

In Vaadin 14 the production mode is recommended to be enabled by is adding a profile to `pom.xml`. All old V8 related production build configuration can be removed. The following code shows the required configuration for enablind a production build in 14 when running the command `mvn package -Pproduction`:

```
<profiles>
  <profile>
    <!-- Production mode is activated using -Pproduction
-->
    <id>production</id>
    <properties>
      <vaadin.productionMode>true</vaadin.productionMode>
    </properties>

    <dependencies>
      <dependency>
        <groupId>com.vaadin</groupId>
        <artifactId>flow-server-production-
mode</artifactId>
      </dependency>
    </dependencies>

    <build>
      <plugins>
        <plugin>
          <groupId>com.vaadin</groupId>
          <artifactId>vaadin-maven-plugin</artifactId>
          <executions>
            <execution>
              <goals>
                <goal>build-frontend</goal>
              </goals>
              <phase>compile</phase>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

For further details on production mode in 14, you can read here[160].

19.8.7. Step 7 - Theming the application

This step is still in progress and its documentation will be added here when it is completed.

----

[92] http://pages.vaadin.com/vaadin-application-assessment-for-migration?utm_campaign=V10%20migration&utm_source=docs
[93] https://vaadin.com/docs/mpr/Overview.html
[94] http://pages.vaadin.com/vaadin-application-assessment-for-migration?utm_campaign=V10%20migration&utm_source=docs
[95] https://github.com/vaadin/flow-and-components-documentation/issues/new
[96] https://vaadin.com/docs/flow/advanced/tutorial-bootstrap.html
[97] https://vaadin.com/docs/flow/advanced/tutorial-service-init-listener.html
[98] https://vaadin.com/docs/flow/advanced/tutorial-push-configuration.html
[99] https://vaadin.com/docs/flow/advanced/tutorial-loading-indicator.html
[100] https://vaadin.com/docs/mpr/Overview.html
[101] https://vaadin.com/training/course/view/v10-intro
[102] https://vaadin.com/api/platform/11.0.1/com/vaadin/flow/component/html/Div.html
[103] https://vaadin.com/components/vaadin-accordion/java-examples
[104] https://vaadin.com/components/vaadin-button/java-examples
[105] https://vaadin.com/components/vaadin-checkbox/java-examples
[106] https://vaadin.com/components/vaadin-checkbox/java-examples
[107] https://vaadin.com/directory
[108] https://vaadin.com/components/vaadin-combo-box/java-examples
[109] https://vaadin.com/components/vaadin-context-menu/java-examples
[110] https://vaadin.com/docs/flow/creating-components/tutorial-component-composite.html
[111] https://vaadin.com/components/vaadin-date-picker/java-examples
[112] https://vaadin.com/components/vaadin-form-layout/java-

examples
[113] https://vaadin.com/components/vaadin-grid/java-examples
[114] https://vaadin.com/components/vaadin-split-layout/java-examples
[115] https://vaadin.com/components/vaadin-list-box/java-examples
[116] https://vaadin.com/components/login/java-examples
[117] https://vaadin.com/components/menu-bar/java-examples
[118] https://vaadin.com/components/select/java-examples
[119] https://vaadin.com/components/vaadin-notification/java-examples
[120] https://vaadin.com/components/vaadin-text-field/java-examples
[121] https://vaadin.com/components/vaadin-progress-bar/java-examples
[122] https://vaadin.com/components/vaadin-radio-button/java-examples
[123] https://vaadin.com/components/rich-text-editor/java-examples
[124] https://vaadin.com/directory
[125] https://vaadin.com/components/vaadin-tabs/java-examples
[126] https://vaadin.com/components/vaadin-text-field/java-examples
[127] https://vaadin.com/components/vaadin-text-field/java-examples
[128] https://github.com/vaadin/vaadin-grid-flow/issues/469
[129] https://vaadin.com/components/vaadin-treegrid/html-examples/grid-tree-demos
[130] https://vaadin.com/components/vaadin-split-layout/java-examples
[131] https://vaadin.com/components/vaadin-upload/java-examples
[132] https://vaadin.com/components/vaadin-dialog/java-examples
[133] https://vaadin.com/directory
[134] https://vaadin.com/components/browse
[135] https://vaadin.com/docs/flow/creating-components/tutorial-component-basic.html
[136] https://vaadin.com/docs/flow/components/tutorial-enabled-state.html
[137] https://vaadin.com/docs/flow/creating-components/tutorial-component-container.html
[138] https://github.com/vaadin/flow/issues/2465
[139] https://developer.mozilla.org/en-US/docs/Web/CSS/flex
[140] https://vaadin.com/directory
[141] https://vaadin.com/docs/flow/creating-components/tutorial-component-basic.html
[142] https://developer.mozilla.org/en-US/docs/Web/CSS/--*

[143] https://vaadin.com/themes/lumo
[144] https://vaadin.com/themes/material
[145] https://vaadin.com/start
[146] https://vaadin.com/start/latest/full-stack-spring
[147] https://github.com/vaadin/cdi
[148] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules
[149] https://github.com/vaadin/designer/blob/master/RELEASE-NOTES.md
[150] https://vaadin.com/docs/mpr/Overview.html
[151] https://vaadin.com/docs/charts/java-api/css-styling.html
[152] https://vaadin.com/docs/charts/java-api/charts-breaking-changes-in-version-6.html
[153] https://github.com/vaadin/bookstore-starter-flow
[154] http://127.0.0.1:8080
[155] https://vaadin.com/docs/flow/advanced/tutorial-service-init-listener.html
[156] https://github.com/vaadin/bookstore-starter-flow/commit/f017602f668527d26f02f1cd2ef862f474ba033b
[157] https://vaadin.com/docs/flow/binding-data/tutorial-flow-data-provider.html
[158] https://vaadin.com/docs/flow/routing/tutorial-router-url-parameters.html
[159] https://github.com/vaadin/bookstore-starter-flow/commit/d628f29b81df8a94dacec72556a19f2d7f0ff019
[160] https://vaadin.com/docs/flow/production/tutorial-production-mode-basic.html

20. Migrating from Vaadin 10-13 to Vaadin 14

20.1. App Layout 2 Migration Guide

Version 2 of `AppLayout` is introduced in Vaadin 14 and comes with several breaking changes.

20.1.1. Routing

`AbstractAppRouterLayout` was removed. `AppLayout` itself now implements `RouterLayout`. If you have a view like that extended `AbstractRouterLayout`

```
public class MyView extends AbstractRouterLayout {
    ...
}
```

It should now extend AppLayout

```
public class MyView extends AppLayout {
    ...
}
```

20.1.2. Menu

`AppLayoutMenu` and `AppLayoutMenuItem` were removed. The same functionality can be achieved by using `vaadin-tabs`. `AppLayoutMenu` can be directly replaced with `Tabs`.

```
Tabs menu = new Tabs();
```

To mimick AppLayoutMenu’s look, use horizontal orientation and place the menu in the navbar.

```
tabs.setOrientation(Tabs.Orientation.HORIZONTAL);
appLayout.addToNavbar(true, tabs);
```

Use `vaadin-tab` for menu items. To create a navigation item, the recommended approach is to add a router link inside the tab.

```
RouterLink link = new RouterLink(null,TargetView.class);
link.add(VaadinIcon.ARROW_RIGHT.create());
link.add("link text");
Tab tab = new Tab();
tab.add(link);
tabs.add(tab);
```

Additionally, use the appropriate theme variant to place the icon on top of the tab.

```
tab.addThemeVariants(TabVariant.LUMO_ICON_ON_TOP);
```

20.1.3. Branding on navbar

To keep the same look of the previous version with the branding on the top left and having the menu center, add both elements to the navbar.

```
Span appName = new Span("Branding");
appName.addClassName("hide-on-mobile");

this.addToNavbar(true, appName, tabs);
```

and add some styles to shared-styles.html

```
<dom-module id="app-layout-theme" theme-for="vaadin-app-
layout">
  <template>
    <style>
      [part="navbar"] {
        align-items: center;
        justify-content: center;
      }
      [part="navbar"]::after {
        content: '';
      }
      [part="navbar"] ::slotted(*:first-child),
      [part="navbar"]::after {
        flex: 1 0 0.001px;
      }
      @media (max-width: 425px) {
        [part="navbar"] ::slotted(.hide-on-mobile) {
          display: none;
        }
        [part="navbar"]::after {
          content: none;
        }
    }
    </style>
  </template>
</dom-module>

<custom-style>
  <style>
    vaadin-app-layout vaadin-tab a:hover {
      text-decoration: none;
    }
    vaadin-app-layout vaadin-tab:not([selected]) a {
      color: var(--lumo-contrast-60pct);
    }
    vaadin-app-layout vaadin-tab iron-icon {
      margin: 0 4px;
      width: var(--lumo-icon-size-m);
      height: var(--lumo-icon-size-m);
      padding: .25rem;
      box-sizing: border-box!important;
    }
  </style>
</custom-style>
```

20.1.4. New `DrawerToggle` to open or close the drawer.

To use it, add a instance to the component, tipically to the navbar. It will cause a button to appear that will toggle the navbar when clicked.

```
appLayout.addToNavbar(new DrawerToggle());
```

20.2. Migration Tool for Polymer Templates

Several steps are required to migrate your project to Vaadin 14 from Vaadin 13, see Vaadin 14 Migration Guide tutorial.

To help with the migration, the Vaadin Maven plugin can convert Polymer 2 HTML templates into Polymer 3 JavaScript modules. The plugin's `migrate-to-p3` goal automates two steps:

- it uses resources directory (by default it is `src/main/webapp`) to locate Polymer 2 templates HTML files, converts them into Polymer 3 format and moves them into `frontend` folder inside your project root directory.
- it finds all Java class declarations annotated with `@HtmlImport` and `@StyleSheet` in the project source files and rewrites annotation to `@JsModule` annotation along with path (`value` parameter) update.

This goal can be executed from command line with

```
mvn vaadin:migrate-to-p3
```

or with most IDEs from a list of configured Maven plugin goals for a project. During August 2019 the template migration tool will also become available as a Java executable for non-Maven projects.

NOTE There are currently some issues in running the migration tool on Windows 10. The error output refers to "bower install failing" or "git not being available on path". To workaround this, you should use the Windows Linux Subsystem[161] and run the tool there instead. Another workaround that has been reported to help, is to install Bower on the system. We are investigating these issues.

NOTE The migration tool takes care about style files and `@StyleSheet` annotations converting them into `@JsModule`. But there is `@CssImport` annotation available which is more convenient to use instead of `@JsModule` for CSS. The migration tool is not able to convert styles using `@CssImport` annotation. This requires manual conversation.

The migration tool doesn’t preserve HTML comments from your original template files. Important HTML comments should therefore be manually transferred to the converted P3 files. To facilitate this, use the **keepOriginal** parameter to prevent removal of the original template files (by default these are removed). See **keepOriginal** parameter description below.

20.2.1. Goal parameters

Here we describe the Maven plugin goal’s parameters.

- **resources** (default: `${project.basedir}/src/main/webapp`): List of folder paths that should be used to locate the P2 resources to convert them into P3 modules. It’s configured in the pom file via `<resources>` parent element and `<resource>` child elements inside it.

- **migrateFolder** (default: `${project.build.directory}/migration`): A temporary directory which is used internally to store copies of the resource files and their conversation to P3. The result files will be moved to the final destination from it.
- **frontendDirectory** (default: `${project.basedir}/frontend`): The resulting directory which will contain converter resource files.
- **keepOriginal** (default: `false`): Whether to keep original resource files or not. By default the converted resource files will be removed.
- **ignoreModulizerErrors** (default: `true`): Whether the Maven build should fail if modulizer internal tool returns non zero exit status. Even if Modulizer exists with error it doesn't mean that conversation wasn't done. So by default the Maven build won't fail even though there were errors.
- **annotationsRewrite** (default: `ALWAYS`): Defines a strategy to rewrite `@HtmlImport`/`@StyleSheet` annotations in Java source files. There are three values available:
  - `ALWAY` means rewrite annotations regardless of resource conversation status
  - `SKIP` means skip annotations rewrite
  - `SKIP_ON_ERROR` means rewrite only if there are no errors during resource conversation

20.3. Vaadin 14 Migration Guide

For existing Vaadin platform projects (versions 10-13), using most Vaadin 14 new features requires just updating the

Vaadin version, adding a dependency for compatibility mode, and addressing any API changes in code, which can be backtracked from the platform release notes[162].

Vaadin 14 also introduces support for new frontend technologies and tools, which change the way the project is built and how frontend dependencies, like web components, are integrated. This is the default starting point for new projects starting from Vaadin 14, and an optional migration to take for existing projects.

First, this document introduces the new technologies and tooling. Second, it gives an overview of the migration path from Platform V10-V13 to V14. This is not intended to be a complete change log, but rather a hands-on guide to help resolving issues users migrating their applications to V14 will be expected to encounter.

For Vaadin Java users, many of the changes described here only have effect on what happens on behind the scenes. But in case you are using templates / Vaadin Designer or are integrating 3rd party frontend dependencies, you will want to read this document.

NOTE There will be later[163] on a migration tool available for converting an existing project to use the new technologies. Its most important part, which is automatic migration of Polymer 2 Html templates into Polymer 3 JavaScript modules, is already ready and available to be used. See Migration Tool document for more information.

20.3.1. New frontend tooling for Vaadin 14

Since Vaadin 10, the client-side UI components have been built as web components. In versions 10 to 13, the web components and their dependencies have been distributed using Bower, a package manager for frontend projects. For Vaadin Java projects, the Bower dependencies have been packaged into `.jar` files using Webjars[164], allowing to use those directly with Maven.

Over time, Bower has been deprecated in favor of better frontend package managers. While Bower still works and is maintained, it is lacking support for the latest web standards like JavaScript modules[165].

To be able to bring all the latest frontend technologies available to the Vaadin users, the frontend technology stack and tooling have been renewed in Vaadin 14.

NOTE The old and the new toolset cannot be used at the same time! This means eg. `@HtmlImports` are completely ignored when running the project in *npm mode*, and vice versa `@JsModule` is ignored when running the project in *Vaadin 13 compatibility mode.*

Web Components: ES6 modules instead of HTML Imports

EcmaScript 6 modules (or JS modules) are a web standard for modulizing and importing JavaScript code in the browser. Vaadin 14 adds support for JS modules as a replacement for HTML imports which was used earlier for building web components with the Polymer library. This means that the Polymer version has been upgraded from version 2 to version 3 (see here[166] what's new in Polymer 3). In Polymer 3, templates are JavaScript modules that fully encapsulate the HTML structure of the component template.

Package management: npm instead of Bower

Behind the scenes, Vaadin 14 manages front-end dependencies using *npm*, the de-facto standard JavaScript package manager (whereas V13 and older used *Bower* & webjars). Using npm means that all JavaScript dependencies will be downloaded to the `node_modules` directory in the root of your project and that `package.json` and `package-lock.json` files will be created to record and resolve the dependencies and their versions respectively.

Do not be alarmed if you are not familiar with these files or the npm way of doing things. The invocation of npm is fully automated by `vaadin-maven-plugin`; the only thing you need to make sure is that *Node.js* (version 10 or later) and npm (version 5.6 or later) are installed. If they are not installed, the application will not be started and there will be a notification about installing Node.js instead.

Serving frontend resources with webpack

Another front-end tool utilized by Vaadin 14 behind the scenes is the module bundler *webpack*. All dynamic frontend resources in your project (JavaScript modules containing components and stylesheets) are now expected to reside in the `frontend` directory immediately under the project root, which is inspected by webpack. As application is started in development mode, the webpack development server is started automatically to handle serving the frontend resources to the browser.

Unlike in Vaadin 10-13, using webpack enables testing your application with Internet Explorer 11 on development mode. Previously this required a production build to get the frontend resources transpiled to IE11 compatible ES5 syntax. The overhead from this transpilation is about 1-2 seconds.

In production mode, webpack is used to create a (transpiled and minified) bundle from the resources inside the `frontend` folder. As long as your project file structure is as specified in this guide, these steps are fully automated by `vaadin-maven-plugin`.

While basic Vaadin usage requires no knowledge about webpack and how it works, for advanced users there is also a possibility to customize the webpack build.

Compatibility mode

Existing Vaadin 10-13 projects can update to 14 without migrating to the new toolset. The only thing that you need to do is to upgrade your Vaadin version to `14.0.0` in your `pom.xml`. No more changes are needed in your project. Behind the scenes, everything works just as before with using Bower & Webjars, Polymer 2 & Html Imports, and Polymer CLI for the production build. This is called *the compatibility mode* in Vaadin 14.

The compatibility mode is only intended to enable a smoother migration path, and should not be used in new projects. The compatibility mode is supported for the full Vaadin 14 support period, but it will be removed permanently in Vaadin 15.

Compatibility mode can be enabled explicitly by including the `flow-server-compatibility-mode` jar. If using Maven, add the following dependency to `pom.xml`:

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>flow-server-compatibility-
mode</artifactId>
</dependency>
```

Alternatively, enable compatibility mode by setting the deployment configuration parameter `vaadin.compatibilityMode` to `true`. Read more about setting configuration parameters here[167].

If using Spring Boot, setting the `vaadin.compatibilityMode` parameter is the mandatory solution as the `web-fragment.xml` added by the `flow-server-compatibility-mode` dependency is not read. To set the property, you may add the following line to `application.properties`:

```
vaadin.compatibilityMode=true
```

Alternatively, add the property as a JVM parameter to the `spring-boot` plugin when running the application via Maven: `mvn spring-boot:run -Dspring -boot.run.jvmArguments=" -Dvaadin.compatibilityMode=true"`

20.3.2. Migration steps

To use the new toolset, any existing Vaadin projects with client-side Polymer 2 based web components must migrate these from Polymer 2 syntax to Polymer 3 (see next section) before they can run on V14. There is a migration tool available as a part of Maven plugin. The goal `migrate-to-p3` converts P2 templates to P3 templates and replaces `@HtmlImport` annotations with `@JsModule` annotations, see Migration Tool tutorial.

1 - Check prerequisites

Install npm

Install npm and Node.js on your development platform of choice if you don't already have them. Either download the installer (https://nodejs.org/en/download/[168]) or use your preferred package management system (Homebrew, dpkg, …).

Miscellaneous

- If you are using Java 9 or newer and `jetty-maven-plugin`, upgrade the plugin to version `9.4.15.v20190215` or newer.
- If you are using Spring Boot, note that the minimum required version of spring-boot-starter-parent is `2.1.0.RELEASE`.

2 - Update project configuration

Update Platform version in `pom.xml`

The first step is to update your maven `pom.xml` configuration file to use the latest V14 release. If the Vaadin version is specified in Maven properties, change it to the following:

```
<properties>
    ...
    <vaadin.version>14.0.0</vaadin.version>
</properties>
```

Add Vaadin Maven plugin

Next, add the Vaadin Maven plugin to the `<build><plugins>` section of `pom.xml` (if your `pom.xml` already included this plugin, update the goals in the `<execution><goals>` section):

```
<build>
    <plugins>
        ...
        <plugin>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-maven-plugin</artifactId>
            <version>${vaadin.version}</version>
            <executions>
                <execution>
                    <goals>
                        <goal>prepare-frontend</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

The `prepare-frontend` goal checks that Node.js and npm are installed and creates or updates `package.json` based on annotations in the project Java code. It also creates `webpack.config.js` if it doesn't exist yet (if needed, you can add your own customized webpack configuration to this file, as it will not be overwritten by future invocations of `prepare-frontend`).

NOTE In V14, you need the `vaadin-maven-plugin` also in development mode. So, make sure that you declare the plugin with `prepare-frontend` in your default Maven profile.

For the production profile plugin you need to have the goal `build-frontend`:

```
<profile>
    <id>production-mode</id>
    ...
    <build>
        <plugins>
            ...
            <plugin>
                <groupId>com.vaadin</groupId>
                <artifactId>vaadin-maven-
plugin</artifactId>
                <version>${vaadin.version}</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>build-frontend</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</profile>
```

The goal `build-frontend` invokes npm to download and cache the npm packages (into directory node_modules) and webpack to process the JavaScript modules.

Move contents of src/main/webapp/frontend

In Vaadin 10-13, files related to front-end, such as HTML templates, stylesheets, JavaScript files and images are stored in the folder `<PROJDIR>/src/main/webapp/frontend`. Depending on the resource type, you may need to move some of these resource files to a new `frontend` folder *at the root of the project*, i.e., at `<PROJDIR>/frontend`. The following list is a rough guide on what to do with each type of resource:

- HTML files containing Polymer templates, should be removed from the `<PROJDIR>/src/main/webapp/frontend` once you finish the migration, but in the meanwhile, you need them as reference to generate the equivalent JS modules under the `<PROJDIR>/frontend` folder as described in the next section.
- Plain .css files used for global styling: keep them in `<PROJDIR>/src/main/webapp/frontend`
- Custom JavaScript files: move them to `<PROJDIR>/frontend`
- Images and other static resources: keep them in `<PROJDIR>/src/main/webapp/frontend` (or move anywhere else under `webapp`; see comments about updating annotations in section 5)

3 - Convert Polymer 2 to Polymer 3

There is a migration tool available which does this conversation. See Migration Tool tutorial.

Templates

Polymer templates defined in HTML files (extension `.html`) should be converted to new ES6 module format files (extension `.js`) which in the basic case only requires the following steps:

a. Change the file extension from `.html` to `.js`.

b. Change the parent class of the element class from `Polymer.Element` to `PolymerElement`.

c. Convert HTML imports for ES6 module imports. For example a local file

```
<link rel=import href="foo.html">
```

becomes

```
import './foo.js';
```

or external import

```
<link rel="import"
    href="../../../bower_components/vaadin-
button/src/vaadin-button.html">
```

becomes

```
import '@vaadin/vaadin-button/src/vaadin-button.js';
```

To see what's the scope of the js module, for vaadin components it's always @vaadin and for other components, you can search the name that comes after `bower_components` here[169] to find the scope.

NOTE The migration tool converts all vaadin imports using correct scope automatically for you. For other js modules you will need to do it manually.

d. Move the template from HTML into a static getter named `template` inside the element class which extends `PolymerElement`.

E.g.

```
<template>
    <vaadin-text-field id="search">
    </vaadin-text-field>
    <vaadin-button id="new">New
    </vaadin-button>
</template>
```

becomes

```
static get template() {
    return html`
        <vaadin-text-field id="search">
        </vaadin-text-field>
        <vaadin-button id="new">New
        </vaadin-button>`;
}
```

e. Remove the `<dom-module>` and `<script>` tags.

As a complete example, the following template

```
<link rel="import" href=
"../../../bower_components/polymer/polymer-element.html">
<link rel="import" href=
"../../../bower_components/vaadin-text-field/src/vaadin-
text-field.html">
<link rel="import" href=
"../../../bower_components/vaadin-button/src/vaadin-
button.html">

<dom-module id="top-bar">
    <template>
        <div>
            <vaadin-text-field id="search">
            </vaadin-text-field>
            <vaadin-button id="new">New
            </vaadin-button>
        </div>
    </template>

    <script>
        class TopBarElement extends Polymer.Element {
            static get is() {
                return 'top-bar'
            }
        }

        customElements.define(TopBarElement.is,
TopBarElement);
    </script>
</dom-module>
```

becomes

```
import {PolymerElement, html} from
'@polymer/polymer/polymer-element.js';
import '@vaadin/vaadin-button/src/vaadin-button.js';
import '@vaadin/vaadin-text-field/src/vaadin-text-
field.js';

class TopBarElement extends PolymerElement {
    static get template() {
        return html`
            <div>
                <vaadin-text-field id="search">
                </vaadin-text-field>
                <vaadin-button id="new">New
                </vaadin-button>
            </div>`;
    }

    static get is() {
        return 'top-bar'
    }
}

customElements.define(TopBarElement.is, TopBarElement);
```

Styles

Converting `<custom-style>` elements is straightforward. The containing HTML file should be converted to a js file and the content of the file, imports excluded, should be added to the head of the document in JavaScript code. Any import should be converted from `<link>` tag to a javascript import statement the same way as for templates. The following example illustrates these steps in practice.

Polymer 2:

```
<link rel="import" href=
"../bower_components/polymer/lib/elements/custom-
style.html">

<custom-style>
    <style>
        .menu-header {
            padding: 11px 16px;
        }

        .menu-bar {
            padding: 0;
        }
    </style>
</custom-style>
```

Polymer 3:

```
import '@polymer/polymer/lib/elements/custom-style.js';
const documentContainer = document.createElement(
'template');

documentContainer.innerHTML = `
    <custom-style>
        <style>
            .menu-header {
                padding: 11px 16px;
            }

            .menu-bar {
                padding: 0;
            }
        </style>
    </custom-style>`;

document.head.appendChild(documentContainer.content);
```

NOTE The migration tool takes care about style files and `@StyleSheet` annotations converting them into `@JsModule`. But there is `@CssImport` annotation available which is more convenient to use instead of `@JsModule` for CSS. The migration tool is not able to convert styles using `@CssImport` annotation. This requires manual conversation.

Polymer modulizer

For more complex cases you can use Polymer 3 upgrade guide[170]. You can also use polymer-modulizer tool that is described in the guide. Vaadin will also release later a migration tool that helps convert a Vaadin 14 application running in the compatibility mode to Vaadin 14 running the new toolset.

4 - Update Java annotations

The migration tool is able to do this step for you automatically, see Migration Tool tutorial.

After converting Polymer templates from HTML to JavaScript modules, every `HtmlImport` annotation in Java classes should be changed to a `JsModule` annotation. Moreover, you should not use a frontend protocol (`frontend://`)in the path of your resources anymore, and add the `./` prefix to the file path. E.g.

```
@HtmlImport("frontend://my-templates/top-bar.html")
```

becomes

```
@JsModule("./my-templates/top-bar.js")
```

WebJars

If you are developing an application or an add-on which depends on web components from webjars, like below:

```
<dependency>

<groupId>org.webjars.bowergithub.polymerelements</groupId
>
    <artifactId>paper-slider</artifactId>
</dependency>
```

then the migration tool won't be able to rewrite correctly the `@HtmlImport` annotation unless it is a Vaadin web component. In this case the `@HtmlImport` will be replaced by `@JsModule` but you should correct the value by yourself since the migration tool is not able to detect the scope of the JS module automatically. Here are the steps you need to do for each WebJar in your project:

- Find the npm package of the web component. You should be able to find it via one of the following ways.
  - Go to the GitHub repository page of the component and most likely the package name is mentioned in the readme file. For the given example, the owner name of the component on GitHub is the last part of the groupId which is `polymerelements`, So, after adding the name of the component, the address of its GitHub repository can be determined as https://github.com/PolymerElements/paper-slider. Then the npm package name can be found under installation section in front of `npm install` command. So, the npm package is `@polymer/paper-slider`.
  - Search on [npmjs.com](https://www.npmjs.com). The name of the web component should be the same. The scope of the package should match the groupId of the

dependency. It can be used to identify the correct npm package if name brings up duplicates. For the given example, you can search for `paper-slider` and among the results, you can see that one of them has the `@polymer` scope has the best match. So, the corresponding npm package is `@polymer/paper-slider`.

- Add `@NpmPackage` annotation to your class. After finding the right npm package and choosing the version that you want to use, you should add `@NpmPackage` annotation with the package name (as `value` parameter) and package version It means that you should add the following annotation to your class. In this example

  ```
  @NpmPackage(value = "@polymer/paper-slider", version =
  "3.0.1")
  ```

  If you want to use the latest minor release of the npm package instead of a fixed version, then you should add a caret before the version. E.g. the above annotation would become like the following.

  ```
  @NpmPackage(value = "@polymer/paper-slider", version =
  "^3.0.1")
  ```

  For more information about the versioning of npm packages, see [this](https://docs.npmjs.com/files/package.json#dependencies).

- Update the value of `@JsModule` annotation with the correct path which can be found on the same page where the npm package is found. For our example, it's `@polymer/paper-slider/paper-slider.js`. So, the annotation would be:

```
@JsModule("@polymer/paper-slider/paper-slider.js")
```

5 - Remove frontend protocol

Apart from `JsModule` annotations, the `frontend://` protocol should also be removed from non-annotation resource accessors in Java code or in JavaScript code. For example in V10-V13 to add a PNG file from `<PROJDIR>/src/main/webapp/img` folder, you would do as follows:

```
String resolvedImage = VaadinServletService.getCurrent()
    .resolveResource("frontend://img/logo.png",
    VaadinSession.getCurrent().getBrowser());

Image image = new Image(resolvedImage, "");
```

In V14, the above becomes:

```
String resolvedImage = VaadinServletService.getCurrent()
    .resolveResource("img/logo.png",
    VaadinSession.getCurrent().getBrowser());

Image image = new Image(resolvedImage, "");
```

6 - Build and maintain the V14 project

Test the new configuration by starting the application. How you do this depends on your application deployment model. For example, if you are using the Jetty maven plugin, run:

```
mvn clean jetty:run
```

You should see Maven log messages confirming that npm is downloading the package dependencies and that webpack is emitting `.js` bundles. If there is any error, go back and re-check the previous steps.

The following files/folders have been generated in the root of your project:

- `package.json` and `package-lock.json`: These files keep track of npm packages and pin their versions. You may want to add these to version control, in particular, if you added any local package directly with npm.
- `node_modules` directory: npm package cache, do not add this to version control!
- `webpack.config.js`: webpack configuration. Include in version control. You can add custom webpack configuration to this file.
- `webpack.generated.js`: Auto-generated webpack configuration imported by `webpack.config.js`. Do not add to version control, as it is always overwritten by `vaadin-maven-plugin` during execution of the `prepare-frontend` goal.

You now have a fully migrated Vaadin 14 project. Enjoy!

----

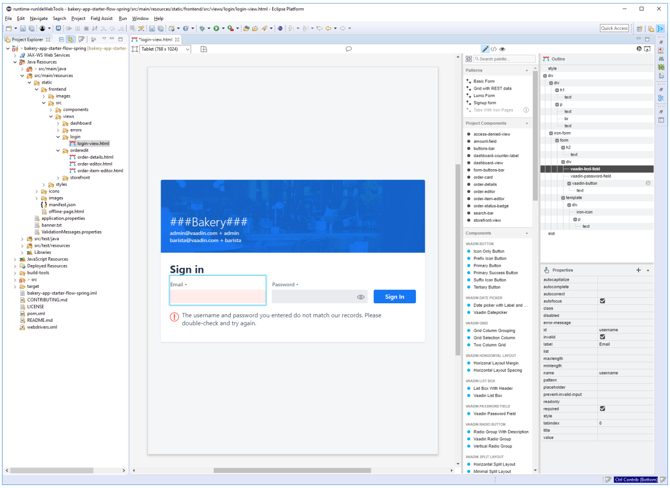
[161] https://docs.microsoft.com/en-us/windows/wsl/install-win10
[162] https://github.com/vaadin/platform/releases
[163] https://github.com/vaadin/flow/issues/5037
[164] https://www.webjars.org/
[165] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules
[166] https://polymer-library.polymer-project.org/3.0/docs/about_30
[167] https://vaadin.com/docs/flow/advanced/tutorial-all-vaadin-properties.html
[168] https://nodejs.org/en/download/
[169] https://www.npmjs.com/search

[170] https://polymer-library.polymer-project.org/3.0/docs/upgrade

21. Vaadin Designer

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 HTML and the Java code, it allows you to design and layout your UIs with speed and confidence.

Vaadin Designer works with Vaadin platform and Vaadin Framework 8 designs. Check the Framework 8 documentation for more information about Framework designs and their declarative format.



runtime-runIdeWebTools - bakery-app-starter-flow-spring/src/main/resources/static/frontend/src/views/login/login-view.html - Eclipse Platform
File Edit Navigate Search Project Field Assist Run Window Help
Project Explorer
*login-view.html
Tablet (768 x 1024)
###Bakery###
admin@vaadin.com + admin
barista@vaadin.com + barista
Sign in
Email
Password
Sign In
Patterns
Project Components
Components
Outline
Properties

The username and password you entered do not match our records. Please double-check and try again.

Figure 1. Vaadin Designer Views

Vaadin Designer is used to create a JavaScript file that defines a UI, or part of a UI. The HTML JavaScript is called a *design*.

For a design, Vaadin Designer can create and update a Java file known as the *companion file*. The companion file exposes elements of the design to Java and provides a way to bind data between Java and the UI defined in the design.

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.

21.1. Installation

21.1.1. Installing in Eclipse

Installing Eclipse and Plug-Ins

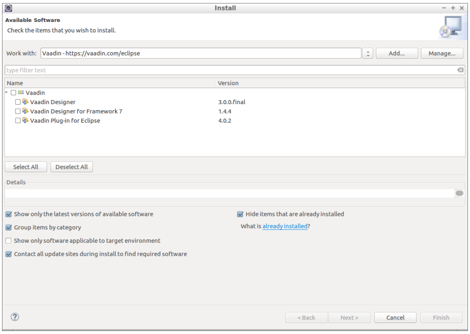
You need to install the following to use Vaadin Designer:

1. Eclipse for Java EE developers
2. Vaadin Designer: https://vaadin.com/eclipse[171]

If you already have Vaadin Designer installed, it will be upgraded to the newest Vaadin Designer.

Installing Vaadin Designer

- Open Eclipse
- Choose Help > Install New Software…
- Type https://vaadin.com/eclipse into the **Work with** field and hit Enter
- On **Vaadin**, select **Vaadin Designer**.



Available Software

Check the items that you wish to install.

Work with: Vaadin - https://vaadin.com/eclipse

Name	Version
Vaadin	
Vaadin Designer	3.0.0.final
Vaadin Designer for Framework 7	1.4.4
Vaadin Plug-in for Eclipse	4.0.2

- [✓] Show only the latest versions of available software
- [✓] Group items by category
- [ ] Show only software applicable to target environment
- [✓] Contact all update sites during install to find required software
- [✓] Hide items that are already installed

What is already installed?

Figure 2. Select Vaadin Designer plugin

- Follow the wizard to finish installing the plugin
- Restart Eclipse to make the plugin active

Installing a preview version

For testing upcoming features, we sometimes provide a preview version of Designer. To install the preview version of Designer for Eclipse, follow the instructions above but use the URL *https://vaadin.com/eclipse/preview* instead of https://vaadin.com/eclipse.

Uninstalling

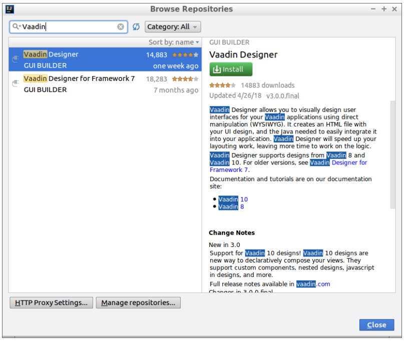
If you want to remove Vaadin Designer from your Eclipse installation, go to "Help > Installation Details", select Vaadin Designer from the list, then click Uninstall.

21.1.2. Installing in IntelliJ IDEA

Installing Vaadin Designer

Vaadin Designer is compatible with both Community and Ultimate Editions.

- Open IntelliJ IDEA
- Choose IntelliJ IDEA > Preferences > Plugins in macOS, File > Settings > Plugins in Windows and Linux.
- Click Browse Repositories…
- Search for **Vaadin**
- Install Vaadin Designer
- Restart IntelliJ IDEA when asked
- Wait for IDEA to restart



GUI BUILDER

Vaadin Designer

14883 downloads

Updated 4/26/18 v3.0.0.final

Vaadin Designer allows you to visually design user interfaces for your Vaadin applications using direct manipulation (WYSIWYG). It creates an HTML file with your UI design, and the Java needed to easily integrate it into your application. Vaadin Designer will speed up your layouting work, leaving more time to work on the logic.

Vaadin Designer supports designs from Vaadin 8 and Vaadin 10. For older versions, see Vaadin Designer for Framework 7.

Documentation and tutorials are on our documentation site:

- Vaadin 10
- Vaadin 8

Change Notes

New in 3.0

Support for Vaadin 10 designs! Vaadin 10 designs are new way to declaratively compose your views. They support custom components, nested designs, javascript in designs, and more.

Full release notes available in vaadin.com

Changes in 3.0.0.final

Figure 3. Install Vaadin Designer in IntelliJ IDEA

Installing a preview version

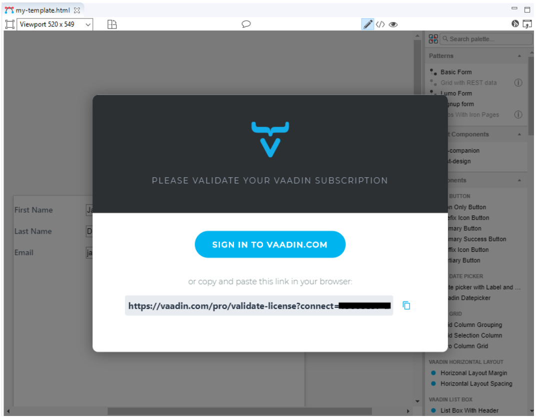
For testing upcoming features, we sometimes provide a preview version of Designer. To install the preview version of Designer for IntelliJ, install the plugin using the instructions above. After installation, go to plugin settings and change the Plugin update channel to EAP. The IDE will suggest to update the plugin automatically. You can also check for updates manually from Preferences > Plugins > Vaadin Designer > Update.

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.

21.1.3. Licensing

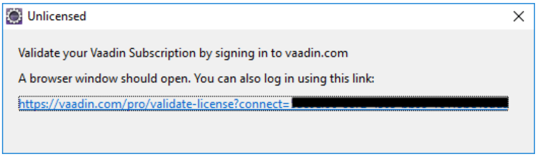
The first time you start Vaadin Designer, it will show a license dialog in the IDE and open vaadin.com in your browser. After logging in to vaadin.com and validating your license, you can start creating your designs. Remember to keep the IDE and the editor open while license validation is in progress.


PLEASE VALIDATE YOUR VAADIN SUBSCRIPTION

or copy and paste this link in your browser:

https://vaadin.com/pro/validate-license?connect=[redacted]

Figure 4. License dialog for a Vaadin platform design


Unlicensed

Validate your Vaadin Subscription by signing in to vaadin.com

A browser window should open. You can also log in using this link:

https://vaadin.com/pro/validate-license?connect=[redacted]

Figure 5. License dialog in Eclipse for Framework design

Please note that a separate license key is required for each developer. If you choose not to supply a license, you will be unable to see your design.

If you for any reason need to remove or change a valid license, it is located in ~/.vaadin/proKey in UNIX systems and C:\Users\<username>\.vaadin\proKey in Windows.

21.1.4. Getting Started

Vaadin Designer works with projects using Vaadin Flow. You can get started with a Flow project with these instructions: https://vaadin.com/start

Creating a Vaadin platform design

With your project selected, find Vaadin platform design from the new file menu of your IDE.

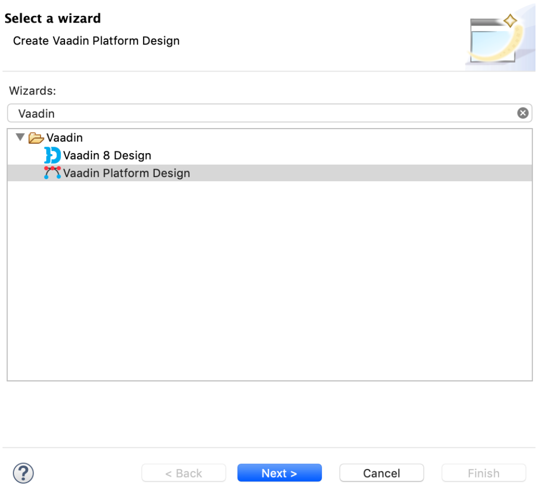
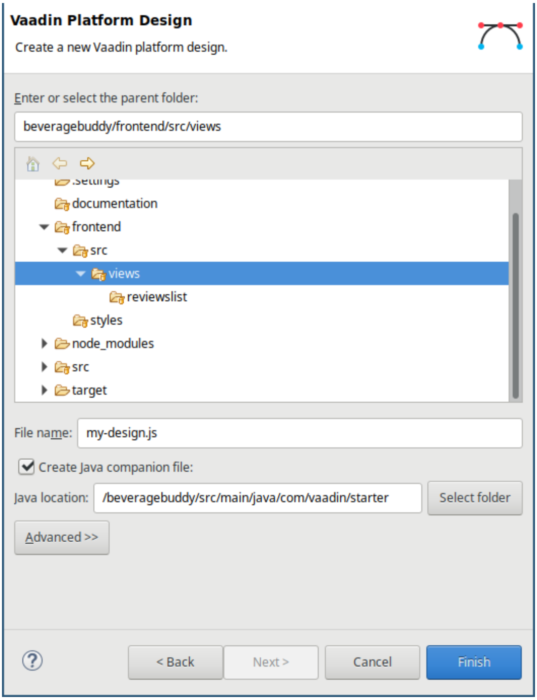


Figure 6. Creating a New Vaadin platform design in Eclipse

In the next step, make sure the locations are correct. The design file must be placed into the *frontend* folder or one of its sub-folders. You can also choose to create a Java companion file together with the new design. The companion file can be located under any of the project's Java source roots.


Vaadin Platform Design

Create a new Vaadin platform design.

Enter or select the parent folder: beveragebuddy/frontend/src/views

File name: my-design.js

Create Java companion file: ✓

Java location: /beveragebuddy/src/main/java/com/vaadin/starter

Figure 7. New Design Parameters

Give your design a descriptive name. The name must be a valid HTML Custom Element name[172].

For example, the name *user-editor-design* will result in *user-editor-design.js* and *UserEditorDesign.java*.

Choose Finish to create the design and open Vaadin Designer.

Vaadin Designer GUI Overview

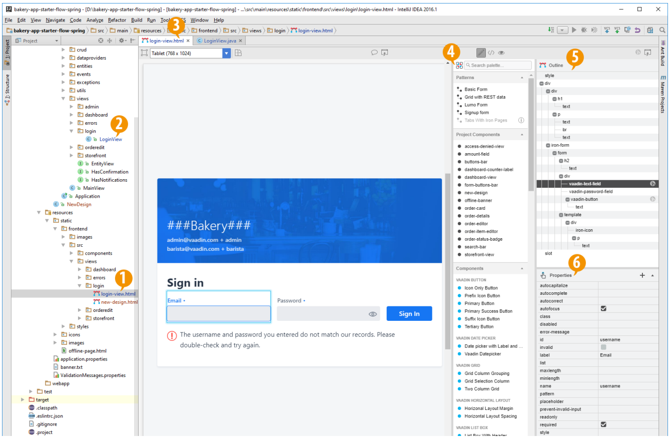
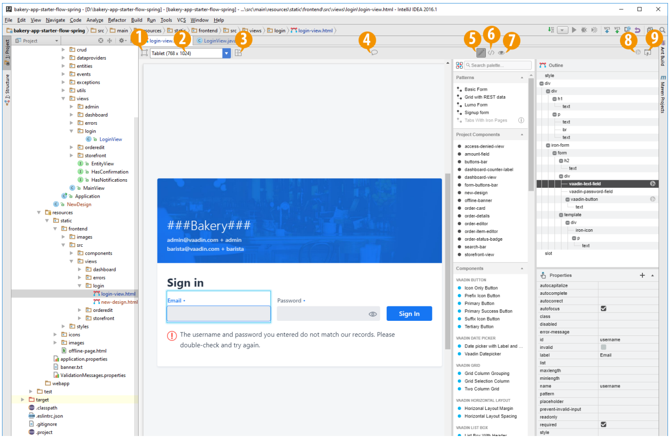


Figure 8. Panels in Vaadin Designer

The elements of the Vaadin Designer are as follows:

1. Design file
2. Companion file
3. Editor (see below for close-up)
4. Palette for web components, HTML elements and snippets
5. Outline - component hierarchy
6. Properties for the selected component

In the editor view, illustrated in Component Editor, you have a number of controls in the toolbar.



The username and password you entered do not match our records. Please double-check and try again.

Figure 9. Component Editor

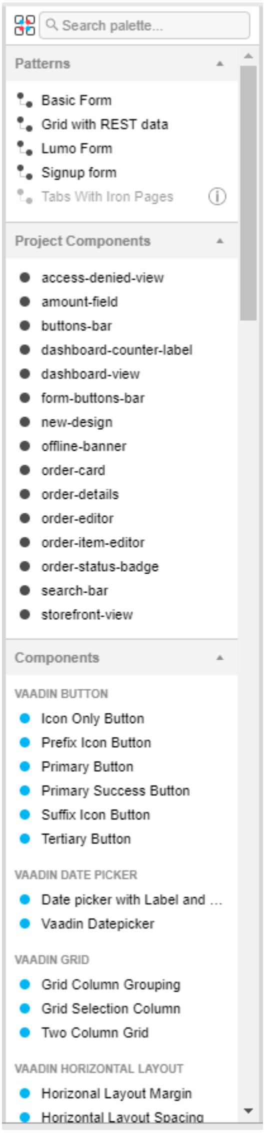
1. Center viewport
2. Viewport size and presets
3. Rotate viewport (portrait / landscape)
4. Send feedback
5. Design mode
6. Source mode
7. Preview mode
8. Companion file connector
9. External preview

IMPORTANT	By default, Vaadin Designer requires Polymer dependency to render the whole viewport. Therefore, your project must have Polymer dependency.

21.1.5. The Palette

The Palette appears on the right side of the editor. The Palette contains the web components available for the current design. Users can drag a component from the Palette and drop it into the desired locations.

When a design is opened, the Designer searches the entire project for web components and loads them into the Palette.



Search palette...

Patterns

- Basic Form
- Grid with REST data
- Lumo Form
- Signup form
- Tabs With Iron Pages

Project Components

- access-denied-view
- amount-field
- buttons-bar
- dashboard-counter-label
- dashboard-view
- form-buttons-bar
- new-design
- offline-banner
- order-card
- order-details
- order-editor
- order-item-editor
- order-status-badge
- search-bar
- storefront-view

Components

VAADIN BUTTON

- Icon Only Button
- Prefix Icon Button
- Primary Button
- Primary Success Button
- Suffix Icon Button
- Tertiary Button

VAADIN DATE PICKER

- Date picker with Label and ...
- Vaadin Datepicker

VAADIN GRID

- Grid Column Grouping
- Grid Selection Column
- Two Column Grid

VAADIN HORIZONTAL LAYOUT

- Horizonal Layout Margin
- Horizontal Layout Spacing

Figure 10. The Palette

There are 4 main groups of components: Patterns, Project Components, Components and Parts.

Patterns

This group contains quick-start solutions to certain design tasks.

Project Components

This group contains the designs from the project so that you can easily reuse them in the currently edited design.

Components

This group contains Vaadin Components[173] with example content and styles so that you can quickly see how they work.

Parts

This group contains two sections: Web Components and HTML Elements.

Web Components

This group contains web components that are included in the project as `npm` dependencies[174].

HTML Elements

This group contains Native HTML5 elements, such as *style, h1 to h6, div, li, ol, p, ul, a, span, img, script, col, table, button, form, input, label, slot* and *template*. If you want to add inner text for an HTML element, you can drag and drop the `text` item from this group to the target element.

TIP Make sure to run `mvn package` before opening your project. Otherwise, web components are not added into the Palette and your project will not work properly.

How the "Project Components" scanning actually works

Whenever you open a Vaadin platform design, Vaadin Designer will scan the whole project for you. All JavaScript files with custom element definition[175] and extends `PolymerElement`[176] will be considered as web components and end up in "Parts / Web Components" section.

If your project has a large number of components, the Palette Search field can help you to find elements quickly.

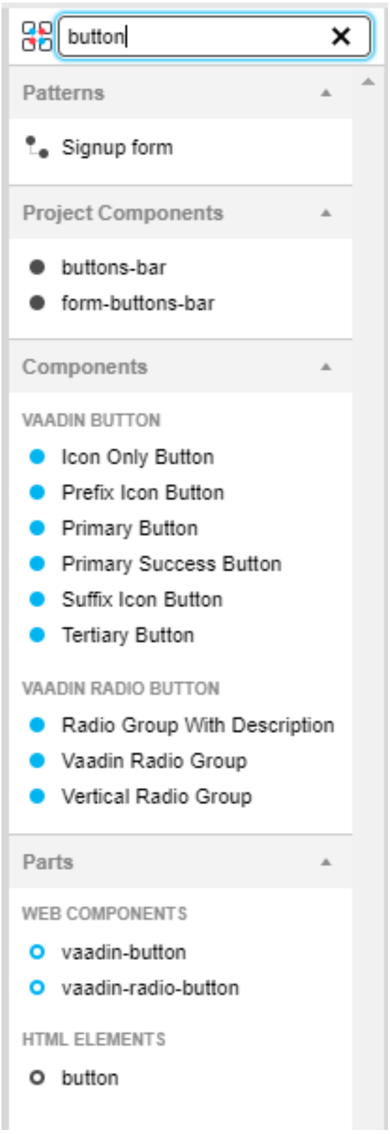
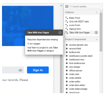


Figure 11. Palette shows the filtered components

If dependencies for items in *Patterns* and *Components* groups are missing, an info indicator appears on each such item. You can hover over it to see more details.



Tabs With Iron Pages

Required dependencies missing:
iron-pages
Add them to project to use Tabs With Iron Pages in designs.

Figure 12. Missing dependency in Palette items

21.1.6. The Outline

The Outline is shown on top-right corner, containing the hierarchy of the opened design. You can drag and drop components from the Palette to the Outline and create your design's structure.

A Vaadin platform design can have many root elements. Layout elements, such as *div, vaadin-form-layout, vaadin-split-layout, vaadin-horizontal-layout* and *vaadin-vertical-layout*, have their width and height expanded to 100% when they are roots.

Inside the Outline, you can also drag and drop a component around to re-arrange it, or press kbd:[Delete] to remove a selected component.

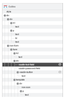
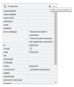


Figure 13. The Outline

21.1.7. The Properties

The Properties lay under the Outline, showing the properties of a selected component.

After selecting a component from the Editor or the Outline, you can edit its properties in the Properties table. It is a good idea to give components at least an `id` 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.



autocapitalize	
autocomplete	
autocorrect	
autofocus	
class	
disabled	
error-message	* Must be at least 8 characters * Must include lowercase and uppercase characters
id	password
invalid	☐
label	Password
list	
maxlength	
minlength	
name	password
on-keydown	_onPasswordKeydown
pattern	
placeholder	
prevent-invalid-input	

Figure 14. The Properties

You can also add a new property by clicking on the *Add a new property* icon.

21.1.8. How do Vaadin 8 and Vaadin platform designs differ?

Vaadin 8 designs are XML stored in .html file. They contain custom markup that is read at runtime by the Vaadin Framework and converted into in-memory component tree. The actual markup is never sent to the browser. Designer for Vaadin 8 reads the markup and uses the Framework to render it in the browser in editable format.

Vaadin platform designs are modern HTML. Technically the design is an independent Polymer template file. The markup itself is not converted at runtime, but is instead sent to the browser to be rendered natively. Designer for Vaadin platform wraps the markup and renders it in the browser in editable format.

While Vaadin 8 and platform designs are not directly compatible, they have a very similar structure.

Simple form design in Vaadin 8

```
<vaadin-vertical-layout>
 <vaadin-text-field caption="Last Name"></vaadin-text-
field>
 <vaadin-text-field caption="First Name"></vaadin-text-
field>
 <vaadin-button plain-text>
   Save
 </vaadin-button>
</vaadin-vertical-layout>
```

Simple form design in Vaadin platform

```
<vaadin-vertical-layout>
  <vaadin-text-field label="First Name"></vaadin-text-
field>
  <vaadin-text-field label="Last Name"></vaadin-text-
field>
  <vaadin-button>
    Save
  </vaadin-button>
</vaadin-vertical-layout>
```

In simple cases elements API is similar. Complexity starts to appear when creating more complex views and using bigger components.

For Vaadin 8 designs there is a limited styling support with the theme variables. Complex styling requires usage of the styleName variable and separate theme file. In platform designs HTML format supports complex styling with rules, which are directly added to the template. The <style> tag can include any CSS for that design.

Same rule is also applied for adding behavior to designs. In Vaadin 8 designs all imperative code must be included in the companion file. In Vaadin platform design can contain any Javascript inside itself.

21.1.9. Frequently Asked Questions

The External Preview only shows a blank page / spinner / connection error.

The browser or device you’re using is unable to connect to your IDE.

- Your computer (with the IDE) and the external browser / device must use the same network.
- The network must allow connections between computers.
- Your computer (firewall) must allow connections to the IDE (you might have been asked something like “Do you want Eclipse to accept incoming connections?”)

If you still have problems

- The external browser or device might have a proxy set up that interferes.
- If you are running a virtual machine (e.g VirtualBox, VMware) it might think it’s on a different network.

How can I download the Designer for offline installation?

Eclipse

The Eclipse runtime allows mirroring of update sites locally. Run the Eclipse executable with these parameters:

```
eclipse -nosplash -application
org.eclipse.equinox.p2.artifact.repository.mirrorApplicat
ion -source https://vaadin.com/eclipse -destination my-
local-updatesite
```

After the command finishes the folder can be added as a local update site.

IntelliJ

Designer for IntelliJ packages can be downloaded as zip files from https://plugins.jetbrains.com/plugin/9519-vaadin-designer and installed via the Install plugin from disk... option.

How do I do responsive views with the Designer?

Creating responsive views with Designer is pretty much the same question than “How to create responsive applications with Vaadin”. We offer a training course on responsive design for Vaadin 8 https://vaadin.com/training/courses/responsive-layouting, and have some resources online as well: https://vaadin.com/docs/v8/framework/themes/themes-responsive.html. For Vaadin platform, creating responsive applications is the same as for frontend applications, so any materials available online will be directly applicable. For example https://developers.google.com/web/fundamentals/design-and-ux/responsive/ is a good resource to get started.

I have a perpetual license for Designer 2, but after updating to Designer I'm asked for a Pro subscription?

Previously it was possible to either buy a Pro subscription, or purchase a single license for a specific major version of a product. From Vaadin platform 10 onwards, we have decided to discontinue the single licenses. In order to use Designer after the trial period, a valid Pro subscription is required.

As IDE's often auto-upgrade plugins, it might be difficult to stay in the 2.x version. To prevent this follow these instructions:

Eclipse

1) Uninstall Vaadin Designer if you already updated

2) Remove the vaadin.com/eclipse update site

3) Add the *https://vaadin.com/eclipse/designer2* URL as an update site, and install Designer 2

IntelliJ

1) Uninstall Vaadin Designer if you already updated

2) Add the https://cdn.vaadin.com/vaadin-designer/intellij-release-2/updatePlugins.xml URL as a repository and install Designer 2

How do Vaadin 8 and Vaadin platform designs differ?

See Platform designs compared to Vaadin 8

21.1.10. Installation issues

“An error occurred while collecting items to be installed” when trying to install Designer for Eclipse

Try to turn off “Contact All Update Sites” while installing (Help → Install New software → Contact All Update Sites.) Please see https://github.com/vaadin/designer-issues/issues/255

Installing Vaadin Designer for Eclipse worked, and it’s shown as Installed Software, but no menu item shows up.

Chances are Eclipse is running on an older version of Java. Please install **at least Java 8**. Note that you can have multiple versions installed, so **make sure Eclipse uses the correct one.** Note that this might also require editing ***eclipse.ini***, which might still point to your old JDK. If all else fails, try uninstalling the old JDK.

If you get the operating system "busy cursor" (e.g "beachball" on OS X):

In rare cases, project settings become inconsistent when updating a plugin in Eclipse. Deleting the project settings seems to make everything work again.

I have problems making layouts behave as I want/look different in application.

Vaadin Designer layouting behaviour matches that of the components - it is a good idea to familiarize yourself with the appropriate component documentation.

I use Linux and the Designer shows strange artifacts or does not render the Property view correctly.

The property view has some issues when rendering under SWT 3 and without Cairo. To improve the situation you can run Eclipse with the following options to use GTK2 and Cairo.

```
env SWT_GTK3=0 GDK_NATIVE_WINDOWS=1 ./eclipse
-Dorg.eclipse.swt.internal.gtk.cairoGraphics=true
-Dorg.eclipse.swt.internal.gtk.useCairo=true
```

Also depending on your Linux distribution you might need to install libwebkitgtk-1.0-0 (Note: It needs to be a 1.x release, if you have a 2.x version install you still need to also install the 1.0 release!). To install use the following command:

```
sudo apt-get install libwebkitgtk-1.0-0
```

I use Linux and the Designer fails to start with the error *IPCException: IPC process exited. Exit code: 127*

The embedded browser used by Designer requires `libXss` and `libCrypto` to be available. Ensure that you have them installed.

Also, on some Debian systems the libraries might be installed in the wrong location resulting in that the embedded browser cannot find them, in that case you can create a symlink to the right location. For example:

```
libcrypto.so.1.0.0 -> ./x86_64-linux-
gnu/libcrypto.so.1.0.2
```

By default, some distros do not have the correct libraries installed that are required by Chromium. Check the logs and install the appropriate libraries. For example, if you see these error messages:

```
There are next missing dependencies:
    browsercore64 => libgconf-2.so.4
    libbrowsercore64.so => libgconf-2.so.4
```

The missing library is `libgconf-2.so.4`. Install the library manually:

```
sudo apt-get install libgconf-2-4
```

Installing Vaadin Designer for Eclipse worked, but launching it hangs or crashes with GTK related errors

Make sure you are running Eclipse with an up to date version of the JRE. At least some versions of OpenJDK and Oracle JDK 8 are known to cause crashes when running Designer.

Does Vaadin Designer support Java 11?

- Starting from Eclipse 2018 running Vaadin Designer with Java 11 is not supported.
- From IntelliJ 2018.2 upwards Designer supports projects running Java 11.

To run Eclipse with a specific Java version: * Open your `eclipse.ini` file in your Eclipse folder * Modify or add the `-vm` parameter as instructed in the Eclipse wiki: https://wiki.eclipse.org/Eclipse.ini#Specifying_the_JVM

Please check https://github.com/vaadin/designer/blob/master/RELEASE-NOTES.md#requirements for more details on supported versions.

21.2. Using Vaadin Designer

21.2.1. Designing

To add a component to your design, drag it from the Palette view and drop it in the desired location - either in the viewport area or in the hierarchical Outline view. Dropping in the desired location on the viewport is a common approach, but in many situations (especially with complex, deeply nested hierarchies) dropping on the Outline view gives more control.

Adding Components

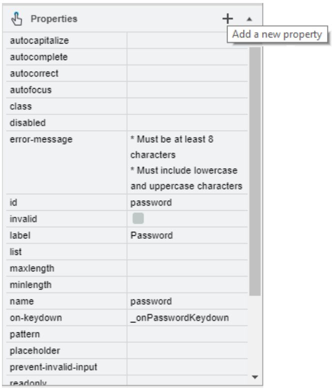
Components can be added by dragging from the Palette view, either to the canvas or to the Outline view. You can also double-click an component in the Palette to add a sibling to the currently selected component.

The component you add will be selected in the editor view, and you can immediately edit its properties.

Editing Properties

You can edit component properties in the Properties view. It is a good idea to give components at least an `id` 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.

Vaadin Designer will discover the defined properties of the selected web component. Public properties (name does not start with an underscore '_') and non-readonly properties will be populated to the properties table. You can also add a new property by clicking the plus button (+) on the Properties view header.



autocapitalize	
autocomplete	
autocorrect	
autofocus	
class	
disabled	
error-message	* Must be at least 8 characters * Must include lowercase and uppercase characters
id	password
invalid	☐
label	Password
list	
maxlength	
minlength	
name	password
on-keydown	_onPasswordKeydown
pattern	
placeholder	
prevent-invalid-input	
readonly	

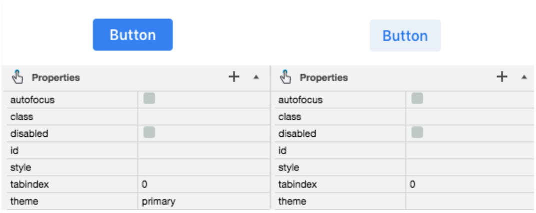
Figure 15. Adding new property

TIP

Some boolean properties might not have a checkbox int the Properties view editor. Vaadin Designer cannot guess the type of the properties without a predefined default value. A workaround for this issue is to add the boolean attribute into the declarative using in **Source mode**, then switch back to the **Edit mode**. For example: `<vaadin-text-field disabled></vaadin-text-field>`

Theme Property

When editing a Vaadin element, theme property is always available in properties table, and you can easily apply styles from Vaadin Themes[177]. For example, to change the visual appearance of a Vaadin Button you can apply the `primary` style.



Properties	
autofocus	☐
class	
disabled	☐
id	
style	
tabindex	0
theme	primary

Properties	
autofocus	☐
class	
disabled	☐
id	
style	
tabindex	0
theme	

Figure 16. Theme property

Theming

Vaadin Designer supports theming the same way as Flow[178]. When a design is opened, Vaadin Designer:

- Loads the selected component theme[179].
- Automatically loads the application theme[180], i.e. `shared-styles.html`.

You can change component theme used by Designer from the project settings. Component themes have different look and feel as well as styles declarations. Changing Designer component theme setting will not affect your Flow project. Likewise, your Flow project theme setting will not be reflected in Designer.

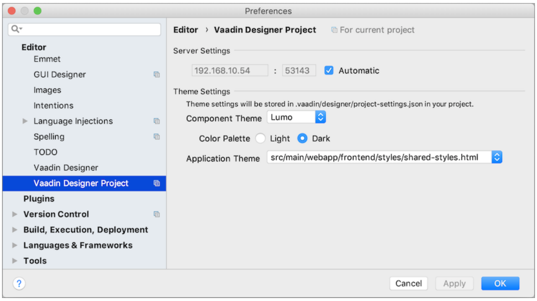
Designer component theme setting only affects how designs are rendered by Designer. Typically, you will match this with your application's component theme.

The default component theme is `Lumo`. `Material` component theme is also available. Both themes have "light" and "dark" color variants[181].

User should provide all styling through the application theme, if `None` component theme was selected.

The None component theme will be used as a fallback if project is missing necessary dependencies for the selected theme, for example if `vaadin-material-styles` JAR is not available in the classpath.

Theme settings will be stored in your project's root folder under .vaadin/designer/project-settings.json so that the settings can be preserved and thereby shared with everyone who works with Designer on the project.



192.168.10.54 : 53143 [✓ Automatic]

Theme settings will be stored in .vaadin/designer/project-settings.json in your project.

Component Theme: Lumo

Color Palette: Light [✓ Dark]

Application Theme: src/main/webapp/frontend/styles/shared-styles.html

Figure 17. Theme settings

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

By resizing the viewport, 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.

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.

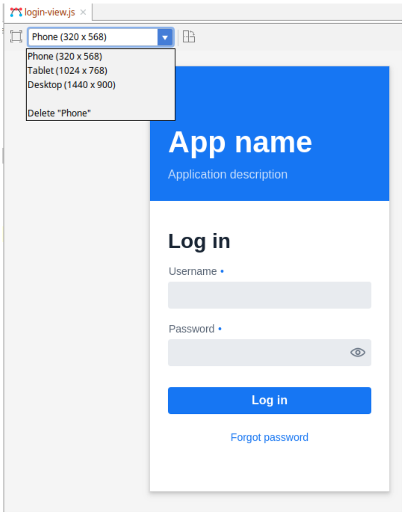


Figure 18. Viewport Preset Sizes

You can also add your own presets - for instance known portlet or dashboard tile sizes, or other specific sizes you want to target.

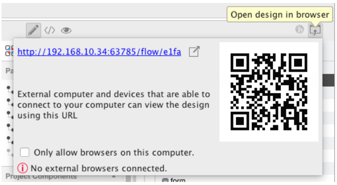
To preview the design in the other orientation (portrait vs. landscape), press the icon right of the viewport size control.

Preview

The Preview is one of the modes available to the right in the toolbar (the other modes being Edit and Source). 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.

External Preview

The external preview popup shows a QR code and its associated URL. By browsing to the URL with 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.



http://192.168.10.34:63785/flow/e1fa

External computer and devices that are able to connect to your computer can view the design using this URL

Only allow browsers on this computer.

No external browsers connected.

Figure 19. 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 the IDE. 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.

21.2.2. Connect to Java

You can connect both components and data between Java and the UI made with Designer. In practice, this is accomplished with a Java class that enables Vaadin Flow to connect your Java code to the UI. In other words, you access the UI defined in a design programmatically through the companion class for the design.

TL;DR To connect a component to Java:

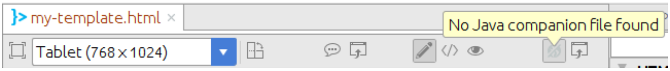
1. Make sure you have a companion Java class with @JsModule and @Tag annotations matching the design.
2. Set the id attribute for the element.
3. Click on the connect button for the element in the Outline.

The Java Companion File

The Java companion file contains the class that connects your Java code to the UI defined in the design. There can be only one companion class for a design.

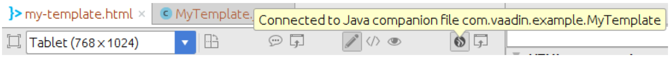
You may have already created the companion file at the time you created the design[182] using the new design wizard. If that is the case you are all set. You can start connecting components and data.

There is a companion file status indicator on the top right of the editor toolbar. The connection has three states: no connected file, connected to a java companion file or connected to multiple java files.



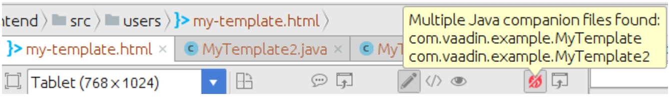
No Java companion file found

Figure 20. Design is not connected to any Java companion file



Connected to Java companion file com.vaadin.example.MyTemplate

Figure 21. Design is connected to a Java companion file



Multiple Java companion files found:
com.vaadin.example.MyTemplate
com.vaadin.example.MyTemplate2

Figure 22. Design is connected to multiple Java companion files

When you have connected your design with a Java companion file, you can simply navigate to the file by clicking on the connected indicator. However, if you do not have a companion file for your design, you need to create one manually. Here is a code snippet for a companion file that is a valid starting point for any design. It has been written as if it was a companion to an imaginary my-design.js. You have to adapt it by providing the correct values for your design to the Tag and JsModule annotations. The class names are not relevant for Designer.

```
import com.vaadin.flow.templatemodel.TemplateModel;
import com.vaadin.flow.component.Tag;
import com.vaadin.flow.component.dependency.JsModule;
import
com.vaadin.flow.component.polymertemplate.PolymerTemplate
;

@Tag("my-design")
@JsModule("./src/views/my-design.js")
public class MyDesign extends PolymerTemplate<MyDesign
.MyDesignModel> {

    public MyDesign() {
        // You can initialise any data required for the
connected UI components here.
    }

    public interface MyDesignModel extends TemplateModel
{
        // Add setters and getters for template
properties here.
    }
}
```

In general, any Java class will be picked up by Designer as a companion file for the design, as long as the class meets the following requirements:

1. It is a descendant of com.vaadin.flow.component.Component
2. It is annotated with com.vaadin.flow.component.Tag annotation. The annotation's value matches the design's tag in custom element definition (e.g. `customElements.define('my-design', MyDesign)`)
3. The value of the com.vaadin.flow.component.dependency.JsModule annotation matches the design path.

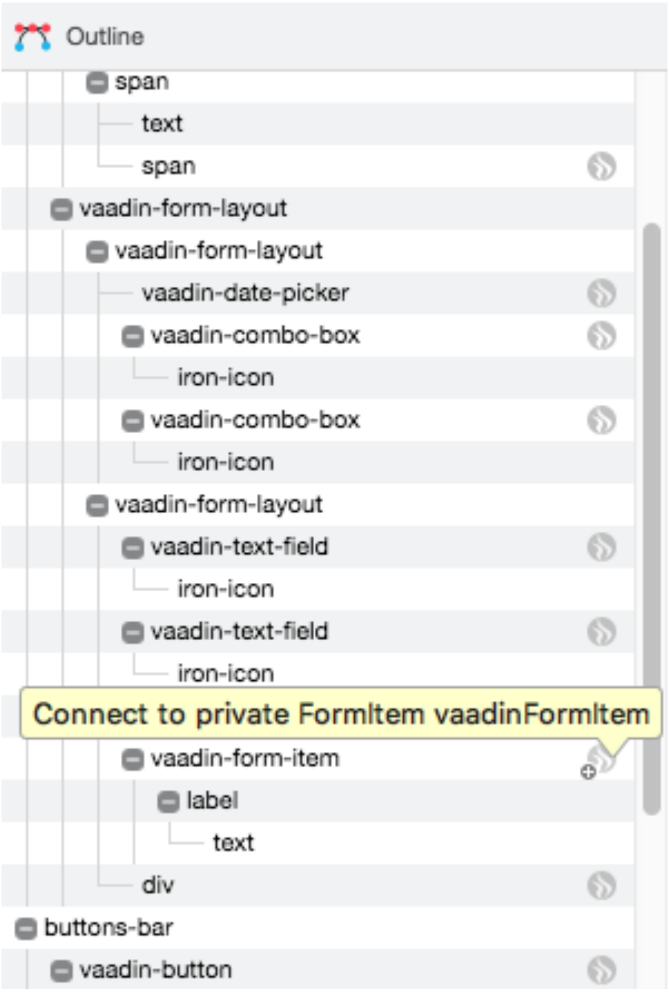
So, if you have a specific need, you can freely customize the companion class to match your demands. You can learn more about connecting designs and Java classes in Flow documentation[183].

Connecting Components

Designer helps to connect the components used in the design to Java but before that can happen you need three things:

1. You need a companion file for the design. See the The Java Companion File for how to get one.
2. The component you want to connect to Java should have its id property set to a unique value (among all the id property values in the same design). If its id is empty, Designer will generate one for you.
3. The project must have Vaadin Flow component integrations as dependencies. Those are needed to correctly set the type of the new field.

When a companion file for the design exists, you can connect components to Java using the Outline view. When you hover over a component in the Outline and the component has a Java API, a connection button will appear on the same row with the component name. By clicking the connection button, you can connect the component to Java. When the component is connected, the connection button will stay visible in the outline. This is illustrated in the following picture.



Connect to private FormItem vaadinFormItem

Figure 23. Adding a Field

NOTE When you make changes in your Java companion file, it will take a few seconds to update the status of the connection buttons in the outline and the connection indicator on the toolbar.

Choosing to add the field in the previous picture will insert the following field to the companion class:

```java
@Id("vaadinFormItem")
private FormItem vaadinFormItem;
```

Flow uses the @Id annotation to connect the UI component to the field. The value in the annotation must match the id property of the component in the design. Otherwise, you are free to change the type, name and visibility of the field. Just be careful not to break it for Flow.

Take a look at the Flow documentation to learn more about binding components in Flow[184].

You can disconnect a component by clicking the connection button of a connected component. Disconnecting a component will erase the corresponding field from the companion class along with its @Id annotation.

You should not have more than one companion class for a design, or more than one field annotated with the same @Id value, but if you do, all of them will be shown in the Java checkbox tooltip so that you can easily locate them to fix the problem manually.

Connecting Data

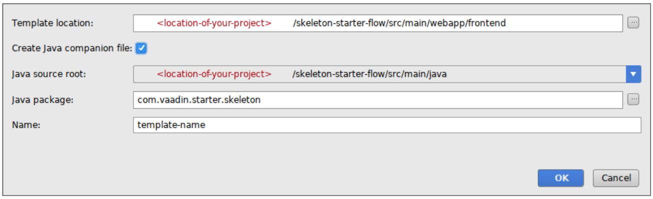
You can also bind data from Java to the UI. Designer provides you with a starting point by adding the template model inner class into the companion file when the file is created. You can learn more about binding data to designs in Flow documentation[185].

21.3. Tutorials

21.3.1. Adding A New View To An Application

In this tutorial, we will add a new view to a starter application. We'll also make our view available by adding a route to that view.

1. First off, let's start with a Project Base from https://vaadin.com/start
2. Build the project using `mvn clean package` so that we get all the necessary components.
3. Import the project into your IDE of choice.
4. Create a new Vaadin platform design through the IDE menu (New > Vaadin platform Design).
5. In the wizard, specify the location of the design HTML file, and also check the companion file checkbox and specify a location for the companion Java-file.



Template location:	<location-of-your-project> /skeleton-starter-flow/src/main/webapp/frontend
Create Java companion file:	✓
Java source root:	<location-of-your-project> /skeleton-starter-flow/src/main/java
Java package:	com.vaadin.starter.skeleton
Name:	template-name

Figure 24. Create a new Vaadin platform design dialog

Open the design HTML file we created. From the palette, drag a `vaadin-vertical-layout` onto the paper. After that, drag a `vaadin-textfield` onto the paper and a `vaadin-button` below the textfield.

Next, connect the button and the textfield to Java by hovering over them in the Outline and clicking the connection button that appears on the hovered row. When we now open the Java file, the button and textfield have appeared as Java fields there.

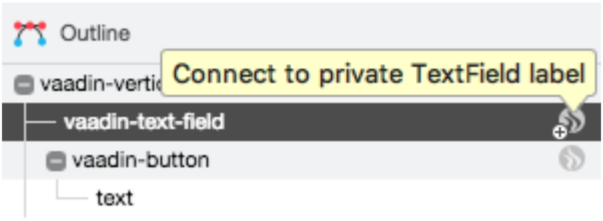


Figure 25. Export a component to Java companion file

Add a constructor for the Java class. In the constructor, we need to add a click listener for the button. In that listener, we will show a notification using the contents of the textfield. The code is as follows:

```java
public MyDesign() {
    vaadinButton.addClickListener( event -> {
        Notification.show(vaadinTextField.getValue(),
1000*10,
            Notification.Position.MIDDLE);
    });
}
```

To show our new view, a route needs to be added. Adding a route is done through the `@Route` annotation. That annotation is added to the design, and it has only one value:

the path of the route to this design.

