Building Modern Web Apps with Spring Boot and Vaadin

A practical introduction to web application development using Java.

Table of Contents

Introduction: Building modern web apps with Spring Boot and Vaadin

This guide is a **practical** introduction to web application development with Spring Boot and Vaadin.

It covers the entire development process, from setup to deployment, following a step-by-step approach. You can replicate each section at your own pace as you follow along.

The content is suitable for anyone familiar with Java who wants to build a web application. To make sure your development experience is enjoyable and productive, we start right from the beginning with setting up your development environment.

TIP	You can skip setting up a Java development environment and go ahead to chapter 4 if you prefer to code using an online IDE.

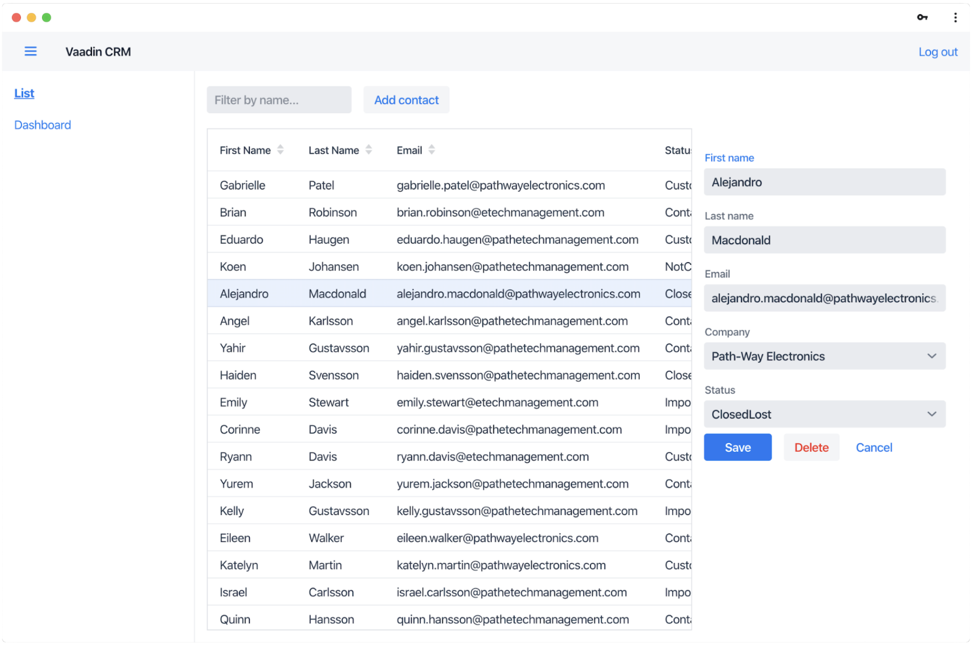
What you'll learn

This guide teaches you how to build a functional, full-stack web app using modern Java. It focuses on real-world developer needs, without diving deeply into theory or academics. Links to relevant further reading are provided for those who are interested.

The application is a customer relationship management (CRM) system for managing contacts. It features:

- A login screen to restrict access.
- A responsive layout with side navigation that works on desktop and mobile.
- A database for persistent data storage.
- A list view that can be sorted and filtered.
- A form to edit and add contacts.
- Data import from a REST API.
- A dashboard view.
- Cloud deployment.
- App installation on mobile and desktop.

Try the completed app here.



First Name	Last Name	Email	Statu
Gabrielle	Patel	gabrielle.patel@pathwayelectronics.com	Custo
Brian	Robinson	brian.robinson@etechmanagement.com	Conta
Eduardo	Haugen	eduardo.haugen@pathetechmanagement.com	Custo
Koen	Johansen	koen.johansen@pathetechmanagement.com	NotC
Alejandro	Macdonald	alejandro.macdonald@pathwayelectronics.com	Close
Angel	Karlsson	angel.karlsson@pathetechmanagement.com	Conta
Yahir	Gustavsson	yahir.gustavsson@pathetechmanagement.com	Conta
Haiden	Svensson	haiden.svensson@pathetechmanagement.com	Close
Emily	Stewart	emily.stewart@etechmanagement.com	Impo
Corinne	Davis	corinne.davis@pathetechmanagement.com	Impo
Ryann	Davis	ryann.davis@etechmanagement.com	Custo
Yurem	Jackson	yurem.jackson@pathetechmanagement.com	Conta
Kelly	Gustavsson	kelly.gustavsson@pathetechmanagement.com	Impo
Eileen	Walker	eileen.walker@pathwayelectronics.com	Conta
Katelyn	Martin	katelyn.martin@pathwayelectronics.com	Custo
Israel	Carlsson	israel.carlsson@pathetechmanagement.com	Impo
Quinn	Hansson	quinn.hansson@pathetechmanagement.com	Conta

First name: Alejandro
Last name: Macdonald
Email: alejandro.macdonald@pathwayelectronics
Company: Path-Way Electronics
Status: ClosedLost

Tools and frameworks

The frameworks and tools used in the guide were chosen for two reasons: they are easy to use, and they are suitable for both learning and production use.

On the **back end**, the application uses **Spring Boot**. This eliminates most of the hassle of setting up and running a Spring-based app and lets you focus on your own code. The main features you'll use are:

- Dependency injection to decouple components.
- Spring Data JPA repositories to work with the database.
- Spring Security to handle access control.
- An embedded Tomcat server to serve the application.
- Spring Boot Developer Tools to provide a smoother development experience.

Don't worry if you don't know what all of these are, we cover each individually as we go.

On the **front end**, the application uses **Vaadin** This is an open-source Java web app framework that comes with:

- A large library of UI components. Each component has a Java API and you can customize the look and feel.
- A router for navigating between views.
- A powerful data-binding system for forms and lists.

Why use Vaadin instead of Spring MVC and Thymeleaf, JSP or Freemarker?

Vaadin is an alternative to using Spring MVC and Thymeleaf, JSP or Freemarker when building web applications with Spring Boot.

The main advantage of Vaadin are:

- Vaadin is designed for building interactive single-page apps (SPA). Spring MVC and templates are better suited for more static content.
- Vaadin offers a Java component API, allowing you to build the entire application in Java.
- Vaadin comes with a large library of customizable UI components.
- Vaadin handles the communication between the server and the browser automatically, you don't need to write any JavaScript to update content dynamically.

Why use Vaadin instead of REST and React or Angular?

Combining a Spring Boot based REST backend with a frontend framework like React or Angular is a popular way of building SPAs. Vaadin allows you to build similar app experiences with less effort.

Advantages of using Vaadin:

- Faster development: you do not need to develop the backend and frontend separately.
- The entire application can be written in typesafe Java.
- Vaadin comes with a large library of customizable UI components.
- Vaadin handles the communication between the server and the browser automatically, you don't need to write any JavaScript to update content dynamically.
- More secure: the Vaadin app runs on the server and doesn't expose application code or extra endpoints to the browser.

Source code

You can find the full source code for this guide on GitHub. Each chapter is in a separate branch. In addition, the individual chapters include links to download the code both before and after the changes in the chapter, so you can easily get your project working again, if something does go wrong. Each chapter has a link to the source code from the previous chapter so you can start the tutorial at any chapter.

Video tutorials

You can find video tutorials of each chapter on the Vaadin YouTube Channel.

Setting up a Java development environment

Before you can start developing your Java app, you need to install the necessary development tools and set up your development environment.

TIP You can skip to chapter 4 if you prefer to code using the online IDE.

For web app development in Java, you need a:

- **Java Development Kit (JDK)**: This is the foundation of your Java development environment. It contains the various tools required to develop a working Java application. Chief among them are the Java Runtime Environment (JRE), which allows you to run a Java application and the Java compiler.
- **Build/project management tool (Maven)**: We will use this to import our project into the IDE and install any dependencies the project needs to function.
- **Version control system (Git)**: Git lets you track and manage your web application's source code. For example, a correctly set-up Git repo would allow you to easily undo any app breaking mistakes.
- **Front-end build tool (Node.js)**: Node.js is a JavaScript run-time environment (much like Java's JRE). Vaadin uses Node internally to build an optimized version of your app.
- **Integrated development environment (IDE)**: The IDE is the tool you use to write and run your code during development.

You can complete this guide using any IDE, but for the sake of simplicity, we use IntelliJ Idea throughout. The IntelliJ Community Edition is free to use and a great choice if you don't already have an IDE setup.

If you have the necessary tools installed on your computer, or if you prefer to code using the online IDE below, you can skip ahead to the Vaadin basics chapter.

Open in online IDE

Which Java version should you install?

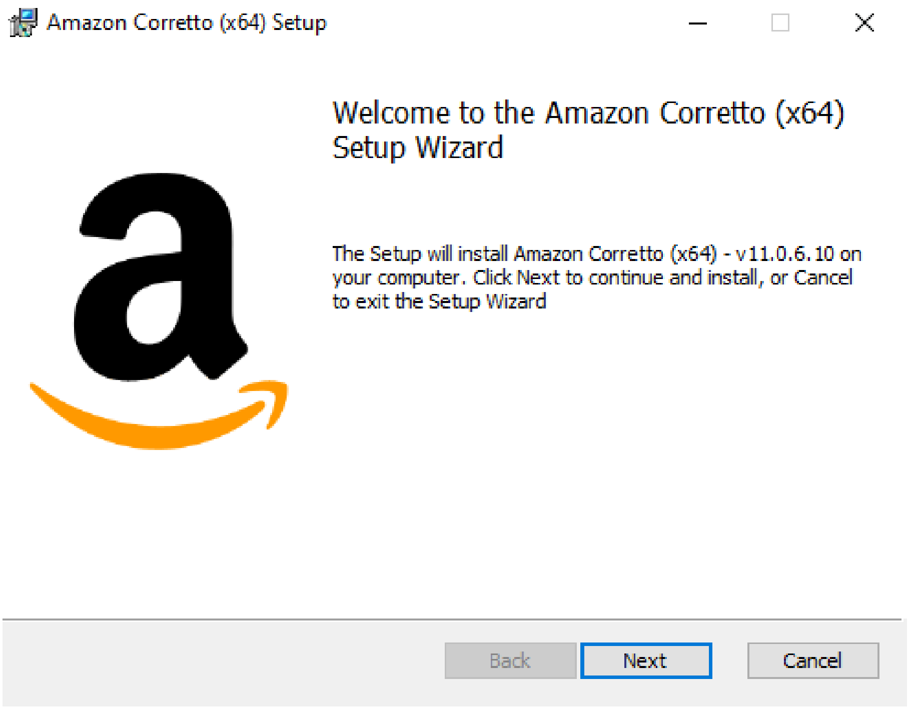
If you're new to Java or haven't used it in a while, you may be surprised by the number of Java runtimes available. A while ago, Oracle changed their release model and made the official Oracle JDK a paid product for commercial projects. Instead of one major release every few years, they now release new major versions every 6 months, and designate a long-term support (LTS) version every 3 years. **We recommend you use the current LTS version, Java SE 11**, if you prefer to use Oracle.

When the Oracle JDK became a commercial product, many developers chose to switch to alternative JDKs. There are many available drop-in replacements that are free to use and come with long-term-support releases. **We recommend you use either Amazon Corretto or OpenJDK**.

Setting up on Windows

To set up your development environment on Windows:

1. Install Java:
   a. Go to the Amazon Corretto 11 download page.
   b. Download and run the Windows installer (`.msi`).
   c. Follow the prompts in the wizard.



Welcome to the Amazon Corretto (x64) Setup Wizard

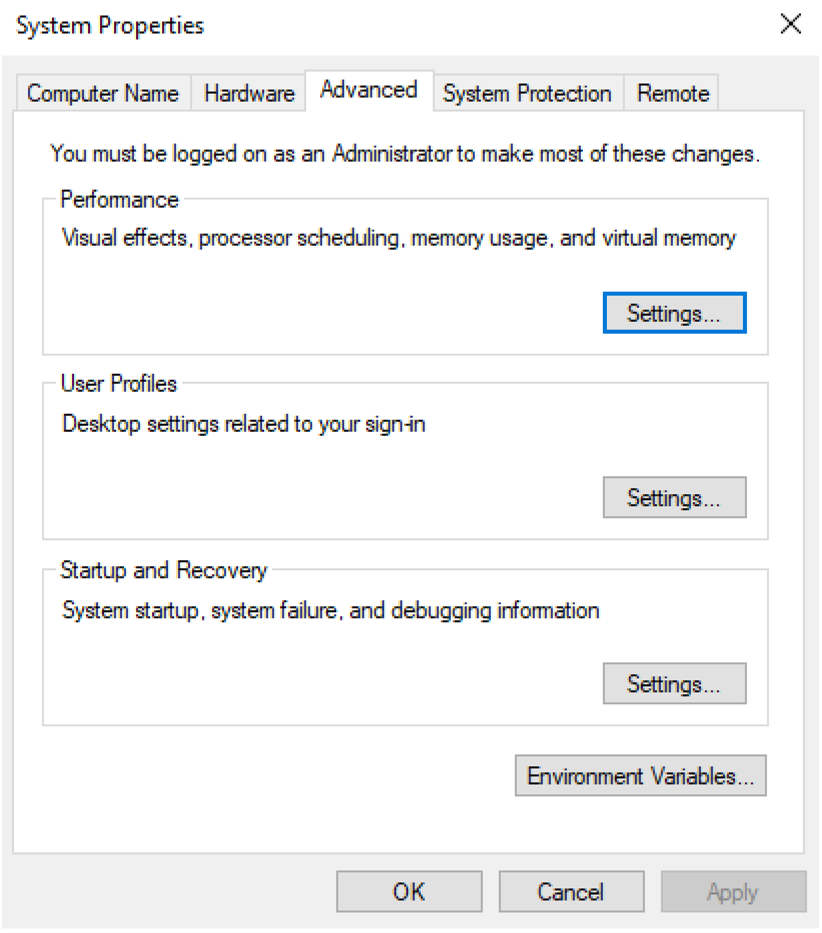
The Setup will install Amazon Corretto (x64) - v11.0.6.10 on your computer. Click Next to continue and install, or Cancel to exit the Setup Wizard

2. Install Maven:
   a. Go to the Maven download page.
   b. Download the Binary zip archive in the Files section.

c. Extract the archive to `C:\Program Files`.

d. In Windows, go to **Control Panel > Systems and Security > System > Advanced system settings**.

e. Select Environment Variables.



System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

Settings...

User Profiles

Desktop settings related to your sign-in

Settings...

Startup and Recovery

System startup, system failure, and debugging information

Settings...

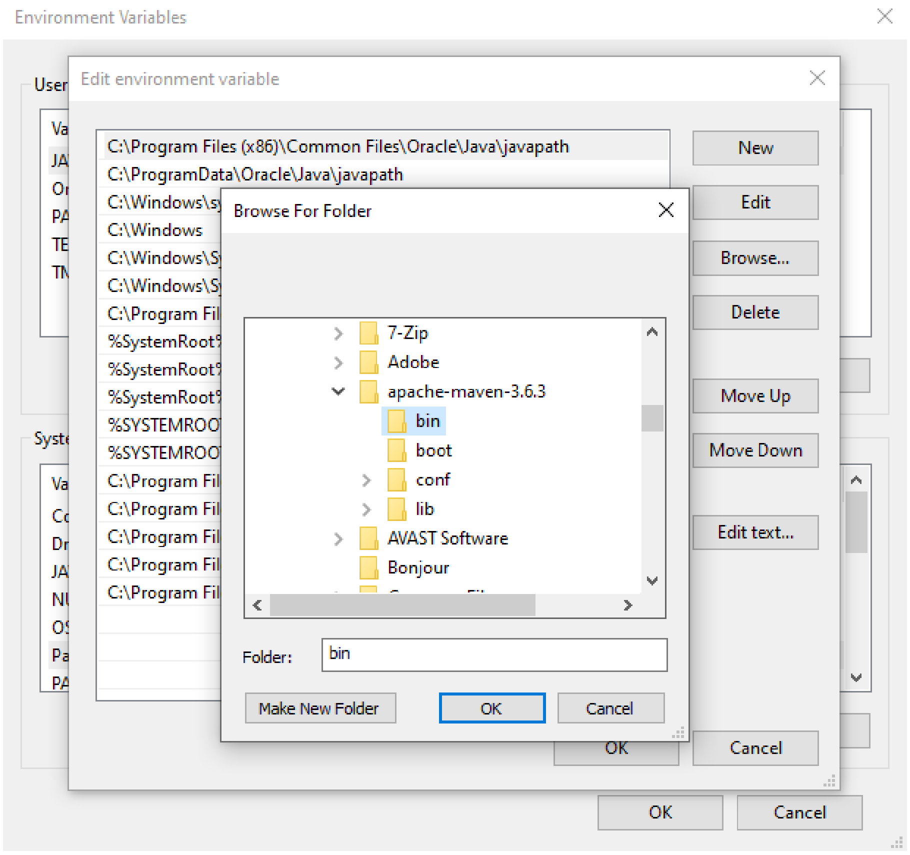
Environment Variables...

OK | Cancel | Apply

f. In Environment Variables, select the Path systems variable (in the bottom box) and then select Edit.

g. Select Browse in the edit dialog.

h. Navigate to and select `C:\Program Files\apache-maven-3.6.3\bin` (substitute the version number you downloaded) and then select OK.


C:\Program Files (x86)\Common Files\Oracle\Java\javapath
C:\ProgramData\Oracle\Java\javapath

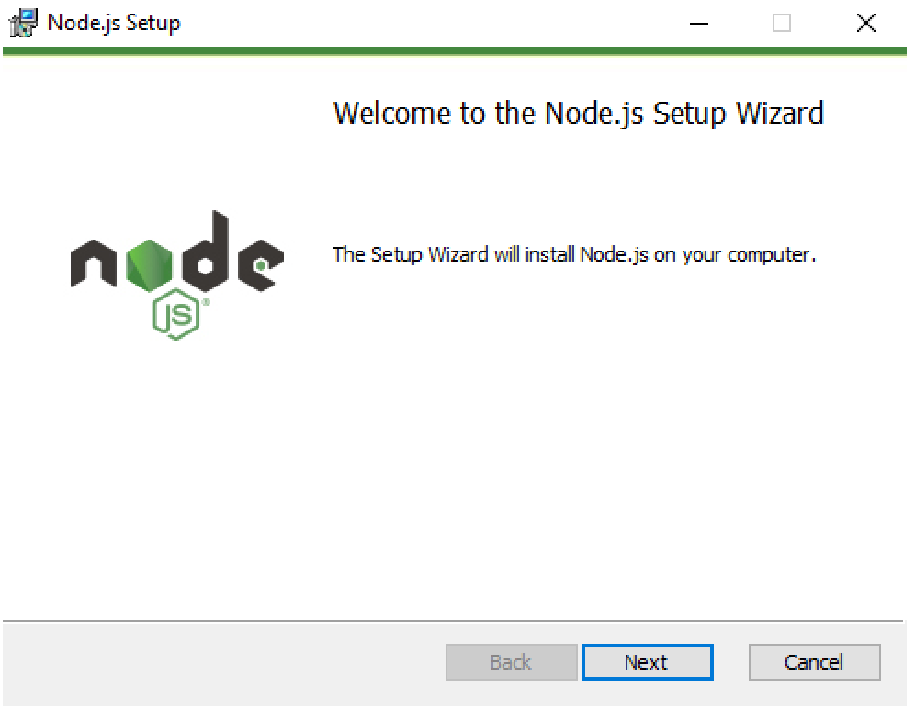
i. Select OK in all open dialogs to close them and save the environment variable.