```
@Tag("my-design")
@HtmlImport("frontend://src/my-design.html")
@Route("myroute")
public class MyDesign extends PolymerTemplate<MyDesign
.MyDesignModel> {
```

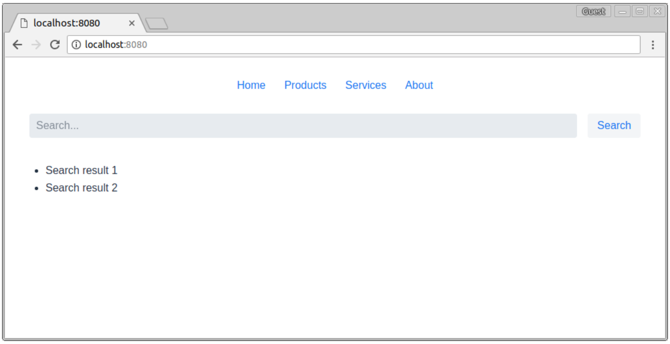
This is what the class declaration should look like for a design named `my-design`.

Running the project with `mvn clean package jetty:run` should make our new view accessible at `http://localhost:8080/myroute`

It is also possible to use the view as a regular component on the server side. Just create a new instance of the view with `new` and add it to a layout as follows: `someLayout.addComponent(new MyDesign());`

21.3.2. Compose Views From Reusable Designs

In this tutorial, we will create a content view, with a top menu and a search bar, by using reusable components in Vaadin Designer. Then we could implement some simple logic to connect the content view and the search bar using Java code. The result will look like the figure below:



- Search result 1
- Search result 2

Figure 26. Simple search view

Prepare workspace

1. First off, let’s start with a Project Base from https://vaadin.com/start
2. Build the project using `mvn clean package` so that we get all the necessary components.
3. Import the project into your IDE of choice. (In this tutorial, we are going to use IntelliJ IDEA)

Create the top menu component

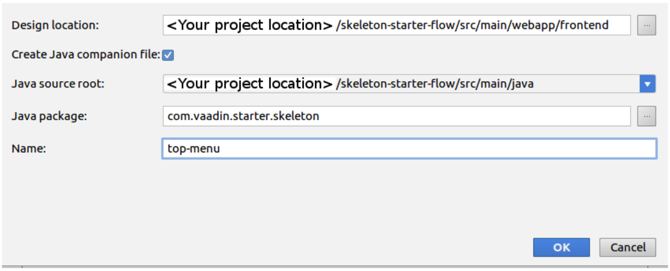
Create a new design by using New Vaadin platform design wizard

1. Create a new Vaadin platform Design via the IDE menu
2. Select the folder `src/main/webapp/frontend` for the Design location
   - Optional: Select the Create Java companion file checkbox to generate a Java companion file for this component.

3. Enter design name, for example: `top-menu`

4. Press OK to create the design

> **TIP** In this tutorial, we will not use the `top-menu` component in Java code, so creating a Java companion file is an optional step.



Design location:	<Your project location> /skeleton-starter-flow/src/main/webapp/frontend
Create Java companion file:	✓
Java source root:	<Your project location> /skeleton-starter-flow/src/main/java
Java package:	com.vaadin.starter.skeleton
Name:	top-menu

Figure 27. Create top-menu design

Now, let's add a `vaadin-horizontal-layout` as a root with four `vaadin-button` as children with the below steps

1. Search for `vaadin-horizontal-layout` in the Search field of the Palette. Use the `vaadin-horizontal-layout` from the category `Parts`.

2. Drag and drop it into the paper or just double-click on it.

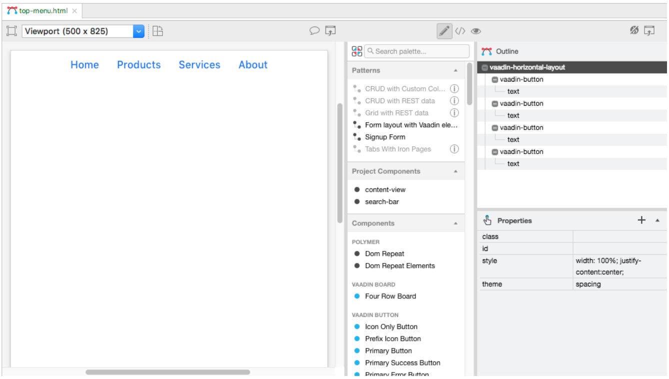
3. Similarly, add four `Tertiary Buttons` into the `vaadin-horizontal-layout`.

   - Tertiary buttons are used in this tutorial because they look more like menu items. You can find more button themes in the vaadin-button page[186]

4. Change the text content of the buttons to anything you want by selecting the `text` under

`vaadin-button` in the Outline, then editing its value in the Properties view.

5. Select the `vaadin-horizontal-layout` and change its `style` property to `width: 100%; justify-content: center;`. This will make our buttons stay in the center of the design.
   - Optional: Using `spacing` theme for `vaadin-horizontal-layout` will append a little bit more space between our buttons



```
vaadin-horizontal-layout
    vaadin-button
        text
    vaadin-button
        text
    vaadin-button
        text
    vaadin-button
        text
```

class	
id	
style	width: 100%; justify-content:center;
theme	spacing

Figure 28. top-menu design

top-menu.html

```
<link rel="import" href=
"../bower_components/polymer/polymer.html">
<link rel="import" href="../bower_components/vaadin-
ordered-layout/src/vaadin-horizontal-layout.html">
<link rel="import" href="../bower_components/vaadin-
button/src/vaadin-button.html">

<dom-module id="top-menu">
    <template>
        <style include="shared-styles">
            :host {
                display: block;
            }
        </style>
        <vaadin-horizontal-layout theme="spacing" style=
"width: 100%; justify-content:center;">
            <vaadin-button theme="tertiary">
                Home
            </vaadin-button>
            <vaadin-button theme="tertiary">
                Products
            </vaadin-button>
            <vaadin-button theme="tertiary">
                Services
            </vaadin-button>
            <vaadin-button theme="tertiary">
                About
            </vaadin-button>
        </vaadin-horizontal-layout>
    </template>

    <script>
        class TopMenu extends Polymer.Element {
            static get is() {
                return 'top-menu';
            }

            static get properties() {
                return {
                    // Declare your properties here.
                };
            }
        }
        customElements.define(TopMenu.is, TopMenu);
    </script>
</dom-module>
```

Create the search bar component

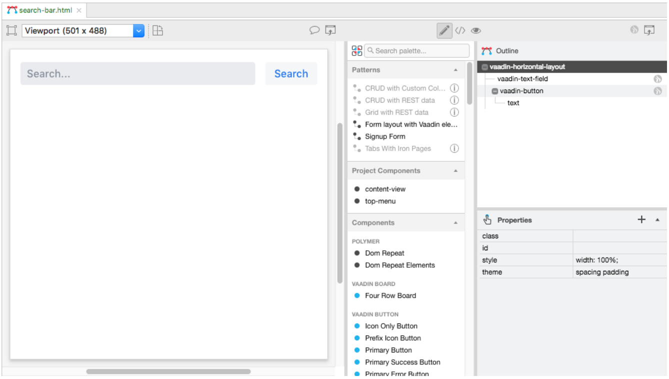
With the same steps from the `top-menu` component creation, let's create another component named `search-bar` and its Java companion file, as well. The `search-bar` component will contain two elements: search field and search button, which line up horizontally. Therefore, we can use `vaadin-horizontal-layout` as the root, `vaadin-text-field` for the search field and `vaadin-button` for the search button.

We can decorate the component with these steps

1. Set `style` property of `vaadin-horizontal-layout` to `width: 100%;` because we don't want the search bar to expand vertically
2. Add `spacing` and `padding` theme for the layout to reserve some spaces between the elements and the padding from the document.
3. Set `style` property of `vaadin-text-field` to `flex-grow: 1;`
4. Set your placeholder text for the search field using `placeholder` property

Later in this tutorial, we might need to use the search field and the search button in Java code. Let's export them by checking their Java checkboxes in the Outline.

The `search-bar` design should look like search-bar design



class	
id	
style	width: 100%;
theme	spacing padding

Figure 29. search-bar design

search-bar.html

```
<link rel="import" href=
"../bower_components/polymer/polymer.html">
<link rel="import" href="../bower_components/vaadin-
ordered-layout/src/vaadin-horizontal-layout.html">
<link rel="import" href="../bower_components/vaadin-text-
field/src/vaadin-text-field.html">
<link rel="import" href="../bower_components/vaadin-
button/src/vaadin-button.html">

<dom-module id="search-bar">
    <template>
        <style include="shared-styles">
            :host {
                display: block;
            }
        </style>
        <vaadin-horizontal-layout theme="spacing padding"
style="width: 100%;">
            <vaadin-text-field placeholder="Search..."
style="flex-grow: 1;"
                                id="vaadinTextField"
></vaadin-text-field>
            <vaadin-button id="vaadinButton">
                Search
            </vaadin-button>
        </vaadin-horizontal-layout>
    </template>

    <script>
        class SearchBar extends Polymer.Element {
            static get is() {
                return 'search-bar';
            }

            static get properties() {
                return {
                    // Declare your properties here.
                };
            }
        }
        customElements.define(SearchBar.is, SearchBar);
    </script>
</dom-module>
```

Create the content view

In the same way as above, we can create a new design called `content-view` along with its Java companion file `ContentView.java`. In this design, we will add a `vaadin-vertical-layout` as the root layout. After that, from the Project Components section of the Palette, we can add the `top-menu` and the `search-bar` as children of the layout.

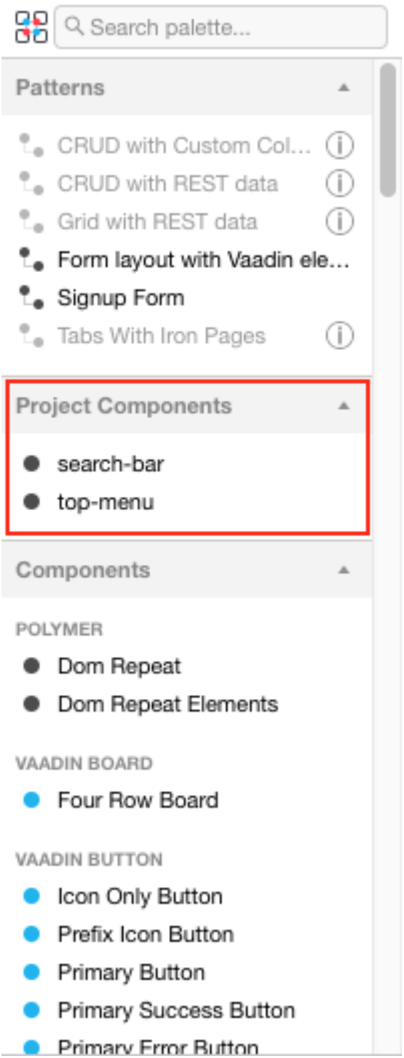


Figure 30. Project Components

We also need a `div` and `ul` as the container for our search result in the view. Then our `content-view` structure will be like content-view design structure.

To prepare for some simple functionalities later, we should export `search-bar` and `ul` to Java.

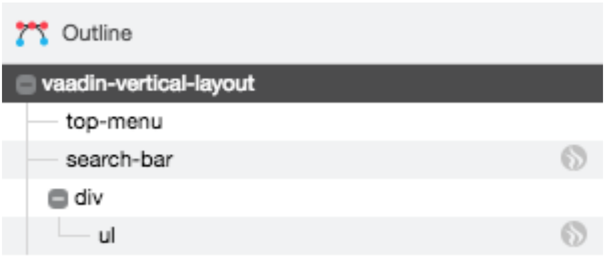


Figure 31. content-view design structure

Let's add some additional styles for the design

1. Set `top-menu` style property to `width: 100%;`
2. Set `search-bar` style property to `width: 100%;`
3. Set `div` style property to `width: 100%; flex-grow: 1;`

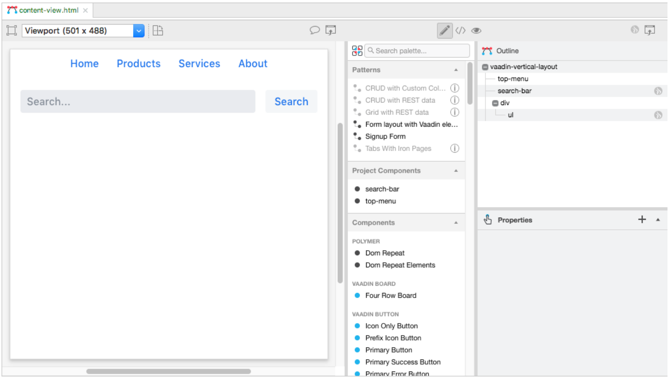


Figure 32. content-view design

content-view.html

```
<link rel="import" href=
"../bower_components/polymer/polymer.html">
<link rel="import" href="../bower_components/vaadin-
ordered-layout/src/vaadin-vertical-layout.html">
<link rel="import" href="top-menu.html">
<link rel="import" href="search-bar.html">

<dom-module id="content-view">
    <template>
        <style include="shared-styles">
            :host {
                display: block;
            }
        </style>
        <vaadin-vertical-layout style="width: 100%;
height: 100%;">
            <top-menu style="width: 100%;"></top-menu>
            <search-bar id="searchBar" style="width:
100%;"></search-bar>
            <div style="width: 100%; flex-grow: 1;">
                <ul id="ul"></ul>
            </div>
        </vaadin-vertical-layout>
    </template>

    <script>
        class ContentView extends Polymer.Element {
            static get is() {
                return 'content-view';
            }

            static get properties() {
                return {
                    // Declare your properties here.
                };
            }
        }
        customElements.define(ContentView.is,
ContentView);
    </script>
</dom-module>
```

Add a route to the view

To add a route to the `content-view`, we need to open the Java companion file (`ContentView.java`) by either navigating via the project explorer, or clicking on the Java connection indicator. Then add `@Route("content-view")` annotation to the `ContentView` class.

ContentView.java

```
...
@Tag("content-view")
@HtmlImport("frontend://src/content-view.html")
@Route("content-view")
public class ContentView extends PolymerTemplate
<ContentView.ContentViewModel> {
...
```

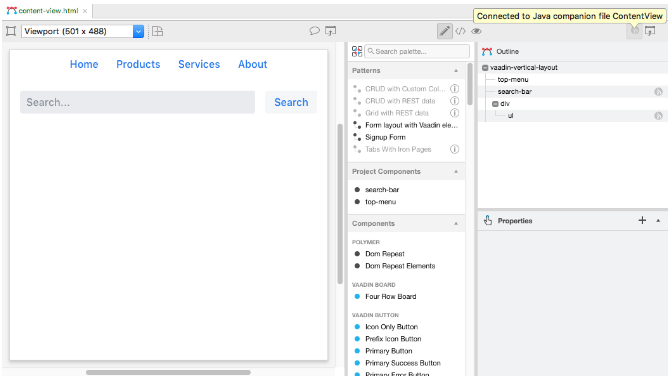


Figure 33. content-view design

Add simple search functionality

Let's add some code to set the content from search field to the content view when pressing the Search button.

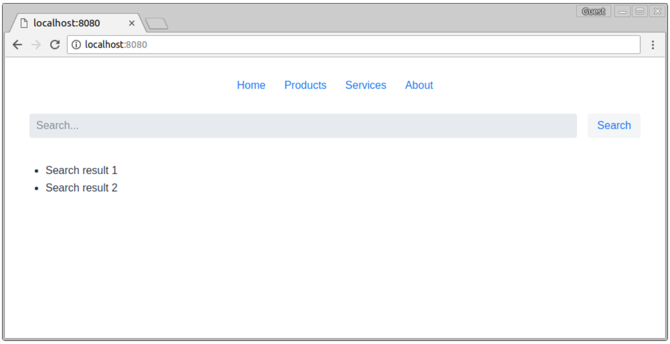
SearchBar.java

```
@Tag("search-bar")
@HtmlImport("frontend://src/search-bar.html")
public class SearchBar extends PolymerTemplate<SearchBar
.SearchBarModel> {

    @Id("vaadinTextField")
    private TextField vaadinTextField;
    @Id("vaadinButton")
    private Button vaadinButton;

    private final List<SearchBarListener> listeners;

    public interface SearchBarModel extends TemplateModel
{

    }

    public SearchBar() {
        listeners = new CopyOnWriteArrayList<>();
        vaadinButton.addClickListener(buttonClickEvent ->
{
            for (SearchBarListener listener :
                    listeners) {
                listener.onSearch(vaadinTextField
.getValue());
            }
        });
    }

    public void addSearchListener(SearchBarListener
listener) {
        listeners.add(listener);
    }

    public void removeSearchListener(SearchBarListener
listener) {
        listeners.remove(listener);
    }

    @FunctionalInterface
    public interface SearchBarListener {
        void onSearch(String text);
    }
}
```

ContentView.java

```
@Tag("content-view")
@HtmlImport("frontend://src/content-view.html")
@Route("content-view")
public class ContentView extends PolymerTemplate
<ContentView.ContentViewModel> {

    @Id("ul")
    private UnorderedList ul;
    @Id("searchBar")
    private SearchBar searchBar;
    private final SearchBar.SearchBarListener
searchBarListener;

    public interface ContentViewModel extends
TemplateModel {

    }

    public ContentView() {
        searchBarListener = text -> ul.add(new ListItem
(text));
        searchBar.addSearchListener(searchBarListener);
    }

    @Override
    protected void onDetach(DetachEvent detachEvent) {
        super.onDetach(detachEvent);
        searchBar.removeSearchListener(searchBarListener
);
    }
}
```

It's time to start the application and see our result by running `mvn jetty:run` from the project folder. Our view is available at http://localhost:8080/content-view



- Search result 1
- Search result 2

Figure 34. Final result

----

[171] https://vaadin.com/eclipse
[172] https://www.w3.org/TR/custom-elements/#valid-custom-element-name
[173] https://vaadin.com/components/
[174] https://www.npmjs.com/
[175] https://developer.mozilla.org/en-US/docs/Web/API/CustomElementRegistry/define
[176] https://polymer-library.polymer-project.org/3.0/docs/devguide/registering-elements
[177] https://vaadin.com/themes
[178] https://vaadin.com/docs/v12/flow/theme/theming-overview.html
[179] https://vaadin.com/docs/v12/flow/theme/using-component-themes.html
[180] https://vaadin.com/docs/v12/flow/theme/application-theming-basics.html
[181] https://vaadin.com/docs/v12/flow/theme/using-component-themes.html#theme-variants
[182] https://vaadin.com/docs/flow/getting-started/designer-getting-started.html#designer.getting-started.design
[183] https://vaadin.com/docs/flow/flow/polymer-templates/tutorial-template-basic.html
[184] https://vaadin.com/docs/flow/flow/polymer-templates/tutorial-template-components.html
[185] https://vaadin.com/docs/flow/flow/polymer-templates/tutorial-template-bindings.html

[186] https://vaadin.com/components/vaadin-button/

22. Vaadin Charts

22.1. Overview

Vaadin Charts is a feature-rich interactive charting library for Vaadin. It provides multiple different chart types for visualizing one- or two-dimensional tabular data, or scatter data with free X and Y values. You can configure all the chart elements with a powerful API as well as the visual style using CSS. The built-in functionalities allow the user to interact with the chart elements in various ways, and you can define custom interaction with events.

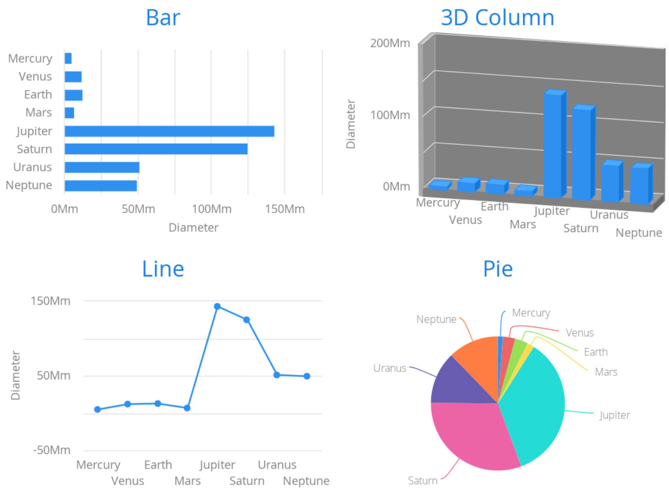


Figure 35. Vaadin Charts

22.1.1. Licensing

Vaadin Charts is a commercial product licensed under the CVAL License (Commercial Vaadin Add-On License). You need to install a license key in order to develop your application with Vaadin Charts.

You can purchase Vaadin Charts or obtain a free trial key from the license section in Vaadin website. You need to be a registered user to obtain the key.

22.2. Installing Vaadin Charts for Flow

As with most components for Vaadin Flow, you can install Vaadin Charts for Flow as a Maven dependency in your project.

Using Vaadin Charts requires a license key, which will be prompted on development time after 24 hours of the first time the application with Vaadin Charts is opened.

22.2.1. Maven Dependency

Install vaadin-charts by adding the dependency to the project (here as a Maven dependency in pom.xml):

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-charts-flow</artifactId>
</dependency>
```

You do not need to specify the version number as long as you have vaadin-bom imported. Otherwise add

```
<version>6.0.0</version>
```

Update the version number to the one you want

You also need to define the Vaadin add-ons repository if not already defined:

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

22.2.2. Installing a License Key

You need to have a valid license in order to develop your application with Vaadin Charts. 24 hours after you open the application with Vaadin Charts in a local browser, you will see a pop-up that asks you to validate your subscription. This popup will open a new tab where you will have to login using your Vaadin account. If the license is valid, it will be saved to the local storage of the browser and you will not see the pop-up again.

More information can be found at "Validating Vaadin Subscription".

22.3. 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.

```
Chart chart = new Chart(ChartType.COLUMN);

//or

Chart chart = new Chart();
chart.getConfiguration().getChart().setType(ChartType.COL
UMN);
...
layout.add(chart);
```

The chart types are described in "Chart Types". The main parts of a chart are illustrated in Chart Elements. Styling a chart is discussed in "CSS Styling"

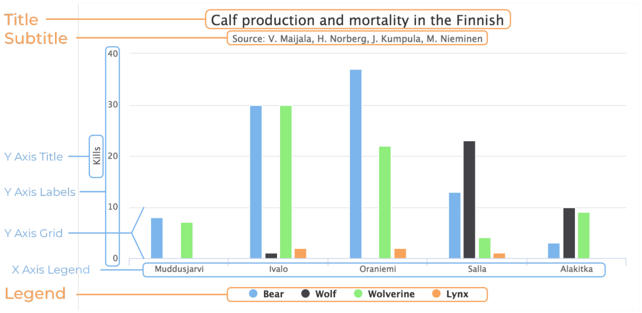


Figure 36. Chart Elements

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.

22.3.1. 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 "Chart Configuration".

22.3.2. 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 Mixed Type Charts.

The shared plot options are described in "Plot Options".

22.3.3. 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,
               51100, 49500);
conf.addSeries(series);
```

The data can be specified with a number of different series types DataSeries, ListSeries, HeatSeries and TreeSeries.

Data point features, such as name and data labels, 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 "Chart Data".

22.3.4. Axis Configuration

One of the most common tasks for charts is customizing its axes. At the 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) + \'
Mm\';}");
yaxis.getLabels().setStep(2);
conf.addyAxis(yaxis);
```

22.3.5. 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[][] = {        // Location:
        {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 Multiple Series in a Chart. Stacking is enabled with setStacking() in PlotOptionsColumn.

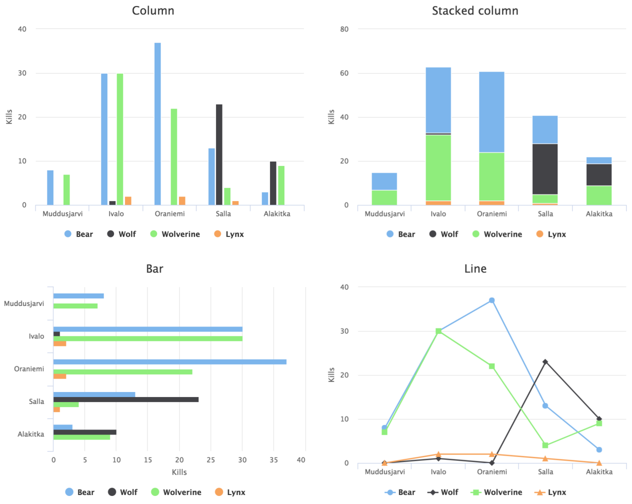


Figure 37. Multiple Series in a Chart

22.3.6. 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 control the animation 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
DataSeries series1 = new DataSeries();
PlotOptionsColumn options1 = new PlotOptionsColumn();
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();
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.

NOTE Gauge and solid gauge series should not be combined with series of other types.

NOTE A bar series inverts the entire chart, combine with care.

22.4. 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`	`candlestick`
`column`	`columnrange`	`errorbar`	`flags`
`funnel`	`gauge`	`heatmap`	`line`
`ohlc`	`pie`	`polygon`	`pyramid`
`scatter`	`solidgauge`	`spline`	`treemap`

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. Configuring `Data Labels` is common to all chart types. Configuring `Markers` is available for all chart types displaying point data.

The basic chart types and their variants are covered in the following subsections.

22.4.1. 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 2. Line Chart Subtypes

ChartType	Plot Options Class
LINE	PlotOptionsLine
SPLINE	PlotOptionsSpline

22.4.2. 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 3. 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 "Range Series", or with DataSeries, described in "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.

See Data Point Markers for plot options regarding markers.

22.4.3. 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 "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 Area and Column Range Charts. They require the use of RangeSeries for defining the lower and upper values.

Table 4. 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.

22.4.4. 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.

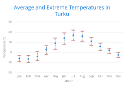


Figure 38. Error Bars in a Scatter Chart

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);

// 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));
...

// Need to be used for series to be recognized as error
bar
PlotOptionsErrorbar barOptions = new PlotOptionsErrorbar
();
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. See the API documentation for details regarding the plot options.

NOTE Although most visual styles are defined in CSS, some options like whiskerLength are set through Java API.

22.4.5. 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 Error Bars, sharing the box and whisker elements.

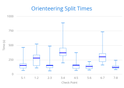


Figure 39. Box Plot Chart

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);

// 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.

For example:

```
// Set median line color and thickness
PlotOptionsBoxplot plotOptions = new PlotOptionsBoxplot(
);
plotOptions.setWhiskerLength("80%");
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 BoxPlotItem. 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);
```

22.4.6. Scatter Charts

Scatter charts display a set of unconnected data points. The name refers to freely given X and Y coordinates, so the DataSeries or DataProviderSeries are usually the most meaningful data series types for scatter charts.

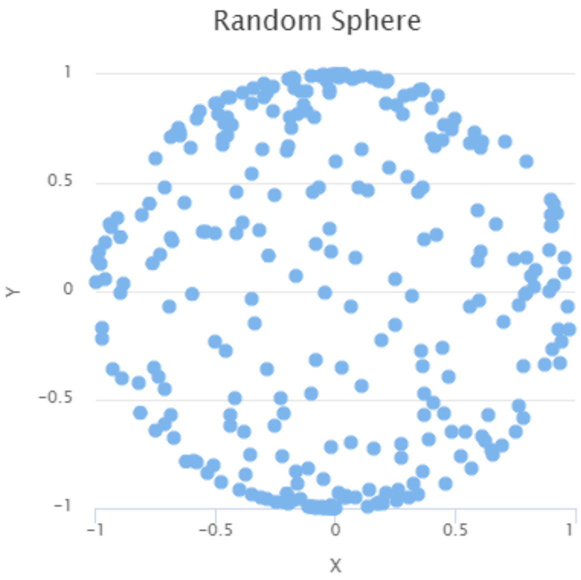


Figure 40. Scatter Chart

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);

// Modify the default configuration a bit
Configuration conf = chart.getConfiguration();
conf.setTitle("Random Sphere");
conf.getLegend().setEnabled(false); // Disable legend
conf.getxAxis().setTitle("X");
conf.getyAxis().setTitle("Y");
conf.getxAxis().setMax(1);
conf.getxAxis().setMin(-1);
conf.getyAxis().setMax(1);
conf.getyAxis().setMin(-1);

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);

    DataSeriesItem point = new DataSeriesItem(x,y);
    series.add(point);
}
conf.addSeries(series);
```

The result was shown in Scatter Chart.

22.4.7. 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 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.

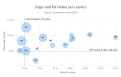


Figure 41. Bubble Chart

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.

22.4.8. 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 Pie Chart.

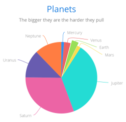


Figure 42. Pie Chart

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 DataSeries or ContainerSeries, which allow labeling the data points, should be used for pie charts.

```
DataSeries series = new DataSeries();
series.add(new DataSeriesItem("Mercury", 4900));
series.add(new DataSeriesItem("Venus", 12100));
...
conf.addSeries(series);
```

If a data point, as defined as a DataSeriesItem in a DataSeries, has the *sliced* property enabled, it is shown as slightly cut away from the pie.

```
// Slice one sector out
DataSeriesItem earth = new DataSeriesItem("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 Overlaid Pie and Donut Chart.

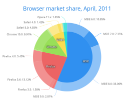


Figure 43. Overlaid Pie and Donut Chart

22.4.9. 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.

NOTE Gauge and solid gauge series should not be combined with series of other types.

NOTE A bar series inverts the entire chart, combine with care.

Let us consider the following gauge:

```
Chart chart = new Chart(ChartType.GAUGE);
```

After the settings done in the subsequent sections, it will show as in A Gauge.

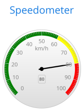


Figure 44. A Gauge

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);

PlotBand green = new PlotBand(0, 60, null);
green.setClassName("green");

PlotBand yellow = new PlotBand(60, 80, null);
yellow.setClassName("yellow");

PlotBand red = new PlotBand(80, 100, null);
red.setClassName("red");

yaxis.setPlotBands(green, yellow, red);

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 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.add(tf);

Button update = new Button("Update", (e) -> {
    Integer newValue = new Integer(tf.getValue());
    series.updatePoint(0, newValue);
});
layout.add(update);
```

22.4.10. 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);
```

After the settings done in the subsequent sections, it will show as in A Solid Gauge.

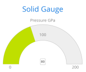


Figure 45. A Solid Gauge

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.setInnerRadius("60%");  // To make it an arc and not
circle
bkg.setOuterRadius("100%"); // Default - not necessary
bkg.setShape(BackgroundShape.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
```

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 "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.add(tf);

Button update = new Button("Update", (e) -> {
    Integer newValue = new Integer(tf.getValue());
    series.updatePoint(0, newValue);
});
layout.add(update);
```

22.4.11. 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 "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);

// 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 Area and Column Range Chart.

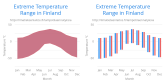


Figure 46. Area and Column Range Chart

22.4.12. 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 Wind Rose and Spiderweb Charts.

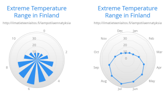


Figure 47. Wind Rose and Spiderweb Charts

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.

22.4.13. 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 dinimishes 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.

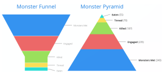


Figure 48. Funnel and Pyramid Charts

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.

22.4.14. 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.

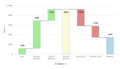


Figure 49. Waterfall Charts

Waterfall charts have chart type WATERFALL.

Waterfall charts can be styled by CSS using the following classes: .highcharts-waterfall-series, .highcharts-point, .highcharts-negative, .highcharts-sum, .highcharts-intermediate-sum.

22.4.15. Heat Maps

A heat map is a two-dimensional grid, where the color of a grid cell indicates a value.

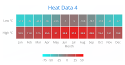


Figure 50. Heat Maps

Heat maps have chart type HEATMAP.

22.4.16. 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.

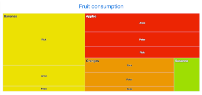


Figure 51. Tree Maps

Tree maps have chart type TREEMAP.

In order to create a Tree Map chart,you need to create a class that extends TreeSeriesItem and add an colorIndex property:

```
public static class MapTreeSeriesItem extends
TreeSeriesItem {
    private Number colorIndex;

    public Number getColorIndex() {
        return colorIndex;
    }

    public void setColorIndex(Number colorIndex) {
        this.colorIndex = colorIndex;
    }
}
```

Then, you need to specify a color index for each of the top levels series items:

```
TreeSeries series = new TreeSeries();

MapTreeSeriesItem apples = new MapTreeSeriesItem();
apples.setId("A");
apples.setName("Apples");
apples.setColorIndex(0);

...

TreeSeriesItem anneA = new TreeSeriesItem("Anne", apples,
5);
TreeSeriesItem rickA = new TreeSeriesItem("Rick", apples,
3);
TreeSeriesItem peterA = new TreeSeriesItem("Peter",
apples, 4);

...

series.addAll(apples, anneA, rickA, peterA);
```

22.4.17. 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.