3. Install Node:

   a. Go to the Node.js download page.

   b. Download and run the Windows Installer (`.msi`) for your system.

   c. Follow the prompts in the wizard.



Welcome to the Node.js Setup Wizard

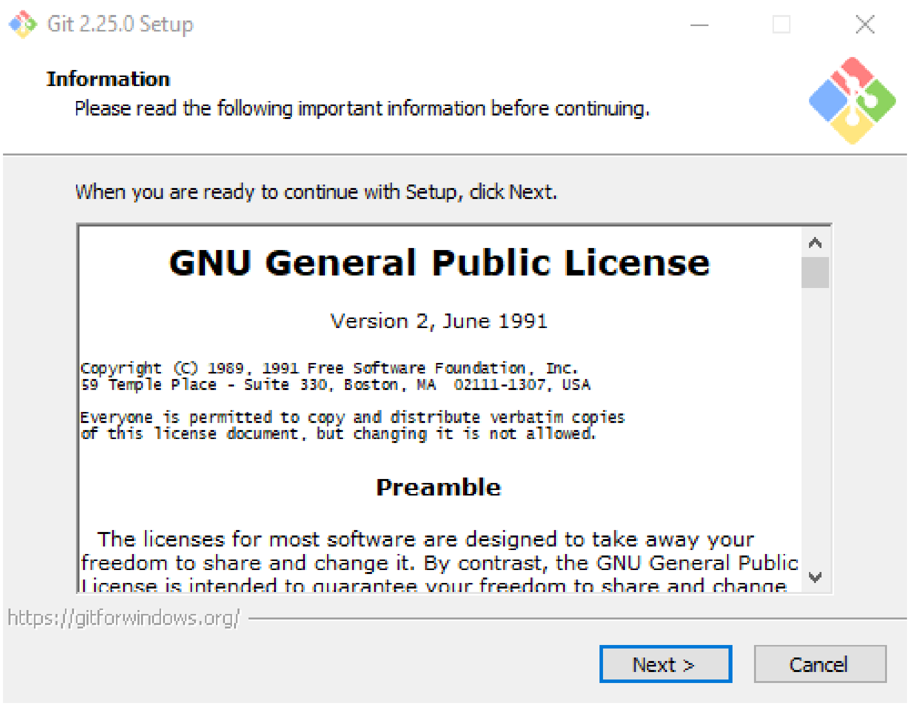
The Setup Wizard will install Node.js on your computer.

4. Install Git:

   a. Go to the Git download page.

   b. Download and run the Windows installer (`.exe`) for your system.

   c. Follow the prompts in the wizard. If you are unsure about any option, use the defaults.


Git 2.25.0 Setup

Information

Please read the following important information before continuing.

When you are ready to continue with Setup, click Next.

GNU General Public License

Version 2, June 1991

```
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place - Suite 330, Boston, MA  02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change

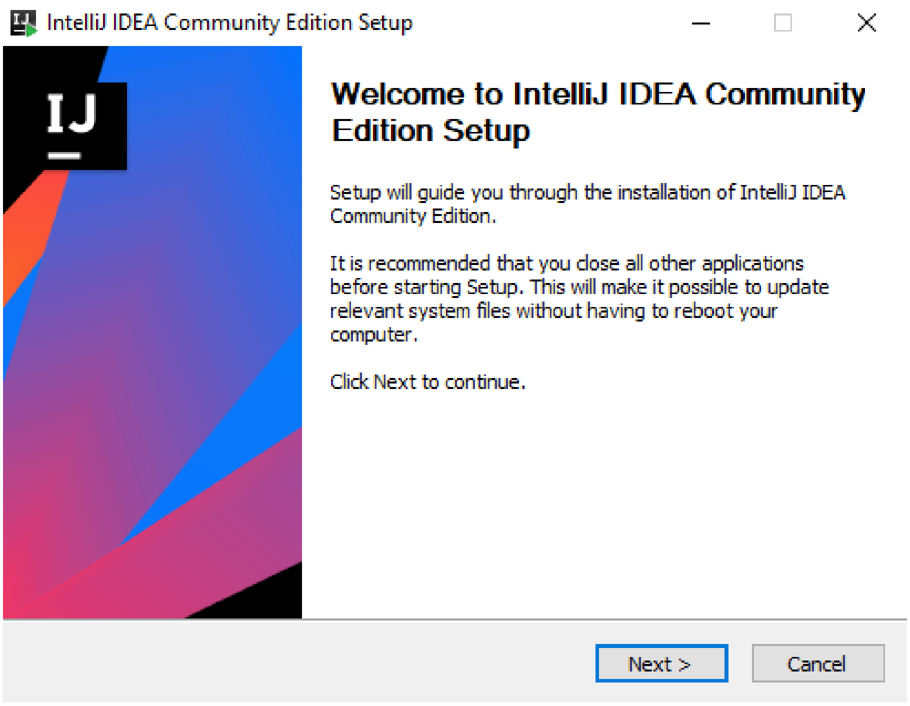
https://gitforwindows.org/

5. Install IntelliJ:

    a. Go to the IntelliJ Windows download page.

    b. Download and run the Community Edition installer (`.exe`).

    c. Follow the prompts in the wizard.


IntelliJ IDEA Community Edition Setup

Welcome to IntelliJ IDEA Community Edition Setup

Setup will guide you through the installation of IntelliJ IDEA Community Edition.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

d. Reboot your computer to finish the setup.

e. Start IntelliJ and set up your preferences. You can use the defaults, unless you have reason not to.

Setting up on macOS

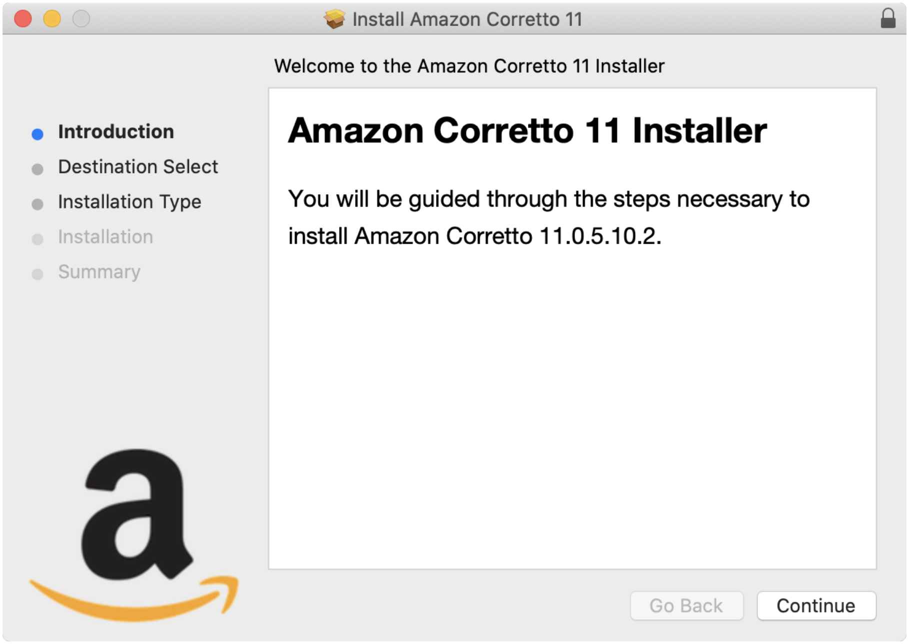
To set up your developer environment on macOS:

1. Install Java:

   a. Go to the Amazon Corretto 11 download page.

   b. Download and run the macOs installer (`.pkg`).

   c. Follow the prompts in the wizard.



Amazon Corretto 11 Installer

You will be guided through the steps necessary to install Amazon Corretto 11.0.5.10.2.

2. Install Homebrew:

   Homebrew, is a package manager, and is the easiest way to install both Maven and Node on macOS. To install Homebrew, paste the following into your terminal:

   ```
   /usr/bin/ruby -e "$(curl -fsSL
   https://raw.githubusercontent.com/Homebrew/install/master/install)"
   ```

3. Install Maven and Node:

   Use this command to install both Maven and Node in Homebrew:

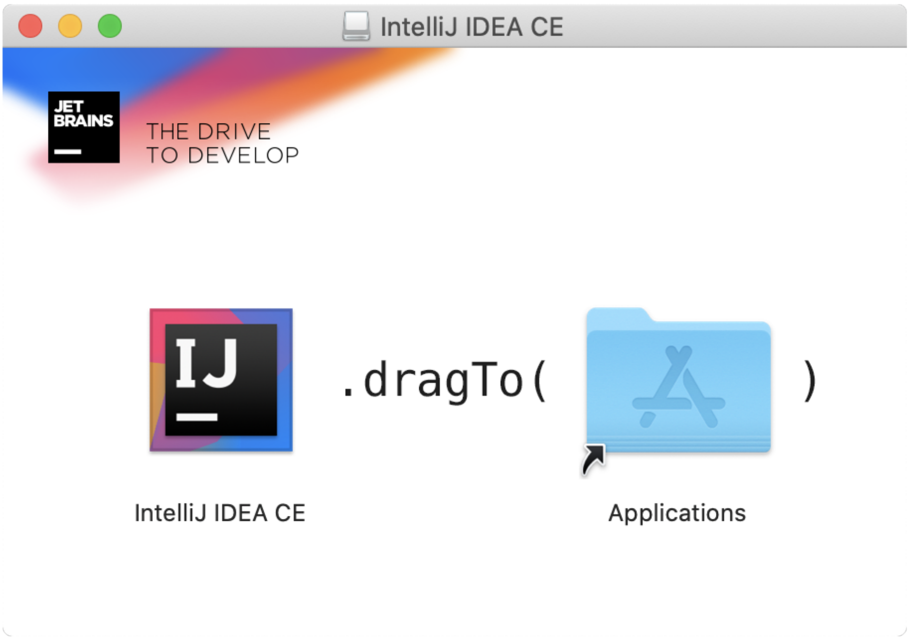
   ```
   brew install node maven
   ```

4. Install IntelliJ:

   a. Go to the IntelliJ Mac download page.

   b. Download the Community Edition installer .

   c. Copy the app to your Applications folder in Finder.



Setting up on Linux

This section contains instructions for Debian and RPM-based distros. Installation on other distributions should be very similar and you can adapt these instructions, if necessary. On Linux, it's easiest to use OpenJDK, as it's available in the package repositories.

1. Install Node.js:

NOTE	You need to install the latest Node.js LTS repository to your package manager. The version available in most distributions is not sufficiently new for our purposes. Nodesource offers packages for all major distros.

   a. Debian-based systems:

      i. For Ubuntu and distributions using sudo, run:

```
curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -
sudo apt-get install -y openjdk-11-jdk maven git nodejs
```

      ii. For Debian, or if you are not using sudo, run the following as root:

```
curl -sL https://deb.nodesource.com/setup_12.x | bash -
apt-get install -y openjdk-11-jdk maven git nodejs
```

b. RPM-based distributions, run:

```
curl -sL https://rpm.nodesource.com/setup_12.x | sudo -E bash -
sudo yum install -y java-11-openjdk-devel.x86_64 maven git nodejs
```

i. Check the Java version:

- To ensure that you are running Java 11, run `java -version`.
- To change to change to Java 11, if necessary, use:

```
sudo alternatives --config java
```

NOTE If you are on a different distro, or aren't comfortable with the automatic repo setup script, you can find a full set of instructions on the NodeSource GitHub repository.

2. Install IntelliJ:

TIP The easiest way to install IntelliJ on Linux is to use the Snap package manager. If you use Ubuntu or a derivative, it is already installed.

a. To install IntelliJ using snap, run:

```
sudo snap install intellij-idea-community --classic
```

b. To install intelliJ manually:

i. Go to the IntelliJ Linux download page.

ii. Download the Community Edition `.tar.gz`.

iii. Extract the archive:

```
sudo mkdir /opt/intellij
sudo tar zxvf ideaIC*.tar.gz -C /opt/intellij --strip-components=1
```

iv. Run the IDE (the start wizard gives you the option to create a desktop shortcut):

```
sh /opt/intellij/bin/idea.sh
```

You now have everything you need to start coding Java. The next tutorial will show you how to import and run a Maven-based Java project in IntelliJ.

Importing, running, and debugging Maven projects in IntelliJ IDEA

The first step in any project is to set up the project and get a base app running.

In this section, you'll learn:

- How to import a project starter into your IDE.
- How to set up your browser to automatically show updates as you build your application.

TIP	You can skip to chapter 4 if you prefer to code using the online IDE.

Importing a Maven project into IntelliJ

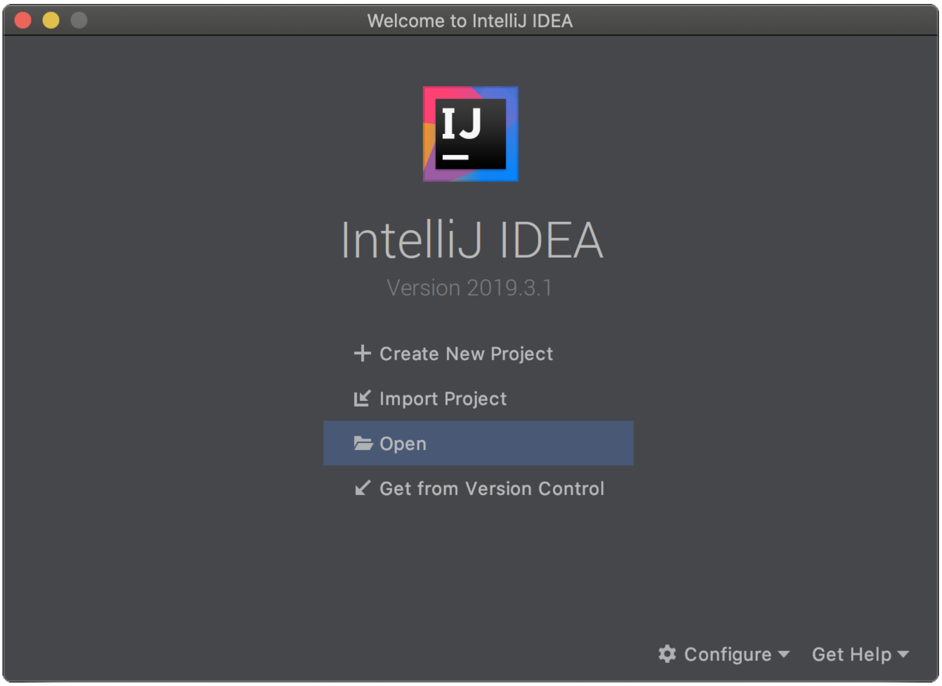
Maven is the most popular project management tool for Java. It takes care of managing your project structure and dependencies, and builds runnable artifacts.

1. To start, download a Spring Boot-based Vaadin starter project:

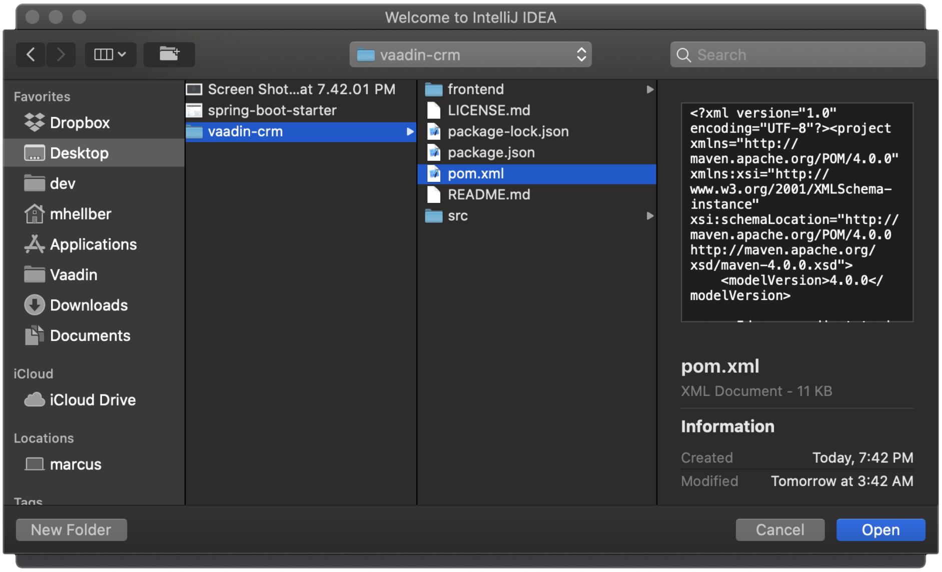
   Use the following link to download a pre-configured starter: https://vaadin.com/vaadincom/start-service/lts/project-base?appName=Vaadin%20CRM&groupId=com.vaadin.tutorial.crm&techStack=spring

2. Unzip the downloaded archive to a file location of your choice. TIP: Avoid unzipping to the download folder, as you could unintentionally delete your project when clearing out old downloads.

3. In IntelliJ, select **Open** in the Welcome screen or **File** menu.



4. Find the extracted folder, and select the pom.xml file.



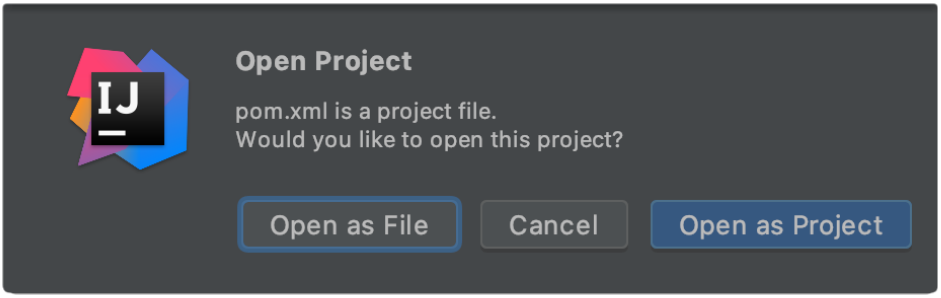
```
<?xml version="1.0"
encoding="UTF-8"?><project
xmlns="http://
maven.apache.org/POM/4.0.0"
xmlns:xsi="http://
www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://
maven.apache.org/POM/4.0.0
http://maven.apache.org/
xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</
modelVersion>
```

pom.xml
XML Document - 11 KB

Information

Created	Today, 7:42 PM
Modified	Tomorrow at 3:42 AM

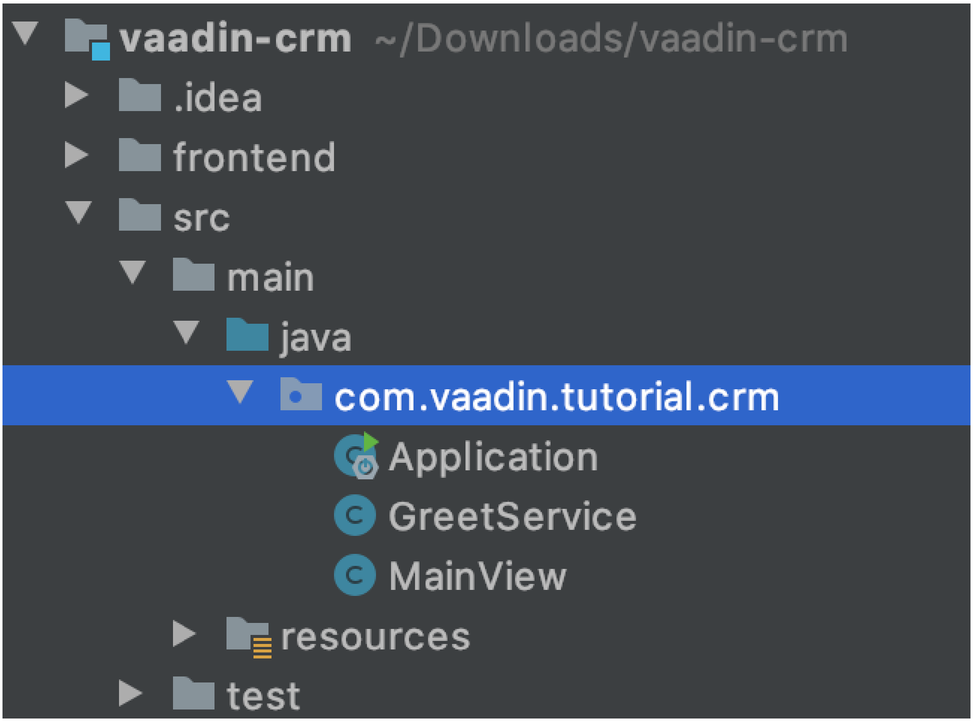
5. Select **Open as Project**. This imports a project based on the POM file.


Open Project
pom.xml is a project file.
Would you like to open this project?
Open as File
Cancel
Open as Project

6. IntelliJ imports the project and downloads all necessary dependencies. This can take several minutes, depending on your internet connection speed.

When the import is complete, your project structure will be similar to this:

- Java source files are in the `src/main/java` folder.
- Test files are in the `src/test` folder (we'll refer to these later).



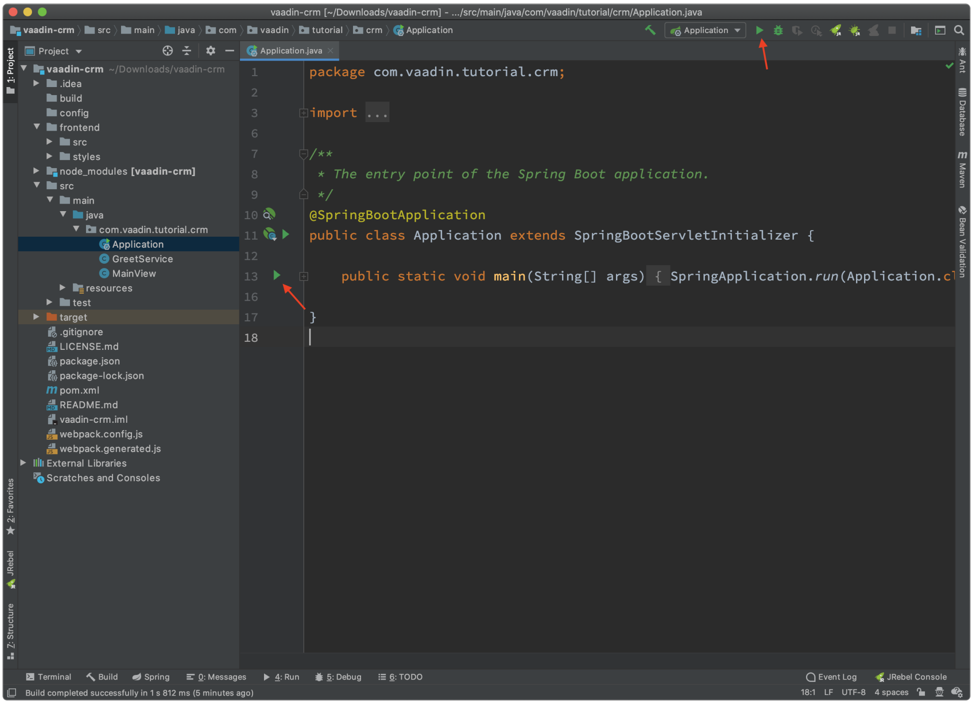
Running a Spring Boot project

Spring Boot makes it easier to run a Java web application, because it takes care of starting and configuring the server.

To run your application, run the Application class that contains the main method that starts Spring Boot. IntelliJ automatically detects that you have a class with a main method and displays it in the run configurations dropdown.

To start your application:

- Open `Application.java` and click the play button next to the code line containing the main method.
- After you have run the app once from the main method, it will show up in run configurations dropdown in the main toolbar. On subsequent runs, you can run the app from there.