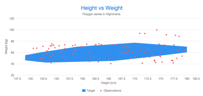


Figure 52. Polygon combined with Scatter

Polygons have chart type POLYGON.

22.4.18. 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.

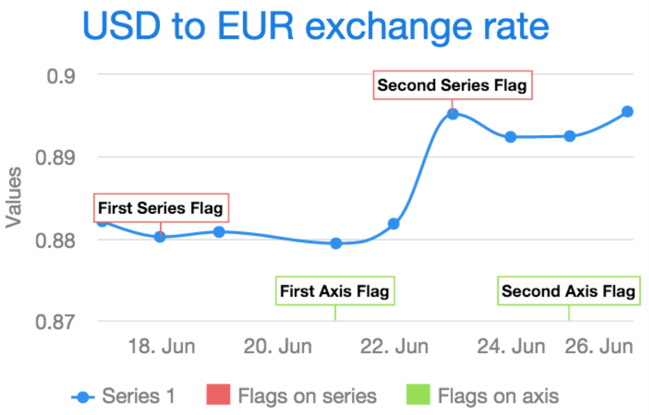


Figure 53. Flags placed on an axis and a series

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.

22.4.19. 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 consist 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.

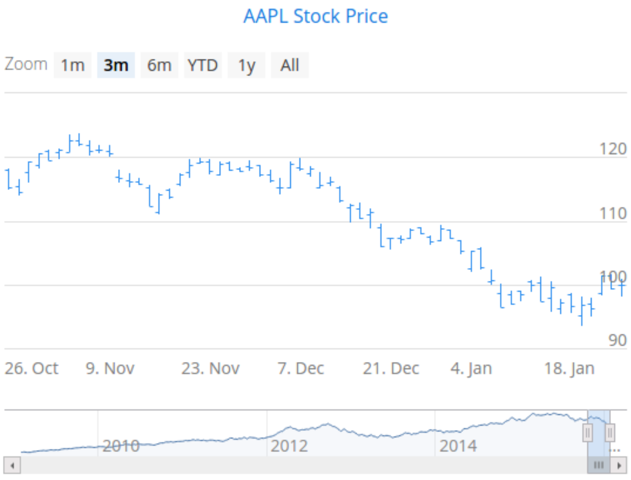


Figure 54. OHLC Chart.

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.

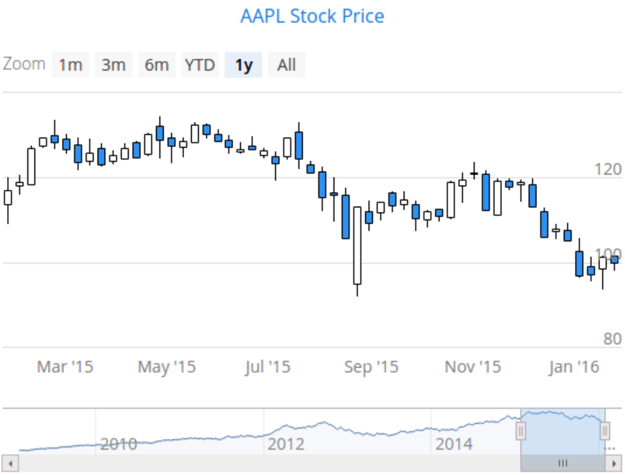


Figure 55. Candlestick Chart.

To attach data to an OHLC or a candlestick chart, you need to use a DataSeries or a ContainerSeries. See "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.getTitle().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();
```

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 "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.

22.4.20. Data Labels

You can change how labels that appears next to data points are displayed for some series types (it’s not available for BOXPLOT and ERRORBAR).

The data labels properties in the DataLabels class are summarized in the following:

- align: HorizontalAlign (left, center, right)
- allowOverlap: Boolean whether to allow data labels to Wrap
- borderRadius: Number with the border radius in pixels
- className: String a class name for the data label to be added to the node to allow custom styles by CSS
- enabled: Boolean whether the data label is enabled or disabled
- format: String a format string for the label (see more at "Using Format Strings")
- formatter: String a format string containing a JavaScript function for the label (see more at "Using a JavaScript Formatter")

Also, data label can be styled by CSS with .highcharts-data-label-box and .highcharts-data-label class names.

22.4.21. Data Point Markers

Lines charts and other charts that display data points, such as scatter 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 DataSeriesItem. 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 stroke and a fill colors, which are set using a CSS selector .highcharts-markers .highcharts-point.

```
// Set radius and symbol
marker.setRadius(10);
marker.setSymbol(MarkerSymbolEnum.DIAMOND);

point.setMarker(marker);
series.add(point);
```

Marker size is determined by the radius parameter, which is given in pixels.

```
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 frontend folder, you can determine its URL as follows:

```
String url = "frontend/img/smiley.png";
marker.setSymbol(new MarkerSymbolUrl(url));
```

You can use width and height to resize the marker. The radius property are not applicable to image symbols.

22.4.22. 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.

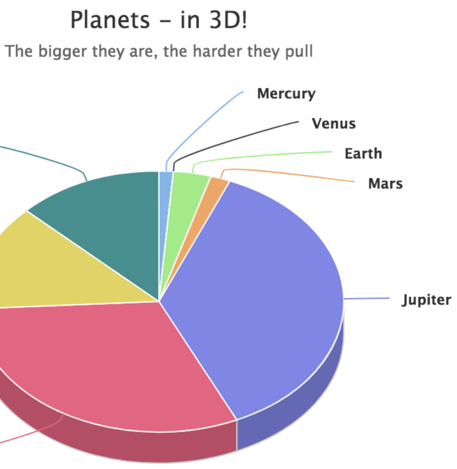


Figure 56. 3D Charts

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(1);
frame.setBack(back);
options3d.setFrame(frame);
```

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 2x2x2 cube centered at the origin. The 3D depth was set to 135 earlier. The result is illustrated in 3D Scatter Chart.

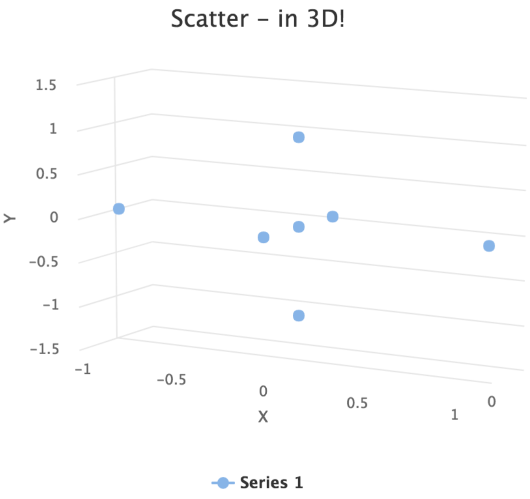


Figure 57. 3D Scatter Chart

22.5. 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)
- legend: Legend (see Legend)
- pane: Pane
- plotoptions: PlotOptions (see Plot Options)

- series: Series
- subtitle: Subtitle
- title: Title
- tooltip: Tooltip
- xAxis: XAxis (see Axes)
- yAxis: YAxis (see Axes)

For data configuration, see "Chart Data". For styling, see "CSS Styling"

22.5.1. Plot Options

The plot options are used to configure the data series in the chart. 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);
```

NOTE GaugeOptions should not be combined with other plot options.

NOTE Gauge and solid gauge series should not be combined with series of other types.

NOTE A bar series inverts the entire chart, combine with care.

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.setAnimation(false);
}
...
PlotOptionsSpline splineOptions = new PlotOptionsSpline(
);
PlotOptionsLine lineOptions = 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*.

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*.

22.5.2. 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 Chart Axis Elements.

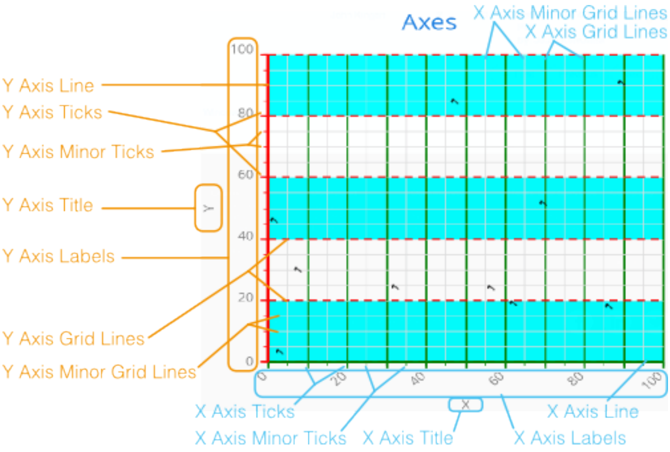


Figure 58. Chart Axis Elements

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 xlabels = xaxis.getLabels();
xlabels.setAlign(HorizontalAlign.CENTER); // Default
xlabels.setRotation(-45);
xlabels.setStep(2); // Every 2 major tick
// The class highcharts-axis-labels can be used to style
further with CSS.
```

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 Formatting Labels. Defaults to " {value}".

formatter

: A JavaScript formatter for the labels, as described in 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;}
```

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
```

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().

22.5.3. 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.

align

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 Legend example.

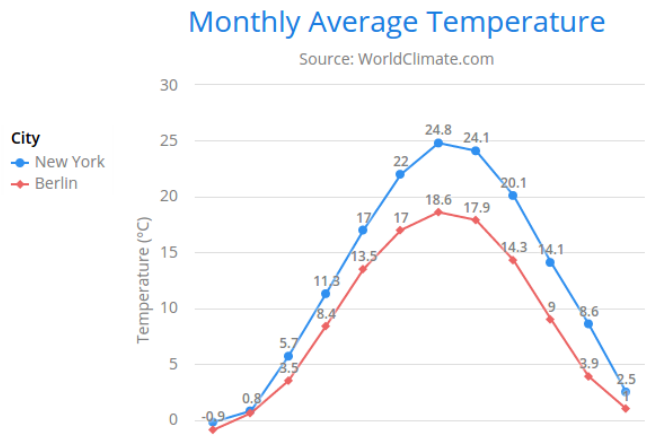


Figure 59. Legend example

22.5.4. 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, point.x in data points and tooltips
- series.name 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.

22.6. 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, HeatSeries, and RangeSeries.

22.6.1. List Series

The ListSeries is essentially a helper type that makes the handling of simple sequential data easier than with

DataSeries. 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().

22.6.2. Generic Data Series

The DataSeries can represent a sequence of data points at an interval as well as scatter data. Data points are represented with the DataSeriesItem class, which has x and y properties for representing the data value. Each item can be given a category name.

```
DataSeries series = new DataSeries();
series.setName("Total Reindeer Population");
series.add(new DataSeriesItem(1959, 181091));
series.add(new DataSeriesItem(1960, 201485));
series.add(new DataSeriesItem(1961, 188105));
series.add(new DataSeriesItem(1962, 177206));

// Modify the radius of one point
series.get(2).getMarker().setRadius(20);
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 described in "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 Range Series.

22.6.3. 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 "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);
```

22.6.4. 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 ContainerDataSeries, 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 `getDescription()`, `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 Chart Bound to a DataProvider.

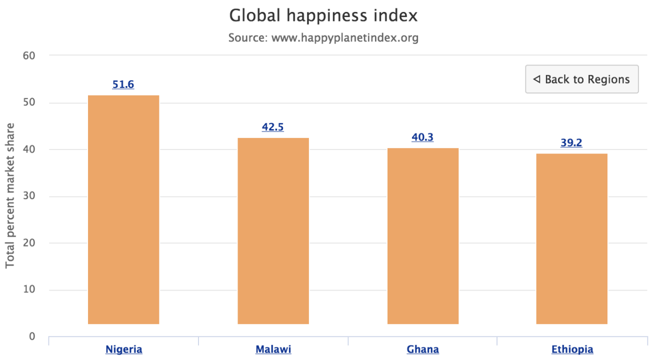


Figure 60. Chart Bound to a DataProvider

NOTE Dynamic changes to the data will be loaded in the data series after calling the refreshAll() method in the DataProvider. This behavior can be disabled by setting the automaticChartUpdateEnabled property to false in DataProviderSeries.

22.6.5. 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 addItemWithDrilldown() method. When the user clicks on a drill-down item, the current series is animated into the the linked drill-down series. A customizable back button is provided to navigate back to the main series, as shown in Detailed series after a drill-down.

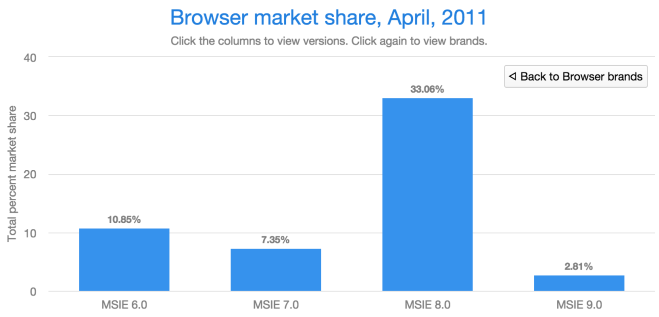


Figure 61. Detailed series after a drill-down

To make use of drill-down, you need to 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.

```
DataSeries series = new DataSeries();

DataSeriesItem mainItem = new DataSeriesItem("MSIE",
55.11);

DataSeries drillDownSeries = new DataSeries("MSIE
versions");
drillDownSeries.setId("MSIE");

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)
);

series.addItemWithDrilldown(mainItem, drillDownSeries);
```

22.7. CSS Styling

Chart appearance is primarily controlled by CSS style rules. A comprehensive list of the supported style classes can be found here[187].

22.7.1. Steps for styling a chart

1. Create a theme file (by convention this should be `webapp/frontend/styles/shared-styles.html`). The theme's dom-module must declare `theme-for=vaadin-chart`.
2. Declare `include="vaadin-chart-default-theme"` on the theme module's style element to customize Chart's default theme.
3. Specify the desired CSS rules in the theme file.
4. If multiple charts are present, each one can be specifically targeted by the host selector e.g `:host(.first-chart-class)`.
5. Import the theme file.

NOTE If there are multiple theme modules **only one** of them should declare the `include` in step 2 above.

22.7.2. Example 1: Chart with Yellow Point Markers and Red Labels

shared-styles.html

```
<link rel="import" href="../bower_components/vaadin-
charts/vaadin-chart-default-theme.html">

<dom-module id="css-style-example" theme-for="vaadin-
chart">
  <template>
    <style include="vaadin-chart-default-theme">
      :host(.first-chart) g.highcharts-markers >
.highcharts-point {
        fill: yellow;
      }

      :host(.first-chart) .highcharts-data-label text {
        fill: red;
      }
    </style>
  </template>
</dom-module>
```

CssStyleExample.java

```
@HtmlImport("frontend://styles/shared-styles.html")
public class CssStyleExample extends Div {

    public CssStyleExample() {
        Chart chart = new Chart();
        Configuration configuration = chart
.getConfiguration();

        configuration.getChart().setType(ChartType.LINE);

        configuration.getxAxis().setCategories("Jan",
"Feb", "Mar", "Apr");

        DataSeries ds = new DataSeries();
        ds.setData(7.0, 6.9, 9.5, 14.5);

        DataLabels callout = new DataLabels(true);
        callout.setShape(Shape.CALLOUT);
        callout.setY(-12);
        ds.get(1).setDataLabels(callout);
        ds.get(2).setDataLabels(callout);
        configuration.addSeries(ds);

        chart.addClassName("first-chart");
        add(chart);
    }
}
```

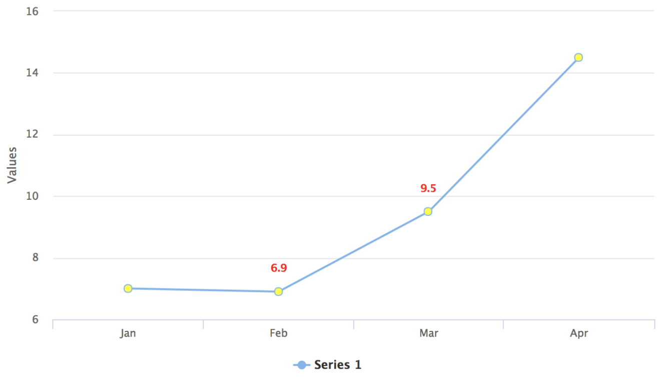


Figure 62. Chart with Yellow Point Markers and Red Labels

22.7.3. Example 2: Exposing a Chart element in Java for CSS Styling

shared-styles.html

```
<link rel="import" href="../bower_components/vaadin-
charts/vaadin-chart-default-theme.html">

<dom-module id="css-style-example" theme-for="vaadin-
chart">
  <template>
    <style include="vaadin-chart-default-theme">
      .huge-axis {
        fill: red;
        font-size: xx-large;
      }
    </style>
  </template>
</dom-module>
```

CssStyleExample.java

```
@HtmlImport("frontend://styles/shared-styles.html")
public class CssStyleExample extends Div {

    public CssStyleExample() {
        Chart chart = new Chart();
        Configuration configuration = chart
.getConfiguration();

        DataSeries ds = new DataSeries();
        ds.setData(7.0, 6.9, 9.5, 14.5);
        configuration.addSeries(ds);

        configuration.getxAxis().setCategories("Jan",
"Feb", "Mar", "Apr");

        // Expose the X-Axis for CSS targeting.
        configuration.getxAxis().setClassName("huge-axis
");

        add(chart);
    }
}
```

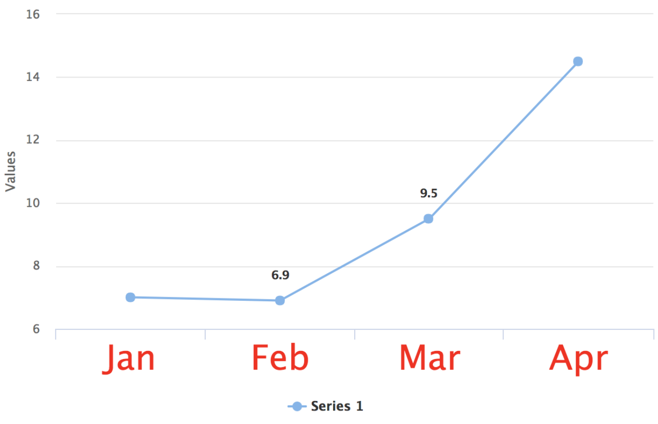


Figure 63. Chart with a Huge X-Axis

22.8. Breaking Changes in Version 6

Vaadin Charts 6 comes with some good enhancements, most notably: CSS styling. This necessitated removal of many Java style configuration API among other changes.

22.8.1. Summary

- Upgraded to HighCharts 5
- Styling is now primarily done with CSS
- Dropped "size with units" sizing properties in favor of strings to take full advantage of browser capabilities
- ZAxis is now a subclass of Axis
- Getting PlotOptionsSeries no longer automatically creates a new instance
- Gradient is no longer supported
- Plot background image is no longer supported
- SVG Generator is no longer supported

22.8.2. Replaced types

Old Type	Replaced By
PinchType	Dimension
ZoomType	Dimension

22.8.3. Dropped types

Type	Used In
Handles	Navigator.handles

22.8.4. Dropped properties

Type	Properties
AbstractDataLabels (and subclasses)	backgroundColor, borderColor, borderRadius, borderWidth, color, reservedSpace, style
AreaOptions (and subclasses)	color, dashStyle, lineColor, lineWidth, negativeColor
Axis (and subclasses)	gridLineColor, gridLineWidth, minorGridLineColor, minorGridLineWidth, tickColor
AxisTitle	reserveSpace
AxisStyle	tickWidth, tickColor, gridLineColor, gridLineWidth
Background	backgroundColor, borderColor, borderWidth
ChartModel	backgroundColor, borderColor, plotBackgroundColor, plotBackgroundImage, plotBorderColor, selectionMarkerFill
ChartStyle	backgroundColor, plotBackgroundColor, plotBorderWidth, plotBorderColor, borderWidth, borderColor
ColumnOptions (and subclasses)	color
ContextButton	symbolFill, symbolSize, symbolStroke, symbolStrokeWidth
Credits	style
GaugeOptions (and subclasses)	zoneAxis, zones
Global	canvasToolsURL

Type	Properties
Hover	lineWidth, lineWidthPlus, fillColor, lineColor
Labels	style
Legend	backgroundColor, borderColor, borderWidth, itemHiddenStyle, itemHoverStyle, itemStyle
LegendNavigation	activeColor, inactiveColor, style
LegendTitle	style
Loading	labelStyle, style
Marker	fillColor, lineColor, lineWidth
Navigation	menuItemHoverStyle, menuItemStyle, menuStyle
Navigator	handles, maskFill, outlineColor, outlineWidth
NoData	style
OhlcOptions (and subclasses)	color,lineWidth
PlotOptionsBoxplot	color, lineWidth, negativeColor
PlotOptionsBubble	color, dashStyle, lineWidth, negativeColor
PlotOptionsCandlestick	lineColor
PlotOptionsFlags	color, lineColor, lineWidth
PlotOptionsPolygon	color, dashStyle, lineWidth, negativeColor

Type	Properties
PlotOptionsSeries	color, dashStyle, lineWidth, negativeColor
PlotOptionsTreemap	color
PlotOptionsWaterfall	dashStyle, lineColor
PointOptions (and subclasses)	color, dashStyle, lineWidth, negativeColor
PyramidOptions (and subclasses)	heightUnit, widthUnit
RangeSelector	buttonTheme, inputStyle, labelStyle
Select	fillColor, lineColor, lineWidth
StackLabels	style
Subtitle	style
Title	style

More information about Charts styling can be obtained in "CSS Styling".

22.8.5. Properties with new types

Property	New Type
ZAxis.title	AxisTitle
ZAxis.type	AxisType
ColumnOptions.zoneAxis	ZoneAxis
Label.textAlign	TextAlign
ChartModel.panKey	PanKey
Exporting.type	ExportingFileType

Property	New Type
Background.shape	BackgroundShape

22.9. 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 `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: `pie`, `gauge`, `solidgauge`, `pyramid`, and `funnel`.

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.

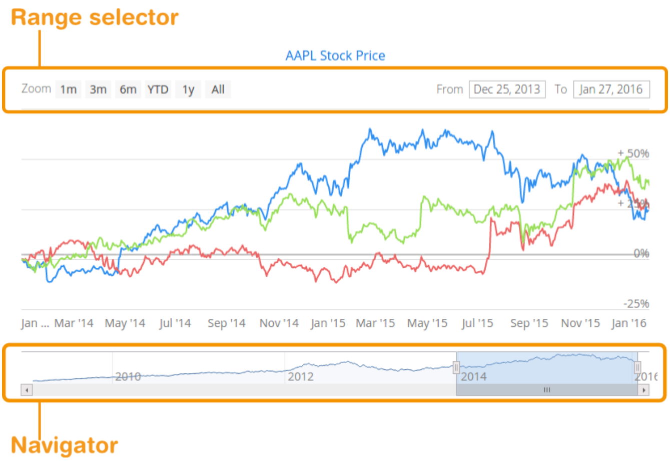


Figure 64. Vaadin chart with a timeline.

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);
```

You can specify the index of the button to appear pre-selected with the setSelected(index) method.

```
RangeSelector rangeSelector = new RangeSelector();
rangeSelector.setSelected(4);

Chart chart = new Chart();
chart.setTimeline(true);
Configuration configuration = chart.getConfiguration();
configuration.setRangeSelector(rangeSelector);
chart.drawChart();
```

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("%YYYY-%MM-%DD:%H:%M");
rangeSelector.setInputEditDateFormat("%YYYY-%MM-
%DD:%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);
```

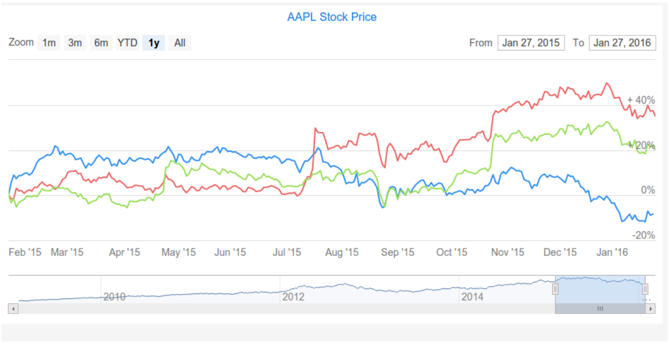


Figure 65. Vaadin chart with a percentage comparison between series.

You can find more examples in the Timeline section of Vaadin Charts Demo[188].

----

[187] https://www.highcharts.com/docs/chart-design-and-style/style-by-css
[188] https://charts.demo.vaadin.com/CompareMultipleSeries

23. Vaadin Testbench

23.1. Overview

Vaadin TestBench is a tool for creating and running browser based integration tests for your Vaadin application. TestBench simulates a user of your application, performs the tasks specified using Java code and verifies that the expected actions take place in the application.

TestBench can also visually inspect your application and detect unintentionally introduced changes, and verify that the application visually looks OK in all the browsers you are testing with. TestBench also includes special support for other Vaadin products, making testing easy and robust compared to generic web testing solutions.

Although not the main purpose of TestBench, you can also use TestBench to automate mundane tasks such as filling out forms.

23.1.1. A Typical Test

A typical test can look like:

1. Start a browser instance.
    - Chrome/Firefox/Safari/IE11/Edge are supported desktop browsers
    - iPhone/iPad/Android simulators are supported for mobile testing
2. Fill in the login form and log in to the application
3. Navigate to the order view

4. Fill in form fields to place an order
5. Verify that the order was placed

In Java code, this could be:

```
@Test
public void fillForm() {
    setDriver(new ChromeDriver());
    getDriver().get("http://localhost:8080");
    LoginViewElement loginView = $(LoginViewElement.class
).first();
    MainViewElement mainView = loginView.login(
"admin@vaadin.com", "admin");
    FormViewElement formView = mainView.navigateTo("form"
);
    formView.clickNew();
    formView.setName("John", "Doe");
    formView.clickSave();
    Assert.assertEquals("John Doe was added", formView
.getMessage());
}
```

The code above uses the page object pattern to hide the implementation details of the view from the main test logic. For more information, see Creating Maintanable Test using Page Objects.

TestBench supports much more complex test cases, both regarding business logic (it's Java, you can do whatever you want), and regarding running tess, e.g. executing on multiple browser instances in parallel, testing that an application works when multiple users interact simultaneously with the same view/data and comparing that the main view of the application still looks like the pregenerated reference screenshot.

23.1.2. Features

The main features of Vaadin TestBench are:

- Control one or several browser instances from Java, both desktop and mobile browsers
- A powerful and robust way to describe your tests so they do not break with application changes
- A high level API for finding the component you want to interact with
- Vaadin Component API for easy interaction with all Vaadin components and HTML elements
- Automatic screen comparison highlighting differences
- Assertion based UI state validation
- Easily running tests in parallel
- Test grid support for speeding up tests by running in parallel on multiple browsers on selected operating systems
- Support for JUnit and other testing frameworks
- All features available in Selenium

23.1.3. Commercial License

Vaadin TestBench is a commercial product and part of the Pro Subscription[189]. You will be asked to validate your license or start a trial period when you start using the tool.

23.2. Getting Started

TIP If you just want to test out TestBench and see how tests are run, the easiest way is to clone the example project at https://github.com/vaadin/testbench-demo. The tests are automatically executed on your local Chrome when you run `mvn verify`.

23.2.1. Setting up your Project

To start using TestBench in an existing project, you need to add the TestBench dependency (`com.vaadin`/`vaadin-testbench`) with a `test` scope. Assuming you have imported the Vaadin platform BOM and have a Maven project, all you need to do is add:

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-testbench</artifactId>
</dependency>
```

The `test` scope and version number is predefined by the Vaadin BOM.

To be able to run tests locally, you might need to install a webdriver for your browser, see Installing Web Drivers for more details.

23.2.2. Creating a Simple Test

The fundamental parts of any TestBench test are:

1. Create an instance of the browser driver for the browser you want to use
2. Open the URL containing the application you want to test

3. Perform test logic and assert that the result was the expected one
4. Close the driver instance to close the browser

The following test example will perform all the above tasks with the test logic consisting of clicking the first available button and checking the the text of the button changes when clicked. If you are adding this test to your own custom application, it will obviously fail unless you modify it.

In the Maven world, all test classes live in the `src/test/java` directory. Create a new class called `SimpleIT` in that directory (`IT` stands for **integration test** and Maven will automatically run all `*IT` classes):

```
import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class SimpleIT extends TestBenchTestCase {

    @Before
    public void setup() throws Exception {
        // Create a new browser instance
        setDriver(new ChromeDriver());
        // Open the application
        getDriver().get("http://localhost:8080/");
    }

    @Test
    public void clickButton() {
        // Find the first button (<vaadin-button>) on the
page
        ButtonElement button = $(ButtonElement.class)
.first();

        // Click it
        button.click();

        // Check the the value of the button is "Clicked"
        Assert.assertEquals("Clicked", button.getText());
    }

    @After
    public void tearDown() throws Exception {
        // close the browser instance when all tests are
done
        getDriver().quit();
    }

}
```

This is all you need to verify that the text of the button is "Clicked" after clicking on it.

NOTE

WebComponents hide their content in the Shadow DOM, that’s why elements inside a WebComponent cannot be found without specifying a search context. For example, `$(TestBenchElement.class).id("content").$(LabelElement.class).first()`, which means label should be found inside the element with `id="content"`, which should be found on the page or current context. For writing real tests use the Page or View Objects, which will improve code readability.

TIP

Don’t place your tests in the root package as in this example. Structure them logically according to your application structure.

23.2.3. Running Tests

The server hosting your application needs to be running at the given URL before you launch your test. If the server is already running and the application is deployed, you only need to ensure that the URL in the test is correct.

If you are using https://vaadin.com/start/v10-project-base, you can launch Jetty using

```
mvn jetty:run
```

If you are using a Spring Boot based starter, you can launch Spring Boot using

```
mvn spring-boot:run
```

You can now launch your test in your IDE (run as JUnit test) or in another terminal:

```
mvn verify
```

You should see a browser window opening, doing something and then closing. If the test fails, put a breakpoint in the `clickButton` method so you can see what happens in the browser before it closes.

TIP

By ending the test name in `IT`, the Maven failsafe plugin will recognize the test as an integration test and is able to automatically start and deploy your application before the test and shut down the server after all tests have been run (tie the server to the `pre-integration-test` and `post-integration-test` phases). See https://github.com/vaadin/testbench-demo for an example.

TIP

Running `mvn test` will only run unit tests (`*Test`) by default while `mvn verify` will also run integration tests (`*IT`)

23.3. Installing Web Drivers

Each browser requires a browser specific web driver to be setup before tests can be run.

TIP

If you are creating a Maven project, consider using the automated web driver plugin. It will automatically download the drivers you need. See https://github.com/vaadin/testbench-demo for an example

If you want to install the drivers, most of them are available through the package manager (e.g. `brew` or `apt-get`). You can also manually download and install the following drivers yourself:

- GeckoDriver for Firefox: https://github.com/mozilla/geckodriver/releases
- ChromeDriver for Chrome: https://sites.google.com/a/chromium.org/chromedriver/downloads
- Microsoft web driver for Edge: https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/
- Selenium IEDriver for IE11: http://selenium-release.storage.googleapis.com/index.html

TIP Safari drivers are pre-installed on Macs and do not need to be manually installed.

NOTE In many cases the web driver is tied to the browser version. You need to make sure that the combination is a supported one, e.g. ChromeDriver 2.35 only supports Chrome 62-64.

23.3.1. Adding Web Driver to System Path

The driver executable must be included in the operating system `PATH` or be given to the test using 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`

In most cases, it is easiest to add it to the `PATH` variable so that it is always available.

23.4. Creating Tests

The test logic in TestBench tests typically consists of two things:

1. Find an element (component) to interact with
2. Interact with an element (component)

For this, TestBench offers the `Element` and `ElementQuery` API.

An `Element` class is a representation of an element on the web page / in the web application. The represented element can be a built-in HTML element such as `<div>` or `<span>`, or it can be a custom element such as `<vaadin-button>` or `<vaadin-grid>`. An `Element` class offers methods to interact with the element in the same way that a user of the application could interact with the element. A `<vaadin-button>` element can for instance be clicked, and you can check what the text on the button says. A `<vaadin-grid>` element has more complex methods for scrolling, checking the visible contents, headers etc. Low level methods applicable to all elements such as `sendKeys()` and `getSize()` are also available when needed.

TIP

`Element` classes are provided for all Vaadin components. If an element class is not availble or some functionality is missing, you can create your own version from scratch or by extending an existing one. See Extending a Page Object.

TIP

An element class is in practice the same as a *page object*, see Creating a Page Object for more details.

An `ElementQuery` is what is used to find a given `Element` (component) on the page so that you can interact with it.

The high-level `ElementQuery` API allows querying Vaadin components in the browser according to their component class type, hierarchy, caption, and other properties. An `ElementQuery` is constructed using a builder like pattern and in the end returns a single element or a list of matching elements. Further queries can be performed on the returned element(s) to find the desired element.

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. The buttons found by the query could be controlled, for example, by clicking them as:

```
for (ButtonElement button : buttons)
    button.click();
```

23.4.1. Element Query Methods

The `$` method creates an `ElementQuery` that looks for the given element class. The method is available both for `TestBenchTestCase` (searches the whole current page) and in `TestBenchElement` (searches inside the given element).

```
// Find the button with id="ok"
ButtonElement button = $(ButtonElement.class).id("ok");
```

```
// Find the first label inside the layout with
id="content"
VerticalLayoutElement layout = $(VerticalLayoutElement
.class).id("content");
LabelElement label = layout.$(LabelElement.class).first(
);
```

NOTE If there is no suitable element class available, you can also use the `$("tag-name")` method to find an element of a given type.

You can use the `ElementQuery` instance returned by `$()` to refine the search query using one of the available methods:

- `id("some-id")` Returns the element with the given id
- `attribute("attributeName", "attributeValue")` Adds a filter to only include elements with the given attribute set to the given value
- `onPage()` Redefines the search context to cover the whole page
- `first()` Returns the first matching element
- `waitForFirst()` Returns the first maching element. If no matches are found, keeps waiting until there is a matching element.
- `last()` Returns the last matching element
- `get(N)` Returns the Nth matching element
- `exists()` Returns `true` if the query matches at least one element
- `all()` Returns a list of all matching elements

23.4.2. Writing Tests

Using Element Queries and Elements you can now compose test methods like:

```
@Test
public void fillForm() {
    $(TextFieldElement.class).id("firstName").setValue(
"John");
    $(TextFieldElement.class).id("lastName").setValue(
"Doe");
    $(ButtonElement.class).id("ok").click();
    Assert.assertEquals("Thank you for submitting the
form", $(DivElement.class).id("result"))
}
```

Do be aware that if you write tests in this manner, you will have a hard time maintaining the tests. A good way to stucture tests is to have only the high level logic in the test itself (your manager should be able to read and understand the test method) and extract the `ElementQuery` parts to a separate **Page Object** class. See Creating Maintainble Tests using Page Objects for more information.

23.5. Creating Maintainable Tests using Page Objects

The Page Object Pattern[190] is an abstraction commonly used when performing actions on a web page. The abstraction hides the implementation details (which elements/components are used, how can they be found on the page etc) from the test methods and allows the test methods to focus on the logic to test. The page object depends on how the page/view is implemented and offer high level methods representing actions that a real user could perform on the page. The separation enables the test method to be independent of the implementation details, so refactoring a view and moving components around only require updates to the page object (if any change is required at all) and not the individual tests.

NOTE The objects are tradititonally called *Page* objects even though they should not represent the whole page but rather a smaller part of it.