```
package com.vaadin.tutorial.crm;

import ...

/**
 * The entry point of the Spring Boot application.
 */
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

    public static void main(String[] args) { SpringApplication.run(Application.cl

}
```

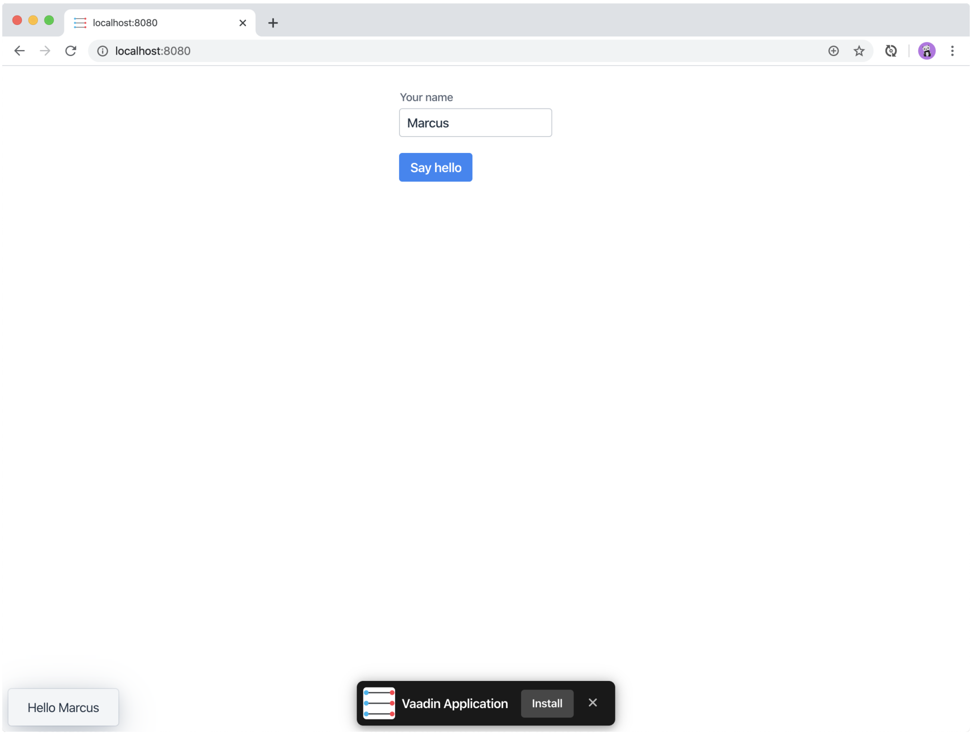
Build completed successfully in 1 s 812 ms (5 minutes ago)

The first time you start a Vaadin application, it downloads front-end dependencies and builds a JavaScript bundle. This can take several minutes, depending on your computer and internet speed.

You'll know that your application has started when you see the following output in the console:

```
Tomcat started on port(s): 8080 (http) with context path ''
Started Application in 80.189 seconds (JVM running for 83.42)
```

You can now open localhost:8080 in your browser. You'll see a **Say hello** button and **Your name** field on the screen. Enter your name and click the button to see the notification that displays.



Debugging with IntelliJ

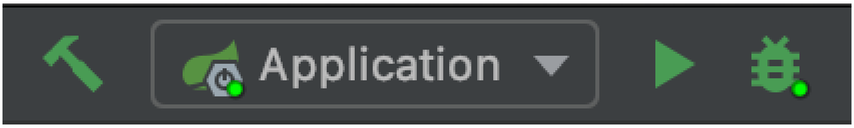
The debugger is a useful tool to understand what's happening in your code when things are not working as expected. Knowing how to use the debugger saves you from having to use a lot of `System.out.println's.

To use the debugger in IntelliJ:

1. If your application is still running from the previous step, click the red stop-button to

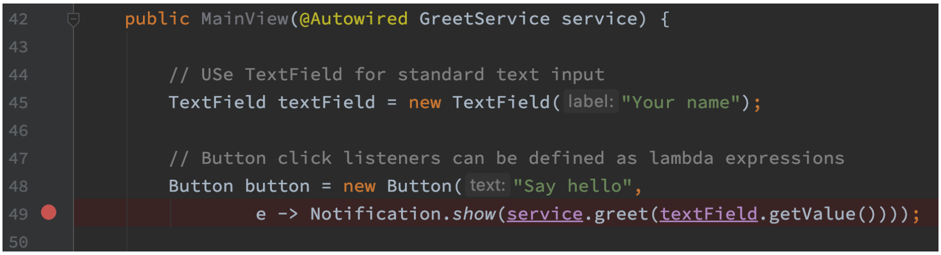
terminate it.

2. Start your application in debug mode, by clicking the bug icon next to the play button. You can now insert a debug point. This tells the debugger to pause the app whenever it gets to the line marked in the code.



You can now insert a debug point. This tells the debugger to pause the app whenever it gets to the line marked in the code.

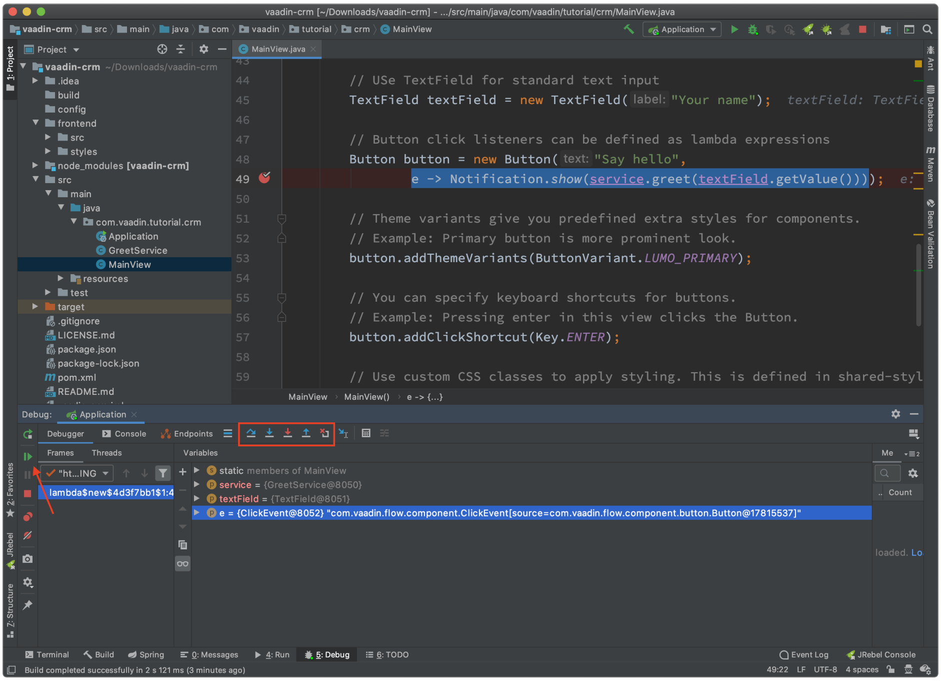
3. In MainView.java, on the line containing Notification.show, click next to the line number to add a breakpoint (debug point). When you click, the code line is highlighted and a red dot displays.



```java
    public MainView(@Autowired GreetService service) {

        // USe TextField for standard text input
        TextField textField = new TextField( label: "Your name");

        // Button click listeners can be defined as lambda expressions
        Button button = new Button( text: "Say hello",
                e -> Notification.show(service.greet(textField.getValue())));

```

If you now open http://localhost:8080 in your browser, and click the Say hello button, nothing happens. This is because the application stops on the line indicated in the IDE.

4. In IntelliJ, have a look at the highlighted code line and the debug panel in the lower part of the screen.



vaadin-crm [~/Downloads/vaadin-crm] - .../src/main/java/com/vaadin/tutorial/crm/MainView.java

```
// USe TextField for standard text input
TextField textField = new TextField( label: "Your name");   textField: TextFie

// Button click listeners can be defined as lambda expressions
Button button = new Button( text: "Say hello",
        e -> Notification.show(service.greet(textField.getValue())));   e:

// Theme variants give you predefined extra styles for components.
// Example: Primary button is more prominent look.
button.addThemeVariants(ButtonVariant.LUMO_PRIMARY);

// You can specify keyboard shortcuts for buttons.
// Example: Pressing enter in this view clicks the Button.
button.addClickShortcut(Key.ENTER);

// Use custom CSS classes to apply styling. This is defined in shared-styl
```

```
static members of MainView
service = {GreetService@8050}
textField = {TextField@8051}
e = {ClickEvent@8052} "com.vaadin.flow.component.ClickEvent[source=com.vaadin.flow.component.button.Button@17815537]"
```

```
Build completed successfully in 2 s 121 ms (3 minutes ago)
```

In the debug panel, you can see values for all variables. There are also controls that allow you to run the app one step at a time, to better understand what's happening. The most important controls are:

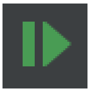
- **Step over**: Continue to the next line in the same file.

- **Step into**: Drill into a method call (for instance, if youwanted to see what's going on inside service.greet()).

- **Step out**: Go back to the line of code that called the methodyou're currently in.

Play around with the debugger to familiarize yourself with it. If you want to learn more, JetBrains has an excellent resource on using the debugger.

5. Click Resume Program when you are done.

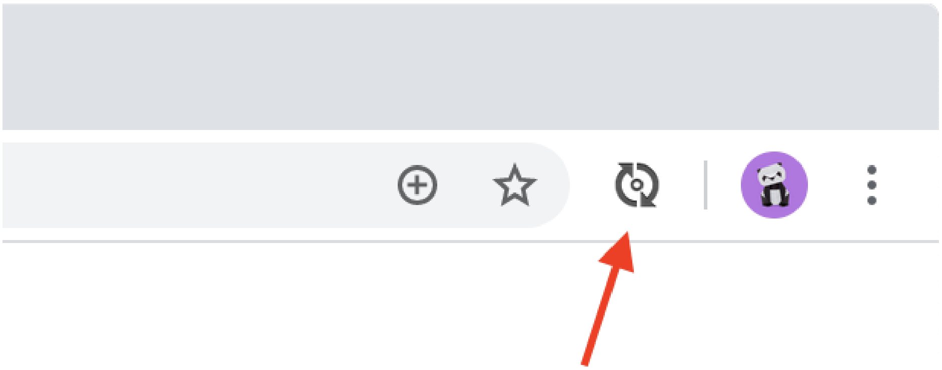


Your code will now run normally and you'll see the notification in your browser.

Enabling live browser reload

One final thing to do before starting to program is to enable live reloading of changes. This provides a far better development experience. All code changes you make are automatically displayed in the browser, without the need to refresh the page manually.

1. Start by downloading the LiveReload plugin for your browser:
   - LiveReload plugin for Chrome and Chromium Edge
   - LiveReload plugin for Firefox
   - LiveReload plugin for Safari
2. Install the plugin, reload your browser window, and click on the LiveReload icon in the top bar of your browser. (Make sure your app is running when you do this.)



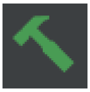
The middle of the icon should turn solid to indicate that LiveReload is working and has connected to your app. If it doesn't, try refreshing the page or reloading the browser.

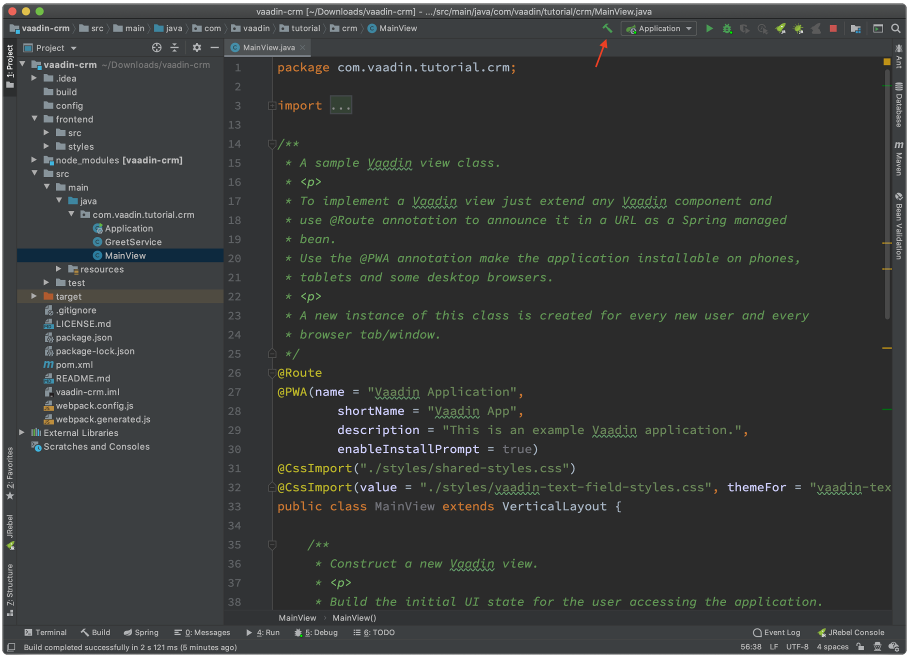
3. When LiveReload is running, verify that it works by making a change in the code:
    a. Create a new H1 heading and add it as the first argument in the add() method on the last line in MainView.

    `MainView.java`

    ```java
    add(new H1("Hello world"), textField, button);
    ```

    b. Click the build icon in IntelliJ (next to the run targets dropdown)



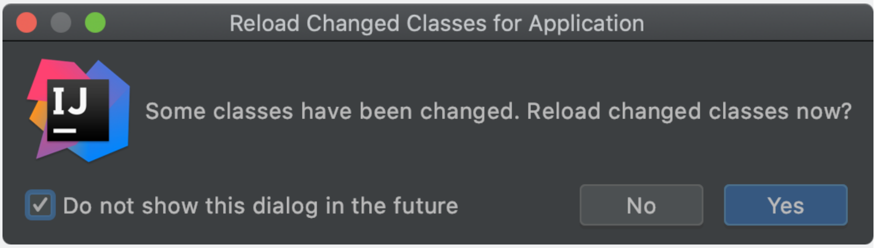


```java
package com.vaadin.tutorial.crm;

import ...