23.5.1. Creating a Page Object

Regardless of the name, a *page object* in reality encapsulates a DOM element and is also sometimes called a *TestBench Element class*.

An element class must:

1. Extend `TestBenchElement`
2. Define an `@Element("tag-name")` annotation

The `@Element` annotation defines the tag name of the element which can be located by the element class. The tag name does not have to be unique as the element query user always defines what type of element she is looking for.

When creating a page object for your view, you should use the root tag of the view in the `@Element` annotation. For views created using Java, this might be e.g. `div` while for templates it is a custom element, e.g. `main-view`.

23.5.2. Page Objects for Templates

A page object for a template based login view can look like:

```
@Element("login-view") // map to <login-view>
public class LoginViewElement extends TestBenchElement {

    protected TextFieldElement getUsernameField() {
        return $(TextFieldElement.class).id("username");
    }

    protected PasswordFieldElement getPasswordField() {
        return $(PasswordFieldElement.class).id("
password");
    }

    protected ButtonElement getLoginButton() {
        return $(ButtonElement.class).id("login");
    }

    public void login(String username, String password) {
        getUsernameField().setValue(username);
        getPasswordField().setValue(password);
        getLoginButton().click();
    }
}
```

When mapping to a template, it’s typically enough to define the tag name as each template has its unique custom tag name.

23.5.3. Page Objects for Java Classes

For views created using Java, the tag name is not unique enough and the page object will find lots of unrelated elements unless you define more restrictions using the `@Attribute` annotation. The `@Attribute` annotation allows you to define additional restrictions using attribute values, in the same way as `attribute(name,value)` on an `ElementQuery`. `@Attribute` can be used for any attribute on the element but there are two attributes it is commonly used for: id and CSS class name.

Given a Java login view:

```
public class LoginView extends Div {
```

and a login view element:

```
@Element("div")
public class LoginViewElement extends TestBenchElement {
```

A query such as `$(LoginViewElement.class).first()` would find the first `<div>` on the page. To make the page object find only the `LoginView`, you can either define an `id`:

```
public class LoginView extends Div {
    public LoginView() {
        setId("login-view");
    }
```

or add a `class name`:

```
public class LoginView extends Div {
    public LoginView() {
        addClassName("login-view");
    }
```

The page object can then use the `id` as:

```
@Element("div")
@Attribute(name = "id", value = "login-view")
public class LoginViewElement extends TestBenchElement {
```

or the `class name` as:

```
@Element("div")
@Attribute(name = "class", contains = "login-view")
public class LoginViewElement extends TestBenchElement {
```

The rest of the page object would be the same in both cases (Template and Java class), the only difference is how you find the element you want to interact with.

NOTE

You should use `contains` when you are matching `class` or similar multi value attributes, so that `login-view` matches even when there are multiple class names, e.g. `class="dark login-view active"`.

TIP

An `@Attribute` `value` or `contains` property can be set to `Attribute.SIMPLE_CLASS_NAME` to make it match the simple class name of the page object class with any `Element` or `PageObject` suffix removed. As `@Attribute` annotations are inherited, you can add this on a base class for your elements.

NOTE

All Vaadin component integrations for TestBench can also be considered *page objects* even though they only provide a high level API for a single component. There is no conceptual difference between creating elements for web components and elements for templates or classes representing a whole view.

23.5.4. Using a Page Object

To be able to use the helper methods from a page object, you need to get an instance of the page object. You use the standard `ElementQuery` methods to retrieve an instance of your page object, e.g. to handle login in a test you can do:

```
public class LoginIT extends TestBenchTestCase {

   // Driver setup and teardown omitted

    @Test
    public void loginAsAdmin() {
        getDriver().open("http://localhost:8080");
        LoginViewElement loginView = $(LoginViewElement
.class).first();
        loginView.login("admin@vaadin.com", "admin");
        // TODO Assert that login actually happened
    }
}
```

23.5.5. Chaining Page Objects

Whenever an action on a page object results in the user being directed to another view, it is good practice to find an instance of the page object for the new view and return that. This allows test methods to chain page object calls and continue to perform actions on the new view.

For the `LoginViewElement` we could accomplish this by updating the `login` method:

```
public MainViewElement login(String username, String
password) {
    getUsernameField().setValue(username);
    getPasswordField().setValue(password);
    getLoginButton().click();
    // Find the page object for the main view the user
ends up on
    // onPage() is needed as MainViewElement is not a
child of LoginViewElement.
    return $(MainViewElement.class).onPage().first();
}
```

TIP When the login view finds the main view element, it automatically validates that the login succeeded or the main view will not be found.

A test method can now do:

```
@Test
public void mainViewSaysHello() {
    getDriver().open("http://localhost:8080");
    LoginViewElement loginView = $(LoginViewElement.
class).first();
    MainViewElement mainView = loginView.login(
"admin@vaadin.com", "admin");
    Assert.assertEquals("Hello", mainView.getBanner());
}
```

You can find a fully functional page object based test example in the demo project at https://github.com/vaadin/testbench-demo/tree/master/src/test/java/com/vaadin/testbenchexample/pageobjectexample.

Extending a Page Object

If you want to add functionality to an existing element, you can extend the original element class and add more helper methods, e.g.

```
public class MyButtonElement extends ButtonElement {

   public void pressUsingSpace() {
     ....
   }
}
```

You can then use your new element by replacing

```
ButtonElement button = $(ButtonElement.class).id("ok");
...
```

by

```
MyButtonElement button = $(MyButtonElement.class).id("ok
");
button.pressUsingSpace();
```

23.6. Low Level Element Interactions

Typically you use the provided, high level element API to interact with components. For the cases where a high level API is not available or does not offer the methods you need, a few helpers are provided.

23.6.1. Getting or Setting Properties

Many interactions with web components can be done by reading or modifying element properties. For this, the following helpers are provided in `TestBenchElement`:

```
String getPropertyString(String... propertyNames)
Boolean getPropertyBoolean(String... propertyNames)
Integer getPropertyInteger(String... propertyNames)
Double getPropertyDouble(String... propertyNames)
Object getProperty(String... propertyNames)
TestBenchElement getPropertyElement(String...
propertyNames)
List<TestBenchElement> getPropertyElements(String...
propertyNames)
```

These methods are typically meant for creating *page objects* or TestBench elements but can also be handy as a workaround when a needed method is not available.

Typically you should use the correct `getPropertyXYZ` depending on the type of the property in JavaScript. If you use another type, the value will be converted using standard JavaScript rules (which may or may not give the result you desire).

23.6.2. Calling Functions

If you need to call a function on an element, you can use `Object callFunction(String methodName, Object… args)` available in `TestBenchElement`, e.g.

```
divElement.callFunction("setAttribute", "title", "Hello"
);
```

23.6.3. Executing JavaScript

Sometimes the available API does not offer what you are looking for and you want to execute a JavaScript snippet to accomplish your task. You can excute any JavaScript snippet using the `executeScript` method available in `TestBenchTestCase` and `TestBenchElement` and add references to elements and other parameters using the `Object… args` parameter. All arguments passed to the method are available through the `arguments` array in JavaScript.

For example to return the `offsetHeight` property of an element you could do

```
TestBenchElement element = ...; // find the element
somehow
Long offsetHeight = (Long)executeScript("return
arguments[0].offsetHeight", element);
```

NOTE The above `executeScript` call would be the same as using `element.getPropertyInteger("offsetHeight");`.

The argument array and the return type support a limited set of types:

- HTML elements are converted to `TestBenchElement` instances
- Decimal numbers are converted to `Double`
- Non-decimal numbers are converted to `Integer`
- Booleans are converted to `Boolean`
- All other values except arrays are converted to `String`
- Returned arrays are converted to `List<Object>`, containing types described above

As there is no way to know what type the JavaScript function returns, you always need to cast the return value.

23.7. 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.

23.7.1. 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.

NOTE If the directory names are not absolute, they are relative to the folder you run the tests from. When using Maven, this is typically the root folder of the project.

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);
}
```

23.7.2. 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 rule automatically calls `quit()` on the driver (controlled by the `true` parameter) so you can remove any calls to `getDriver().quit()`.

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");
    ...
}
```

23.7.3. 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(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(String) takes a base name of the screenshot. It is appended with browser identifier and the file extension.

```
assertTrue(testBench(driver).compareScreen("oneplustwo"))
;
```

The `compareScreen(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 Screenshot Parameters.

For example, the error caused by a missing reference image could be written to `screenshot/errors/oneplustwo_mac_chrome_64.png`.

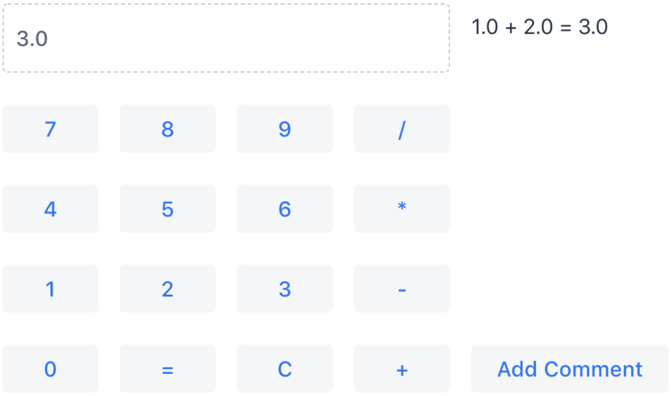


Figure 66. A screenshot taken by a test run

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

Screenshot Parameters. To create a reference image, just copy a screenshot from the `errors/` directory to the `reference/` directory.

For example:

```
$ cp screenshot/errors/oneplustwo_mac_chrome_64.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.

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. The text "Image for this run" is displayed in the top-left corner of the screenshot to distinguish it from the reference image, for example:

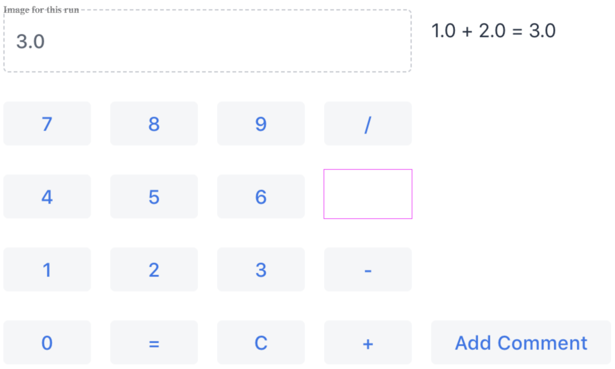


Figure 67. A highlighted error image

23.7.4. 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.

23.8. Advanced Testing Concepts

The following testing concepts are not typically needed in your tests. The cases when you need to disable automatic waiting or scrolling into view are so rare that chances are you have hit a bug if you are considering using these.

23.8.1. Waiting for Vaadin

Traditional web pages 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 SPAs (Single Page 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. A major advantage of using TestBench compared to other testing solutions is that TestBench knows when something is still being rendered on the page and automatically waits for that rendering to finish before moving on with the test.

In most cases, this is not something you need to take into account as waiting is automatically enabled. It might be necessary to disable it in some cases though and 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.

23.8.2. Waiting Until a Condition is Met

In addition to waiting for Vaadin, it is also possible to wait until a condition is met. This could, for example, be used to wait until an element is visible on the web page.

```
waitUntil(ExpectedConditions.presenceOfElementLocated(By.
id("first")));
```

The above waits until the specified element is present or times out after waiting for 10 seconds by default.

`waitUntil(condition, timeout)` allows the timeout duration to be controlled.

23.8.3. Scrolling

To be able to interact with an element, it needs to be visible on the screen. This limitation is set so that test which are run using a web driver shall simulate a normal user as closely as possible. TestBench handles this automatically by ensuring an element is in view before an interaction is triggered. In some cases you might want to disable this behavior and can

then use
`TestBenchCommands.setAutoScrollIntoView(false)`.

23.8.4. 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.

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.

`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.

The following example is given in the `VerifyExecutionTimeITCase.java` [191] 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()
        .getValue());
}
```

23.9. Making Tests Reliable

There are different types of problems which can cause problems in your tests:

- Tests are not understood by other developers/testers and are disabled or accidentally broken
- Changes in the application causes tests to fail
- Problems in the testing environment causes tests to fail

You need to take these into account or will quickly end up with a test suite where a few test always fail. As experienced testers can tell you, this test suite is as good as having no tests at all, because a test suite which is always "a bit red" is not taken seriously by any developer.

TIP You should make sure that your test suite is run on a regular basis. Having a manually triggered test suite which is run only after a lot of changes has been done to the application makes maintenance extermely difficult. The best approach is if you can run the test suite on every change.

23.9.1. Creating Readable Tests

Just as with code, it is important to write tests so that the reader understands the intent. When each test contains high level, meaningful calls, the reader will immediately grasp what is being tested. When/if she wants to know more details about some part of the test, she can then dig into that part. If the test is full of low level details about how you locate the parts of the application you want to interact with, it becomes completely overwhelming to try to decode what the test is actually trying to verify.

By using page/view objects you can abstract away the low level details about how the view is built and what exact components are used. You can also use BDD to describe your test scenarios using normal English sentences.

23.9.2. Guarding Against Application Changes

If your application never changes, you can test it manually just once and you will know that it works properly. In most cases though, your application will be developed forward and you need to maintain the tests when the application evolves.

As long as you abstract away the details from the tests to page/view objects, you only need to take care that your page/view objects are built in a robust way.

You should avoid by all means necessary to depend on the HTML DOM structure. If you depend on finding a `<div>` inside a `<span>` or anything similar, you will have to update the page/view object for every small detail that changes in the application.

Similarly, you should avoid depending on strings targeted for humans in your application. While it is in many cases tempting to find the button with the text "Save", you will run into unnecessary problems when somebody decides to change the text to "Store", or decides to internationalize the application.

Define Ids for the Components

For most cases, it makes sense to define `ids` for all the elements you want to interact with inside your page/view object. The `ids` are only created to be able to identify a given element and there is typically no reason to change them when the application evolves.

When using templates, you also do not need to worry about global `ids` and `ids` colliding with each other, as the id of a given element only needs to be unique inside the shadow root, i.e. the template. For layouts and components outside templates (an inside a single template), you should take care that you do not use the same `id` in multiple places.

TIP

Use `ids` which describe the action which will occur when pressing the button, not `ids` describing e.g. where in the hierarchy the button is. If your `id` is tied to the hierarchy, you will indirectly depend on the hierarchy and lose many benefits of using `ids`.

23.9.3. Dealing with Test Environment Problems

When dealing with browser based tests, and especially older browser such as IE11, you need to take into account that the environment is not always as stable as you would want it to be. Ideally the test would fire up the browser, execute the actions and terminate the browser nicely. Always. In practice, there is potential to have network problems (especially when using a cloud based browser provider), there can be browser problems causing randomness or even browser crashes (yes, this is about you IE11).

When the point of failure is outside your control, e.g. a temporary network failure, your options are very limited. To deal with all kinds of unexpected randomness, in the network or the browsers, TestBench offers a `RetryRule`, which is simply a way to automatically run the test again to see if the temporary problem has disappeared.

`RetryRule` is used as a JUnit 4 `@Rule`, with an parameter describing the maximum number of times the test should be run, e.g:

```
public class RandomFailureTest extends TestBenchTestCase
{

    // Run the test max two times
    @Rule
    public RetryRule rule = new RetryRule(2);

    @Test
    public void doStuff() {
      ...
    }

}
```

If the test passes on the first attempt, it will not be re-run. Only if the first attempt fails, it will try again until either the test passes or the maximum number of attempst has been reached.

NOTE RetryRule affects all the test methods in the class and also child classes.

NOTE The default value of maxAttempts is 1, meaning that test is run only once. You can change the value of maxAttempts globally using the Java system property: `-Dcom.vaadin.testbench.Parameters.maxAttempts=2`.

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.

23.10. Behavior-Driven Development

Behavior-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 Behavior - 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 Behavior. 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:

```java
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 Creating Maintainble Tests using Page Objects.

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[192].

23.11. Running Tests with Maven

A Maven build is divided into different lifecycle phases where the relevants are:

- `compile` Compiles the code
- `test` Runs unit tests which do not require a packaged project
- `pre-integration-test` Makes preparations for integration tests
- `integration-test` Executes integration tests
- `post-integration-test` Cleans up after integration tests

TestBench tests naturally fit into the `integration-test` phase. The `pre-integration-test` phase is the place to start a server and deploy the package and the `post-integration-test` phase is where you would stop the server.

TIP

If you name your tests `*Test`, they will automatically be run in the `test` phase. Name your TestBench tests `*IT` instead, and they will automatically be run in the `integration-test` phase.

NOTE

Never execute TestBench in the `test` phase because they cannot be run without a packaged or deployed project.

23.11.1. Starting the Server Automatically

For applications without external dependencies, it is often handy to start a test as part of the build. As an example, if you are using Jetty for running the project you can use the `jetty-maven-plugin` to start the serve in the `pre-integration-test` and stop it in the `post-integration-test` phase as follows:

```
<plugin>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-maven-plugin</artifactId>
    <version>9.2.3.v20140905</version>
    <configuration>
        <stopPort>9966</stopPort>
        <stopKey>something-goes-here</stopKey>
    </configuration>
    <executions>
        <execution>
            <id>start-jetty</id>
            <phase>pre-integration-test</phase>
            <goals>
                <goal>start</goal>
            </goals>
        </execution>
        <execution>
            <id>stop-jetty</id>
            <phase>post-integration-test</phase>
            <goals>
                <goal>stop</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

The `stopPort` and `stopKey` are Jetty specific parameters which must be given so that Jetty is able to stop the correct server instance. A fully working example of running Jetty as part of the build can be found in https://github.com/vaadin/testbench-demo/blob/master/pom.xml.

If you are using Spring Boot, you can use the `spring-boot-maven-plugin` to achieve the same thing. See the Bakery starter for Spring for an example.

If you are using JavaEE, you can start TomEE, WildFly or a Liberty server in a similar way. See the Bakery starter for JavaEE (at the time of writing only available for Vaadin Framework 8) for an example.

23.11.2. Executing Tests in the Integration Test Phase

In Maven, unit tests are executed by the `maven-surefire-plugin`, automatically included in all projects. Integration tests on the other hand are executed by the `maven-failsafe-plugin`, which needs to be included manually in the project as

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>2.19.1</version>
    <executions>
        <execution>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <trimStackTrace>false</trimStackTrace>
    </configuration>
</plugin>
```

The `<executions>` part is needed to actually execute the plugin during the `integration-test` phase. The `<configuration>` part is optional but by including it you get the full stack trace when an error occurs, which typically makes it easier to figure out what went wrong in a test.

23.11.3. Downloading Web Drivers Automatically

There is a plugin called `driver-binary-downloader-maven-plugin` which downloads webdrivers for a given browser and platform and making them available for the TestBench tests. By downloading these as part of the build, you do not need to do any setup on the machine where you are running the tests. The plugin can be enabled as follows:

```
<plugin>
    <groupId>com.lazerycode.selenium</groupId>
    <artifactId>driver-binary-downloader-maven-
plugin</artifactId>
    <version>1.0.17</version>
    <configuration>

<downloadedZipFileDirectory>${project.basedir}/webdriver/
zips</downloadedZipFileDirectory>

<rootStandaloneServerDirectory>${project.basedir}/webdriv
er</rootStandaloneServerDirectory>

<customRepositoryMap>${project.basedir}/webdrivers.xml</c
ustomRepositoryMap>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>selenium</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

This will download the webdrivers defined in `webdrivers.xml` (called a repository map) in the project root during the `test-compile` phase, i.e. well before the integration tests start. The downloaded webdrivers will be placed in the `webdriver/zips` folder in the project and unpacked to the `webdriver` folder. The `webdrivers.xml` define which version of the various webdrivers to download, an example can be found at https://github.com/vaadin/testbench-demo/blob/master/webdrivers.xml.

TIP | There is a repository map which is kept quite up to date available at https://github.com/Ardesco/selenium-standalone-server-plugin

In addition to downloading the webdrivers, the location of the unpacked drivers must be passed to the `maven-failsafe-plugin` so that the TestBench tests can find them during execution. This can be done by defining system propertys for in the `<configuration>` section of the `maven-failsafe-plugin`:

```
<configuration>
    <trimStackTrace>false</trimStackTrace>
    <systemPropertyVariables>

<webdriver.chrome.driver>${webdriver.chrome.driver}</webd
river.chrome.driver>
        <!-- Similarly for other browsers -->
    </systemPropertyVariables>
</configuration>
```

23.12. Running Tests on a CI Server

Your tests can be run on a CI server as part of the build in the same way as you would run other tests but there are a few things to take into account:

1. The application must be deployed to a server and the server started
2. The URL used in the test must match the URL for the deployed application
3. The browsers you want to run on must be available
4. If run in parallel, tests should be truly independent of each other
5. You need to install a license file on the CI server

23.12.1. Deploying the Application

Deployment of the application can be done in several different ways depending on the setup and your preferences. The important thing is that the applicatiton is deployed and ready to accept requests before the tests are started.

For applications without external dependencies, it is often handy to start a test as part of the build. If you are building with Maven, see Running Tests with Maven for information on how you can start and stop the server as part of the build.

If you have an external server that you deploy the application to, you will typically copy the result of the build to that server in one build step, then wait for the deployment to finish by querying the server or polling the URL where the application should be. The following build step will then execute the TestBench tests using the predefined URL.

23.12.2. Using the Correct URL

In tests you typically use an URL like `http://localhost:8080/` when running on your local machine. On a build server this is usually ok if you are running the server and the tests all on the same build agent.

If only the server is running on the build agent and the browsers are running on a separate machine or on a cloud based browser provider, you might need to define and use a public IP of the build agent. Either you need to pass the IP address to the build in some way and use it in your test, or you can use the provided `IPAddress.findSiteLocalAddress()` helper in your test as e.g.

```
getDriver().get("http://" + IPAddress
.findSiteLocalAddress() + ":8080/");
```

If you are deploying on another host name, you need to pass that information to the tests in a suitable way, e.g. as a system property or environment variable you read in the test code.

NOTE If you are not using site local addresses (10.x.y.z, 172.16.x.y or 192.168.x.y) then you can use `IPAddress.findIPAddress(..)` instead.

23.12.3. Making Sure the Browsers are Available

When running the tests on your local machine you need to have a suitable browser installed. If the test creates a `ChromeDriver` instance, you need to have Chrome installed and so on. The same goes for the CI server (the build agent) if you are running tests directly on a local browser (as opposed to a test cluster described in Running Tests on Multiple Browsers in a Grid).

In addition to installing the browsers on the build agent, you must take into account that browsers typically require a GUI to be run. This is not available directly on your typical build system. The options for running on such a system are:

1. Run Chrome or Firefox in headless mode. Then no GUI is needed.
2. If it is a Linux based system, start `xvfb` which provides a GUI environment for the browser without actually showing the GUI environment anywhere.

To run Chrome in headless mode, you need to pass the `--headless` (and `--disable-gpu` on Windows) parameter to the `ChromeDriver` when starting the browser. The parameters can be defined using `ChromeOptions`:

```
ChromeOptions options = new ChromeOptions();
options.addArguments("--headless", "--disable-gpu");
setDriver(new ChromeDriver(options));
```

Similarly, to run Firefox in headless mode, you need to pass the `-headless` parameter to the `FirefoxDriver`:

```
FirefoxOptions options = new FirefoxOptions();
options.addArguments("--headless", "--disable-gpu");
setDriver(new FirefoxDriver(options));
```

NOTE The `--disable-gpu` flag is only needed on Windows and until http://crbug.com/737678 is resolved but it should not hurt on other platforms.

NOTE Previously PhantomJS was recommended as a good way to do headless testing. You should no longer use PhantomJS as it has fallen far behind the latest browser versions and will likely not work properly with Vaadin platform.

23.12.4. Truly Independent Tests

The easiest way to ensure that tests do not interfere with each other is to have a separate test database initialized from scratch for each test. How you do this is typically connected to what stack you are using. If you are using e.g. Spring Boot, you can use the `SpringRunner` and set your test to rollback any transaction at the end of the test. Other environments might have different options for this.

If you are not resetting the database for each test, you should typically not run the tests in parallel as it will be very hard to understand where something went wrong when a lot of tests suddenly fail. When running tests in sequence you can, even though it is not a good practice, take into account in what way the previous test modified the data set. A better approach is typically to try not to alter the global state in the test, or at least set up data needed by the test in the test itself. An example would be that when you test a CRUD view, you should start by creating an entity instead of selecting an existing entity randomly. You can then delete your test entity in the end of the test. Both of these approaches will though cause a lot of tests to fail if one test fails in the beginning of the set and you always need to hunt down the initial problem and then rerun the whole set to find additional errors.

23.12.5. Installing the License

The license for your subscription is stored on your local machine in

```
~/.vaadin/proKey (Mac/Linux)
%HOMEPATH%\.vaadin\proKey (Windows)
```

You need to copy the file from your local machine to the CI server to enable running tests on the CI server. The CVAL3 license allows you to use your personal license on the CI server also. This is the preferred way as it will always make the license available to all builds running on the same server.

If you do not have access to the build agent running your builds on the CI server, you can also supply the license information using a system property:

```
-Dvaadin.proKey=<username>/<proKey>
```

where the `username` and `proKey` values come from your local `proKey` license file.

NOTE If you use a system property then it needs to be supplied to the process running the tests. It might not be enough to supply the system property to the build command starting the build.

23.13. Running Tests on Multiple Browsers in a Grid

A distributed test environment ("Test grid") consists of a hub and a number of test nodes. The hub acts as an orchestrator, tracking what browsers are available in the nodes and making sure that a node is only used by one test at a time. The nodes have one or several browsers installed and a node is where the actual test is executed.

When running a test on a hub, the TestBench test asks the hub for a certain browser (based on a list of *capabilities*) instead of launching a local browser. The hub waits until a suitable browser is available on some node, reserves that and redirects the test to that given node. The test is then executed and after it has finished, the node reservation is removed and the node used for another test.

NOTE When running on a hub, you do not need a local webdriver installed. The webdriver must be installed on the node instead.

23.13.1. Preparing your Tests for Running on a Test Grid

The tests created previously are setup only to run on a single browser, as a single `ChromeDriver` (or other `WebDriver`) instance is created in a `@Before` method. When running on multiple browsers in parallel, it's easier not to handle the driver instances manually but instead let TestBench handle creation and destruction when needed. To do this, you need to:

1. Extend `ParallelTest` instead of `TestBenchTestCase`. The `ParallelTest` class takes care of creating and destroying driver instances as needed.
2. Define the grid hub URL using either
   1. `@RunOnHub("hub.testgrid.mydomain.com")` on the test class (or a super class) or the system property `com.vaadin.testbench.Parameters.hubHostname`
   2. Configure Sauce Labs credentials and use Sauce Connect proxy to use Sauce Labs test grid. See Using Sauce Labs Test Grid

A test class extending `ParallelTest` will automatically:

- Execute test methods in parallel on the hub defined using

`@RunOnHub` or the corresponding system property

- Create a suitable webdriver instance
- Terminate the driver after the test ends
- Grab a screenshot if the test fails
- Support running the test locally on only one browser for debugging, using `@RunLocally` or the corresponding system property

NOTE

When changing the super class of the test, you need to remove any calls to `setDriver(new ChromeDriver())` or similar, and also any `@After` method which does `getDriver().quit()`.

TIP

In almost all cases you want to configure something for all your grid tests so it makes sense to create a common superclass, e.g. `public abstract class AbstractIT extends ParallelTest`. Then you can add a `@RunOnHub` annotation on that class.

NOTE

Up to 50 test methods in any `ParallelTest` class will be executed simultaneously by default. The limit can be set using the `com.vaadin.testbench.Parameters.testsInParallel` system property. If your tests do not work in parallel set the parameter to `1`.

NOTE

When running tests in parallel, you need to ensure that the tests are independent and do not affect each other in any way.

NOTE

You can handle creation and destruction of the webdriver manually also when running on a hub if you do not want to use `ParallelTest` for one reason or another. In this case you should create a `RemoteWebdriver` with the correct hub URL and set suitable `DesiredCapabilities` on the driver.

23.13.2. Using Sauce Labs Test Grid

For using Sauce Labs you will need:

1. Provide valid Sauce Labs credentials as a system property `sauce.user` and `sauce.sauceAccessKey` or environment variables `SAUCE_USERNAME` and `SAUCE_ACCESS_KEY`
2. Configure pom.xml to have `sauce-connect-plugin` open a tunnel with a tunnel identifier to Sauce Labs test grid
3. Write your tests as described in Preparing your Tests for Running on a Test Grid

After you have your Sauce Labs credentials you can pass them to your build e.g. `mvn verify -Dsauce.user=<yourusername> -Dsauce.sauceAccessKey=<youraccesskey>`.

Sauce Connect plugin opens a tunnel before the integration tests are run and closes it after they are run. The plugin needs an identifier for the tunnel. The identifier is passed with `--tunnel-identifier` attribute in `sauce.options` system property e.g. in pom.xml `<sauce.options>--tunnel-identifier ${maven.build.timestamp}</sauce.options>`.

```
<plugins>
    <plugin>
        <groupId>com.saucelabs.maven.plugin</groupId>
        <artifactId>sauce-connect-plugin</artifactId>
        <version>2.1.23</version>
        <executions>
            <!-- Start Sauce Connect prior to running the
integration tests -->
            <execution>
                <id>start-sauceconnect</id>
                <phase>pre-integration-test</phase>
                <goals>
                    <goal>start-sauceconnect</goal>
                </goals>
            </execution>
            <!-- Stop the Sauce Connect process after the
integration tests have
                finished -->
            <execution>
                <id>stop-sauceconnect</id>
                <phase>post-integration-test</phase>
                <goals>
                    <goal>stop-sauceconnect</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
```

23.13.3. Defining the Browsers to Run Tests On

You can define the tested browsers and their versions in an environment variable `TESTBENCH_GRID_BROWSERS` or system property `com.vaadin.testbench.Parameters.gridBrowsers` with a comma separated list e.g. `-Dcom.vaadin.testbench.Parameters.gridBrowsers=chrome,firefox,safari-11,safari-9`.

If you don’t want to use the environment variable, you can define the configuration in your test class, in a method annotated with @BrowserConfiguration. It returns a list of DesiredCapabilities, typically describing what platform, os and browser name/version should be used. Typically this is defined in a superclass for the test so that all tests are run on the same browsers, e.g.

```
@RunOnHub("hub.testgrid.mydomain.com")
public abstract class AbstractIT extends ParallelTest {

   @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;
   }
}
```

NOTE

The BrowserUtil helper methods create a DesiredCapability object which works in many cases. To customize the versions and other values, annotate your test class using @BrowserFactory(MyBrowserFactory.class) and implement MyBrowserFactory by extending DefaultBrowserFactory.

TIP

To run a multi browser test locally, you can use the `com.vaadin.testbench.Parameters.runLocally` system property (or a `@RunLocally` annotation on the test class) to override what browser to run on. The value of the property or annotation should be the browser to run on, e.g. `chrome` or `@RunLocally(Brwoser.CHROME)`. When `RunLocally` is used, any hub configuration is also ignored and a local webdriver is used.

23.14. Setting up your Own Test Grid

TestBench is based on Selenium and does not contain any modifications to the grid hub/node part. This means that you can run TestBench tests on any available Selenium grid and setting up a grid is also exactly like setting up a Selenium grid.

23.14.1. Setting up the Docker Based Selenium Grid

There are ready made Docker images for setting up a Selenium Grid available at https://github.com/SeleniumHQ/docker-selenium. To use the images, you first need to install Docker[193]. Once you have Docker installed, you can create your own test grid e.g. using `docker-compose`.

First create the following `docker-compose.yaml` in an empty folder:

```
version: '2'
services:
  firefox:
    image: selenium/node-firefox:3.9.1-actinium
    volumes:
      - /dev/shm:/dev/shm
    depends_on:
      - hub
    environment:
      HUB_HOST: hub

  chrome:
    image: selenium/node-chrome:3.9.1-actinium
    volumes:
      - /dev/shm:/dev/shm
    depends_on:
      - hub
    environment:
      HUB_HOST: hub

  hub:
    image: selenium/hub:3.9.1-actinium
    ports:
      - "4444:4444"
```

This defines a grid with one Chrome node and one Firefox node in addition to the hub.

The whole grid can then be started as

```
docker-compose up
```

This will start a grid on http://localhost:4444, with the console at http://localhost:4444/grid/console so you can run your tests on the hub using `@RunOnHub("localhost")`.

NOTE The communication protocol used by the grid is standardized so it should not be critical to match the Selenium version that TestBench is based on with the version of the grid. If you run into some strange issues, try matching the versions. You can check the Selenium version for TestBench here[194]

23.14.2. Setting up a Custom Selenium Grid

The process for setting up your own custom Selenium grid is described at https://seleniumhq.github.io/docs/grid.html#rolling_your_own_grid. All the instructions for Selenium apply also for TestBench.

23.14.3. 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 cursor blinking
- Use the exact same operating system and browser version on every host
- Turn off any software that may suddenly pop up a new window
- Turn off the screen saver

If you are using Windows and Internet Explorer, you should also turn on `Allow active content to run in files on My Computer` in `Security settings`.

23.14.4. 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[195] and the AndroidDriver[196].

23.15. Migrating to Vaadin 10

23.15.1. Introduction

Vaadin TestBench is part of the Vaadin platform and is intended to primarily be used to test applications created using the same platform version.

While the features are primarily the same as in TestBench 5 for Vaadin Framework 8 and TestBench 4 for Vaadin Framework 7, the API has been tuned a bit to better match Flow component API and features.

23.15.2. ElementQuery Changes

The ElementQuery method `caption(String)` has been removed as there is no generic `caption` concept across all web components. The method `state(String,String)` was also tied to the Vaadin Framework "shared state" feature and has been removed. A more generic finder method `attribute(String name, String value)` has been added instead. This can be used to find an element with any given attribute value. The old `caption("OK")` can in some cases be replaced by `attribute("label","OK")` and `state("something","value")` also by `attribute("something", "value")`, depending on the used component.

The query methods `in()`, `child()` and `$$()` were rarely used and have been removed to simplify the query language.

23.15.3. Element API Changes

The Element API has been made consistent with the API provided by the element (web component) itself. The feature set is largely the same as in older versions but the exact method naming differs in some cases.

23.15.4. Applications using both Vaadin Framework and Vaadin platform

If you have an application which is using both a Vaadin Framework version and a Vaadin platform version, you should keep the tests for each version in a separate module in the project. This allows you to use an older TestBench version for the Vaadin Framework tests and a new version for the Vaadin platform tests.

23.15.5. Selenium Version

The Selenium version has been upgraded to the latest available version. While it is mostly compatible, some small API changes might require your attention.

23.15.6. PhantomJS

It is no longer recommended to use PhantomJS for headless testing. PhantomJS is lacking behind the latest browser versions in features and will in many cases just not work with Vaadin platform. You should instead run using headless Chrome (using `--headless --disable-gpu`) or using headless Firefox (using `-headless`).

----

[189] https://vaadin.com/pricing
[190] https://martinfowler.com/bliki/PageObject.html
[191] https://github.com/vaadin/testbench-demo/blob/master/src/test/java/com/vaadin/testbenchexample/VerifyExecutionTimeITCase.java
[192] http://jbehave.org/
[193] https://www.docker.com/
[194] https://github.com/vaadin/testbench/blob/master/vaadin-testbench-core/pom.xml
[195] http://ios-driver.github.io/ios-driver/
[196] http://selendroid.io/mobileWeb.html

24. Vaadin Multiplatform Runtime

The Vaadin Multiplatform Runtime (or MPR for short) allows the developer to run applications and components written with a Legacy Framework (Vaadin 7 or Vaadin 8) inside a Vaadin 14 (Flow) application.