/**
 * A sample Vaadin view class.
 * <p>
 * To implement a Vaadin view just extend any Vaadin component and
 * use @Route annotation to announce it in a URL as a Spring managed
 * bean.
 * Use the @PWA annotation make the application installable on phones,
 * tablets and some desktop browsers.
 * <p>
 * A new instance of this class is created for every new user and every
 * browser tab/window.
 */
@Route
@PWA(name = "Vaadin Application",
        shortName = "Vaadin App",
        description = "This is an example Vaadin application.",
        enableInstallPrompt = true)
@CssImport("./styles/shared-styles.css")
@CssImport(value = "./styles/vaadin-text-field-styles.css", themeFor = "vaadin-tex
public class MainView extends VerticalLayout {

    /**
     * Construct a new Vaadin view.
     * <p>
     * Build the initial UI state for the user accessing the application.
```

```
Build completed successfully in 2 s 121 ms (5 minutes ago)
```

NOTE

The first time you make a change with the debugger active, you'll see a "Reload changed classes now?" dialog. Select **Do not ask again** and click **No**. Spring Boot DevTools will take care of the reload for us.

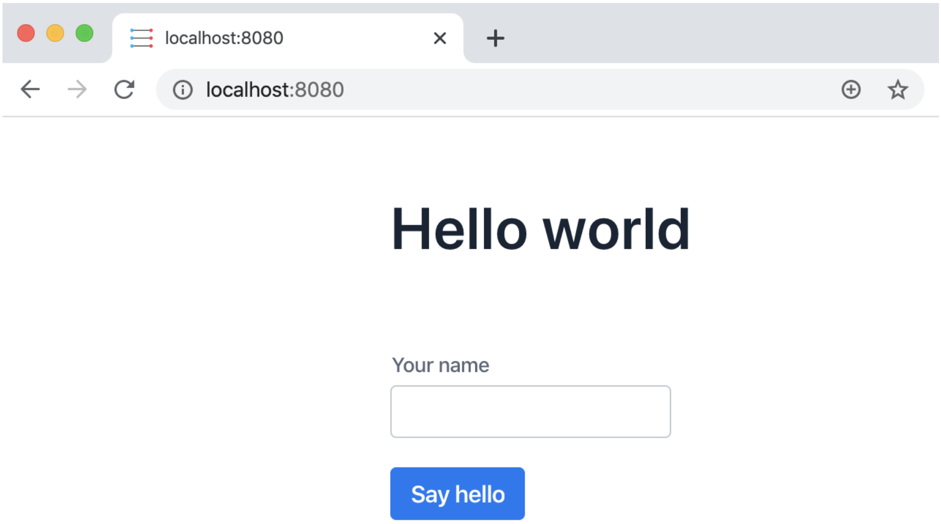


Reload Changed Classes for Application

Some classes have been changed. Reload changed classes now?

[✓] Do not show this dialog in the future

1. If all goes well, you'll see a notification that the build was successful, and your browser will reload automatically to show the change. Magic.



Hello world

Your name

NOTE

You may sometimes see error messages like this in the browser after a reload.

```
Could not navigate to ''
Reason: Couldn't find route for ''
Available routes:
This detailed message is only shown when running in development mode.
```

or

```
There was an exception while trying to navigate to '' with the exception
message 'Error creating bean with name 'com.vaadin.tutorial.crm.MainView':
Unsatisfied dependency expressed through constructor parameter 0
```

These errors are caused by a Spring DevTools reload timing issue. You may be able to alleviate the issue by adding the following two properties to src/main/resources/application.properties and adjusting the intervals to work with your computer. Stop and restart the server after adding the properties.

application.properties

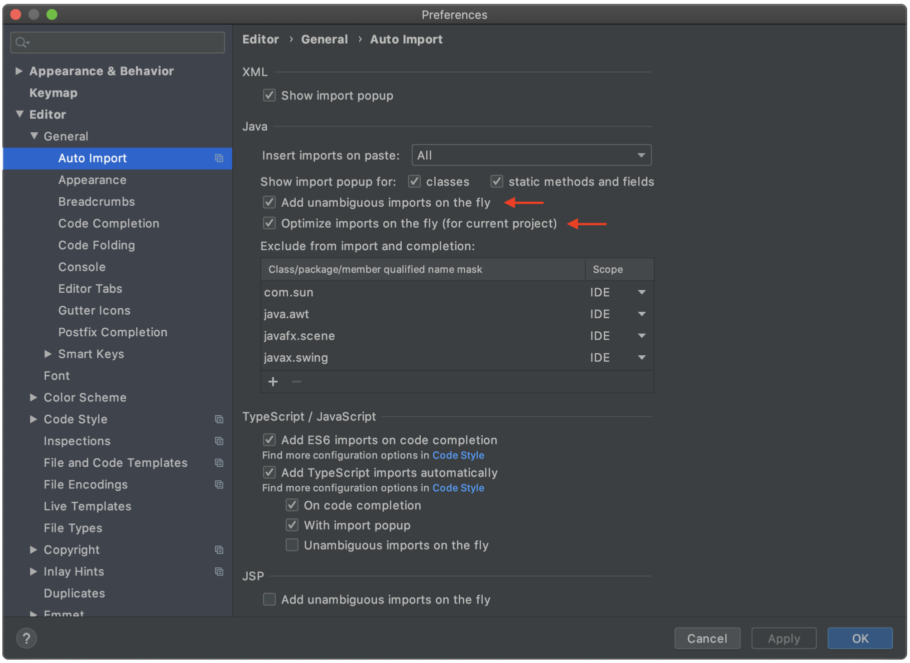
```
spring.devtools.restart.poll-interval=2s
spring.devtools.restart.quiet-period=1s
```

Enabling auto import

You can configure IntelliJ to automatically resolve imports for Java classes. This makes it easier to copy code from this tutorial into your IDE.

To enable auto import in IntelliJ:

1. Open the **Preferences/Settings** window and navigate to **Editor > General > Auto Import**.
2. Enable the following two options:
   - **Add unambiguous imports on the fly**.
   - **Optimize imports on the fly**.



Editor › General › Auto Import

XML

[✓] Show import popup

Java

Insert imports on paste: All

Show import popup for: [✓] classes [✓] static methods and fields

[✓] Add unambiguous imports on the fly

[✓] Optimize imports on the fly (for current project)

Exclude from import and completion:

Class/package/member qualified name mask	Scope
com.sun	IDE
java.awt	IDE
javafx.scene	IDE
javax.swing	IDE

TypeScript / JavaScript

[✓] Add ES6 imports on code completion
Find more configuration options in Code Style

[✓] Add TypeScript imports automatically
Find more configuration options in Code Style

[✓] On code completion

[✓] With import popup

[ ] Unambiguous imports on the fly

JSP

[ ] Add unambiguous imports on the fly

Vaadin shares many class names (like Button) with Swing, AWT, and JavaFX.

3. If you don't use Swing, AWT, or JavaFX in other projects, add the following packages to the **Exclude from import and completion** list to help IntelliJ select the correct classes automatically.

   - `com.sun`
   - `java.awt`
   - `javafx.scene`
   - `javax.swing`
   - `jdk.internal`
   - `sun.plugin`

Now that you have a working development environment, we can start building a web app.

You can find the completed source code for this tutorial on GitHub.

Vaadin basics: building UIs with components and layouts

In the previous two tutorials, you learned to set up a Java development environment and to import and run a Maven-based Spring Boot project.

In this chapter we introduce you to core Vaadin concepts and get our project ready for coding.

You can download the completed source code at the bottom. The code from the previous tutorial chapter can be found here, if you want to jump directly into this chapter.

TIP If you prefer to use a drag and drop editor to build the UI, check out the Vaadin Designer version of this tutorial.

Quick introduction to Vaadin

First, let’s review a few of the core Vaadin concepts we’ll be working with.

What is Vaadin?

Vaadin is a Java framework for building web applications. It has a component-based programming model that allows you to build user interfaces.

Vaadin UI components

The Vaadin platform includes a large library of UI components that you can use as the building blocks of your application.

You create a new component by initializing a Java object. For instance, to create a `Button`, you write:

```
Button button = new Button("I'm a button");
```

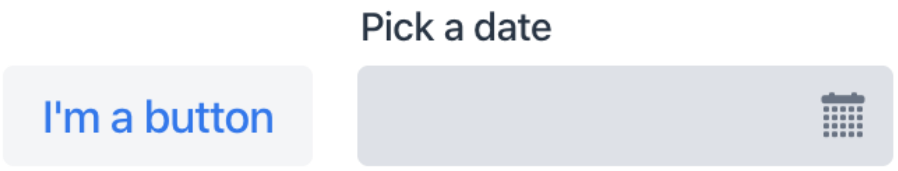
Layouts

Layouts determine how components display in the browser window. The most common layout components are `HorizontalLayout`, `VerticalLayout`, and `Div`. The first two set the content orientation as horizontal or vertical, respectively, whereas `Div` lets you control the positioning with CSS.

You add components to layouts using the `add()` method.

`HorizontalLayout` and `VerticalLayout` provide methods to align items on both the primary and the cross axis. For instance, if you want all components, regardless of their height, to be aligned with the bottom of a `HorizontalLayout`, you can set the default alignment to `Alignment.END`:

```
Button button = new Button("I'm a button");
HorizontalLayout layout = new HorizontalLayout(button, new DatePicker("Pick a date"
));


layout.setDefaultVerticalComponentAlignment(Alignment.END);
add(layout);
```



Events

You can add functionality to your application by listening to events, such as, click events from buttons, or value-change events from select components.

This example adds the text "Clicked!" to the layout when the button is clicked.

```
button.addClickListener(clickEvent ->
add(new Text("Clicked!")));
```

Where's the HTML?

One unique Vaadin feature is that you can build web applications entirely in Java. This higher level of abstraction makes development more productive and debugging easier.

Vaadin also supports HTML-templates and customizing the code that runs in the browser, but in most cases you needn't worry about this.

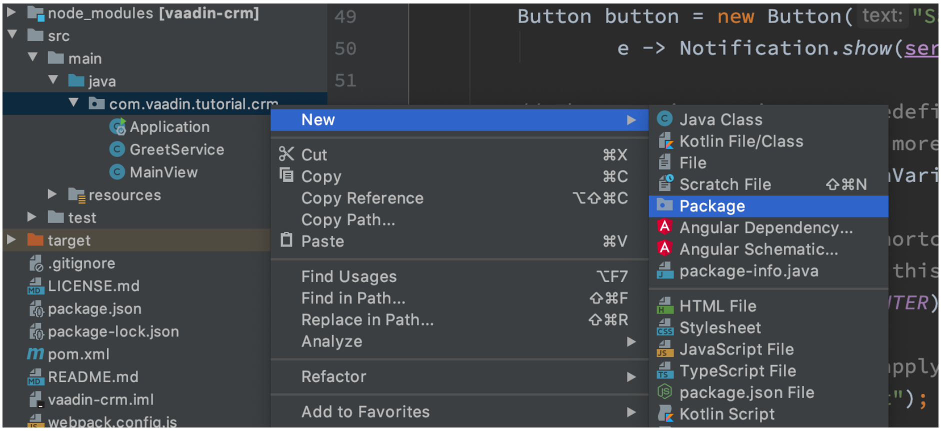
Preparing the project

Defining packages

Our app has both UI and backend code. To keep our code organized, we need to define separate packages for each in the project structure.

To define packages:

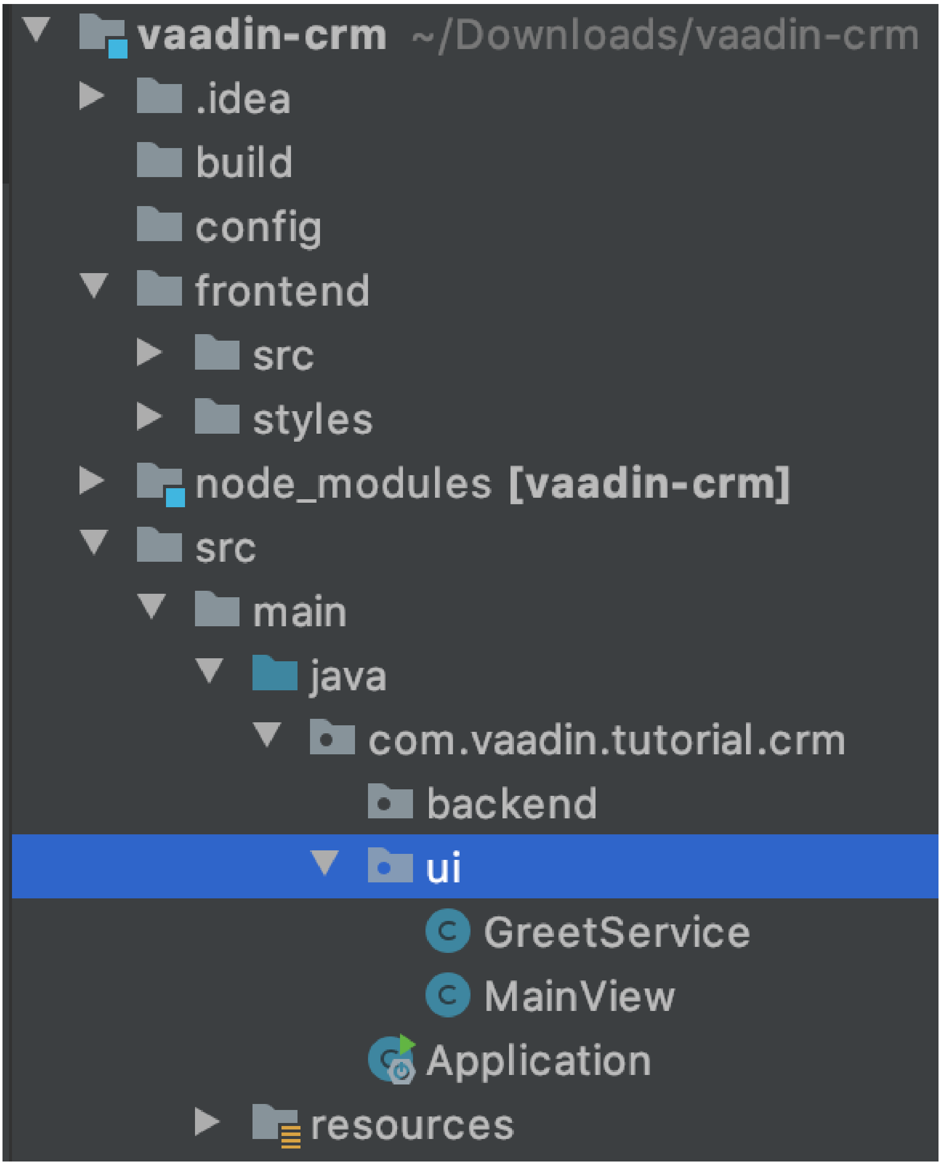
1. Right-click the `com.vaadin.tutorial.crm` package.
2. Select **New > Package** and create a package named `com.vaadin.tutorial.crm.ui`.
3. Repeat this process to create another package named `com.vaadin.tutorial.crm.backend`.



```
Button button = new Button( text: "S
        e -> Notification.show(ser
```

4. Drag `MainView` into the `ui` package. If IntelliJ asks you if you want to refactor the code, say yes.

   Your project structure should now look like this:



Setting up the main layout

Next, we clean out unnecessary code and set up our main layout.

To do this:

1. Delete the content of `MainView` and replace it with the code shown below. This removes all unnecessary code and ensures that we start with a clean slate.

   `MainView.java`

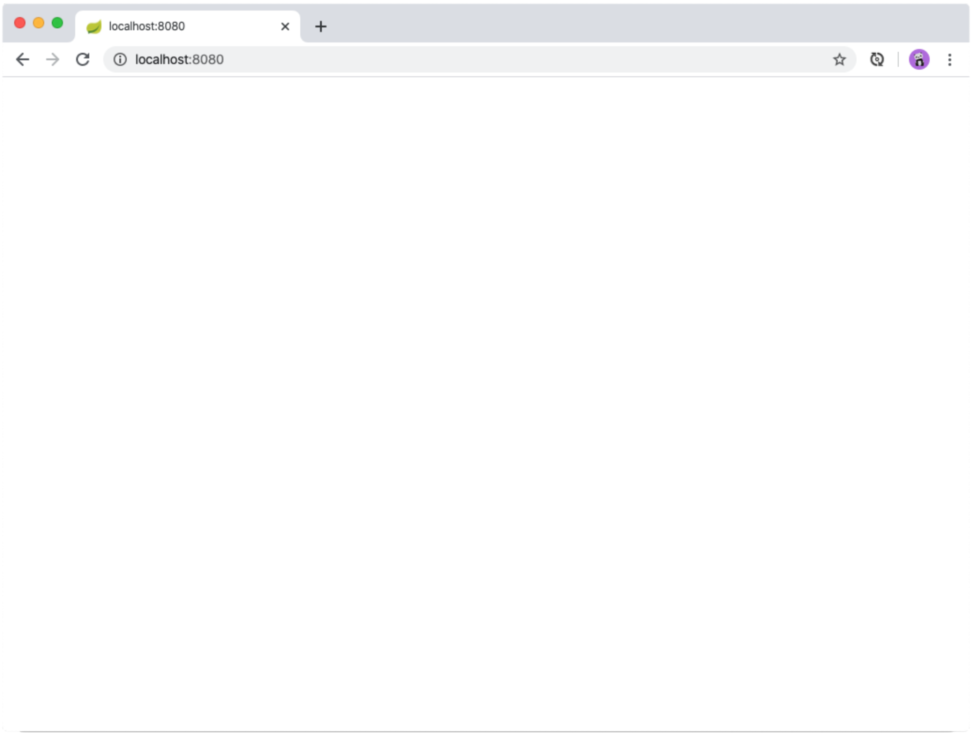
   ```
   package com.vaadin.tutorial.crm.ui;

   import com.vaadin.flow.component.orderedlayout.VerticalLayout;
   import com.vaadin.flow.router.Route;

   @Route("") ①
   public class MainView extends VerticalLayout {

       public MainView() {

       }

   }
   ```

   ① `@Route("")` maps the view to the root.

2. Next, **delete** the following unnecessary files:
   - `GreetService.java`
   - `frontend/styles/vaadin-text-field-styles.css`

3. Verify that you are able to run your application.

   You should see an empty window in the browser, and no errors in the console.



Before we can start building the UI, we need data to work with. In the next chapter, we'll set up a database and populate it with test data.

You can find the completed source code for this tutorial on GitHub.

Creating a Spring Boot backend: database, JPA repositories, and services

Most real-life applications need to persist and retrieve data from a database. In this tutorial, we use an in-memory H2 database. You can easily adapt the configuration to use another database, like MySQL or Postgres.

There are a fair number of classes to copy and paste to set up your backend. You can make your life easier by downloading a project with all the changes, if you prefer. The download link is at the end of this chapter.

The code from the previous tutorial chapter can be found here, if you want to jump directly into this chapter.

Installing the database dependencies

We use Spring Data for data access. Under the hood, it uses Hibernate to map Java objects to database entities through the Java Persistence API. Spring Boot takes care of configuring all these tools for you.

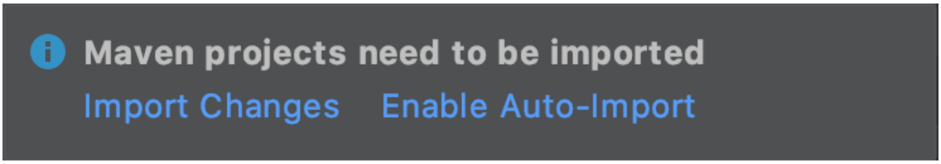
To add database dependencies:

1. In the <dependencies> tag in your pom.xml file, add the following dependencies for H2 and Spring Data:

   pom.xml

   ```xml
   <dependencies>
     <!--all existing dependencies -->