The Multiplatform Runtime is available to all Vaadin customers in the Prime subscription[197] tier. The project is licensed under the Commercial Vaadin Add-On License version 3 (CVAL).

Issues can be reported on the MPR issues public repository[198].

24.1. Step by step migration guide

- Follow the Migration Guide to configure your project to use MPR and port your legacy application to Flow

24.2. Configuration and advanced topics

- Legacy theme in MPR
- Custom widgetset and MPR
- Push and MPR
- Using sessions
- Using custom legacy UIs (advanced)

24.2.1. Using Legacy Components In a Flow Application

- Adding Legacy Components in a Flow Layout

24.2.2. Production mode

- Setting up production mode

24.2.3. Known Limitations

- Limitations of MPR

24.3. Step-by-step migration guide

24.3.1. Step-by-step Migration Guide

The Multiplatform Runtime allows you to use components and views developed with Vaadin 7 or Vaadin 8 inside a Vaadin 14 application using Vaadin Flow. This document will guide you through a series of steps to properly migrate a working Vaadin 7 or Vaadin 8 application to Vaadin Flow.

NOTE It is easier to get started by changing things inside the existing Vaadin application than starting from a Flow Starter.

Step 1 - pom.xml configuration

The first step is to configure the Maven dependencies and plugins for MPR to work properly. There are different settings depending on which Vaadin version do you use:

- My application uses Vaadin 7 →
- My application uses Vaadin 8 →

Or:

- ← Go back to the overview[199]

24.3.2. Step 1 - pom.xml configuration for Vaadin 7

Maven setup

When using Vaadin 14

MPR is part of the Vaadin platform, so the supported version of the project is set for you when importing the plaform in your project. In other words, you only need to define the platform version:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-bom</artifactId>
            <type>pom</type>
            <scope>import</scope>
            <version>14.0.0</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

i. and then declare the usage of `vaadin-core` and `mpr-v7`:

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-core</artifactId>
</dependency>
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>mpr-v7</artifactId>
</dependency>
```

Framework 7 dependency

When using MPR the minimum requirement for Vaadin 7 version is 7.7.14 or newer.

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-server</artifactId>
    <version>7.7.17</version>
    <exclusions>
        <exclusion>
            <groupId>org.jsoup</groupId>
            <artifactId>jsoup</artifactId>
        </exclusion>
        <exclusion>
            <groupId>com.google.gwt</groupId>
            <artifactId>gwt-elemental</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-themes</artifactId>
    <version>7.7.17</version>
</dependency>
```

NOTE You also need to remove the dependency on the `vaadin-client-compiled`, since a custom widgetset is served by the MPR project.

NOTE

When using MPR you can not use CDN for the widgetset. This means that the configuration `<vaadin.widgetset.mode>cdn</vaadin.widgetset.mode>` or `<vaadin.widgetset.mode>fetch</vaadin.widgetset.mode>` should be removed.

Maven Plugins

If not already added in your build section, you need to add the `vaadin-maven-plugin` for it to manage the custom legacy widgetset. Maven plugin version used at the moment is 7.7.17.

```
<build>
     <plugins>
         <plugin>
             <groupId>com.vaadin</groupId>
             <artifactId>vaadin-maven-plugin</artifactId>
             <version>7.7.17</version>
             <executions>
                 <execution>
                     <goals>
                         <goal>resources</goal>
                         <goal>update-widgetset</goal>
                         <goal>compile</goal>
                     </goals>
                 </execution>
             </executions>
         </plugin>
     </plugins>
 </build>
```

Vaadin 14 too requires a Maven plugin for processing frontend resources during development time. Because the `vaadin-maven-plugin` can only be defined with one version, you'll have to use the `flow-maven-plugin` instead. Unfortunately this forces you to manually define the plugin version, since Maven does not allow you to define a plugin version in BOM (bill of materials).

```
<build>
     <plugins>
         <plugin>
             <groupId>com.vaadin</groupId>
             <artifactId>flow-maven-plugin</artifactId>
             <version>2.0.7</version>
             <executions>
                 <execution>
                     <goals>
                         <goal>prepare-frontend</goal>
                     </goals>
                 </execution>
             </executions>
         </plugin>
     </plugins>
 </build>
```

Logging

To display Flow application logs, any slf4j implementation should be added to the project. The easiest way would be to use `slf4j-simple` dependency:

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.25</version>
</dependency>
```

Next step

- Step 2 - Removing legacy servlets →

Or:

- ← Go back to the overview[200]

Appendix: sample pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.mycompany</groupId>
    <artifactId>my-mpr-app</artifactId>
    <packaging>war</packaging>
    <version>0.1</version>

    <properties>
        <vaadin.version>7.7.17</vaadin.version>
        <vaadin.plugin.version>
${vaadin.version}</vaadin.plugin.version>
        <!-- Flow version needs to be defined manually
for Flow Maven plugin,
            because Maven BOMs do not support plugin
versions or defining properties.
            The Flow version to use can be checked from
vaadin-bom. -->
        <flow.version>2.0.7</flow.version>

        <slf4j.version>1.7.25</slf4j.version>
        <jetty.plugin.version>
9.4.19.v20190610</jetty.plugin.version>
        <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
        <maven.compiler.source>
1.8</maven.compiler.source>
        <maven.compiler.target>
1.8</maven.compiler.target>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
```

```
                <groupId>com.vaadin</groupId>
                <artifactId>vaadin-bom</artifactId>
                <type>pom</type>
                <scope>import</scope>
                <version>14.0.0</version>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependencies>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-core</artifactId>
        </dependency>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>mpr-v7</artifactId>
        </dependency>

        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-server</artifactId>
            <version>${vaadin.version}</version>
            <exclusions>
                <exclusion>
                    <groupId>org.jsoup</groupId>
                    <artifactId>jsoup</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>com.google.gwt</groupId>
                    <artifactId>gwt-
elemental</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-themes</artifactId>
            <version>${vaadin.version}</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>${slf4j.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>com.vaadin</groupId>
                <artifactId>vaadin-maven-
plugin</artifactId>
                <version>
${vaadin.plugin.version}</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>resources</goal>
                            <goal>update-widgetset</goal>
                            <goal>compile</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>

            <!-- Since the Vaadin Maven plugin can only
be defined with one version,
                The Flow Maven plugin is used instead for
handling Vaadin 14+ frontend
                resources for development and production
builds. -->
            <plugin>
                <groupId>com.vaadin</groupId>
                <artifactId>flow-maven-
plugin</artifactId>
                <version>${flow.version}</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>prepare-frontend</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>

            <!-- The Jetty plugin allows us to easily
test the development build by
                running jetty:run on the command line.
-->
```

```
            <plugin>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>jetty-maven-
plugin</artifactId>
                <version>
${jetty.plugin.version}</version>
                <configuration>
                    <scanIntervalSeconds>
2</scanIntervalSeconds>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

24.3.3. Step 1 - pom.xml configuration for Vaadin 8

Maven setup

MPR is part of the Vaadin platform, so the supported version of the project is set for you when importing the plaform in your project. In other words, you only need to define the platform version:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-bom</artifactId>
            <type>pom</type>
            <scope>import</scope>
            <version>14.0.0</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

i. and then declare the usage of `vaadin-core` and `mpr-v8`:

```xml
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-core</artifactId>
</dependency>
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>mpr-v8</artifactId>
</dependency>
```

Framework 8 dependency

When using MPR the minimum requirement for Vaadin 8 version is 8.6.0 or newer:

```xml
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-server</artifactId>
    <version>8.7.0</version>
    <exclusions>
        <exclusion>
            <groupId>org.jsoup</groupId>
            <artifactId>jsoup</artifactId>
        </exclusion>
        <exclusion>
            <groupId>com.google.gwt</groupId>
            <artifactId>gwt-elemental</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-themes</artifactId>
    <version>8.7.0</version>
</dependency>
```

NOTE You also need to remove the dependency on the `vaadin-client-compiled`, since a custom widgetset is served by the MPR project.

NOTE When using MPR you can not use CDN for the widgetset. This means that the configuration `<vaadin.widgetset.mode>cdn</vaadin.widgetset.mode>` or `<vaadin.widgetset.mode>fetch</vaadin.widgetset.mode>` should be removed.

Maven Plugins

If not already added in your build section, you need to add the `vaadin-maven-plugin` for it to manage the custom widgetset. Maven plugin version used at the moment is 8.7.0.

```
<build>
     <plugins>
         <plugin>
             <groupId>com.vaadin</groupId>
             <artifactId>vaadin-maven-plugin</artifactId>
             <version>8.7.0</version>
             <executions>
                 <execution>
                     <goals>
                         <goal>resources</goal>
                         <goal>update-widgetset</goal>
                         <goal>compile</goal>
                     </goals>
                 </execution>
             </executions>
         </plugin>
     </plugins>
 </build>
```

Vaadin 14 too requires a Maven plugin for processing frontend resources during development time. Because the `vaadin-maven-plugin` can only be defined with one version, you'll have to use the `flow-maven-plugin` instead. Unfortunately this forces you to manually define the plugin version, since Maven does not allow you to define a plugin version in BOM (bill of materials).

```
<build>
     <plugins>
         <plugin>
             <groupId>com.vaadin</groupId>
             <artifactId>flow-maven-plugin</artifactId>
             <version>2.0.7</version>
             <executions>
                 <execution>
                     <goals>
                         <goal>prepare-frontend</goal>
                     </goals>
                 </execution>
             </executions>
         </plugin>
     </plugins>
 </build>
```

Logging

To display Flow application logs, any slf4j implementation should be added to the project. The easiest way would be to use `slf4j-simple` dependency:

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.25</version>
</dependency>
```

Vaadin 7 compatibility package

If your project is using components from the Vaadin 7 compatibility package, then you also need to add:

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-compatibility-server</artifactId>
    <version>8.7.0</version>
</dependency>

<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-compatibility-client</artifactId>
    <version>8.7.0</version>
    <scope>provided</scope>
</dependency>
```

Next step

- Step 2 - Removing legacy servlets →

Or:

- ← Go back to the overview[201]

Appendix: sample pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.mycompany</groupId>
    <artifactId>my-mpr-app</artifactId>
    <packaging>war</packaging>
    <version>0.1</version>

    <properties>
        <vaadin.version>8.7.0</vaadin.version>
        <vaadin.plugin.version>
${vaadin.version}</vaadin.plugin.version>
```

```
        <!-- Flow version needs to be defined manually
for Flow Maven plugin,
            because Maven BOMs do not support plugin
versions or defining properties.
            The Flow version to use can be checked from
vaadin-bom. -->
        <flow.version>2.0.7</flow.version>

        <slf4j.version>1.7.25</slf4j.version>
        <jetty.plugin.version>
9.4.19.v20190610</jetty.plugin.version>
        <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
        <maven.compiler.source>
1.8</maven.compiler.source>
        <maven.compiler.target>
1.8</maven.compiler.target>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>com.vaadin</groupId>
                <artifactId>vaadin-bom</artifactId>
                <type>pom</type>
                <scope>import</scope>
                <version>14.0.0</version>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependencies>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-core</artifactId>
        </dependency>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>mpr-v8</artifactId>
        </dependency>

        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-server</artifactId>
            <version>${vaadin.version}</version>
            <exclusions>
                <exclusion>
                    <groupId>org.jsoup</groupId>
                    <artifactId>jsoup</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>com.google.gwt</groupId>
                    <artifactId>gwt-
elemental</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-themes</artifactId>
            <version>${vaadin.version}</version>
        </dependency>

        <!-- Vaadin 7 compatibility packages -->
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-compatibility-
server</artifactId>
            <version>${vaadin.version}</version>
        </dependency>

        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-compatibility-
client</artifactId>
            <version>${vaadin.version}</version>
            <scope>provided</scope>
        </dependency>

        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>${slf4j.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>com.vaadin</groupId>
                <artifactId>vaadin-maven-
plugin</artifactId>
                <version>
${vaadin.plugin.version}</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>resources</goal>
                            <goal>update-widgetset</goal>
                            <goal>compile</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>

            <!-- Since the Vaadin Maven plugin can only
be defined with one version,
                The Flow Maven plugin is used instead for
handling Vaadin 14+ frontend
                resources for development and production
builds. -->
            <plugin>
                <groupId>com.vaadin</groupId>
                <artifactId>flow-maven-
plugin</artifactId>
                <version>${flow.version}</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>prepare-frontend</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>

            <!-- The Jetty plugin allows us to easily
test the development build by
                running jetty:run on the command line.
-->
            <plugin>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>jetty-maven-
plugin</artifactId>
                <version>
${jetty.plugin.version}</version>
```

```
                <configuration>
                    <scanIntervalSeconds>
2</scanIntervalSeconds>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

24.3.4. Step 2 - Removing legacy Servlets

The MPR framework manages the VaadinServlets to make sure the correct requests are routed to the right frameworks (either Flow or Vaadin 7/8). For that to work properly, all legacy VaadinServlets need to be removed. If you need some custom functionality, you can use the VaadinServlet provided by Flow instead.

See Flow documentation on Dynamic content[202] for details.

Next step

- Step 3 - Converting legacy UIs →

Or:

- ← Go back to step 1
- ← Go back to the overview[203]

24.3.5. Step 3 - Converting legacy UIs

The UI object, which represents the `<body>` element in the page, is controlled by Flow when running the MPR, so anything extending `UI` should be converted. Actually, `UI` class should not be used for layouting or navigation handling anymore. At the end of step 3, you should not have any `UI` class unless you have a real use case for it.

There are several conversion paths, depending on what’s used in the project:

- My application uses Spring Boot →
- My application uses CDI →
- My application uses Navigator →
- My application doesn’t use any of those. Continue to Converting a UI →

Or:

- ← Go back to step 2
- ← Go back to the overview[204]

24.3.6. Step 3 - Running a Spring Boot application with MPR and Flow

NOTE This step is needed in case your Vaadin 7 or 8 application uses Spring Boot. If it is not the case, go back to the framework selection.

Updating to the correct Spring version

Update parent `org.springframework.boot:spring-boot-starter-parent` to `2.1.7.RELEASE` or newer.

The dependency `com.vaadin:vaadin-spring-boot-starter` should not have a version defined as it comes from `vaadin-bom`.

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
</parent>

<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-spring-boot-starter</artifactId>
</dependency>
```

NOTE Also take a look at the Use Flow with Spring[205] tutorial on how Flow integrates with Spring.

Handling of SpringUI

The `@SpringUI` can be replaced with a `@Route`. For example this UI:

```java
@SpringUI
@Theme("valo")
public class TodoUI extends UI {
    @Override
    protected void init(VaadinRequest vaadinRequest) {
        setContent(new HorizontalLayout());
    }
}
```

Can be replaced with:

```
@Route("")
public class TodoUI extends Div implements
HasLegacyComponents {
    @PostConstruct
    private void buildLayouts() {
        setSizeFull();
        add(new HorizontalLayout());
    }
}
```

NOTE Annotations in the UI, such as `@Theme` and `@Title` and so on, will be dealt with later on in the tutorial. Most of them have similar counterpart in either Flow or MPR.

Update imports

Then any `com.vaadin.spring.annotation` imports needs to be changed to `com.vaadin.flow.spring.annotation`.

NOTE The V14 Spring add-on doesn't have a feature comparable with `ViewScope`

What to do with SpringView

Any `@SpringView` should be updated to a Flow Route by wrapping them as a `MprRouteAdapter<? extends View>` or re-writing it to be a Flow Component. See Migrating Views to Flow Routes for details.

Things to keep in mind

- When porting the UI to a flow component, you lose the ability to use UI methods, such as `setErrorHandler`. You can still access those by using `UI.getCurrent()`. The method `setContent` is not support though - you should use the `add` method from your Flow layout instead.

- When running MPR with Spring, the Spring integration is done with Flow (and not anymore with Vaadin 7 or 8), so in some cases you will need to import classes from the old `vaadin-spring` project in order to make your MPR project to compile, since those classes are not present anymore in the new versions of `vaadin-spring`. The source code of `vaadin-spring` can be found on GitHub[206]. Examples of such classes:
  - com.vaadin.spring.access.SecuredViewAccessControl;
  - com.vaadin.spring.access.ViewAccessControl;
  - com.vaadin.spring.internal.SpringBeanUtil;
  - com.vaadin.spring.internal.VaadinSpringComponentFa ctory;
  - com.vaadin.spring.server.SpringVaadinServletService;
- If your routes are defined in a different package than the Spring application itself, you need to annotate your application with `@EnableVaadin`, in order to Spring to scan the appropriate folders for beans. For example:

```
// Assuming that Application is in a different package
than the classes
// annotated with @Route
@SpringBootApplication
@EnableVaadin("com.mycompany.views")
public class Application extends
SpringBootServletInitializer {
```

Next step

- Step 4 - Configuring UI parameters →

Or:

- ← Go back to step 2
- ← Go back to the overview[207]

24.3.7. Step 3 - Running a Vaadin Legacy CDI application with MPR and Flow

NOTE This step is needed in case your Vaadin 7 or 8 application uses CDI. If it is not the case, go back to the framework selection.

Updating to the correct CDI version

Remove any version from `com.vaadin:vaadin-cdi` as the proper Vaadin 14 compatible version for it is managed by the `vaadin-bom`:

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-cdi</artifactId>
</dependency>
```

Handling of CDIUI annotation

Instead of `@CDIUI` use `@Route`.

```
@CDIUI
@Theme("valo")
public class TodoUI extends UI {
    @Override
    protected void init(VaadinRequest vaadinRequest) {
        setContent(new HorizontalLayout());
    }
}
```

can for instance be replaced with

```
@Route("")
public class TodoUI extends Div implements
HasLegacyComponents {
    @PostConstruct
    private void buildLayouts() {
        setSizeFull();
        add(new HorizontalLayout());
    }
}
```

NOTE: Annotations in the UI, such as `@Theme` and `@Title` and so on, will be dealt with later on in the tutorial. Most of them have similar counterpart in either Flow or MPR.

What to do with CDIView

Any `@CDIView` should be updated to a Flow Route by wrapping them as a `MprRouteAdapter<? extends View>` or re-writing it to be a Flow Component. See Migrating Views to Flow Routes

What to do with ViewScopes

All ViewScopes should be changed to RouteScopes e.g.

- `@ViewScoped` to `@RouteScoped`
- `@NormalViewScoped` to `@NormalRouteScoped`

> **NOTE** In some projects CDI has ignored the archive and not instantiated objects as expected. This is fixed by adding a `beans.xml` (empty is fine) file to `src/main/webapp/WEB_INF`.

Next step

- Step 4 - Configuring UI parameters →

Or:

- ← Go back to step 2
- ← Go back to the overview[208]

24.3.8. Step 3 - Navigation using Navigator in Flow with MPR

> **NOTE** This step is needed in case your Vaadin 7 or 8 application uses Navigator. If it is not the case, go back to the framework selection.

The navigation with MPR can be done in three ways. You can choose the one most suitable for your application, but do not mix them together. Also, note that no other approach is supported at the moment and you are on your own if you do it your own way.

1. Using the Navigator together Flow's Router: this is suitable for creating new views in Flow while maintaining the old views to be routed by the `Navigator`.

2. Using the Navigator without mixing with Flow: this is suitable for projects with complex custom navigators.
3. Using only the Flow's Router: this is suitable for basic navigation setups that can be easily ported, or as the final stage of an incremental porting process.

NOTE Keep in mind that the old `Navigator` uses URLs with the "hash-bang" (`#!`) prefix. That prefix is not used at all by the Flow's Router.

Mixing navigation and Flow routing

It is possible to use the legacy Navigator and Flow routing together.

Starting from the legacy application:

```
public class NavigatorUI extends UI {
  @Override
  protected void init(VaadinRequest request) {
    CssLayout viewDisplay = new CssLayout();
    Navigator navigator = new Navigator(this,
viewDisplay);

    navigator.addView("", HomeView.class);
    navigator.addView("away", AwayView.class);

    VerticalLayout content = new VerticalLayout
(viewDisplay);
    setContent(content);
  }
}
```

We would make the `UI` into a Flow route by extending `MprNavigatorRoute`.

```
@Route("")
public class MyNavigatorRoute extends MprNavigatorRoute {
    @Override
    public void configureNavigator(Navigator navigator) {
        navigator.addView("", HomeView.class);
        navigator.addView("away", AwayView.class);
    }
}
```

For a more complex sample we could have a `MainMenu` component that is visible at all times and used to navigate between the views.

```
public class MyNavigatorUI extends UI {
  @Override
  protected void init(VaadinRequest request) {
    CssLayout viewDisplay = new CssLayout();
    Navigator navigator = new Navigator(this,
viewDisplay);

    navigator.addView("", HomeView.class);
    navigator.addView("away", AwayView.class);

    setContent(new VerticalLayout(new MainMenu(),
viewDisplay));
  }
}

public class MainMenu extends HorizontalLayout {
    public MainMenu() {
        Button home = new Button("Home",
                event -> getUI().getNavigator()
.navigateTo(""));
        Button away = new Button("Away",
                event -> getUI().getNavigator()
.navigateTo("away"));

        addComponents(home, away);
    }
}
```

Here we can move the `MainMenu` to its own `RouterLayout` that will be used on all Routes that have it as the parent layout. All we need to do is to create a Flow component, e.g. `MainLayout`, that contains the `MainMenu` component and add that to the `@Route` annotation.

```java
// Flow router target
@Route(value = "", layout = MainLayout.class)
public class MyNavigatorRoute extends MprNavigatorRoute {
    @Override
    public void configureNavigator(Navigator navigator) {
        navigator.addView("", HomeView.class);
        navigator.addView("away", AwayView.class);
    }
}

// Flow layout, used by the router
public class MainLayout extends VerticalLayout implements
RouterLayout {
    public MainLayout() {
        add(new LegacyWrapper(new MainMenu()));
    }
}
```

This way we can make a single `MainLayout` that can be used both with the old navigator views as well as with the new Flow views.

To add a Flow view we just need to create the route target.

```java
@Route(value = "flow", layout = MainLayout.class)
public class FlowView extends Div {
}
```

and then add it to the `MainMenu` as a `Button`.

```
public class MainMenu extends HorizontalLayout {
    public MainMenu() {
        Button home = new Button("Home",
                event -> getUI().getNavigator()
.navigateTo(""));
        Button away = new Button("Away",
                event -> getUI().getNavigator()
.navigateTo("away"));
        Button flow = new Button("Flow",
                event -> getUI().getNavigator()
.navigateTo("flow"));

        addComponents(home, away, flow);
    }
}
```

Now the menu can be used to navigate from a legacy view to a Flow view and back.

When requesting the `Navigator` to navigate to a view that isn't registered in the `Navigator` we will navigate to a corresponding Flow view if available.

Also navigation from a Flow route to a legacy View will work through the `Navigator`.

NOTE By default the `MprNavigatorRoute` creates a `<div>` on the client-side, but this can be changed by annotating the subclass with `@Tag`.

NOTE `MainMenu`, `HomeView` and `AwayView` are legacy Vaadin 7 components and, `FlowView` and `MainLayout` are Flow components. `HomeView` and `AwayView` also implement `View`.

Use navigator without mixing with Flow

Navigator can be used as is by having a view display component that is wrapped in a `LegacyWrapper`.

Consider the following simple legacy navigator setup:

```
public class MyUI extends UI {

    @Override
    protected void init(VaadinRequest request) {
        Navigator navigator = new Navigator(this, this);
        navigator.addView("", DefaultView.class);
        navigator.addView("subview", SubView.class);
    }
}
```

This would just be changed to:

```
@Route("")
public class Root extends Div {
    private final CssLayout content = new CssLayout();

    public Root() {
        add(new LegacyWrapper(content));

        Navigator navigator = new Navigator(UI.
getCurrent(), content);
        navigator.addView("", DefaultView.class);
        navigator.addView("subview", SubView.class);
    }
}
```

Now, navigation to `localhost` would show `DefaultView` and `localhost#!subview` would show `SubView` as is expected.

The thing to note in this case is that Flow doesn't receive any view change events.

Migrating Views to Flow Routes

Another open path for navigator migration is to wrap the existing `View` classes into a `MprRouteAdapter<? extends View>` and give the adapter class a `Route`.

So then the `navigator.addView("away", AwayView.class);` configuration in the previous example would be changed to:

```
@Route(value = "away", layout = MainLayout.class)
public class AwayRoute extends MprRouteAdapter<AwayView>
{
}
```

NOTE: By default the `MprRouteAdapter` creates a `<div>` on the client-side, but this can be changed by annotating the subclass with `@Tag`.

Now, there is no need to setup a `Navigator` and the View will still receive a `ViewChangeEvent` as it did with the navigator.

NOTE: Any `ViewChangeListener` should be replaced with a `BeforeEnterListener` for the `beforeViewChange` and an `AfterNavigationListener` for the `afterViewChange` to the Flow UI. See Routing lifecycle[209] documentation.

Next step

- Step 4 - Configuring UI parameters →

Or:

- ← Go back to step 2

- ← Go back to the overview[210]

24.3.9. Step 3 - Converting a UI when not using other frameworks

NOTE This step is needed in case your Vaadin 7 or 8 application does not use Spring Boot, CDI or Navigator. If it uses any of those, go back to the framework selection.

Converting UIs

When not using a Navigator, you can just replace the UI class with a Flow component by changing `init(VaadinRequest)` to a constructor and have UI.setContent to be `add(new LegacyWrapper(content))` instead.

Also remember to register `Route` for the class.

For example, this code:

```
@Theme("valo")
public class AddressbookUI extends UI {
    private HorizontalLayout content = new
HorizontalLayout();

    @Override
    protected void init(VaadinRequest vaadinRequest) {
        content.setSizeFull();
        setContent(content);
    }
}
```

Should be converted to this:

```
@Route("")
public class AddressbookLayout extends Div {
    private HorizontalLayout content = new
HorizontalLayout();

    public AddressbookLayout() {
        content.setSizeFull();
        add(new LegacyWrapper(content));
    }
}
```

NOTE Annotations in the UI, such as `@Theme` and `@Title` and so on, will be dealt with later on in the tutorial. Most of them have similar counterpart in either Flow or MPR.

To make the code look less busy you can also implement the HasLegacyComponents[211] interface so you do not need to use `new LegacyWrapper`.

```
@Route("")
public class AddressbookLayout extends Div implements
HasLegacyComponents {
    private HorizontalLayout content = new
HorizontalLayout();

    public AddressbookLayout() {
        content.setSizeFull();
        add(content);
    }
}
```

Next step

- Step 4 - Configuring UI parameters →

Or:

- ← Go back to step 2
- ← Go back to the overview[212]

24.3.10. Step 4 - Converting UI parameters

For this step, not all actions need to be done. It depends on what is configured in your original UI.

Please refer to each specific tutorial for details.

- My application uses a custom widgetset → [213]
- My application uses a custom theme → [214]
- My application uses push → [215]
- My application needs to manage the VaadinSessions → [216]
- My application uses an advanced custom UI logic → [217]

Other parameters

- For `@Title`, you should use `@PageTitle` from the `com.vaadin.flow.router` package;
- For `@Viewport`, you should use `@ViewPort` from the `com.vaadin.flow.component.page` package.

After converting those parameters, you can progress to the next step.

Next step

- Step 5 - Adding legacy components to Flow layouts →

Or:

- ← Go back to step 3
- ← Go back to the overview[218]

24.3.11. Step 5 - Adding legacy components to Flow layouts

At this stage have everything you need to make a Vaadin 7 or 8 application to run inside a Flow application.

And since this is a Flow application, you can add Flow components to the layout alongside the legacy components. You can also create legacy components and add them dynamically, for example:

```
add(new com.vaadin.flow.component.html.NativeButton(
        "Flow button that adds a FW7 Label", e -> {
            add(new LegacyWrapper(
                    new com.vaadin.ui.Label("Legacy
label")));
        }));
add(new LegacyWrapper(new com.vaadin.ui.NativeButton(
        "Legacy button that adds a Flow Label", e -> {
            add(new com.vaadin.flow.component.html.Label
("Flow label"));
        })));
```

(Fully qualified names were used in this example just to make it clear which class comes from which framework)

Check the Adding Legacy Components in a Flow Layout[219] tutorial for more details.

Adding new views to your application

We highly recommend that the new views added to the application follow the Flow’s routing system. In Flow, "Views" are called "RouteTargets", and are managed by primarily by the `@Route` annotation. For details on the differences in navigation between Flow and previous Vaadin versions, check the Routing and Navigation[220] migration guide.

For more about Flow’s navigation mechanism, check the Routing Annotation[221] tutorial.

Run in production mode

After your application is built and it’s running with MPR, you should consider packaging it for production. Check the Setting up production mode[222] tutorial for details.

- ← Go back to step 4
- ← Go back to the overview[223]

24.4. Configuration and advanced topics

24.4.1. Adding Legacy Components in a Flow Layout

As shown in the Adding Legacy components to Flow layouts [224] tutorial, you can use the `LegacyWrapper` class to wrap up any legacy component and add it to a Flow layout. In this tutorial, we are going to explore different ways of adding Components and how to customize them.

LegacyWrapper

The `LegacyWrapper` class is the most direct way of adding legacy components to your Flow application. You can add any Components, Containers or Views this way. But keep in mind that this wrapper class also creates a wrapping div around the component on the client-side.

```
Button button = new Button("Legacy button");
add(new LegacyWrapper(button));
```

By default, the style of this wrapper div has display: inline-block, and width and height set to inherit. This means that the LegacyWrapper component uses whatever size is defined on its parent element to determine its own size.

But since LegacyWrapper is a Flow component, you can customize it as much as needed. For example, you can set it to have full size, to better accommodate some legacy framework layout:

```
// Vaadin 7 or 8 VerticalLayout
VerticalLayout legacyLayout = new VerticalLayout();
LegacyWrapper wrapper = new LegacyWrapper(legacyLayout);
wrapper.setSizeFull();
add(wrapper);
```

HasLegacyComponents

In most of the cases, there's no need to customize the LegacyWrapper at all. In these situations you can use a Flow component that implements the HasLegacyComponents mixin interface, and use the add method directly for both Flow and legacy components, without having to wrap the components (the wrapping is done automatically for you).

```
// Flow layout
public class MainLayout extends Div implements
HasLegacyComponents {

    public MainLayout() {
        Button button = new Button("Legacy button");
        add(button); // no wrapping is needed
    }
}
```

The `HasLegacyComponents` interface also brings methods to remove legacy components, without having to deal with the wrappers.

← Go back to the overview[225]

24.4.2. Legacy theme in MPR

By default the theme used with MPR is 'valo' and this can be changed with adding the `MprTheme` annotation with the wanted theme name to your root navigation level, RouterLayout or to the top level @Route.

The closest instance found will be used for first initialization for a `UI` instance, but the recommendation would be to put it always on the top most `RouterLayout` in the view chain.

NOTE Runtime changing of the theme is not supported

Sample theme definition

```
@MprTheme("reindeer")
public class MainLayout extends Div implements
RouterLayout {
}

@Route(value = "", layout = MainLayout.class)
public class RootTarget extends Div {
    public RootTarget() {
      LegacyWrapper addressbookWrapper = new
LegacyWrapper(
                  new AddressbookLayout());
      add(addressbookWrapper);
    }
}
```

The theme can be a old legacy `styles.css` theme or a `styles.scss` theme. In the case of a SASS theme, on-the-fly compilation works out of the box without any changes.

Using your custom theme

Using your own Vaadin legacy theme remains the same as it was. Create your theme by following the instructions in the themes documentation for Vaadin 7[226] or Vaadin 8[227].

Then just add the `@MprTheme` annotation with your theme name on the root level navigation target and your theme will be used for the legacy framework part.

By default there is no need for a custom widgetset as MPR will function by using the `AppWidgetset` that is automatically built and configured by scanning the dependencies. For more information on the AppWidgetset and widgetset compilation see Add-ons[228] and Widget Set part of application environment[229].

NOTE When using MPR you can not use CDN for the widgetset. This means that the configuration `<vaadin.widgetset.mode>cdn</vaadin.widgetset.mode>` should be removed.

24.4.3. Custom widgetset and MPR

To use a custom widgetset for the legacy framework embedded with MPR, just add `MprWidgetset` annotation to your root navigation level, RouterLayout or to the top level @Route.

The closest instance found will be used for first initialization for a `UI` instance, but the recommendation would be to put it always on the top most `RouterLayout` in the view chain.

Sample widgetset definition

```
@MprWidgetset("com.vaadin.mpr.DemoWidgetset")
public class MainLayout extends Div implements
RouterLayout {
}

@Route(value = "", layout = MainLayout.class)
public class RootTarget extends Div {
    public RootTarget() {
      LegacyWrapper addressbookWrapper = new
LegacyWrapper(
                  new AddressbookLayout());
      add(addressbookWrapper);
    }
}
```

Generally the AppWidgetset will contain widgetsets things found by scanning the dependencies, but at times you might only want to have specific widgetsets included or you have the need for an optimized widgetset with eager and lazy parts.

NOTE The widgetset should start with `<!-- WS Compiler: manually edited -->` in the module so that it's not manually updated with imports

NOTE The custom widgetset xml needs to import MprWidgetSet e.g. `<inherits name="com.vaadin.mpr.MprWidgetSet"/>`

TIP After changing the widgetset xml, remember to recompile it. When using Vaadin Maven plugin, you can run `mvn vaadin:compile`.

← Go back to the overview[230]

24.4.4. Limitations of MPR

Using MPR in your project to port a legacy application to Vaadin Flow has some known limitations. This is the current list of limitations - keep in mind that it can change over time as new features are implemented.

It's possible to add a legacy component to a Flow layout, but not a Flow component in a legacy layout

The `LegacyWrapper` class and the `HasLegacyComponents` mixin interface only work for adding legacy components or views in a Flow layout, and not the other way around.

Custom UIs are supported, but not building the application in there

Custom legacy UIs can be used to host configuration settings, but can't be used as a layout. You need to convert your UIs layouting to be in components, and then wrap them with a `LegacyWrapper` and add them to a Flow layout. Custom UIs can be used as long as they extend MprUI, the root navigation target is annotated with `@LegacyUI(*.class)` and you don't use `UI.setContent();`.

Multiple UIs are not supported

Also because of the need of the `MprUI`, multiple legacy UIs are not supported. They need to be converted to a legacy layout and then wrapped in a `LegacyWrapper` for Flow to use them.

Custom legacy VaadinServlets are not supported

MPR has a special servlet (called `MprServlet`) that knows how to map each request to the appropriate framework. This makes legacy VaadinServlets unusable in an application controlled by the MPR. If you need some custom functionality, you can use the VaadinServlet provided by Flow instead. See Flow documentation on Dynamic content [231] for details.

Only Vaadin 7.7.14+ and Vaadin 8.6.0+ are supported

The 7.7.14 and 8.6.0 releases introduced the changes needed for MPR to work with Vaadin 7 and Vaadin 8 respectively. Versions before 7.7.14 and 8.6.0 are not supported.

CDN and FETCH are not supported for the widgetset mode

When using MPR you can not use CDN for the widgetset. This means that the configuration

- `<vaadin.widgetset.mode>cdn</vaadin.widgetset.mode>` or
- `<vaadin.widgetset.mode>fetch</vaadin.widgetset.mode>`

should be removed from the `pom.xml`.

Runtime changing of the legacy theme is not supported

when using the `@MprTheme` annotation, the legacy theme is set at startup time, and can’t be changed dynamically after the application has been started.

No ViewScope in Flow Spring add-on

The Vaadin 14 Spring add-on doesn’t have a feature comparable with `@ViewScope`, so when using MPR with Spring, that scope is not supported.

`UI.getCurrent()` is no longer automatically inherited into the spawned thread

This code no longer works when running with MPR:

```
button.addClickListener(event -> {
    new Thread(() -> {
        UI.getCurrent()
            .access(() -> Notification.show("Hello from
thread"));
    }).start();
});
```

The workaround for this is to store `UI.getCurrent()` already in the click listener into an effectively final variable that the thread can use:

```
button.addClickListener(event -> {
    UI ui = UI.getCurrent();
    new Thread(() -> {
        ui.access(() -> Notification.show("Hello from
thread"));
    }).start();
});
```

Code hot swap during development time is not supported

When MPR is on the classpath, it's currently not possible to use hot swap of code for fast deployment of the application during development time. See this issue[232] for details.

PhantomJS is not supported

The PhantomJS project is not maintained anymore, and Flow doesn't officially support it. Old Vaadin projects that rely on PhantomJS should use alternatives when using MPR, such as headless browsers.

Java 8+ is required

Since the application managed by the MPR is a Flow application, it requires Java 8+ runtime to work.

Old browsers are not supported

Only the browsers supported by Flow are supported in an application with the MPR. Those include IE11 (with transpilation), and evergreen browsers (latest versions of Chrome, Firefox, Opera, Safari and Edge).

← Go back to the overview[233]

24.4.5. Setting up production mode

To run Flow+MPR in production mode you need to update the project as told in Taking your Application into Production[234].

NOTE The flow-server-production-mode dependency sets `productionMode=true` using a `web-fragment.xml` that then also reflects to Vaadin 7/8 production mode setting.

Sample production mode profile for MPR

```xml
<profile>
    <!-- Production mode is activated using -Pproduction
-->
    <id>production</id>
    <properties>
        <vaadin.productionMode>
true</vaadin.productionMode>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>flow-server-production-
mode</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>com.vaadin</groupId>
                <artifactId>flow-maven-
plugin</artifactId>
                <version>${flow.version}</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>build-frontend</goal>
                        </goals>
                        <phase>compile</phase>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</profile>
```

← Go back to the overview[235]

24.4.6. Push and MPR

In order to enable push mode for any navigation target in MPR, place `@Push` Flow annotation on it. The annotation has similar parameters (except for the deprecated ones) as the ones used in the Legacy Framework. Refer to Flow push documentation[236] or javadocs for particular description on each parameter.

When enabled, push uses Flow implementation, no Legacy Framework push is used. Although all Legacy methods such as `UI::access` and `UI::push` work as, if nothing's changed, hence no code updates are needed here.

← Go back to the overview[237]

24.4.7. Using sessions with MPR

The state of the in an MPR project is managed by the `com.vaadin.flow.server.VaadinSession` class, but the methods from the legacy `com.vaadin.server.VaadinSession` class can also be used, since both wrap the same `javax.servlet.http.HttpSession`.

Invalidating a session

To invalidate a session (and possibly start a new one), you can invalidate the session managed by Flow and reload the page.

For example:

```
Button close = new Button("Close session", event -> {
    VaadinSession.getCurrent().getSession().invalidate();
    UI.getCurrent().getPage().reload();
});
```

← Go back to the overview[238]

24.4.8. Using a custom legacy UI class

NOTE This is intended for advanced cases only, where using the recommended migration path is not enough to cover specific logic that cannot be easily ported to Flow.

If you have a need for a specific UI class to be used for the legacy Vaadin UI you can have the UI class extend MprUI. Note that the UI **can not** be used for layouting purposes.

```
public class MyCustomUI extends MprUI {
    @Override
    protected void init(VaadinRequest request) {
        super.init(request);
    }
}
```

NOTE You need to call `super.init(request)` if you need to override the init method

Then you need to tell the application that this class should be used with the annotation `@LegacyUI()`.

```
@Route("")
@LegacyUI(MyCustomUI.class)
public class MainLayout extends Div {
}
```

Now when navigating to the "" (root) route you will get a MyCustomUI instead of the default MprUI.

← Go back to the overview[239]

24.4.9. Creating V7 and V14 CDI applications side-by-side

If you have an application developed using Vaadin 7 and CDI, you have the option to keep your legacy code untouched and continue developing new pages with V14.

You will also be able to use CDI beans, e.g. `SessionScoped` beans, in both V14 and Vaadin 7 parts of your application. The following instructions are step-by-step guide on how to adopt this approach.

1. Add Vaadin 14 to your maven dependencies.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-bom</artifactId>
            <type>pom</type>
            <scope>import</scope>
            <version>14.0.0</version>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>com.vaadin</groupId>
        <artifactId>vaadin-core</artifactId>
    </dependency>
</dependencies>
```

2. Exclude conflicted dependencies between Vaadin 7 and Vaadin 14 which are `jsoup` and `atmosphere-runtime` from Vaadin 7 in your `pom.xml`, like shown in the following example:

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-server</artifactId>
    <version>${framework.7.version}</version>
    <exclusions>
        <exclusion>
            <groupId>
com.vaadin.external.atmosphere</groupId>
            <artifactId>atmosphere-
runtime</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.jsoup</groupId>
            <artifactId>jsoup</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

3. Remove dependency of `vaadin-cdi 1.*` and add a dependency to `mpr-cdi-v7 1.0.0`.

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>mpr-cdi-v7</artifactId>
    <version>1.0.0.alpha1</version>
    <!--
        At the moment mpr-cdi-v7 is in pre-release
stage and you need to use
        the version 1.0.0.alpha1.
    -->
</dependency>
```

NOTE Since, a pre-release version of a Vaadin product is being used you need to add the `vaadin-prereleases` repository to your `pom.xml`, like this:

```
<repositories>
    <repository>
        <id>vaadin-prereleases</id>
        <url>https://maven.vaadin.com/vaadin-
prereleases</url>
    </repository>
</repositories>
```

At the moment `mpr-cdi-v7` is in pre-release stage and you may need to use e.g. version `1.0.0.alpha1`.

4. Add the `vaadin-cdi` dependency. The versions is not needed as it is defined by the `vaadin-bom`.

```
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-cdi</artifactId>
</dependency>
```

5. Since the root path of your application is managed by Vaadin 7, you need to define the Vaadin 14 servlet manually, and set its url pattern to a value that doesn't collide with any of the V7 servlets.

```
@WebServlet(name = "Flow Servlet", urlPatterns = {
        MyFlowServlet.FLOW_SERVLET_ROOT + "/*" })
public class MyFlowServlet extends CdiVaadinServlet {
    public static final String FLOW_SERVLET_ROOT =
"flow";
}
```

6. Now, you can have both Vaadin 7 and Vaadin 14 parts of your application in one project. To navigate from Vaadin 7

part to Vaadin 14 part you can simply use the following line of code.

```
getUI().getPage().setLocation(MyFlowServlet.FLOW_SERVL
ET_ROOT);
```

And, to navigate from Vaadin 14 part to a Vaadin 7 view you can for example use an `Anchor` like the below code.

```
Anchor anchor = new Anchor("/#!home", "Home");
add(anchor);
```

7. To have shared data between Vaadin 14 and Vaadin 7 parts, you can use `SessionScoped` beans that are shared for both V7 and V14 applications.

```
@SessionScoped
public class SecurityContext implements Serializable {
    private User currentUser = new User();

    public boolean signIn(String username, String
password) {
        if (username == null || username.isEmpty())
            return false;

        currentUser.setUsername(username);

        return true;
    }
}
```

----

[197] https://vaadin.com/pricing
[198] http://github.com/vaadin/multiplatform-runtime
[199] https://vaadin.com/docs/flow/Overview.html
[200] https://vaadin.com/docs/flow/Overview.html
[201] https://vaadin.com/docs/flow/Overview.html

[202] https://vaadin.com/docs/flow/advanced/tutorial-dynamic-content.html#using-custom-servlet-and-request-parameters
[203] https://vaadin.com/docs/flow/Overview.html
[204] https://vaadin.com/docs/flow/Overview.html
[205] https://vaadin.com/docs/flow/spring/tutorial-spring-basic.html
[206] https://github.com/vaadin/spring
[207] https://vaadin.com/docs/flow/Overview.html
[208] https://vaadin.com/docs/flow/Overview.html
[209] https://vaadin.com/docs/flow/routing/tutorial-routing-lifecycle.html
[210] https://vaadin.com/docs/flow/Overview.html
[211] https://vaadin.com/docs/flow/configuration/adding-legacy-components.html#hasLegacyComponents
[212] https://vaadin.com/docs/flow/Overview.html
[213] https://vaadin.com/docs/flow/configuration/legacy-widgetset.html
[214] https://vaadin.com/docs/flow/configuration/legacy-theme.html
[215] https://vaadin.com/docs/flow/configuration/push.html
[216] https://vaadin.com/docs/flow/configuration/session.html
[217] https://vaadin.com/docs/flow/configuration/custom-ui.html
[218] https://vaadin.com/docs/flow/Overview.html
[219] https://vaadin.com/docs/flow/configuration/adding-legacy-components.html
[220] https://vaadin.com/docs/flow/migration/4-routing-navigation.html
[221] https://vaadin.com/docs/flow/routing/tutorial-routing-annotation.html
[222] https://vaadin.com/docs/flow/configuration/production-mode.html
[223] https://vaadin.com/docs/flow/Overview.html
[224] https://vaadin.com/docs/flow/introduction/step-5-adding-legacy-components.html
[225] https://vaadin.com/docs/flow/Overview.html
[226] https://vaadin.com/docs/v7/framework/themes/themes-overview.html
[227] https://vaadin.com/docs/v8/framework/themes/themes-overview.html
[228] https://vaadin.com/docs/v7/framework/addons/addons-overview.html#installing
[229] https://vaadin.com/docs/v7/framework/application/application-environment.html

[230] https://vaadin.com/docs/flow/Overview.html
[231] https://vaadin.com/docs/flow/advanced/tutorial-dynamic-content.html#using-custom-servlet-and-request-parameters
[232] https://github.com/vaadin/multiplatform-runtime/issues/19
[233] https://vaadin.com/docs/flow/Overview.html
[234] https://vaadin.com/docs/flow/production/tutorial-production-mode-basic.html
[235] https://vaadin.com/docs/flow/Overview.html
[236] https://vaadin.com/docs/flow/advanced/tutorial-push-configuration.html
[237] https://vaadin.com/docs/flow/Overview.html
[238] https://vaadin.com/docs/flow/Overview.html
[239] https://vaadin.com/docs/flow/Overview.html

25. Advanced Topics

25.1. 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.

25.1.1. Deployment

Before a Vaadin application can be used, it has to be deployed to a Java web server. 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.

There is no need to define your own servlet class (which should extend the `VaadinServlet` class) if you are using Servlet 3.0 specification. You just need to have at least one class annotated with `@Route` annotation and a `VaadinServlet` instance will be registered for you automatically and Vaadin will register all servlets required automatically.

Automatic servlet registration

When starting, Vaadin application tries to registed the following servlets:

- Vaadin application servlet, mapped to `/*` path

This servlet is needed to serve the main application files.

The servlet won’t be registered, if any {@link VaadinServlet} is registered already or if there are no classes annotated with {@link Route} annotation.

- Frontend files servlet, mapped to `/frontend/*` path

This servlet is required in the development mode to serve the WebJar contents and is only registered when the application is started in the development mode.

In addition to the rules mentioned above, a servlet won’t be registered, if * there is a servlet that had been mapped to the same path already * or if `disable.automatic.servlet.registration` system property is set to `true`

Undeploying and Redeploying

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

25.1.2. Vaadin Servlet and Service

The VaadinServlet 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 handles all tasks common to servlets in a VaadinService. It manages sessions, gives access to the deployment configuration information, handles system messages, and does various other tasks. Any further servlet specific tasks are handled in the corresponding VaadinServletService. 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 class and override the createServletService() to create a custom service object.

25.1.3. User Session

A user session begins when a user first makes a request to a Vaadin servlet by opening the URL for a particular UI. All server requests belonging to a particular UI class are processed by the VaadinServlet 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, through a WrappedSession. You can also access the deployment configuration through VaadinSession.

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

Vaadin Servlet 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
    }
}
```

25.1.4. 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.

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 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 BootstrapHandler.js script whose content is inlined into the page.

25.1.5. 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.

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.

25.1.6. 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 via JS execution.

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.

25.1.7. 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 closeIdleSessions deployment configuration parameter of the servlet is set to true 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.

See "Flow runtime configuration" section about setting configuration parameters.

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

25.1.8. 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.

```java
@Route("")
public class MainLayout extends Div {

    protected void onAttach(AttachEvent attachEvent) {
        UI ui = getUI().get();
        Button button = new Button("Logout", event -> {
            // Redirect this page immediately
            ui.getPage().executeJs(
"window.location.href='logout.html'");

            // Close the session
            ui.getSession().close();
        });

        add(button);

        // Notice quickly if other UIs are closed
        ui.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.

25.2. I18N localization

To use localization and translation strings the application only needs to implement `I18NProvider` and define the fully qualified class name in the property `i18n.provider`.

25.2.1. Defining the i18n provider property

The `i18n.provider` property can be set from the command line as a system property, as a Servlet init parameter in the `web.xml` or using the `@WebServlet` annotation.

As a system property the parameter needs the `vaadin` prefix e.g.:

```
mvn jetty:run
-Dvaadin.i18n.provider=com.vaadin.example.ui.TranslationP
rovider
```

When using the annotation you could have the servlet class as:

```
@WebServlet(urlPatterns = "/*", name = "slot",
asyncSupported = true, initParams = {
        @WebInitParam(name = Constants.I18N_PROVIDER,
value = "com.vaadin.example.ui.TranslationProvider") })
public class ApplicationServlet extends VaadinServlet {
}
```

Or when using the `web.xml` file:

```
<?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>i18n.provider</param-name>
      <param-
value>com.vaadin.example.ui.TranslationProvider</param-
value>
    </init-param>
  </servlet>

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

You may provide a `I18NProvider` as a bean in case you are using Spring. All you need in this case it's just annotate your implementation with `@Component` so it becomes available as a Spring bean. Spring add-on will automatically use it in case if it's available. See the class SimpleI18NProvider.java[240] implemented in the tutorial project as an example.

25.2.2. Locale selection for new session

The initial locale is decided by matching the locales provided by the `I18NProvider` against the `Accept-Language` header in the initial response from the client.

If an exact match (language + country) is found that will then be used, else we will try to match on only language. If neither is found the locale will be set to the first 'supported' locale from `I18NProvider.getProvidedLocales()` and if that is empty `Locale.getDefault()` will be used.

25.2.3. Provider sample for translation

For this example we enable Finnish and English to be used with Finnish being the **"default"** that is used if the user client doesn’t specify english as an accepted language.

In this sample the language `.properties` files start with **"translate"** e.g. `translate.properties` (for default), `translate_fi_FI.properties` and `translate_en_GB.properties`

The translation properties files are in the example loaded using the class loader so they should be located on the classpath for example in the resources folder e.g. `src/main/resources` for a default maven setup.

The `LoadingCache` used in the implementation is from the Google Guava package.

Sample i18n provider implementation

```
public class TranslationProvider implements I18NProvider
{

    public static final String BUNDLE_PREFIX = "
translate";

    public final Locale LOCALE_FI = new Locale("fi", "FI
");
    public final Locale LOCALE_EN = new Locale("en", "GB
");
```

```
    private List<Locale> locales = Collections
            .unmodifiableList(Arrays.asList(LOCALE_FI,
LOCALE_EN));

    private static final LoadingCache<Locale,
ResourceBundle> bundleCache = CacheBuilder
            .newBuilder().expireAfterWrite(1, TimeUnit
.DAYS)
            .build(new CacheLoader<Locale,
ResourceBundle>() {

                @Override
                public ResourceBundle load(final Locale
key) throws Exception {
                    return initializeBundle(key);
                }
            });

    @Override
    public List<Locale> getProvidedLocales() {
        return locales;
    }

    @Override
    public String getTranslation(String key, Locale
locale, Object... params) {
        if (key == null) {
            LoggerFactory.getLogger(TranslationProvider
.class.getName())
                    .warn("Got lang request for key with
null value!");
            return "";
        }

        final ResourceBundle bundle = bundleCache
.getUnchecked(locale);

        String value;
        try {
            value = bundle.getString(key);
        } catch (final MissingResourceException e) {
            LoggerFactory.getLogger(TranslationProvider
.class.getName())
                    .warn("Missing resource", e);
            return "!" + locale.getLanguage() + ": " +
key;
        }
        if (params.length > 0) {
            value = MessageFormat.format(value, params);
        }
        return value;
    }

    private static ResourceBundle initializeBundle(final
Locale locale) {
        return readProperties(locale);
    }

    protected static ResourceBundle readProperties(final
Locale locale) {
        final ClassLoader cl = TranslationProvider.class
.getClassLoader();

        ResourceBundle propertiesBundle = null;
        try {
            propertiesBundle = ResourceBundle.getBundle
(BUNDLE_PREFIX, locale,
                    cl);
        } catch (final MissingResourceException e) {
            LoggerFactory.getLogger(TranslationProvider
.class.getName())
                    .warn("Missing resource", e);
        }
        return propertiesBundle;
    }
}
```

Using localization in the application

Using the internationalization in the application is a combination of using the I18NProvider and updating the translations on locale change.

To make this simple the application classes that control the captions and texts that are localized can implement the `LocaleChangeObserver` to receive events for locale change.

This observer will also be notified on navigation in the attach phase of before navigation after any url parameters are set, so that the state from a url parameter can be used.

```
public class LocaleObserver extends Div implements
LocaleChangeObserver {

    @Override
    public void localeChange(LocaleChangeEvent event) {
        setText(getTranslation("my.translation",
getUserId()));
    }
}
```

Using localization without using LocaleChangeObserver

I18NProvider without LocaleChangeObserver

```
public class MyLocale extends Div {

    public MyLocale() {
        setText(getTranslation("my.translation",
getUserId()));
    }
}
```

25.3. Modifying the bootstrap page

The application bootstrap page is created for you by the framework and normally there is no need to modify it. For instance Flow includes its internal JavaScripts to be able to provide its core functionality. Also it is possible to include additional JavaScripts, HTML imports and Style Sheets using annotations `@JavaScript`, `HtmlImport` and `@StyleSheet` (see Including Style Sheets[241] and Importing html/javascript[242]).

Sometimes you may want to customize the page header and add there some additional data, e.g. custom `meta` tags. Such markup is required to enable your web page to become a rich object in a social graph using OpenGraph protocol[243].

25.3.1. Viewport annotation

Viewport meta-tag can be set to the initial page by annotating the navigation target with `@Route` or the top most `RouterLayout` that builds the navigation target chain.

```
@Route("")
@Viewport("width=device-width")
public class MyApp extends Div {
  public MyApp() {
    setText("Hello world");
  }
}
```

```
@Route(value = "", layout = MyLayout.class)
public class MyView extends Div {
  public MyView() {
    setText("Hello world");
  }
}

@Viewport("width=device-width")
public class MyLayout extends Div implements RouterLayout
{
}
```

NOTE If the `Viewport annotation is not on a `@Route` Component or a top `RouterLayout` an exception will be thrown on startup.

25.3.2. Inline annotation

The initial page can be modified by using the `@Inline` annotations to add file contents to either the `<head>` or the `<body>`. Inline is repeatable, so multiple `@Inline` annotations can be added at once.

Inline will add the contents of a classloader resource by default as appended to the head of the initial page with type decided by the file suffix.

The configurations available for inlining contents are:

- TargetElement [**HEAD**, BODY]
- Position [**APPEND**, PREPEND]
- Wrapping [**AUTOMATIC**, NONE, STYLESHEET, JAVASCRIPT]

```
@Route(value = "", layout = MyInline.class)
public class MyRoot extends Div {
  public MyRoot() {
  }
}

@Inline("initialization.js")
@Inline("initial_style.css")
@Inline(value = "important_styles", wrapping = Inline
.Wrapping.STYLESHEET)
public class MyInline extends Div implements RouterLayout
{
}
```

NOTE: If the `Inline` annotation is not on a `@Route` Component or a top `RouterLayout` an exception will be thrown on startup.

25.3.3. PageConfigurator

To be able to modify default bootstrap page and add your custom meta tags on the page you can implement the `PageConfigurator` on the navigation target with `@Route` or the top most `RouterLayout` that builds the navigation target chain. The `PageConfigurator` gives you easy access to customize the `LoadingIndicatorConfiguration`, `ReconnectDialogConfiguration` and `PushConfiguration` for the initial response.

With the `PageConfigurator` you can prepend or append `javascript`, `html` and `css` to the head by giving a file on the classpath or as content string. Also supported is adding links and meta tags which can also be either prepended or appended.

Setting the viewport meta tag using `InitialPageSettings::setViewport` will override any viewport set through a `@Viewport` annotation.

By default everything is appended, but if needed you can give the position `InitialPageSettings.Position.PREPEND` to have the item prepended to head instead.

Here is the code for the `PageConfigurator` implementation on the top `RouterLayout` which modifies the header of the page:

```
@Route(value = "", layout = MainLayout.class)
public class Root extends Div {
}

public class MainLayout extends Div
        implements RouterLayout, PageConfigurator {

    @Override
    public void configurePage(InitialPageSettings
settings) {
        settings.addInlineFromFile(InitialPageSettings
.Position.PREPEND,
                "inline.js", InitialPageSettings.
WrapMode.JAVASCRIPT);

        settings.addMetaTag("og:title", "The Rock");
        settings.addMetaTag("og:type", "video.movie");
        settings.addMetaTag("og:url",
                "http://www.imdb.com/title/tt0117500/");
        settings.addMetaTag("og:image",
                "http://ia.media-
imdb.com/images/rock.jpg");

        settings.addLink("shortcut icon",
"icons/favicon.ico");
        settings.addFavIcon("icon", "icons/icon-192.png",
"192x192");
    }
}
```

NOTE — If the PageConfigurator implementation is not on a `@Route` Component or a `RouterLayout` used from a route it will not be used.

25.3.4. Setting the body size styles

By default, the body element in a Flow application has size properties `height = "100vh", width = "100vw"`, which makes the page fill the entire viewport. To change the width and height properties of the body you can either use the

@BodySize annotation or the PageConfigurator.

For @BodySize you just add it to the the navigation target with @Route or the top most RouterLayout that builds the navigation target chain. You can pass custom height and width properties for the annotation, or leave them out to just prevent the default size to be applied for the body:

```
@Route("")
@BodySize
public static class BodySizeAnnotatedRoute extends Div {
}
```

With the PageConfigurator you can just addInlineContent like:

```
@Route("")
public static class InitialPageConfiguratorBodyStyle
extends Div
        implements PageConfigurator {
    @Override
    public void configurePage(InitialPageSettings
settings) {
        settings.addInlineWithContents("body {width:
100vw; height:100vh;}",
                InitialPageSettings.WrapMode.STYLESHEET);
    }
}
```

NOTE Only one way should be used as else the later statement will override the earlier one. In practise this would mean that by default the PageConfigurator will override the @BodySize except if the inlining is done as a PREPEND then the @BodySize will be the deciding one.

NOTE If the `BodySize` annotation is not on a `@Route` Component or a top `RouterLayout` an exception will be thrown on startup.

NOTE When using an empty `@BodySize` annotation (which doesn't apply any sizing for the UI / body), you will not be able to use relative sizing (% as unit) for any component, unless the component has a parent that has defined size using anything else than % as the unit. For that reason, it is recommended to use the default settings for the body size, by omitting the `@BodySize` annotation, or to declare a specific size for it.

25.3.5. BootstrapListener

To be able to modify default bootstrap page and add your custom meta tags on the page you should use your `BootstrapListener` implementation and add it to the `ServiceInitEvent` instance available in a `VaadinServiceInitListener`.

Here is the code for the `BoostrapListener` implementation which modifies the header of the page:

```
public class CustomBootstrapListener implements
BootstrapListener {

    public void modifyBootstrapPage(BootstrapPageResponse
response) {
        Document document = response.getDocument();

        Element head = document.head();

        head.appendChild(createMeta(document, "og:title",
"The Rock"));
        head.appendChild(createMeta(document, "og:type",
"video.movie"));
        head.appendChild(createMeta(document, "og:url",
                "http://www.imdb.com/title/tt0117500/"));
        head.appendChild(createMeta(document, "og:image",
                "http://ia.media-
imdb.com/images/rock.jpg"));
    }

    private Element createMeta(Document document, String
property,
            String content) {
        Element meta = document.createElement("meta");
        meta.attr("property", property);
        meta.attr("content", content);
        return meta;
    }
}
```

Now this listener should be added to a `ServiceInitEvent` which is sent when a Vaadin service is initialized. Take a look on the ServiceInitListener tutorial[244] on how to configure it.

25.3.6. Adding static HTML contents

The framework provides multiple ways of adding static content to the page. Here we cover three different ways of adding a favicon.

- using `InitialPageSettings#addLink()`

```
public class Layout1 extends Div implements RouterLayout,
PageConfigurator {

    @Override
    public void configurePage(InitialPageSettings
settings) {
        HashMap<String, String> attributes = new HashMap
<>();
        attributes.put("rel", "shortcut icon");
        settings.addLink("icons/favicon.ico", attributes
);
    }
}
```

- using `InitialPageSettings#addInlineWithContents()`

```
public class Layout2 extends Div implements RouterLayout,
PageConfigurator {

    @Override
    public void configurePage(InitialPageSettings
settings) {
        settings.addInlineWithContents(
                "<link rel=\"shortcut icon\" href=
\"icons/favicon.ico\">",
                InitialPageSettings.WrapMode.NONE);
    }
}
```

- using `BootstrapListener#modifyBootstrapPage()` (documentation)

```
public class Layout3 extends Div
            implements RouterLayout, BootstrapListener {

        @Override
        public void modifyBootstrapPage
(BootstrapPageResponse response) {
            final Element head = response.getDocument()
.head();
            head.append(
                    "<link rel=\"shortcut icon\" href=
\"icons/favicon.ico\">");
        }
    }
```

But most commonly, you will deal with quite many files, in this case, you can see that it causes a lot of hard coding easily. To avoid this, we recommend you to move all the contents into a file, (e.g. `your-content.html`) and inline this file in your `PageConfigurator`

```
public class Layout4 extends Div implements RouterLayout,
PageConfigurator {

        @Override
        public void configurePage(InitialPageSettings
settings) {
            settings.addInlineFromFile("your-
content.html",
                    InitialPageSettings.WrapMode.NONE);
        }
    }
```

25.4. Changing Flow behavior with runtime configuration.

Flow application have extra parameters that may be set to change its behavior.

25.4.1. How to set parameters

Parameters can be set the following way:

- by setting the system property

In this case, `vaadin.` prefix is needed to be specified before the parameter names. For instance, Spring Boot can be configured in application.properties[245] file. For more details, refer to Flow Spring configuration

Alternatively, system properties can be set via command line:

```
mvn jetty:run
-Dvaadin.frontend.url.es6=http://mydomain.com/es6/
-Dvaadin.frontend.url.es5=http://mydomain.com/es5/
```

- by setting the servlet init parameters

You can use the traditional `web.xml` file or the Servlet 3.0 `@WebServlet` annotation:

```
@WebServlet(urlPatterns = "/*", name = "myservlet",
asyncSupported = true, initParams = {
        @WebInitParam(name = "frontend.url.es6", value =
"http://mydomain.com/es6/"),
        @WebInitParam(name = "frontend.url.es5", value =
"http://mydomain.com/es5/") })
@VaadinServletConfiguration(productionMode = false)
public class MyServlet extends VaadinServlet {
}
```

Or when using the `web.xml` file:

```xml
<?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>frontend.url.es6</param-name>
      <param-value>http://mydomain.com/es6/</param-value>
    </init-param>

    <init-param>
      <param-name>frontend.url.es5</param-name>
      <param-value>http://mydomain.com/es5/</param-value>
    </init-param>
  </servlet>

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

NOTE System properties override application properties ergo if you have both properties with the same name specified, the system one will be used.

25.4.2. Parameters and description

- disable.webjars - if set to `true`, webjars would be ignored during request resolving, allowing Flow to use external source of web components' files.

NOTE Webjars are enabled for development mode and disabled for production mode by default, unless explicitly overridden by parameter specified. In the future, webjars are expected to the always enabled by default.

Next group of parameters are paths to external web component’s locations in development and production modes.

Development mode:

- frontend.url.dev (default value is `context://frontend`) – a location Flow searches web components' files in development mode. Supports changes reload on a working application and should be set as a directory, containing `bower.json` file.

Production mode:

- productionMode (default value is `false`) – turns application to work in production mode
- frontend.url.es5 (default value is `context://frontend-es5`) - a location Flow searches web components' files in production mode when the request is coming from older browsers, not supporting es6[246], default web components' development language version.
- frontend.url.es6 (default value is `context://frontend-es6`) - a location Flow searches web components' files in production mode for requests from modern browsers
- load.es5.adapters (default value is `true`) – include polyfills for browsers that does not support ES6 to their initial page. In order for web components to work, extra libraries (polyfills) are required to be loaded, can be turned off if different versions or libraries should be included instead.

25.5. The Loading Indicator

To inform the user that loading is in progress and that the *UI* is currently unresponsive, a *loading indicator* is displayed. A longer loading time might be due to e.g. bad network conditions. The framework automatically displays a loading indicator after a configurable delay when a server request starts, and hides it after the response processing has ended.

By default, the loading indicator is shown at the top of the viewport after a delay. You can turn the indicator off, change delays or customize the looks of the indicator. The theming targets the `<div class="v-loading-indicator"></div>` element located inside the `<body>` element. **You need to also need to toggle the default theming off**.

```
<body>
  <!-- application root level element omitted -->
  <!-- "the framework removes "display: none" when
indicator shown -->
  <div class="v-loading-indicator first"
       style="display: none;">
  </div>
</body>
```

The loading indicator can be configured from Java by accessing the configuration object from `UI`. The easiest way to do this is with the `PageConfigurator` (see bootstrap page docs for more information) and the changes are applied already on the initial response.

```
public class MainLayout extends Composite<Div> implements
PageConfigurator, RouterLayout {

    // other implementation omitted
    @Override
    public void configurePage(InitialPageSettings
settings) {
        LoadingIndicatorConfiguration conf = settings
.getLoadingIndicatorConfiguration();

        // disable default theme -> loading indicator
will not be shown
        conf.setApplyDefaultTheme(false);
    }
}
```

The configuration object can be used for configuring the delays after which the indicator changes "stages". The indicator is shown after a delay of 300ms by default and a class name `first` is set to it. There are two additional delays which you can configure. After the delays, class names `second` and `third` are set and can be used to change the style of the loading indicator after certain time has passed.

```
public class MainLayout extends Composite<Div> implements
PageConfigurator, RouterLayout {

    @Override
    public void configurePage(InitialPageSettings
settings) {
        LoadingIndicatorConfiguration conf = settings
.getLoadingIndicatorConfiguration();

        /*
         * Delay for showing the indicator and setting
the 'first' class name.
         */
        conf.setFirstDelay(300); // 300ms is the default

        /* Delay for setting the 'second' class name */
        conf.setSecondDelay(1500); // 1500ms is the
default

        /* Delay for setting the 'third' class name */
        conf.setThirdDelay(5000); // 5000ms is the
default
    }
}
```

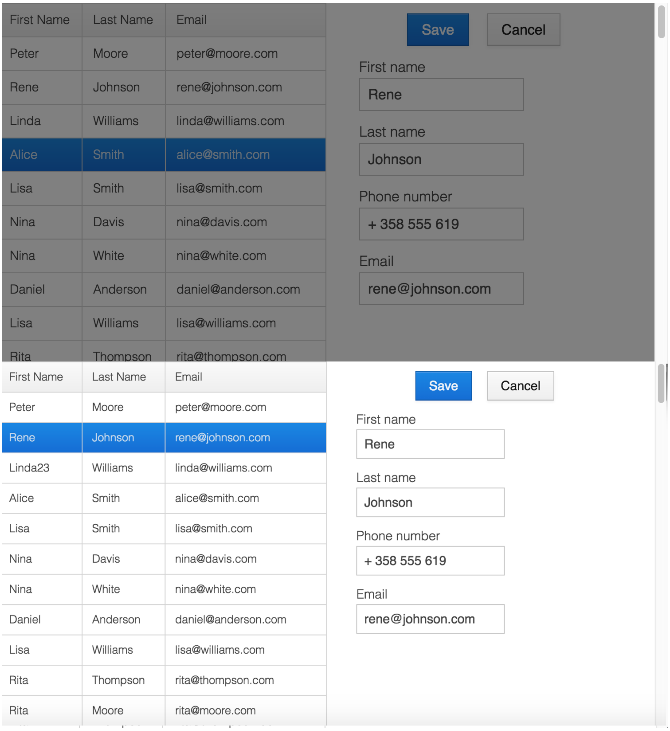
25.5.1. Displaying a Modal Curtain

To show an alternative for the default loading indicator theme, this examples demonstrates how to show a loading indicator that simply darkens the UI. The darkening is animated, so that it does not flash the screen. The darkening starts after the server side round-trip takes over 0.5 seconds (300ms delay configured in java + 200ms animation delay).

NOTE In addition to the css, the default theme should be explicitly disabled via Java, as shown in the previous chapter.

```
.v-loading-indicator {
  position: fixed; /* Occupy whole screen even if
scrolled */
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  /* Eat mouse events when visible, doesn't prevent
scrolling */
  pointer-events: auto;
  /* Animated with delay to prevent flashing */
  animation: fadein 0.3s ease-out 0.2s normal 1 both;
  z-index: 2147483647;
}
@keyframes fadein {
  0% {
    background: rgba(0,0,0,0);
  }
  100% {
    background: rgba(0,0,0,.5); /* Darkens the UI */
  }
}
```

The next image illustrates an application with the modal curtain visible during loading (above) compared to the normal state (below).



First Name	Last Name	Email
Peter	Moore	peter@moore.com
Rene	Johnson	rene@johnson.com
Linda	Williams	linda@williams.com
Alice	Smith	alice@smith.com
Lisa	Smith	lisa@smith.com
Nina	Davis	nina@davis.com
Nina	White	nina@white.com
Daniel	Anderson	daniel@anderson.com
Lisa	Williams	lisa@williams.com
Rita	Thompson	rita@thompson.com

First name	Rene
Last name	Johnson
Phone number	+ 358 555 619
Email	rene@johnson.com

First Name	Last Name	Email
Peter	Moore	peter@moore.com
Rene	Johnson	rene@johnson.com
Linda23	Williams	linda@williams.com
Alice	Smith	alice@smith.com
Lisa	Smith	lisa@smith.com
Nina	Davis	nina@davis.com
Nina	White	nina@white.com
Daniel	Anderson	daniel@anderson.com
Lisa	Williams	lisa@williams.com
Rita	Thompson	rita@thompson.com
Rita	Moore	rita@moore.com

First name	Rene
Last name	Johnson
Phone number	+ 358 555 619
Email	rene@johnson.com

25.5.2. Displaying a Changing Loading Indicator

Once the loading indicator is displayed, it gets the class name `first`. After the second and third configurable delays, it gets the `second` and the `third` class names respectively. You can use those class names in your styling to let the look reflect how long time the user has been waiting.

The following style snippet demonstrates how to create an animation that changes color as the user is waiting.

NOTE In addition to the css, the default theme should be explicitly disabled via Java.