     <!--database dependencies -->
     <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-data-jpa</artifactId>
     </dependency>
     <dependency>
       <groupId>com.h2database</groupId>
       <artifactId>h2</artifactId>
       <scope>runtime</scope>
     </dependency>
   </dependencies>
   ```

2. Save your file and when IntelliJ asks if you want to enable automatic importing of Maven dependencies, select Enable Auto-Import.



If IntelliJ doesn't ask you to import dependencies, or if you use another IDE, type `mvn install` in the command line (while in the root of your project folder) to download the dependencies.

NOTE

H2 is a great database for tutorials because you don't need to install external software. If you prefer, you can easily change to another database. See:

- Setting up MySQL
- Setting up Postgres

The instructions in the remainder of this tutorial are the same, regardless of which database you use. To keep things simple, we recommend sticking with H2.

Defining the data model

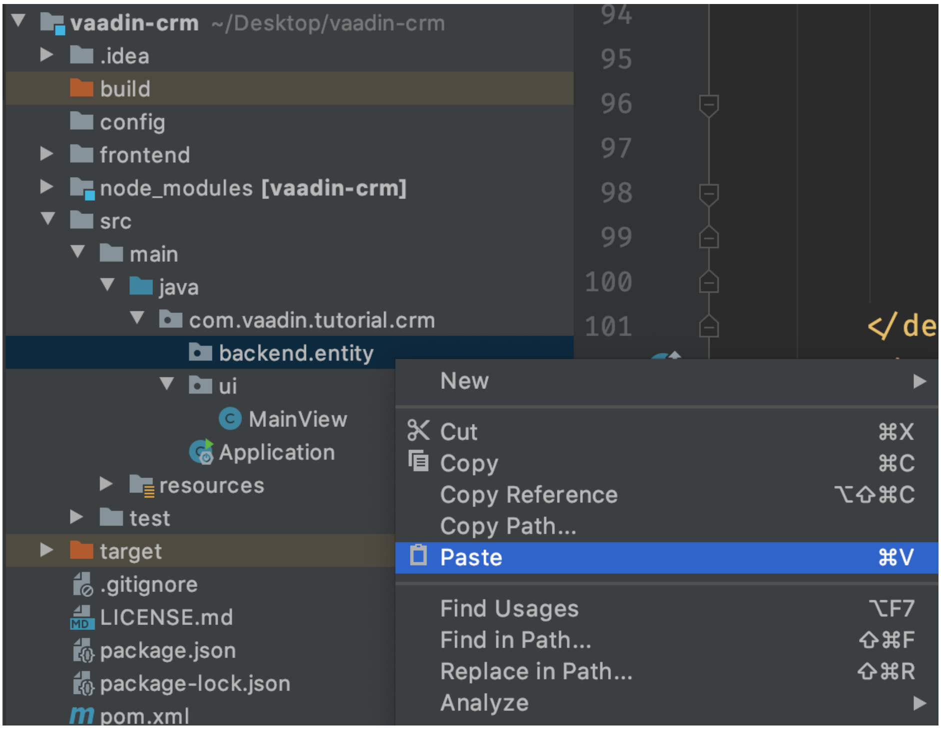
Our application is a customer relationship management (CRM) system that manages contacts and companies. To map content to our database, we need to create the following entity classes:

- `Contact`: An employee at a company.
- `Company`: An entity that can have several employees.
- `AbstractEntity`: A common superclass for both.

To create your entity classes:

1. Create a new package: `com.vaadin.tutorial.crm.backend.entity`.
2. Create three classes, `AbstractEntity`, `Contact`, and `Company`, in the new package, using the code detailed below.

   The easiest way to do this is to copy the full class and paste it into the package in the project view. IntelliJ (and most other IDEs) will automatically create the Java file for you.



a. Start by adding `AbstractEntity`, the common superclass. It defines how objects ids are generated and how object equality is determined.

AbstractEntity.java

```java
package com.vaadin.tutorial.crm.backend.entity;

import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.MappedSuperclass;

@MappedSuperclass
public abstract class AbstractEntity {
  @Id
  @GeneratedValue(strategy= GenerationType.SEQUENCE)
  private Long id;

  public Long getId() {
    return id;
  }

  public boolean isPersisted() {
    return id != null;
  }

  @Override
  public int hashCode() {
    if (getId() != null) {
      return getId().hashCode();
    }
    return super.hashCode();
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    }
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    AbstractEntity other = (AbstractEntity) obj;
 if (getId() == null || other.getId() == null) {
      return false;
    }
    return getId().equals(other.getId());
  }
}
```

b. Next, create the `Contact` class:

Contact.java

```
package com.vaadin.tutorial.crm.backend.entity;

import javax.persistence.*;
import javax.validation.constraints.Email;
import javax.validation.constraints.NotEmpty;
import javax.validation.constraints.NotNull;

@Entity
public class Contact extends AbstractEntity implements Cloneable {

  public enum Status {
    ImportedLead, NotContacted, Contacted, Customer, ClosedLost
  }

  @NotNull
  @NotEmpty
  private String firstName = "";

  @NotNull
  @NotEmpty
  private String lastName = "";

  @ManyToOne
  @JoinColumn(name = "company_id")
  private Company company;

  @Enumerated(EnumType.STRING)
  @NotNull
  private Contact.Status status;

  @Email
  @NotNull
  @NotEmpty
  private String email = "";

  public String getEmail() {
    return email;
  }

  public void setEmail(String email) {
    this.email = email;
  }

  public Status getStatus() {
    return status;
  }

  public void setStatus(Status status) {
    this.status = status;
  }

  public String getLastName() {
    return lastName;
  }
  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public void setCompany(Company company) {
    this.company = company;
  }

  public Company getCompany() {
    return company;
  }

  @Override
  public String toString() {
    return firstName + " " + lastName;
  }

}
```

c. Finally, copy over the `Company` class:

Company.java

```
package com.vaadin.tutorial.crm.backend.entity;

import javax.persistence.*;
import java.util.LinkedList;
import java.util.List;

@Entity
public class Company extends AbstractEntity {
  private String name;

  @OneToMany(mappedBy = "company", fetch = FetchType.EAGER)
  private List<Contact> employees = new LinkedList<>();

  public Company() {
  }

  public Company(String name) {
    setName(name);
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public List<Contact> getEmployees() {
    return employees;
  }
}
```

3. Verify that you're able to build the project successfully.

   If you see a lot of errors about missing classes, double check the Maven dependencies and run `mvn install` to make sure they are downloaded.

Creating repositories to access the database

Now that you have defined the data model, the next step is to create repository classes to access the database. Spring Boot makes this a painless process. All you need to do is define an interface that describes the entity type and primary key type, and Spring Data will configure it for you.

To create your repository classes:

1. Create a new package `com.vaadin.tutorial.crm.backend.repository`.
2. Copy the following two repository classes into the package:

`ContactRepository.java`

```java
package com.vaadin.tutorial.crm.backend.repository;

import com.vaadin.tutorial.crm.backend.entity.Contact;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import java.util.List;

public interface ContactRepository extends JpaRepository<Contact, Long> {

}
```

`CompanyRepository.java`

```java
package com.vaadin.tutorial.crm.backend.repository;

import com.vaadin.tutorial.crm.backend.entity.Company;
import org.springframework.data.jpa.repository.JpaRepository;

public interface CompanyRepository extends JpaRepository<Company, Long> {
}
```

Creating service classes for business logic

It’s good practice to not let UI code access the database directly. Instead, we create service classes that handle business logic and database access. This makes it easier for you to control access and to keep your data consistent.

To create your service classes:

1. Create a new package `com.vaadin.tutorial.crm.backend.service`.
2. Copy the following two service classes into the package:

ContactService.java

```java
package com.vaadin.tutorial.crm.backend.service;

import com.vaadin.tutorial.crm.backend.entity.Contact;
import com.vaadin.tutorial.crm.backend.repository.CompanyRepository;
import com.vaadin.tutorial.crm.backend.repository.ContactRepository;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

@Service ①
public class ContactService {
    private static final Logger LOGGER = Logger.getLogger(ContactService.class
.getName());
    private ContactRepository contactRepository;
    private CompanyRepository companyRepository;

    public ContactService(ContactRepository contactRepository,
                                                CompanyRepository
companyRepository) { ②
        this.contactRepository = contactRepository;
        this.companyRepository = companyRepository;
    }

    public List<Contact> findAll() {
        return contactRepository.findAll();
    }

    public long count() {
        return contactRepository.count();
    }

    public void delete(Contact contact) {
        contactRepository.delete(contact);
    }

    public void save(Contact contact) {
        if (contact == null) { ③
            LOGGER.log(Level.SEVERE,
                    "Contact is null. Are you sure you have connected your form
to the application?");
            return;
        }
        contactRepository.save(contact);
    }
}
```

① The `@Service` annotation lets Spring know that this is a service class and makes it available for injection. This allows you to easily use it from your UI code later on.

② The constructor takes 2 parameters: `ContactRepository` and `CompanyRepository`.

Spring provides instances based on the interfaces we defined earlier.

③ For now, most operations are just passed through to the repository. The only exception is the save method, which checks for null values before attempting to save.

CompanyService.java

```
package com.vaadin.tutorial.crm.backend.service;

import com.vaadin.tutorial.crm.backend.entity.Company;
import com.vaadin.tutorial.crm.backend.repository.CompanyRepository;
import org.springframework.stereotype.Service;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Service
public class CompanyService {

  private CompanyRepository companyRepository;

  public CompanyService(CompanyRepository companyRepository) {
    this.companyRepository = companyRepository;
  }

  public List<Company> findAll() {
    return companyRepository.findAll();
  }

}
```

Populating with test data

Next, we add a method that generates test data to populate our database. This makes it easier to work with the application.

To do this, add the following method at the end of ContactService:

ContactService.java

```java
@PostConstruct ①
public void populateTestData() {
    if (companyRepository.count() == 0) {
        companyRepository.saveAll( ②
            Stream.of("Path-Way Electronics", "E-Tech Management", "Path-E-Tech
Management")
                .map(Company::new)
                .collect(Collectors.toList()));
    }

    if (contactRepository.count() == 0) {
        Random r = new Random(0);
        List<Company> companies = companyRepository.findAll();
        contactRepository.saveAll( ③
            Stream.of("Gabrielle Patel", "Brian Robinson", "Eduardo Haugen",
                "Koen Johansen", "Alejandro Macdonald", "Angel Karlsson", "Yahir
Gustavsson", "Haiden Svensson",
                "Emily Stewart", "Corinne Davis", "Ryann Davis", "Yurem Jackson",
"Kelly Gustavsson",
                "Eileen Walker", "Katelyn Martin", "Israel Carlsson", "Quinn
Hansson", "Makena Smith",
                "Danielle Watson", "Leland Harris", "Gunner Karlsen", "Jamar Olsson
", "Lara Martin",
                "Ann Andersson", "Remington Andersson", "Rene Carlsson", "Elvis
Olsen", "Solomon Olsen",
                "Jaydan Jackson", "Bernard Nilsen")
                .map(name -> {
                    String[] split = name.split(" ");
                    Contact contact = new Contact();
                    contact.setFirstName(split[0]);
                    contact.setLastName(split[1]);
                    contact.setCompany(companies.get(r.nextInt(companies.size())));
                    contact.setStatus(Contact.Status.values()[r.nextInt(Contact
.Status.values().length)]);
                    String email = (contact.getFirstName() + "." + contact
.getLastName() + "@" + contact.getCompany().getName().replaceAll("[\\s-]", "") +
".com").toLowerCase();
                    contact.setEmail(email);
                    return contact;
                }).collect(Collectors.toList()));
    }
}
```

① The @PostConstruct annotation tells Spring to run this method after constructing ContactService.

② Creates 3 test companies.

③ Creates test contacts.

Restart the server to pick up all the new dependencies

You need to stop and restart the application to make sure all the new POM dependencies are picked up correctly.

You can download the project with a fully set-up back end below. Unzip the project and follow the instructions in the importing chapter.

Download from GitHub

In the next chapter, we'll use the back end to populate data into a data grid in the browser.

Adding data and configuring columns in Vaadin Grid

Now that your back end is in place, you’re ready to begin building the UI. We start by listing all contacts in a data grid. To do this, we need to create the necessary UI components and hook them up to the back end.

You can download the completed source code at the bottom. The code from the previous tutorial chapter can be found here, if you want to jump directly into this chapter.

Creating and configuring the Grid

To create and configure the grid:

1. Edit `MainView` and add the following code:

   `MainView.java`

```java
// Package and imports omitted

@Route("")
public class MainView extends VerticalLayout {

    private Grid<Contact> grid = new Grid<>(Contact.class); ①

    public MainView() {
        addClassName("list-view"); ②
        setSizeFull(); ③
        configureGrid(); ④

        add(grid); ⑤
    }

    private void configureGrid() {
        grid.addClassName("contact-grid");
        grid.setSizeFull();
        grid.setColumns("firstName", "lastName", "email", "status"); ⑥

    }

}
```

   ① Defines a new field grid and instantiates it to a Grid of type Contact.

   ② Gives the component a CSS class name to help with styling.

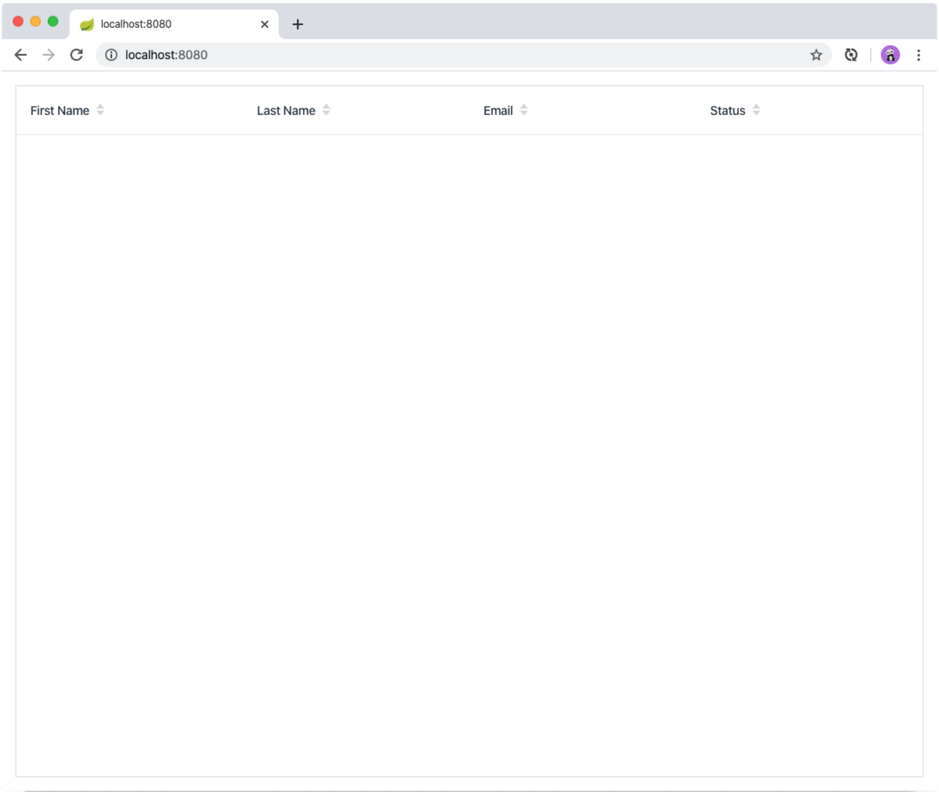
   ③ Calls `setSizeFull()` to make `MainView` take up the full size of the browser window.

④ Splits the grid configuration into a separate method. We will add more components to the class later on and this helps to keep things easy to understand.

⑤ Adds the grid to the main layout using the `add(grid)` method.

⑥ Defines the properties of a Contact to shown using the `grid.setColumns(..)` method.

2. Run the application and verify that you now see an empty grid with the correct columns.



Populating the Grid with data from the backend

Next, we hook up the view to the back end, so that we can fetch contacts to show in the grid. We take advantage of Spring's dependency injection to get hold of the back end service, by adding it as a parameter to the constructor. Spring passes it in when `MainView` is created.

To hook your view to the back end:

1. Amend `MainView` as follows:

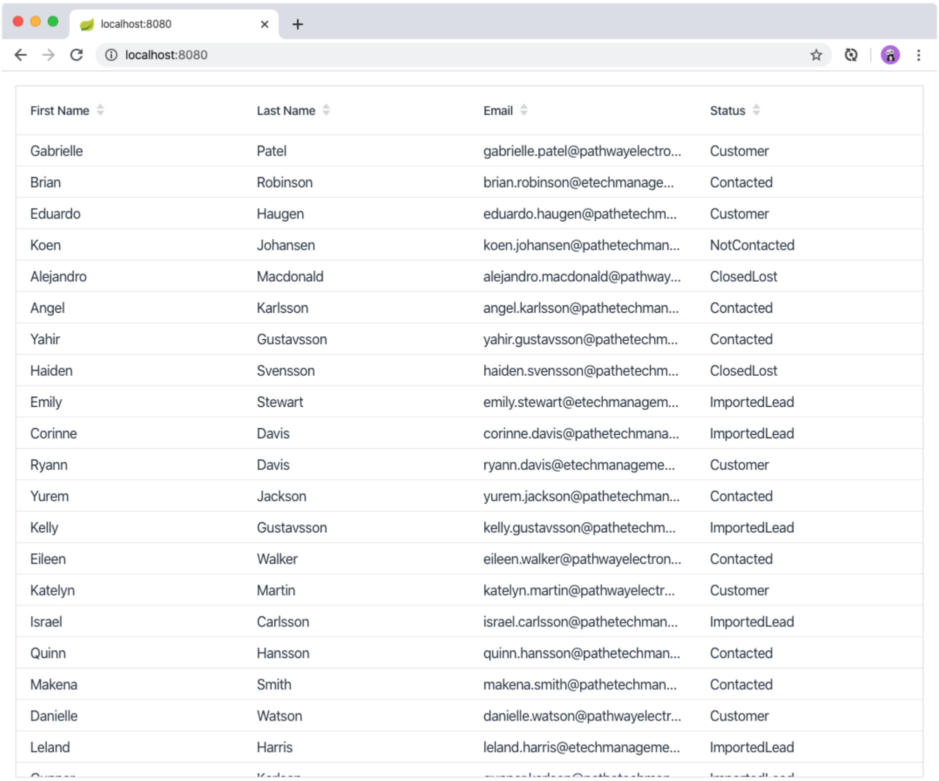
   `MainView.java`

```java
@Route("")
public class MainView extends VerticalLayout {

    private ContactService contactService;
    private Grid<Contact> grid = new Grid<>(Contact.class);

    public MainView(ContactService contactService) {
        this.contactService = contactService; ①
        addClassName("list-view");
        setSizeFull();
        configureGrid();

        add(grid);
        updateList(); ②
    }

    private void configureGrid() {
        grid.addClassName("contact-grid");
        grid.setSizeFull();
        grid.setColumns("firstName", "lastName", "email", "status");
    }

    private void updateList() {
        grid.setItems(contactService.findAll());
    }

}
```

   ① Saves ContactService in a field, so you have easy access to it later.

   ② Creates a new method, updateList(), that fetches all contacts from the service, and passes them to the grid.

2. Build the project and verify that you now see contacts listed in the grid.



localhost:8080

First Name	Last Name	Email	Status
Gabrielle	Patel	gabrielle.patel@pathwayelectro...	Customer
Brian	Robinson	brian.robinson@etechmanage...	Contacted
Eduardo	Haugen	eduardo.haugen@pathetechm...	Customer
Koen	Johansen	koen.johansen@pathetechman...	NotContacted
Alejandro	Macdonald	alejandro.macdonald@pathway...	ClosedLost
Angel	Karlsson	angel.karlsson@pathetechman...	Contacted
Yahir	Gustavsson	yahir.gustavsson@pathetechm...	Contacted
Haiden	Svensson	haiden.svensson@pathetechm...	ClosedLost
Emily	Stewart	emily.stewart@etechmanagem...	ImportedLead
Corinne	Davis	corinne.davis@pathetechmana...	ImportedLead
Ryann	Davis	ryann.davis@etechmanageme...	Customer
Yurem	Jackson	yurem.jackson@pathetechman...	Contacted
Kelly	Gustavsson	kelly.gustavsson@pathetechm...	ImportedLead
Eileen	Walker	eileen.walker@pathwayelectron...	Contacted
Katelyn	Martin	katelyn.martin@pathwayelectr...	Customer
Israel	Carlsson	israel.carlsson@pathetechman...	ImportedLead
Quinn	Hansson	quinn.hansson@pathetechman...	Contacted
Makena	Smith	makena.smith@pathetechman...	Contacted
Danielle	Watson	danielle.watson@pathwayelectr...	Customer
Leland	Harris	leland.harris@etechmanageme...	ImportedLead

Adding a custom column to the grid

In the previous chapter, when we set up the data model, we defined that each `Contact` belongs to a `Company`. We now want to show that company information as a column in the grid. The problem is that if we add `"company"` to the column list, the grid doesn't show the company name, but something like "com.vaadin.tutorial.crm.backend.e..." instead.

The company property is a reference to another object, and the grid shows the fully qualified class name because it doesn't know how we want to display the object. To fix this, we need to change how the company column is defined.

To define the company column:

1. Amend `MainView` as follows:

MainView.java

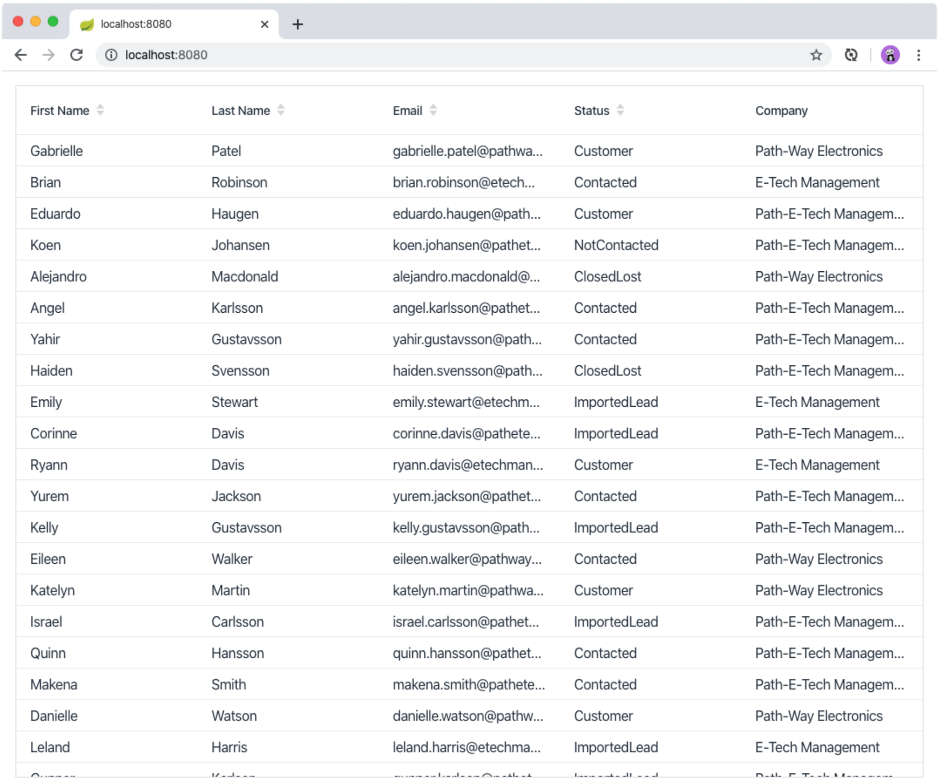
```java
private void configureGrid() {
    grid.addClassName("contact-grid");
    grid.setSizeFull();
    grid.removeColumnByKey("company"); ①
    grid.setColumns("firstName", "lastName", "email", "status");
    grid.addColumn(contact -> { ②
        Company company = contact.getCompany();
        return company == null ? "-" : company.getName();
    }).setHeader("Company");
}
```

① Removes the default column definition with the `removeColumnByKey` method.

② Uses the `addColumn` method to add a custom column.

- `addColumn` gets a `contact` parameter, and returns how it should be shown in the grid. In this case, we show the company name, or a dash if it’s empty.
- `setHeader` defines the column header for the custom column.

2. Build the application, and you should now see the company names listed in the grid.



First Name	Last Name	Email	Status	Company
Gabrielle	Patel	gabrielle.patel@pathwa...	Customer	Path-Way Electronics
Brian	Robinson	brian.robinson@etech...	Contacted	E-Tech Management
Eduardo	Haugen	eduardo.haugen@path...	Customer	Path-E-Tech Managem...
Koen	Johansen	koen.johansen@pathet...	NotContacted	Path-E-Tech Managem...
Alejandro	Macdonald	alejandro.macdonald@...	ClosedLost	Path-Way Electronics
Angel	Karlsson	angel.karlsson@pathet...	Contacted	Path-E-Tech Managem...
Yahir	Gustavsson	yahir.gustavsson@path...	Contacted	Path-E-Tech Managem...
Haiden	Svensson	haiden.svensson@path...	ClosedLost	Path-E-Tech Managem...
Emily	Stewart	emily.stewart@etechm...	ImportedLead	E-Tech Management
Corinne	Davis	corinne.davis@pathete...	ImportedLead	Path-E-Tech Managem...
Ryann	Davis	ryann.davis@etechman...	Customer	E-Tech Management
Yurem	Jackson	yurem.jackson@pathet...	Contacted	Path-E-Tech Managem...
Kelly	Gustavsson	kelly.gustavsson@path...	ImportedLead	Path-E-Tech Managem...
Eileen	Walker	eileen.walker@pathway...	Contacted	Path-Way Electronics
Katelyn	Martin	katelyn.martin@pathwa...	Customer	Path-Way Electronics
Israel	Carlsson	israel.carlsson@pathet...	ImportedLead	Path-E-Tech Managem...
Quinn	Hansson	quinn.hansson@pathet...	Contacted	Path-E-Tech Managem...
Makena	Smith	makena.smith@pathete...	Contacted	Path-E-Tech Managem...
Danielle	Watson	danielle.watson@pathw...	Customer	Path-Way Electronics
Leland	Harris	leland.harris@etechma...	ImportedLead	E-Tech Management
Gunnar	Karlsson	gunnar.karlsson@pathet...	ImportedLead	Path-E-Tech Managem...

Defining column widths

By default, the grid makes each column equally wide. Let's turn on automatic column sizing so that the email and company fields, which are typically longer, get more space. Automatic column sizing tries to make the column wide enough to fit the widest content.

To turn on automatic column sizing:

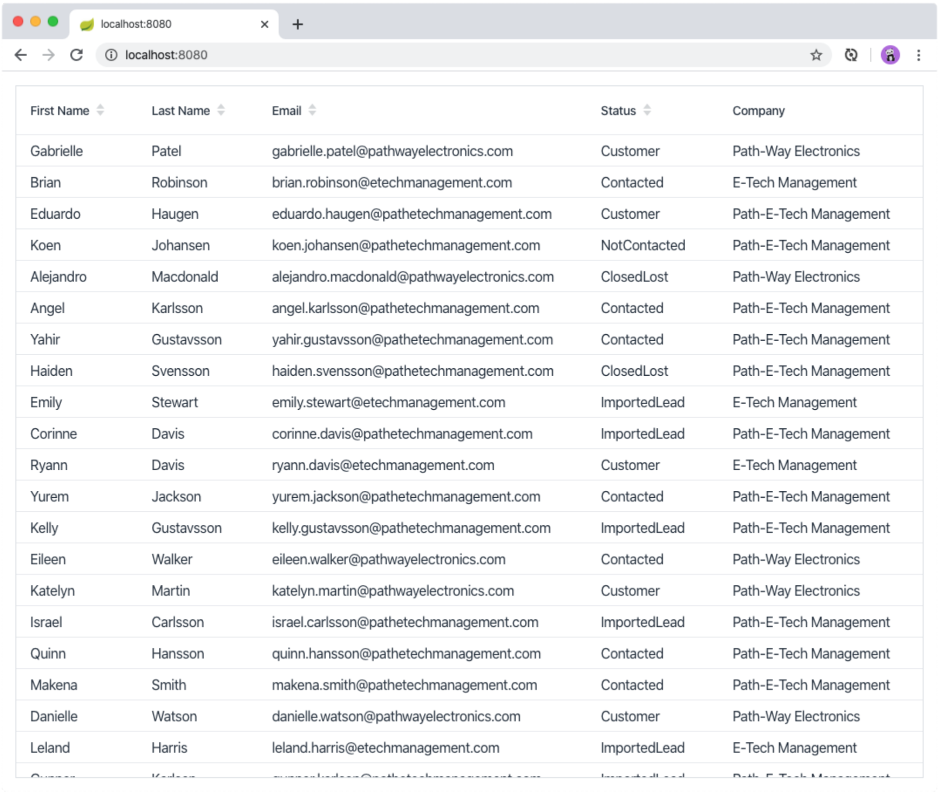
1. Amend `MainView` as follows:

MainView.java

```
private void configureGrid() {
    // column definitions omitted

    grid.getColumns().forEach(col -> col.setAutoWidth(true)); ①
}
```

① Automatic width needs to be turned on for each column separately. The easiest way to do it is to call `grid.getColumns()` and then use `forEach` to loop over all of them.

2. Build the app and you should now see that the columns are sized more appropriately.



First Name	Last Name	Email	Status	Company
Gabrielle	Patel	gabrielle.patel@pathwayelectronics.com	Customer	Path-Way Electronics
Brian	Robinson	brian.robinson@etechmanagement.com	Contacted	E-Tech Management
Eduardo	Haugen	eduardo.haugen@pathetechmanagement.com	Customer	Path-E-Tech Management
Koen	Johansen	koen.johansen@pathetechmanagement.com	NotContacted	Path-E-Tech Management
Alejandro	Macdonald	alejandro.macdonald@pathwayelectronics.com	ClosedLost	Path-Way Electronics
Angel	Karlsson	angel.karlsson@pathetechmanagement.com	Contacted	Path-E-Tech Management
Yahir	Gustavsson	yahir.gustavsson@pathetechmanagement.com	Contacted	Path-E-Tech Management
Haiden	Svensson	haiden.svensson@pathetechmanagement.com	ClosedLost	Path-E-Tech Management
Emily	Stewart	emily.stewart@etechmanagement.com	ImportedLead	E-Tech Management
Corinne	Davis	corinne.davis@pathetechmanagement.com	ImportedLead	Path-E-Tech Management
Ryann	Davis	ryann.davis@etechmanagement.com	Customer	E-Tech Management
Yurem	Jackson	yurem.jackson@pathetechmanagement.com	Contacted	Path-E-Tech Management
Kelly	Gustavsson	kelly.gustavsson@pathetechmanagement.com	ImportedLead	Path-E-Tech Management
Eileen	Walker	eileen.walker@pathwayelectronics.com	Contacted	Path-Way Electronics
Katelyn	Martin	katelyn.martin@pathwayelectronics.com	Customer	Path-Way Electronics
Israel	Carlsson	israel.carlsson@pathetechmanagement.com	ImportedLead	Path-E-Tech Management
Quinn	Hansson	quinn.hansson@pathetechmanagement.com	Contacted	Path-E-Tech Management
Makena	Smith	makena.smith@pathetechmanagement.com	Contacted	Path-E-Tech Management
Danielle	Watson	danielle.watson@pathwayelectronics.com	Customer	Path-Way Electronics
Leland	Harris	leland.harris@etechmanagement.com	ImportedLead	E-Tech Management

In the next tutorial, we'll add filtering to the application, so it's easier to find the right contact.

You can find the completed source code for this tutorial on GitHub.

Filtering rows in Vaadin Grid

In the previous tutorial, we created a data grid and filled it with contact details stored in the database. In this tutorial, we continue by adding a text field to filter the contents of the grid.

You can download the completed source code at the bottom. The code from the previous tutorial chapter can be found here, if you want to jump directly into this chapter.

Adding a text field for filtering

Start by adding a text field above the grid. Remember, `MainView` is a `VerticalLayout`, so you need to add the text field before the grid.

`MainView.java`

```
public class MainView extends VerticalLayout {
    private ContactService contactService;

    private Grid<Contact> grid = new Grid<>(Contact.class);
    private TextField filterText = new TextField(); ①

    public MainView(ContactService contactService) {
        this.contactService = contactService;
        addClassName("list-view");
        setSizeFull();
        configureFilter(); ②
        configureGrid();

        add(filterText, grid); ③
        updateList();
    }

    private void configureFilter() {
        filterText.setPlaceholder("Filter by name..."); ④
        filterText.setClearButtonVisible(true); ⑤
        filterText.setValueChangeMode(ValueChangeMode.LAZY); ⑥
        filterText.addValueChangeListener(e -> updateList()); ⑦
    }
    // Grid configuration omitted
}
```

① Creates a field for the `TextField`.

② Calls the `configureFilter()` method to configure what the filter should do.

③ Updates the `add()` method call to add both `filterText` and `grid`.

④ Sets placeholder text so users know what to type in the field.

⑤ Sets the clear button to visible so users can easily clear the filter.

⑥ Sets the value change mode to `LAZY` so the text field will notify you of changes automatically after a short timeout in typing.

⑦ Calls the `updateList` method whenever the value changes. We'll update the logic to filter the content shortly.

Implementing filtering in the back end

We could implement the filtering in two ways:

1. Keep a copy of the contacts list in the view and filter it using Java streams.
2. Defer the filtering to the back end (database).

It's a best practice to avoid keeping references to lists of objects in Vaadin views, as this can lead to excessive memory usage.

We'll add filtering support to the back end:

1. Amend `ContactService` as follows:

ContactService.java

```
public class ContactService {
    private static final Logger LOGGER = Logger.getLogger(ContactService.class
.getName());
    private ContactRepository contactRepository;
    private CompanyRepository companyRepository;

    public ContactService(ContactRepository contactRepository,
                                                CompanyRepository
companyRepository) {
        this.contactRepository = contactRepository;
        this.companyRepository = companyRepository;
    }

    public List<Contact> findAll() {
        return contactRepository.findAll();
    }

    public List<Contact> findAll(String stringFilter) { ①
        if (stringFilter == null || stringFilter.isEmpty()) { ②
            return contactRepository.findAll();
        } else {
            return contactRepository.search(stringFilter); ③
        }
    }

         // remaining methods omitted
}
```

① Adds a new `findAll` method that takes a filter text as a parameter.

② If the filter text is not empty, search the database for that text.

③ Otherwise, return all contacts

2. Add the `search` method to the contacts repository.

ContactRepository.java

```
public interface ContactRepository extends JpaRepository<Contact, Long> {

  @Query("select c from Contact c " +
      "where lower(c.firstName) like lower(concat('%', :searchTerm, '%')) " +
      "or lower(c.lastName) like lower(concat('%', :searchTerm, '%'))") ①
    List<Contact> search(@Param("searchTerm") String searchTerm); ②
}
```

① Uses the `@Query` annotation to define a custom query. In this case, it checks if the string matches the first or the last name, and ignores the case. The query uses Java Persistence Query Language (JPQL) which is an SQL-like language for querying JPA managed databases.

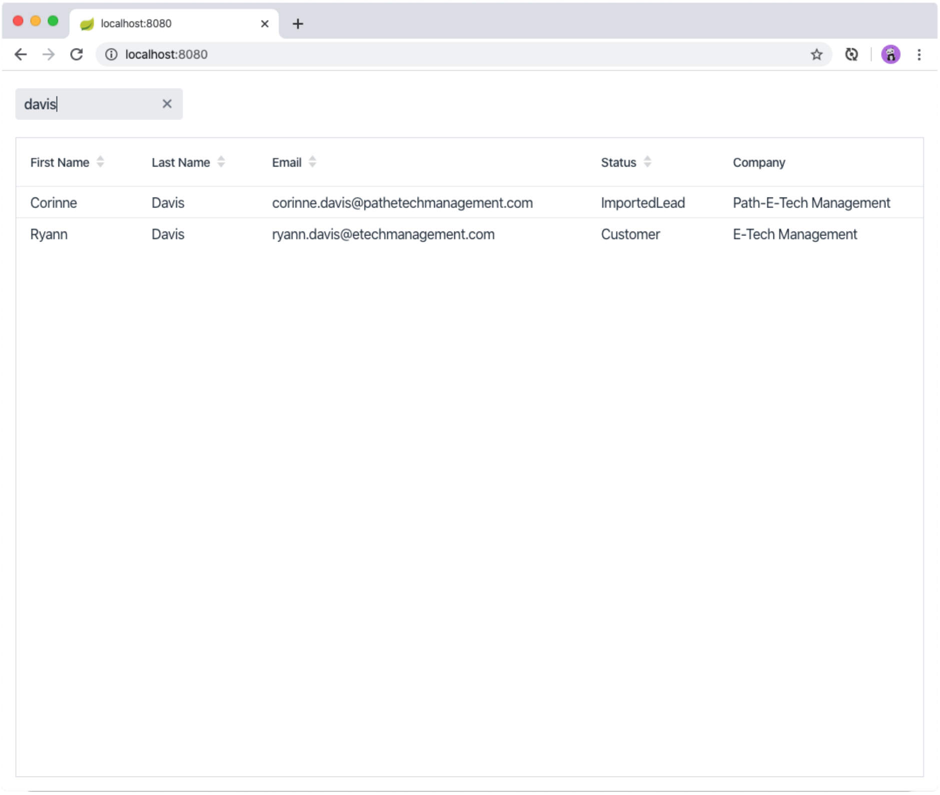
② Selects the Spring Framework import for `@Param`.

3. Update the way `MainView` updates the contacts. This is the method that is called every time the filter text field changes.

   `MainView.java`

   ```java
   private void updateList() {
       grid.setItems(contactService.findAll(filterText.getValue()));
   }
   ```

4. Build the application and try out the filtering. You should be able to filter the contacts by entering a term in the text field.



First Name	Last Name	Email	Status	Company
Corinne	Davis	corinne.davis@pathetechmanagement.com	ImportedLead	Path-E-Tech Management
Ryann	Davis	ryann.davis@etechmanagement.com	Customer	E-Tech Management

So far, we've created an application that shows and filters contacts that are stored in a database. Next, we'll add a form to add, remove, and edit contacts.

You can find the completed source code for this tutorial on GitHub.

Creating your own reusable components in Vaadin

In the previous tutorial, we added filtering to the grid that lists contacts stored in a database. In this tutorial, we build a form component to add, remove, and edit contacts.

You can download the completed source code at the bottom. The code from the previous tutorial chapter can be found here, if you want to jump directly into this chapter.

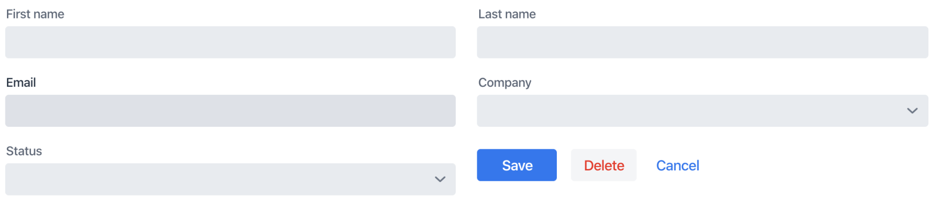
Vaadin is a component-based framework. You've already worked with several components, like `Grid`, `TextField`, and `VerticalLayout`. But, the real power of the component-based architecture is in the ability to create your own components.

Instead of building an entire view in a single class, your view can be composed of smaller components that each handle different parts of the view. The advantage of this approach is that individual components are easier to understand and test. The top-level view is used mainly to orchestrate the components.

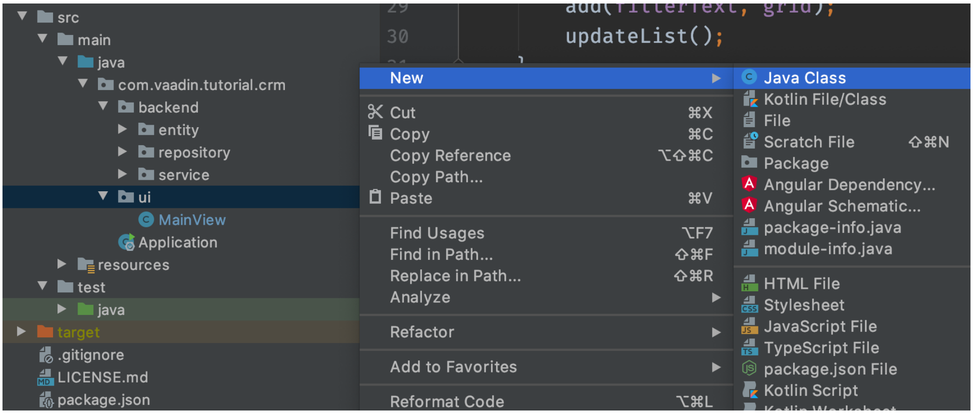
Creating a form component

Our form component will have:

- Text fields for the first and last name.
- An email field.
- Two select fields: one to select the company and the other to select the contact status.



To create the form, copy the code below and paste it into the `ui` package. IntelliJ will create a new Java class, `ContactForm`.