```
.v-loading-indicator {
  position: fixed;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  pointer-events: auto;
  z-index: 2147483647;
}
.v-loading-indicator:before {
  width: 76px;
  height: 76px;

  position: absolute;
  top: 50%;
  left: 50%;

  margin: -38px 0 0 -38px;

  border-radius: 100%;
  animation: bouncedelay 1.2s infinite 0.4s ease-in-out
both;
  content: "";
}

.v-loading-indicator.first:before {
  background-color: skyblue;
}

.v-loading-indicator.second:before {
  background-color: salmon;
}

.v-loading-indicator.third:before {
  background-color: red;
}

@keyframes bouncedelay {
  0%, 80%, 100% {
    transform: scale(0);
  } 40% {
    transform: scale(1.0);
  }
}
```

25.6. Server Push Configuration

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.

This section describes how to configure server push in your application. See Asynchronous Updates for a description on how to use server push from your application code and Creating Collaborative Views for a full multi-user example.

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 the Atmosphere framework[247] for client-server communication.

25.6.1. Enabling Push in your application

To enable server push, you need to define the push mode either in the deployment descriptor or with the `@Push` annotation for the main layout or individual views of your application.

25.6.2. Push Modes and Transports

You can use server push in two modes: `automatic` and `manual`. The automatic mode pushes changes to the browser automatically 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+XHR is the default transport.

25.6.3. The `@Push` annotation

You can enable server push for the main layout or individual view of an application with the `@Push` annotation as follows. It defaults to automatic mode (`PushMode.AUTOMATIC`).

```
@Push
public class MyLayout extends Div implements RouterLayout
{
```

To enable manual mode, you need to give the `PushMode.MANUAL` parameter as follows:

```
@Push(PushMode.MANUAL)
public class MyLayout extends Div implements RouterLayout
{
```

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 MyLayout extends Div implements RouterLayout
{
```

25.6.4. Servlet Configuration

If you are configuring your servlet manually, you should ensure the `async-supported` parameter is set.

You can enable server push and define the push mode for an entire application in the servlet configuration with the `pushmode` parameter for the servlet in the `web.xml` deployment descriptor or a corresponding `@WebServlet` annotation.

In addition to this, it is possible to configure the url to use for push requests by setting the `pushURL` parameter. This is useful for servers that require a predefined URL to push.

25.7. Asynchronous Updates

This section describes to use server push from your application code. See Server Push Configuration for an overall description on what server push means and how to configure your application to use server push.

Making changes to a UI from another thread and pushing them to the browser requires locking the user session. 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 as parameter a `Command` to execute while the session is locked.

For example:

```
ui.access(new Command() {
    @Override
    public void execute() {
        statusLabel.setText(statusText);
    }
});
```

You also use a simple lambda expression to define your access command.

```
ui.access(() -> statusLabel.setText(statusText));
```

If the push mode is `manual`, you need to push the pending UI changes to the browser explicitly with the `push()` method.

```
ui.access(() -> {
    statusLabel.setText(statusText);
    ui.push();
});
```

Below is a complete example of a case where we make UI changes from another thread.

```
@Push
@Route("push")
public class PushyView extends VerticalLayout {
    private FeederThread thread;

    @Override
    protected void onAttach(AttachEvent attachEvent) {
        add(new Span("Waiting for updates"));

        // Start the data feed thread
        thread = new FeederThread(attachEvent.getUI(),
this);
        thread.start();
    }

    @Override
    protected void onDetach(DetachEvent detachEvent) {
        // Cleanup
        thread.interrupt();
        thread = null;
    }

    private static class FeederThread extends Thread {
        private final UI ui;
        private final PushyView view;

        private int count = 0;

        public FeederThread(UI ui, PushyView view) {
            this.ui = ui;
            this.view = view;
        }

        @Override
        public void run() {
            try {
                // Update the data for a while
                while (count < 10) {
                    // Sleep to emulate background work
                    Thread.sleep(500);
                    String message = "This is update " +
count++;

                    ui.access(() -> view.add(new Span
(message)));
                }

                // Inform that we are done
                ui.access(() -> {
                    view.add(new Span("Done updating"));
                });
            } 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 in Creating Collaborative Views.

25.8. Creating Collaborative Views

This section describes to use server push to create a view where changes made by one user is immediately shown to all users. See Server Push Configuration for an overall description on what server push means and how to configure your application to use server push.

Broadcasting messages to be pushed to UIs in other user sessions requires some sort of message-passing mechanism that sends the messages to all UIs that are registered as recipients. As processing server requests for different UIs is done concurrently in different threads of the application server, locking the data structures properly is very important to avoid deadlock situations.

25.8.1. The Broadcaster

The standard pattern for sending messages to other users is to use a *broadcaster* singleton that registers recipients 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 a single thread is usually the easiest and safest way. The methods in the class are defined as `synchronized` to prevent race conditions.

```
public class Broadcaster {
    static Executor executor = Executors
.newSingleThreadExecutor();

    static LinkedList<Consumer<String>> listeners = new
LinkedList<>();

    public static synchronized Registration register(
            Consumer<String> listener) {
        listeners.add(listener);

        return () -> {
            synchronized (Broadcaster.class) {
                listeners.remove(listener);
            }
        };
    }

    public static synchronized void broadcast(String
message) {
        for (Consumer<String> listener : listeners) {
            executor.execute(() -> listener.accept
(message));
        }
    }
}
```

Receiving Broadcasts

The receivers need register a consumer to the broadcaster to receive the broadcasts. The registration should be removed when the component is no longer attached. When updating the UI in a receiver, it should be done safely by executing the update through the `access()` method of the UI, as described in Asynchronous Updates.

```
@Push
@Route("broadcaster")
public class BroadcasterView extends Div {
    VerticalLayout messages = new VerticalLayout();
    Registration broadcasterRegistration;

    // Creating the UI shown separately

    @Override
    protected void onAttach(AttachEvent attachEvent) {
        UI ui = attachEvent.getUI();
        broadcasterRegistration = Broadcaster.register
(newMessage -> {
            ui.access(() -> messages.add(new Span
(newMessage)));
        });
    }

    @Override
    protected void onDetach(DetachEvent detachEvent) {
        broadcasterRegistration.remove();
        broadcasterRegistration = null;
    }
}
```

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.

```
public BroadcasterView() {
    TextField message = new TextField();
    Button send = new Button("Send", e -> {
        Broadcaster.broadcast(message.getValue());
        message.setValue("");
    });

    HorizontalLayout sendBar = new HorizontalLayout
(message, send);

    add(sendBar, messages);
}
```

25.9. Modifying how dependencies are loaded with DependencyFilters

As seen on the tutorials about using `@JavaScript`, `@HtmlImport` and `@StyleSheet` (see Including Style Sheets[248] and Importing html/javascript[249]), you can use annotations or an imperative API to add resources (or dependencies) to your application when needed. But in some cases, a more fine control is needed: for example, when bundling resources into multiple different bundles, you may want to control the application to import the right bundle when some specific resource is requested.

To control how the dependencies are loaded, and which files are effectively added or removed from the loading process, you can use DependencyFilters.

Here is one example - it removes all dependencies and add one single bundle when running in production mode:

```
public class BundleFilter implements DependencyFilter {
    @Override
    public List<Dependency> filter(List<Dependency>
dependencies,
            FilterContext filterContext) {

        if (filterContext.getService()
.getDeploymentConfiguration()
                .isProductionMode()) {
            dependencies.clear();
            dependencies.add(new Dependency(Dependency
.Type.HTML_IMPORT,
                    "my-bundle.html", LoadMode.EAGER));
        }

        return dependencies;
    }
}
```

TIP You can also use the `frontend://` and `context://` protocols on dependencies returned by the DependencyFilter. These protocols are resolved after the filters are applied. The `context://` protocol is resolved to the servlet context root and the `frontend://` protocol is resolved to a `frontend` folder in the servlet context root.

The DependencyFilters are called in two particular situations: when a PolymerTemplate[250] is parsed for the first time, and when a set of dependencies are about to be sent to the client.

- When a Polymer template is parsed, all `@HtmlImport` of the class are analyzed and sent to the DependencyFilters for evaluation. The filter must return a dependency that contains the definition of the template, so it can be parsed. In the example provided above, the `my-bundle.html` file must contain the definition of the Polymer templates needed by the application.

- When a route changes and a new set of components are requested, all dependencies are gathered in a list and sent to the filters for evaluation. The filters can change, remove or add new dependencies as needed.

WARNING	DependencyFilters allow you to change, add and remove any dependencies. You may leave your application in a broken state if you remove a required dependency for your project without providing a suitable replacement. With great power comes great responsibility.

With your DependencyFilter in place, you need to add it to a `ServiceInitEvent` which is sent when a Vaadin service is initialized. Take a look on the ServiceInitListener tutorial on how to configure it.

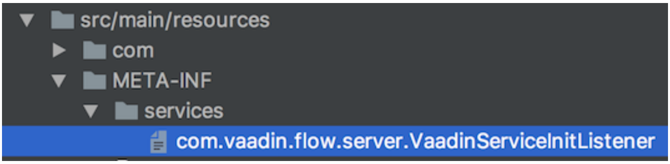
25.10. VaadinServiceInitListener

`VaadinServiceInitListener` can be used to configure RequestHandlers, BootstrapListeners and DependencyFilters. You can also use it to dynamically register routes during the application startup.

The listener gets a `ServiceInitEvent` which is sent when a Vaadin service is initialized.

```
public class ApplicationServiceInitListener
        implements VaadinServiceInitListener {

    @Override
    public void serviceInit(ServiceInitEvent event) {
        event.addBootstrapListener(response -> {
            // BoostrapListener to change the bootstrap
page
        });

        event.addDependencyFilter((dependencies,
filterContext) -> {
            // DependencyFilter to add/remove/change
dependencies sent to
            // the client
            return dependencies;
        });

        event.addRequestHandler((session, request,
response) -> {
            // RequestHandler to change how responses are
handled
            return false;
        });
    }

}
```

This listener should be registered as a provider via Java SPI loading facility. To do this you should create `META-INF/services` resource directory and a provider configuration file with the name `com.vaadin.flow.server.VaadinServiceInitListener`. This is a text file and it should contain the fully qualified name of the `ApplicationServiceInitListener` class on its own line. It allows to discover the `ApplicationServiceInitListener` class, instantiate it and register as a service init listener for the application.



The content of the file should be like this:

```
com.mycompany.ApplicationServiceInitListener
```

TIP See https://docs.oracle.com/javase/tutorial/ext/basics/spi.html#register-service-providers and https://docs.oracle.com/javase/7/docs/api/java/util/ServiceLoader.html for details about Java SPI loading.

25.11. Dynamic Content

There are two options to generate content dynamically based on data provided by the current application state:

- You can use a `StreamResource` which will handle URLs automatically.
- You can build a custom URL including parameters with `String` type parameters. In this case you will need one more servlet which handles the URL.

The first option is preferable since it doesn't require additional servlet and allows to use data with any type from the application state.

25.11.1. Using custom servlet and request parameters

You can create a custom servlet which handles "image" as a relative URL:

```
@WebServlet(urlPatterns = "/image", name =
"DynamicContentServlet")
public class DynamicContentServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req,
HttpServletResponse resp)
            throws ServletException, IOException {
        resp.setContentType("image/svg+xml");
        String name = req.getParameter("name");
        if (name == null) {
            name = "";
        }
        String svg = "<?xml version='1.0' encoding='UTF-
8' standalone='no'?>"
                + "<svg
xmlns='http://www.w3.org/2000/svg' "
                +
"xmlns:xlink='http://www.w3.org/1999/xlink'>"
                + "<rect x='10' y='10' height='100'
width='100' "
                + "style=' fill: #90C3D4'/><text x='30'
y='30' fill='red'>"
                + name + "</text>" + "</svg>";
        resp.getWriter().write(svg);
    }
}
```

The following code should be used in the application (which has its own servlet). It generates the resource URL on the fly based on the current application state. The property value of the input component is used here as a state:

```
Input name = new Input();

Element image = new Element("object");
image.setAttribute("type", "image/svg+xml");
image.getStyle().set("display", "block");

NativeButton button = new NativeButton("Generate Image");
button.addClickListener(event -> {
    String url = "image?name=" + name.getValue();
    image.setAttribute("data", url);
});

UI.getCurrent().getElement().appendChild(name.getElement(
), image,
    button.getElement());
```

Using `StreamResource`

Use `StreamResource` to generate dynamic content within the same servlet. In this case the application will generate the URL transparently for you and register an internal handler for this URL. The code below shows how to implement the same functionality as above using `StreamResource`.

```
Input name = new Input();

Element image = new Element("object");
image.setAttribute("type", "image/svg+xml");
image.getStyle().set("display", "block");

NativeButton button = new NativeButton("Generate Image");
button.addClickListener(event -> {
    StreamResource resource = new StreamResource(
"image.svg",
            () -> getImageInputStream(name));
    image.setAttribute("data", resource);
});

UI.getCurrent().getElement().appendChild(name.getElement(
), image,
    button.getElement());
```

The `data` attribute value is set to the StreamResource, which will automatically be converted into a URL. A `StreamResource` uses a dynamic data provider to produce the data. The file name given to a `StreamResource` is used as a part of the URL and will also become the filename if the user selects to download the resource. And here is an example how to create a data provider:

```
private InputStream getImageInputStream(Input name) {
    String value = name.getValue();
    if (value == null) {
        value = "";
    }
    String svg = "<?xml version='1.0' encoding='UTF-8'
standalone='no'?>"
        + "<svg  xmlns='http://www.w3.org/2000/svg' "
        + "xmlns:xlink='http://www.w3.org/1999/xlink'>"
        + "<rect x='10' y='10' height='100' width='100' "
        + "style=' fill: #90C3D4'/><text x='30' y='30'
fill='red'>"
        + value + "</text>" + "</svg>";
    return new ByteArrayInputStream(svg.getBytes
(StandardCharsets.UTF_8));
}
```

25.12. History API

The *History API* allows you to access the browser navigation history from the server-side. The history is always bound to the current browser window / frame, so you can access it through the *Page* object (available through the *UI*).

```
History history = UI.getCurrent().getPage().getHistory();
```

25.12.1. Traversing History

With the methods `forward()`, `back()` and `go(int)` you can programmatically traverse the browser's history entries. The methods correspond to the user actions on the browser's back and forward buttons.

```
history.back(); // navigates back to the previous entry

history.forward(); // navigates forward to the next entry

history.go(-2); // navigates back two entries
history.go(1); // equal to history.forward();
history.go(0); // will reload the current page
```

NOTE

Triggering the `forward`, `back` and `go` methods will asynchronously trigger a *HistoryStateChangeEvent* if the history entries are for the same document, e.g. the entries share the same origin[251].

25.12.2. Handling user navigation

If you want to manually handle navigation events you can replace it by setting a handler for navigation events using the `history.setHistoryStateChangeHandler(HistoryStateChangeHandler)`. It will be notified when:

- the user navigates back or forward using the browser buttons
- the navigation was done programmatically from server-side java code or client-side JavaScript
- the user clicks a link marked with the *router-link* attribute

```
history.setHistoryStateChangeHandler(this::onHistoryState
Change);

private void onHistoryStateChange(HistoryStateChangeEvent
event) {
    // site base url is www.abc.com/
    // user navigates back from abc.com/dashboard to
abc.com/home
    event.getLocation().getPath(); // returns "home"
}
```

NOTE The server side history state change event is not fired if only the hash[252] has changed. Hash is always stripped from the location sent to server. Hash is a browser feature not intended for use on the server side.

25.12.3. Changing history

You can update the history by either pushing new entries to the history, or by replacing the current entry. You may optionally provide a json value as the *state* parameter. This state value will be available via `LocationChangeEvent:getState()` when the entry is being revisited the next time.

```
// adds a new history entry for location "home", no state
history.pushState(null, "home");

// replaces the current entry with location "about" and a
state object
JsonValue state = Json.create("preview-mode");
history.replaceState(state, "about");
```

NOTE The *url* used with `pushState` and `replaceState` must be for the same origin[253] as the current *url*; otherwise browser will throw an exception and the history is not updated.

NOTE If you use either `pushState` or `replaceState`, the framework internal scroll position restoration on navigation won't work, since it stores data in the `history.state` to keep track of the scroll positions.

25.13. StreamReceiver for receiving incoming data stream

To receive upload from the client we need to register a `StreamReceiver` that will get a URL that will handle receiving of an upload stream.

For creating a StreamReceiver we first need to create a `StreamVariable` that handles terminal Upload monitors and controls the upload during the time it is being streamed.

Then the stream can be registered through the Element API.

```
StreamReceiver streamReceiver = new StreamReceiver(
    getElement().getNode(), "upload", getStreamVariable(
));
getElement().setAttribute("target", streamReceiver);
```

25.14. UIInitListener

A UIInitListener can be used to receive an event each time a new UI has been created and initialized.

The ideal place to add UIInitListeners would be inside a VaadinServiceInitListener

```
public class ServiceListener implements
VaadinServiceInitListener {

    @Override
    public void serviceInit(ServiceInitEvent event) {
        event.getSource().addUIInitListener(
                initEvent -> LoggerFactory.getLogger
(getClass())
                        .info("A new UI has been
initialized!"));
    }
}
```

25.15. Making a component add-on OSGi-compatible

In order to use a component jar as an OSGi bundle, the manifest file needs to have additional headers. The headers describe the bundle and provide additional information. Some of the headers are as follows.

- **Bundle-SymbolicName** is the only mandatory header. It specifies a unique identifier for the bundle, based on the reverse domain name convention. e.g. `com.vaadin.flow.component.button`
- **Bundle-Name** defines a human-readable name. e.g. vaadin-button-flow
- **Bundle-License** specifies the license information of bundle. e.g. http://www.apache.org/licenses/LICENSE-2.0
- **Bundle-ManifestVersion** indicates the OSGi specification to use for reading this bundle. The value 1 indicates OSGi release 3, and the value 2 indicates OSGi release 4 and later.
- **Bundle-Version** specifies the version of this bundle which

consists of up to four parts separated with dots.

- **Import-Package** declares the imported packages for this bundle.
- **Export-Package** contains a declaration of exported packages.
- **Require-Capability** specifies that this bundle requires other bundles to provide a capability e.g. osgi.ee;filter:="(&(osgi.ee=JavaSE)(version=1.8))"

After generating the MANIFEST.MF file, manually or using any tool, it should be added to the output jar file. This job can be done by configuring `maven-jar-plugin` like this:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <configuration>
        <archive>

<manifestFile>${project.build.outputDirectory}/META-
INF/MANIFEST.MF</manifestFile>
        </archive>
    </configuration>
</plugin>
```

NOTE Static resources do not work in OSGi environment for now and it is required to unpack them in the application project.

25.15.1. Tools

Although the headers can be added to MANIFEST.MF manually, it is recommended to use an automated tool to create them. Here two tools, both maven plugins, are briefly introduced.

Bnd maven plugin

This plugin generates required manifest entries based on specified instructions which are declared in either a file (with default name of bnd.bnd) or the plugin `<configuration>` in the pom. The plugin also set default values to some headers derived from pom elements. For example, `Bundle-SymbolicName` is set to `artifactId`, and `Bundle-Version` is deducted from artifact version. The following is an example of the usage of the plugin.

```
<plugin>
    <groupId>biz.aQute.bnd</groupId>
    <artifactId>bnd-maven-plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>bnd-process</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <bnd><![CDATA[
             -exportcontents:
org.example.api,org.example.types
             -sources: true
             Private-Package: org.example.internal.*
             Bundle-Activator:
org.foo.myproject.impl.Activator
             ]]>
        </bnd>
    </configuration>
</plugin>
```

In this example, the instructions are provided using `<![CDATA[ ]]>` section in `bnd` parameter. In addition to the instructions, that start with a minus sign ('-'), manifest headers (e.g. `Private-Package`) can also be added here. For more information about instructions see Bnd Instruction Reference.

For more information about this plugin see bnd-maven-plugin documentation on GitHub.

Apache Felix Maven Bundle Plugin

This plugin is based on Bnd tool with this change that you can provide headers and instructions as nested tags in `<configuration>` section. So, it may be a better choice when we want to have instructions in the `pom` file. Although the values of the required entries in the manifest file can be set manually, this plugin generates reasonable default values for various headers. Here is an example on how to use the plugin.

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <executions>
        <execution>
            <id>bundle-manifest</id>
            <phase>process-classes</phase>
            <goals>
                <goal>manifest</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <instructions>

<_exportcontents>org.example.api,org.example.types</_expo
rtcontents>
            <_sources>true</_sources>
            <Private-Package>
org.example.internal.*</Private-Package>
            <Bundle-
Activator>org.foo.myproject.impl.Activator</Bundle-
Activator>
        </instructions>
    </configuration>
</plugin>
```

With these instructions, the plugin generates for instance `Bundle-Version` deducted from `${pom.version}`, `Bundle-Name` set to `${pom.name}` and `Bundle-SymbolicName` computed from `groupId` and `artifactId`. For more information see Apache Felix Maven Bundle Plugin documentation.

25.16. All Vaadin properties

There are a number of properties that you can use to configure your Flow application.

This document summarizes the properties that are defined in `com.vaadin.server.DeploymentConfiguration` and `com.vaadin.server.Constants` classes.

These properties can be set from the command line as a system property, as a Servlet init parameter in the `web.xml` or using the `@WebServlet` annotation. For further information, please consult the tutorial Changing Flow behavior with runtime configuration.

NOTE If you use Spring Boot then you should add the prefix `vaadin.`, e.g. vaadin.productionMode=true.

Table 5. Vaadin properties

Property	Description
productionMode	Turns application to work in production mode. Production mode disables most of the logged information that appears on the console because logging and other debug features can have a significant performance impact. In addition to this, development mode JS functions are not exported, `push` is given as a minified JS file instead of full size and static resources are cached.
requestTiming	If request timing info should be made available. If true, in each response the server includes some basic timing information. This can be used for performance testing.
disable-xsrf-protection	Cross-site request forgery protection. This protection is enabled by default, but it might need to be disabled to allow a certain type of testing. For these cases, the check can be disabled by setting the init parameter.

Property	Description
heartbeatInterval	Vaadin UIs that are open on the client side send a regular heartbeat to the server to indicate they are still alive, even though there is no ongoing user interaction. When the server does not receive a valid heartbeat for a given UI, it will eventually remove that UI from the session. Vaadin UIs that are open on the client side send a regular heartbeat to the server to indicate they are still alive, even though there is no ongoing user interaction. When the server does not receive a valid heartbeat for a given UI, it will eventually remove that UI from the session.
closeIdleSessions	When it is set to true (the default is false), the session will be closed if no UI is active. Heartbeat requests are just like any other request from the servlet container’s viewpoint. This means that as long as there is an open UI, the session never expires even though there is no user interaction. You can control this behavior by setting an init parameter named closeIdleSessions to true.
pushMode	The permitted values are "disabled" or "manual". Please consult Server Push Configuration documentation.
pushURL	It is the url to use for push requests. Some servers require a predefined URL to push. Please consult Server Push Configuration documentation.

Property	Description
syncIdCheck	Returns whether sync id checking is enabled. The sync id is used to gracefully handle situations when the client sends a message to a connector that has recently been removed on the server. By default, it is true.
sendUrlsAsParameters	Returns whether the sending of URL’s as GET and POST parameters in requests with content-type <code>application/x-www-form-urlencoded</code>is enabled or not.
pushLongPollingSuspendTimeout	When using the long polling transport strategy, it specifies how long it accepts responses after each network request. Number of milliseconds.
load.es5.adapters	Include polyfills for browsers that do not support ES6 to their initial page. In order for web components to work, extra libraries (polyfills) are required to be loaded, can be turned off if different versions or libraries should be included instead.
frontend.url.es5	A location Flow searches web components' files in production mode when the request is coming from older browsers, not supporting es6, default web components' development language version.
frontend.url.es6	A location Flow searches web components' files in production mode for requests from modern browsers.
disable.webjars	Configuration name for the parameter that determines if Flow should use webJars or not. If set to true, webjars would be ignored during request resolving, allowing Flow to use an external source of web components' files.

Property	Description
original.frontend.resources	Configuration name for the parameter that determines if Flow should use bundled fragments or not.
i18n.provider	I18N provider property. To use localization and translation strings the application only needs to implement `I18NProvider` and define the fully qualified class name in the property `i18n.provider`. Please consult I18N localization documentation.
disable.automatic.servlet.registration	Configuration name for the parameter that determines if Flow should automatically register servlets needed for the application to work.

Property	Description
compatibilityMode	When set to `true`, enables Vaadin 13 compatibility mode (Vaadin uses Bower and Webjars to handle frontend resources instead of npm and webpack). The purpose of this mode is to ease migration and it will no longer be supported in Vaadin 15. See Compatibility mode in Vaadin 14 Migration Guide[254] for more information. :leveloffset: 1 = Vaadin Bakery App Starter :leveloffset: 2 :imagesdir: ../../../target/unzip/bakery-starter = Bakery App Starter for Flow and Spring Bakery is an App Starter to give you a head start building your business application based on Vaadin 10+ with Flow and Spring. It includes an end-to-end technology stack covering each layer that is needed to build a production grade application. The App Starter is opinionated and reflects Vaadin's view on what is the best way to build business applications. See a live demo of the application.[255] image::img/overview.png[Bakery on desktop,align=center] == Features === Full stack architecture Bakery

Property	Description
===	Property
Values	Example
FontFamily	MONOSPACE
component.addClassName(LumoStyles.FontFamily.MONOSPACE);	FontSize
XXS, XS, S, M (default), L, XL, XXL, XXXL	UIUtils.setFontSize(FontSize.XL, component);
FontWeight	BOLD, BOLDER, LIGHTER, NORMAL, _100, _200, _300, _400, _500, _600, _700, _800, _900
UIUtils.setFontWeight(FontWeight.BOLD, component);	Headings
H1, H2, H3, H4, H5, H6	component.addClassName(LumoStyles.Heading.H1);
IconSize	S, M, L
UIUtils.createSmallIcon(VaadinIcon.HOME);, UIUtils.createLargeIcon(VaadinIcon.HOME);	TextColor

Property	Description
HEADER, BODY, SECONDARY, TERTIARY, DISABLED, PRIMARY, PRIMARY_CONTRAST, ERROR, ERROR_CONTRAST, SUCCESS, SUCCESS_CONTRAST	UIUtils.setTextColor(TextColor.SUCCESS, component);
=== == Colors https://cdn.vaadin.com/vaadin-lumo-styles/1.4.2/demo/colors.html	===
Color	Values
Base	BASE_COLOR
Primary	_10, _50, _100
Error	_10, _50, _100
Success	_10, _50, _100
Tint	_5, _10, _20, _30, _40, _50, _60, _70, _80, _90, _100
Shade	_5, _10, _20, _30, _40, _50, _60, _70, _80, _90, _100
Contrast	_5, _10, _20, _30, _40, _50, _60, _70, _80, _90, _100

Property	Description
=== [source,java] ---- UIUtils.setBackgroundColor(LumoStyles.Color.Contrast._20, component); ---- == Styles https://cdn.vaadin.com/vaadin-lumo-styles/1.4.2/demo/styles.html	===
Color	Values
Example	`BorderRadius`
`S`, `M`, `L`, `_50`	`UIUtils.setBorderRadius(BorderRadius.L, component);`
`BoxShadowBorders`	`BOTTOM`, `LEFT`, `RIGHT`, `TOP`
`component.addClassName(BoxShadowBorders.BOTTOM);`	`Shadow`
`S`, `M`, `L`, `XL`	`UIUtils.setShadow(Shadow.L, component);`

Property	Description
=== == Sizing and Spacing https://cdn.vaadin.com/vaadin-lumo-styles/1.4.2/demo/sizing-and-spacing.html	===
Property	Size
Direction	`Margin`
`XS`, `S`, `M` (default), `L`, `XL`	`BOTTOM`, `LEFT`, `RIGHT`, `TOP`, `HORIZONTAL`, `VERTICAL`, `TALL`, `UNIFORM` (default), `WIDE`
`Padding`	`XS`, `S`, `M` (default), `L`, `XL`
`BOTTOM`, `LEFT`, `RIGHT`, `TOP`, `HORIZONTAL`, `VERTICAL`, `TALL`, `UNIFORM` (default), `WIDE`	`Spacing`
`XS`, `S`, `M` (default), `L`, `XL`	`BOTTOM`, `LEFT`, `RIGHT`, `TOP`, `HORIZONTAL`, `VERTICAL`, `TALL`, `UNIFORM` (default), `WIDE`

Property	Description
=== [source,java] ---- component.addClassNames(LumoStyles.Margin.Left.S , LumoStyles.Padding.Vertical.XL, LumoStyles.Spacing.Bottom.M); flexBoxLayout.setMargin(Left.S); flexBoxLayout.setPadding(Vertical.XL); flexBoxLayout.setSpacing(Bottom.S); ---- == Utility Classes A number of utility classes, most importantly `UIUtils`,	===

Property	Description
Variant	UIUtils method
Primary	`createPrimaryButton`
Tertiary	`createTertiaryButton`, `createTertiaryInlineButton`
Success	`createSuccessButton`, `createSuccessPrimaryButton`
Error	`createErrorButton`, `createErrorPrimaryButton`
Contrast	`createContrastButton`, `createContrastPrimaryButton`
Size	`createSmallButton`, `createLargeButton`

Property	Description
===	===
Combinations can be created with `createButton(String, ButtonVariant…)`, `createButton(VaadinIcon, ButtonVariant…)` and `createButton(String, VaadinIcon, ButtonVariant…)`.	
[source,java]	

UIUtils.createPrimaryButton("Primary");	
UIUtils.createSuccessButton(VaadinIcon.CHECK);	
UIUtils.createErrorButton("Error", VaadinIcon.	

Property	Description
Type	UIUtils method
Color	`createLabel(TextColor, String)`
Size	`createLabel(FontSize, String)`
Size & color	`createLabel(FontSize, TextColor, String)`
Heading	`createH1Label(String)`, `createH2Label(String)`, `createH3Label(String)`, `createH4Label(String)`, `createH5Label(String)`, `createH6Label(String)`
=== === Icons	===
Variant	UIUtils method
Primary	`createPrimaryIcon(VaadinIcon)`
Secondary	`createSecondaryIcon(VaadinIcon)`
Tertiary	`createTertiaryIcon(VaadinIcon)`
Disabled	`createDisabledIcon(VaadinIcon)`
Success	`createSuccessIcon(VaadinIcon)`
Error	`createErrorIcon(VaadinIcon)`
Small	`createSmallIcon(VaadinIcon)`
Large	`createLargeIcon(VaadinIcon)`

Property	Description
=== Combinations can be created with `createIcon(IconSize, TextColor, VaadinIcon)`. === Numbers	===
UIUtils method	Description
`formatAmount(Double)`	Formats a decimal amount for improved legibility.
`createAmountLabel(Double)`	Initialises a monospaced H5 label for improved legibility of decimal values.
`formatUnits(Integer)`	Formats an integer amount for improved legibility.
`createUnitsLabel(Integer)`	Initialises a monospaced H5 label for improved legibility of integer values.
=== === Dates	===
UIUtils method	Description
`formatDate(LocalDate)`	Formats a `LocalDate` according to the format defined in `UIUtils`.

Property	Description
=== === Misc	===
UIUtils method	Description
`setColSpan(Integer, Components… )`	Sets the column span for components in a `FormLayout`.
`createFloatingActionButton(VaadinIcon)`	Initialises a `Button` positioned in the bottom right corner of its container. Used for primary actions.
`createInitials(String)`	Creates an avatar with the given initials.

----

[240] https://github.com/vaadin/flow-spring-tutorial/blob/master/src/main/java/org/vaadin/spring/tutorial/SimpleI18NProvider.java
[241] https://vaadin.com/docs/flow/importing-dependencies/tutorial-include-css.html
[242] https://vaadin.com/docs/flow/importing-dependencies/tutorial-importing.html
[243] http://ogp.me/
[244] https://vaadin.com/docs/flow/advanced/tutorial-service-init-listener.html
[245] https://github.com/netgloo/spring-boot-samples/blob/master/spring-boot-mysql-springdatajpa-hibernate/src/main/resources/application.properties
[246] http://es6-features.org/
[247] https://github.com/Atmosphere/atmosphere
[248] https://vaadin.com/docs/flow/importing-dependencies/tutorial-include-css.html
[249] https://vaadin.com/docs/flow/importing-dependencies/tutorial-importing.html
[250] https://vaadin.com/docs/flow/polymer-templates/tutorial-template-basic.html

[251] https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
[252] https://developer.mozilla.org/en-US/docs/Web/Events/hashchange
[253] https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
[254] https://vaadin.com/docs/flow/v14-migration/v14-migration-guide.html#compatibility-mode
[255] https://bakery-flow.demo.vaadin.com/
[256] https://vaadin.com/start
[257] https://vaadin.com/license/cvtl-1
[258] https://github.com/mozilla/geckodriver/releases
[259] https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-devtools.html#using-boot-devtools-customizing-classload
[260] http://hotswapagent.org/mydoc_quickstart.html
[261] http://hotswapagent.org/mydoc_setup_intellij_idea.html#other-way-its-explicit-agent-configuration-without-plugin
[262] http://hotswapagent.org/mydoc_setup_eclipse.html
[263] http://hotswapagent.org/mydoc_setup_intellij_idea.html#start-with-hotswapagent-plugin-for-intellij-idea
[264] https://vaadin.com/license/cvtl-1.0
[265] https://vaadin.com/elements/browse#charts
[266] https://vaadin.com/elements/vaadin-board
[267] https://vaadin.com/testbench
[268] https://vaadin.com/pro/validate-license
[269] https://vaadin.com/pro/validate-license
[270] https://vaadin.com/designer
[271] http://localhost:8080/about
[272] https://www.polymer-project.org/3.0/docs/devguide/style-shadow-dom
[273] https://cdn-origin.vaadin.com/vaadin-lumo-styles/1.0.0/demo/
[274] https://cdn-origin.vaadin.com/vaadin-lumo-styles/1.4.0/demo/
[275] https://vaadin.com/docs/flow/flow/polymer-templates/tutorial-template-basic.html
[276] https://www.polymer-project.org/3.0/docs/devguide/style-shadow-dom
[277] https://cdn-origin.vaadin.com/vaadin-lumo-styles/1.0.0/demo/
[278] https://cdn-origin.vaadin.com/vaadin-lumo-styles/1.0.0/demo/colors.html#dark-palette
[279] https://css-tricks.com/favicon-quiz
[280] https://docs.spring.io/spring-boot/docs/current/reference/html/

boot-features-sql.html
[281] https://aws.amazon.com/free
[282] http://aws.amazon.com/documentation/elasticbeanstalk
[283] https://aws.amazon.com/answers/web-applications/aws-web-app-deployment-java/
[284] https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create_deploy_Java.html
[285] https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/customdomains.html
[286] https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/configuring-https.html
[287] https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.managing.db.html
[288] https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3.html
[289] https://developers.google.com/web/progressive-web-apps/
[290] https://developers.google.com/web/fundamentals/primers/service-workers/
[291] https://developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.webmanifest
[292] https://vaadin.com/progressive-web-applications/learn/how-are-pwa-different-than-normal-web-apps
[293] https://vaadin.com/blog/progressive-web-apps-in-java
[294] https://developers.google.com/web/ilt/pwa/introduction-to-progressive-web-app-architectures
[295] https://cdn.vaadin.com/vaadin-lumo-styles/1.4.2/demo/badges.html
[296] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout
[297] https://vaadin.com/docs/v13/flow/creating-components/tutorial-component-container.html
[298] https://vaadin.com/docs/v13/business-app/simple-viewframe-example.html
[299] https://vaadin.com/docs/v13/flow/routing/tutorial-routing-annotation.html
[300] https://vaadin.com/docs/v13/business-app/theming.html
[301] https://vaadin.com/docs/v13/business-app/overview.html
[302] https://vaadin.com/themes/lumo
[303] https://github.com/vaadin/vaadin-themable-mixin/wiki