```
add(filterText, grid);
updateList();
```

ContactForm.java

```java
public class ContactForm extends FormLayout { ①

  TextField firstName = new TextField("First name"); ②
  TextField lastName = new TextField("Last name");
  EmailField email = new EmailField("Email");
  ComboBox<Contact.Status> status = new ComboBox<>("Status");
  ComboBox<Company> company = new ComboBox<>("Company");

  Button save = new Button("Save"); ③
  Button delete = new Button("Delete");
  Button close = new Button("Cancel");

  public ContactForm() {
    addClassName("contact-form"); ④
    add(firstName,
        lastName,
        email,
        company,
        status,
        createButtonsLayout()); ⑤
  }

  private HorizontalLayout createButtonsLayout() {
    save.addThemeVariants(ButtonVariant.LUMO_PRIMARY); ⑥
    delete.addThemeVariants(ButtonVariant.LUMO_ERROR);
    close.addThemeVariants(ButtonVariant.LUMO_TERTIARY);

    save.addClickShortcut(Key.ENTER); ⑦
    close.addClickShortcut(Key.ESCAPE);

    return new HorizontalLayout(save, delete, close); ⑧
  }
}
```

① `ContactForm` extends `FormLayout`: a responsive layout that shows form fields in 1 or 2 columns depending on viewport width.

② Creates all the UI components as fields in the component.

③ Uses the `com.vaadin.ui` import for `Button`, not the one from the `crud` package.

④ Gives the component a CSS class name so we can style it later

⑤ Adds all the UI components. The buttons require a bit of extra configuration so we create and call a new method, `createButtonsLayout()`.

⑥ Makes the buttons visually distinct from each other using built-in theme variants.

⑦ Defines keyboard shortcuts: `Enter` to save and `Escape` to close the editor

⑧ Returns a `HorizontalLayout` containing the buttons to place them next to each other.

Adding the form to main view

The next step is to add the form to the main view.

To do this, amend `MainView` as follows:

```java
public class MainView extends VerticalLayout {
    private ContactService contactService;

    private Grid<Contact> grid = new Grid<>(Contact.class);
    private TextField filterText = new TextField();
    private ContactForm form;  ①

    public MainView(ContactService contactService) {
        this.contactService = contactService;
        addClassName("list-view");
        setSizeFull();

        configureGrid();
        configureFilter();

        form = new ContactForm(); ②

        Div content = new Div(grid, form); ③
        content.addClassName("content");
        content.setSizeFull();

        add(filterText, content); ④
        updateList();

    }
    // other methods omitted.
}
```

① Creates a field for the form so you have access to it from other methods later on.

② Initialize the form in the constructor.

③ Creates a `Div` that wraps the `grid` and the `form`, gives it a CSS class name, and makes it full size.

④ Adds the `content` layout to the main layout.

Making the layout responsive

To make the layout responsive and usable on both mobile and desktop, we need to add CSS.

To do this, replace the content of `<project root>/frontend/shared-styles.css` with the following styles:

`shared-styles.css`

```css
/* List view */
.list-view .content {
    display: flex; ①
}

.list-view .contact-grid {
    flex: 2; ②
}

.list-view .contact-form {
    flex: 1;
    padding: var(--lumo-space-m);  ③
}

@media all and (max-width: 1100px) {  ④
    .list-view.editing .toolbar,
    .list-view.editing .contact-grid {
        display: none;
   }
}
```

① Uses CSS Flexbox to manage the layout

② Allocates 2/3 of the available width to the grid and 1/3 to the form.

③ Uses the Vaadin Lumo theme custom property, `--lumo-space-m`, to add standard padding in the form

④ Hides the toolbar and grid when editing on narrow screens (we'll add some logic to handle this shortly).

Importing CSS styles into main the view

Next, we load the CSS file by adding a `CssImport` annotation in `MainView`.

To add and load the new CSS styles:

1. Amend `MainView` as follows:

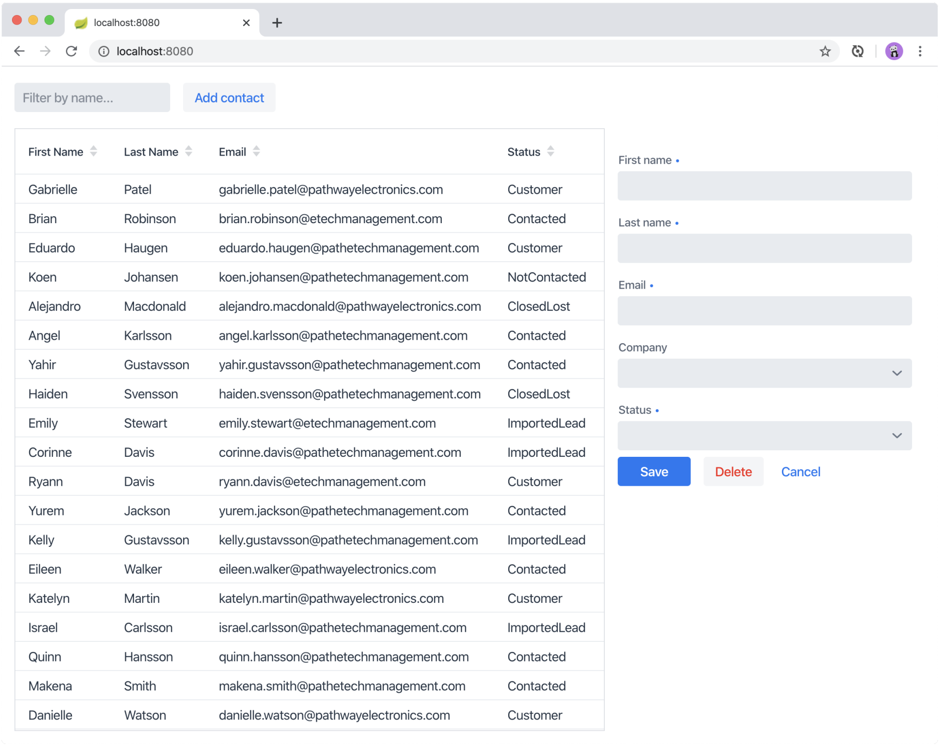
   `MainView.java`

   ```
   @Route("")
   @CssImport("./styles/shared-styles.css") ①
   public class MainView extends VerticalLayout {
       ...
   }
   ```

   ① The import path needs to be relative to the `frontend` folder

2. Stop and restart the server to ensure the CSS is loaded.

3. Verify that the main view looks the way it should. The form should now display next to the grid.



First Name	Last Name	Email	Status
Gabrielle	Patel	gabrielle.patel@pathwayelectronics.com	Customer
Brian	Robinson	brian.robinson@etechmanagement.com	Contacted
Eduardo	Haugen	eduardo.haugen@pathetechmanagement.com	Customer
Koen	Johansen	koen.johansen@pathetechmanagement.com	NotContacted
Alejandro	Macdonald	alejandro.macdonald@pathwayelectronics.com	ClosedLost
Angel	Karlsson	angel.karlsson@pathetechmanagement.com	Contacted
Yahir	Gustavsson	yahir.gustavsson@pathetechmanagement.com	Contacted
Haiden	Svensson	haiden.svensson@pathetechmanagement.com	ClosedLost
Emily	Stewart	emily.stewart@etechmanagement.com	ImportedLead
Corinne	Davis	corinne.davis@pathetechmanagement.com	ImportedLead
Ryann	Davis	ryann.davis@etechmanagement.com	Customer
Yurem	Jackson	yurem.jackson@pathetechmanagement.com	Contacted
Kelly	Gustavsson	kelly.gustavsson@pathetechmanagement.com	ImportedLead
Eileen	Walker	eileen.walker@pathwayelectronics.com	Contacted
Katelyn	Martin	katelyn.martin@pathwayelectronics.com	Customer
Israel	Carlsson	israel.carlsson@pathetechmanagement.com	ImportedLead
Quinn	Hansson	quinn.hansson@pathetechmanagement.com	Contacted
Makena	Smith	makena.smith@pathetechmanagement.com	Contacted
Danielle	Watson	danielle.watson@pathwayelectronics.com	Customer

The visual part of the form is now complete. In the next tutorial, we'll make it functional.

You can find the completed source code for this tutorial on GitHub.

Vaadin form data binding and validation

In the previous tutorial, we started building a reusable form component. In this tutorial, we define the form component API and use data binding to make the form functional.

You can download the completed source code at the bottom. The code from the previous tutorial chapter can be found here, if you want to jump directly into this chapter.

Binding object properties to input fields

A form is a collection of input fields that are connected to a data model, a `Contact` in this case. Forms validate user input and make it easy to get an object populated with input values from the UI.

Vaadin users the `Binder` class to build forms. It binds UI fields to data object fields by name. For instance, it takes a UI field named `firstName` and maps it to the `firstName` field of the data object, and the `lastName` field to the `lastName` field, and so on. This is why the field names in `Contact` and `ContactForm` are the same.

NOTE

Binder also supports an advanced API where you can configure data conversions and additional validation rules, but for this application, the simple API is sufficient.

Binder can use validation rules that are defined on the data object in the UI. This means you can run the same validations in both the browser and before saving to the database, without duplicating code.

Creating the binder

The first step is to create a new binder field in the contact form.

To do this, add the `BeanValidationBinder` field to `ContactForm` as follows:

`ContactForm.java`

```java
// Other fields omitted
Binder<Contact> binder = new BeanValidationBinder<>(Contact.class); ①

public ContactForm() {
    addClassName("contact-form");
    binder.bindInstanceFields(this); ②
    // Rest of constructor omitted
}
```

① `BeanValidationBinder` is a `Binder` that is aware of bean validation annotations. By passing it in the `Contact.class`, we define the type of object we are binding to.

② `bindInstanceFields` matches fields in `Contact` and `ContactForm` based on their names.

With these two lines of code, you've made the UI fields ready to be connected to a contact. We'll do that next.

Decoupling components

Object-oriented programming allows you to decouple objects and this helps to increase their reusability.

In our app, we create a form component and use it in the `MainView`. The most straightforward approach would appear to be to let the form call methods on `MainView` directly to save a contact. But what happens if you need the same form in another view? Or if you want to write a test to verify that the form works as intended? In both cases, the dependency on `MainView` makes it more complex than is necessary. Coupling a component to a specific parent typically makes it more difficult to reuse and test.

Instead, we should aim to make components that work in the same way as a `Button` component: you can use them anywhere. You configure the component by setting properties, and it notifies you of events through listeners.

Creating a reusable component is as simple as making sure it can be configured through setters, and that it fires events whenever something happens. Using the component should not have side effects, for instance it shouldn't change anything in the database by itself.

Defining the form component API

With the visual part of the component complete, the next step is to define the form component API. This is how developers interact with the form.

A good rule of thumb when designing an API for a reusable component is: ***properties in, events out***. Users should be able to fully configure a component by setting properties.They should be notified of all relevant events, without the need to manually call getters to see if things have changed.

With this in mind, here's what our API will cover:

Properties in:

- Set the contact.
- Set the list of companies.

Events out:

- Save.
- Delete.
- Close.

Setting the company and contact status

The two properties that need to be handled are the contact shown in the form and the list of companies shown in the dropdown.

To set the company and contact status:

1. In `ContactForm`, add the company list as a constructor parameter. We do this first, because the company list is needed before a contact can be edited.

   ContactForm.java

   ```
   public ContactForm(List<Company> companies) { ①
     addClassName("contact-form");
     binder.bindInstanceFields(this);

     company.setItems(companies); ②
     company.setItemLabelGenerator(Company::getName); ③
     status.setItems(Contact.Status.values()); ④

      //omitted
   }
   ```

   ① Adds a list of `Company` objects as a parameter to the constructor.

   ② Sets the list of `companies` as the items in the company combo box.

   ③ Tells the combo box to use the name of the company as the display value.

   ④ Populates the status dropdown with the values from the `Contact.Status` enum.

   WARNING You will get a compilation error if you build the application at this point. This is because you have not yet passed a list of companies in `MainView`.

2. In `MainView`, update the constructor to take `CompanyService` as a parameter, and then use this service to pass a list of all companies.

`MainView.java`

```java
public MainView(ContactService contactService,
                CompanyService companyService) { ①
    this.contactService = contactService;
    addClassName("list-view");
    setSizeFull();

    configureGrid();
    configureFilter();

    form = new ContactForm(companyService.findAll()); ②

    add(filterText, grid, form);
    updateList();
}
```

① Auto wires (injects) `CompanyService` as a constructor parameter.

② Finds all companies and passes them to `ContactForm`.

Updating the contact

Next, we need to create a setter for the `contact` because it can change over time as a user browses through the contacts.

To do this, add the following in the `ContactForm` class:

`ContactForm.java`

```java
public class ContactForm extends FormLayout {
    private Contact contact;

    // other methods and fields omitted

    public void setContact(Contact contact) {
        this.contact = contact; ①
        binder.readBean(contact); ②
    }
}
```

① Save a reference to the contact so we can save the form values back into it later.

② Calls `binder.readBean` to bind the values from the contact to the UI fields. `readBen` copies the values from the Contact to an internal model, that way we don't accidentally overwrite values if we cancel editing.

Setting up events

Vaadin comes with an event-handling system for components. We've already used it to listen to value-change events from the filter text field. We want the form component to have a similar way of informing parents of events.

To do this, add the following at the end of the `ContactForm` class:

ContactForm.java

```java
// Events
public static abstract class ContactFormEvent extends ComponentEvent<ContactForm> {
  private Contact contact;

  protected ContactFormEvent(ContactForm source, Contact contact) { ①
    super(source, false);
    this.contact = contact;
  }

  public Contact getContact() {
    return contact;
  }
}

public static class SaveEvent extends ContactFormEvent {
  SaveEvent(ContactForm source, Contact contact) {
    super(source, contact);
  }
}

public static class DeleteEvent extends ContactFormEvent {
  DeleteEvent(ContactForm source, Contact contact) {
    super(source, contact);
  }

}

public static class CloseEvent extends ContactFormEvent {
  CloseEvent(ContactForm source) {
    super(source, null);
  }
}

public <T extends ComponentEvent<?>> Registration addListener(Class<T> eventType,
    ComponentEventListener<T> listener) { ②
  return getEventBus().addListener(eventType, listener);
}
```

① `ContactFormEvent` is a common superclass for all the events. It contains the `contact` that was edited or deleted.

② The `addListener` method uses Vaadin's event bus to register the custom event types.

Select the `com.vaadin` import for `Registration` if IntelliJ asks.

Saving, deleting, and closing the form

With the event types defined, we can now inform anyone using `ContactForm` of relevant events.

To add `save`, `delete` and `close` event listeners, add the following to the `ContactForm` class:

`ContactForm.java`

```java
private Component createButtonsLayout() {
  // omitted

  save.addClickListener(event -> validateAndSave()); ①
  delete.addClickListener(event -> fireEvent(new DeleteEvent(this, contact))); ②
  close.addClickListener(event -> fireEvent(new CloseEvent(this))); ③


  binder.addStatusChangeListener(e -> save.setEnabled(binder.isValid())); ④
  return new HorizontalLayout(save, delete, close);
}

private void validateAndSave() {
  try {
    binder.writeBean(contact); ⑤
    fireEvent(new SaveEvent(this, contact)); ⑥
  } catch (ValidationException e) {
    e.printStackTrace();
  }
}
```

① The save button calls the `validateAndSave` method

② The delete button fires a delete event and passes the currently-edited contact.

③ The cancel button fires a close event.

④ Validates the form every time it changes. If it is invalid, it disables the save button to avoid invalid submissions.

⑤ Write the form contents back to the original `contact`.

⑥ Fire a save event so the parent component can handle the action.

In the next tutorial, we'll connect the form to the main view so that the selected contact in the form can be edited.

You can find the completed source code for this tutorial on GitHub.

Passing data and events between Vaadin components

In the previous tutorial, we created a reusable form component to edit contacts. In this tutorial we'll hook it up to our application. Our form will:

- Show the contact currently-selected in the grid.
- Be hidden when no contact is selected.
- Save and delete contacts in the database.

You can download the completed source code at the bottom. The code from the previous tutorial chapter can be found here, if you want to jump directly into this chapter.

Showing selected contacts in the form

The first step is to show the selected grid row in the form. This is also known as creating a master-detail view.

To do this, amend `MainView` as follows:

MainView.java

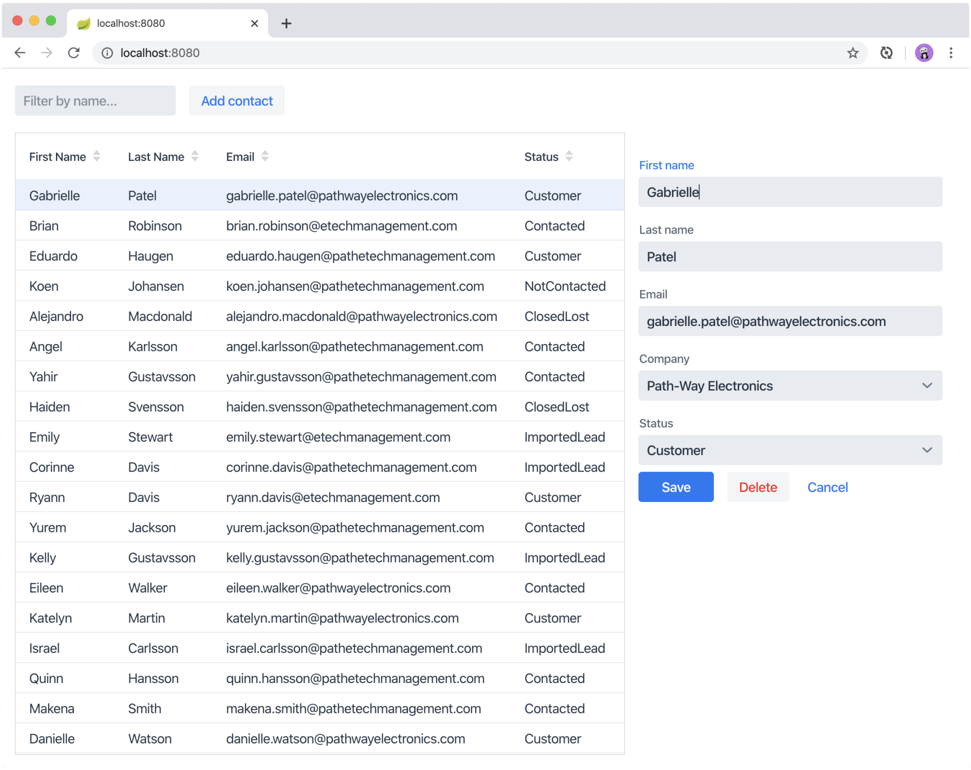
```java
public MainView(ContactService contactService,
                CompanyService companyService) {
    //omitted
    closeEditor(); ①
}

private void configureGrid() {
  // Omitted

  grid.asSingleSelect().addValueChangeListener(event -> ②
      editContact(event.getValue()));
}

public void editContact(Contact contact) { ③
    if (contact == null) {
        closeEditor();
    } else {
        form.setContact(contact);
        form.setVisible(true);
        addClassName("editing");
    }
}

private void closeEditor() {
    form.setContact(null);
    form.setVisible(false);
    removeClassName("editing");
}

// Remaining methods omitted
```

① The `closeEditor()` call at the end of the constructor Sets the form contact to `null`, clearing out old values. Hides the form. Removes the `"editing"` CSS class from the view.

② `addValueChangeListener` adds a listener to the grid. The `Grid` component supports multi and single-selection modes. We only want to select a single `Contact`, so we use the `asSingleSelect()` method. The `getValue()` method returns the `Contact` in the selected row or null if there's no selection.

③ `editContact` sets the selected contact in the `ContactForm` and hides or shows the form, depending on the selection. It also sets the `"editing"` CSS class name when editing.

Build the application. You should now be able to select contacts in the grid and see them in the form. But, none of the buttons work yet.



First Name	Last Name	Email	Status
Gabrielle	Patel	gabrielle.patel@pathwayelectronics.com	Customer
Brian	Robinson	brian.robinson@etechmanagement.com	Contacted
Eduardo	Haugen	eduardo.haugen@pathetechmanagement.com	Customer
Koen	Johansen	koen.johansen@pathetechmanagement.com	NotContacted
Alejandro	Macdonald	alejandro.macdonald@pathwayelectronics.com	ClosedLost
Angel	Karlsson	angel.karlsson@pathetechmanagement.com	Contacted
Yahir	Gustavsson	yahir.gustavsson@pathetechmanagement.com	Contacted
Haiden	Svensson	haiden.svensson@pathetechmanagement.com	ClosedLost
Emily	Stewart	emily.stewart@etechmanagement.com	ImportedLead
Corinne	Davis	corinne.davis@pathetechmanagement.com	ImportedLead
Ryann	Davis	ryann.davis@etechmanagement.com	Customer
Yurem	Jackson	yurem.jackson@pathetechmanagement.com	Contacted
Kelly	Gustavsson	kelly.gustavsson@pathetechmanagement.com	ImportedLead
Eileen	Walker	eileen.walker@pathwayelectronics.com	Contacted
Katelyn	Martin	katelyn.martin@pathwayelectronics.com	Customer
Israel	Carlsson	israel.carlsson@pathetechmanagement.com	ImportedLead
Quinn	Hansson	quinn.hansson@pathetechmanagement.com	Contacted
Makena	Smith	makena.smith@pathetechmanagement.com	Contacted
Danielle	Watson	danielle.watson@pathwayelectronics.com	Customer

First name: Gabrielle
Last name: Patel
Email: gabrielle.patel@pathwayelectronics.com
Company: Path-Way Electronics
Status: Customer

Handling form events We designed the `ContactForm` API to be reusable: it is configurable through properties and it fires the necessary events. So far, we've passed a list of companies and the contact to the form. Now all we need to do is listen for the events to complete the integration.

To handle event listeners, amend `MainView` as follows:

MainView.java

```java
public MainView(ContactService contactService,
                CompanyService companyService) {
    this.contactService = contactService;
    addClassName("list-view");
    setSizeFull();

    configureGrid();
    configureFilter();

    form = new ContactForm(companyService.findAll());
    form.addListener(ContactForm.SaveEvent.class, this::saveContact); ①
    form.addListener(ContactForm.DeleteEvent.class, this::deleteContact); ②
    form.addListener(ContactForm.CloseEvent.class, e -> closeEditor()); ③


    Div content = new Div(grid, form);
    content.addClassName("content");
    content.setSizeFull();

    add(filterText, content);
    updateList();
    closeEditor();
}

private void saveContact(ContactForm.SaveEvent event) {
    contactService.save(event.getContact());
    updateList();
    closeEditor();
}

private void deleteContact(ContactForm.DeleteEvent event) {
    contactService.delete(event.getContact());
    updateList();
    closeEditor();
}
```

① Save calls `saveContact`, it:

a. Uses contactService to save the contact in the event to the database.

b. Updates the list.

c. Closes the editor.

② Delete calls `deleteContact`, it:

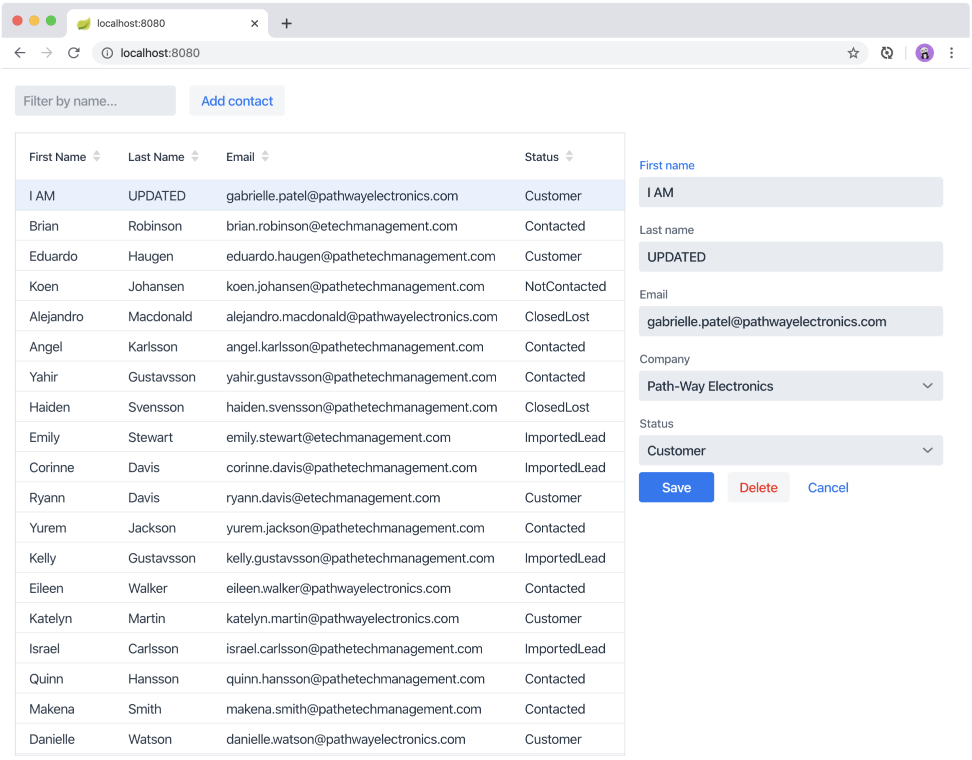
a. Uses `contactService` to delete the contact from the database.

b. Updates the list.

c. Closes the editor.

③ Close closes the editor.

Build the application and verify that you are now able to update and delete contacts.



First Name	Last Name	Email	Status
I AM	UPDATED	gabrielle.patel@pathwayelectronics.com	Customer
Brian	Robinson	brian.robinson@etechmanagement.com	Contacted
Eduardo	Haugen	eduardo.haugen@pathetechmanagement.com	Customer
Koen	Johansen	koen.johansen@pathetechmanagement.com	NotContacted
Alejandro	Macdonald	alejandro.macdonald@pathwayelectronics.com	ClosedLost
Angel	Karlsson	angel.karlsson@pathetechmanagement.com	Contacted
Yahir	Gustavsson	yahir.gustavsson@pathetechmanagement.com	Contacted
Haiden	Svensson	haiden.svensson@pathetechmanagement.com	ClosedLost
Emily	Stewart	emily.stewart@etechmanagement.com	ImportedLead
Corinne	Davis	corinne.davis@pathetechmanagement.com	ImportedLead
Ryann	Davis	ryann.davis@etechmanagement.com	Customer
Yurem	Jackson	yurem.jackson@pathetechmanagement.com	Contacted
Kelly	Gustavsson	kelly.gustavsson@pathetechmanagement.com	ImportedLead
Eileen	Walker	eileen.walker@pathwayelectronics.com	Contacted
Katelyn	Martin	katelyn.martin@pathwayelectronics.com	Customer
Israel	Carlsson	israel.carlsson@pathetechmanagement.com	ImportedLead
Quinn	Hansson	quinn.hansson@pathetechmanagement.com	Contacted
Makena	Smith	makena.smith@pathetechmanagement.com	Contacted
Danielle	Watson	danielle.watson@pathwayelectronics.com	Customer

First name: I AM
Last name: UPDATED
Email: gabrielle.patel@pathwayelectronics.com
Company: Path-Way Electronics
Status: Customer

Adding new contacts

The final step is to add a button to add new contacts. We'll position the button next to the filter field.

1. In `MainView`, create a `HorizontalLayout` that wraps the text field and the button, rename the `configureFilter` method to `getToolbar`, and replace its contents, as follows:

MainView.java

```java
private HorizontalLayout getToolbar() { ①
    filterText.setPlaceholder("Filter by name...");
    filterText.setClearButtonVisible(true);
    filterText.setValueChangeMode(ValueChangeMode.LAZY);
    filterText.addValueChangeListener(e -> updateList());

    Button addContactButton = new Button("Add contact");
    addContactButton.addClickListener(click -> addContact()); ②

    HorizontalLayout toolbar = new HorizontalLayout(filterText, addContactButton
); ③
    toolbar.addClassName("toolbar");
    return toolbar;
  }
```

① Returns a `HorizontalLayout`.

② The `"Add contact"` button calls `addContact` when clicked.

③ Adds a `HorizontalLayout` with the filter input field and a button, gives it a CSS class name `"toolbar"` that is used for the responsive layouting.

2. Define the `addContact()` method as follows:

MainView.java

```java
void addContact() {
    grid.asSingleSelect().clear(); ①
    editContact(new Contact()); ②
}
```

① Deselects the grid so that a previously selected `Contact` is no longer highlighted when the user adds a new contact.

② Creates a new `Contact` and passes it to `editContact`.

3. Update the `MainView` constructor to use the new toolbar as follows:

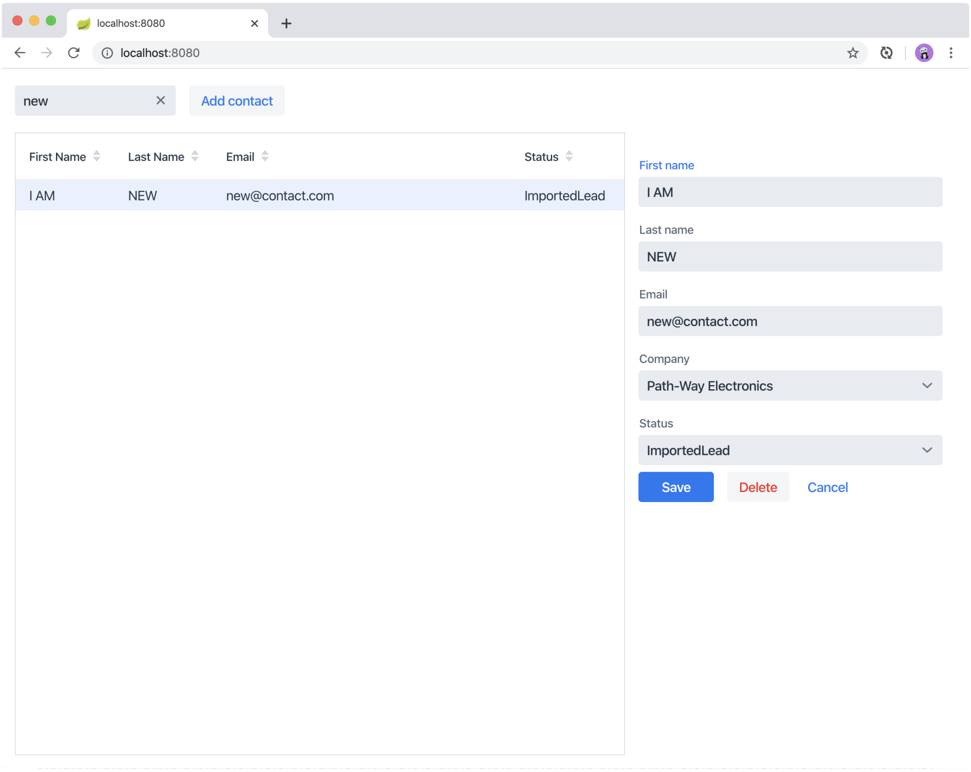
MainView.java

```
public MainView(ContactService contactService,
                  CompanyService companyService) {
      this.contactService = contactService;
      addClassName("list-view");
      setSizeFull();
①
      configureGrid();


      form = new ContactForm(companyService.findAll());
      form.addListener(ContactForm.SaveEvent.class, this::saveContact);
      form.addListener(ContactForm.DeleteEvent.class, this::deleteContact);
      form.addListener(ContactForm.CloseEvent.class, e -> this.closeEditor());
      closeEditor();

      Div content = new Div(grid, form);
      content.addClassName("content");
      content.setSizeFull();

      add(getToolbar(), content); ②
      updateList();
  }
```

① Removes the `configureFilter()` method call.

② Replaces the `filterText` component with a call to `getToolbar()`.

Build the application and verify that you are now able to add new contacts. New contacts are added at the end of the list, so you may need to scroll or use the filter to find them.



First Name	Last Name	Email	Status
I AM	NEW	new@contact.com	ImportedLead

First name: I AM
Last name: NEW
Email: new@contact.com
Company: Path-Way Electronics
Status: ImportedLead

In the next tutorial, we'll add a second screen to the application and learn how to navigate between views.

You can find the completed source code for this tutorial on GitHub.

Navigating between views in Vaadin

So far in this tutorial series, we've built a CRM application for listing and editing contacts. In this chapter, we add a dashboard view to the application. We also add a responsive application layout, with a header and a navigation sidebar that can be toggled on small screens.

You can download the completed source code at the bottom. The code from the previous tutorial chapter can be found here, if you want to jump directly into this chapter.

Defining routes

Any Vaadin component can be made a navigation target by adding an `@Route("<path>")` annotation. Routes can be nested by defining the parent layout in the annotation: `@Route(value = "list", parent=MainView.class)`.

Creating the parent list and child view

We want our application to have:

- A shared parent layout with two child views:

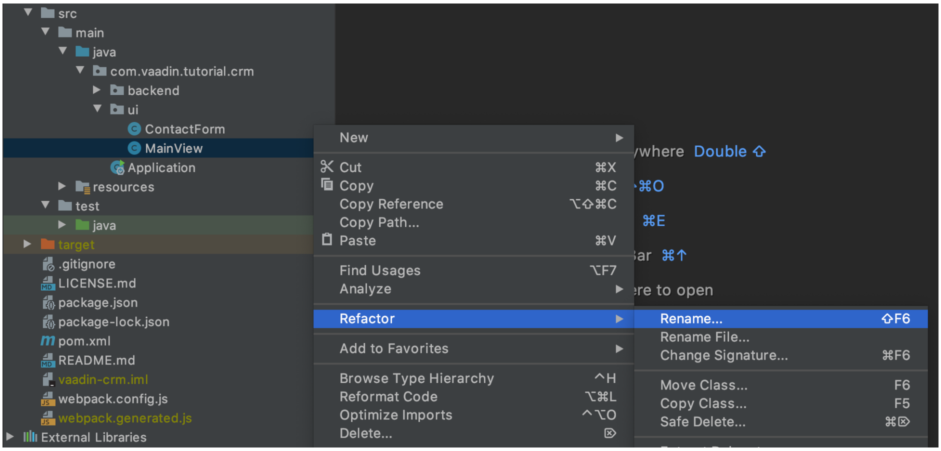
  `MainLayout`: AppLayout with header and navigation:

  a. `ListView`: The default view, mapped to `""`.

  b. `DashboardView`: Mapped to `"dashboard"`.
- A responsive app layout and navigation links.

Let's begin:

1. Start by renaming `MainView` to `ListView`. Right click `MainView` and select **Refactor > Rename**. When IntelliJ asks if you want to rename the file, answer yes.



2. Create a new Java class named `MainLayout` with the following content. This is the shared parent layout of both views in the application.

MainLayout.java

```
@CssImport("./styles/shared-styles.css") ①
public class MainLayout extends AppLayout { ②
    public MainLayout() {
        createHeader();
        createDrawer();
    }

    private void createHeader() {
        H1 logo = new H1("Vaadin CRM");
        logo.addClassName("logo");

        HorizontalLayout header = new HorizontalLayout(new DrawerToggle(), logo);
③

        header.setDefaultVerticalComponentAlignment(
             FlexComponent.Alignment.CENTER); ④
        header.setWidth("100%");
        header.addClassName("header");


        addToNavbar(header); ⑤

    }

    private void createDrawer() {
        RouterLink listLink = new RouterLink("List", ListView.class); ⑥
        listLink.setHighlightCondition(HighlightConditions.sameLocation()); ⑦

        addToDrawer(new VerticalLayout(listLink)); ⑧
    }
}
```

① Adds the `@CssImport` annotation to `MainLayout`. We'll remove it from `ListView` later.

② `AppLayout` is a Vaadin layout with a header and a responsive drawer.

③ `DrawerToggle` is a menu button that toggles the visibility of the sidebar.

④ Centers the components in the `header` along the vertical axis.

⑤ Adds the `header` layout to the app layout's nav bar.

⑥ Creates a `RouterLink` with the text "List" and `ListView.class` as the destination view

⑦ Sets setHighlightCondition(HighlightConditions.sameLocation()) to avoid highlighting the link for partial route matches. (Technically, every route starts with an empty route, so without this it would always show up as active even though the user is not on the view)

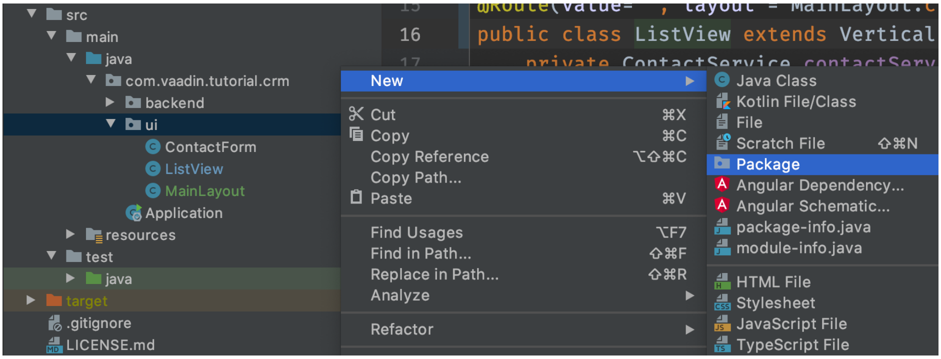
⑧ Wraps the link in a `VerticalLayout` and adds it to the `AppLayout's drawer

3. Add the following CSS to `frontend/styles/shared-styles.css`

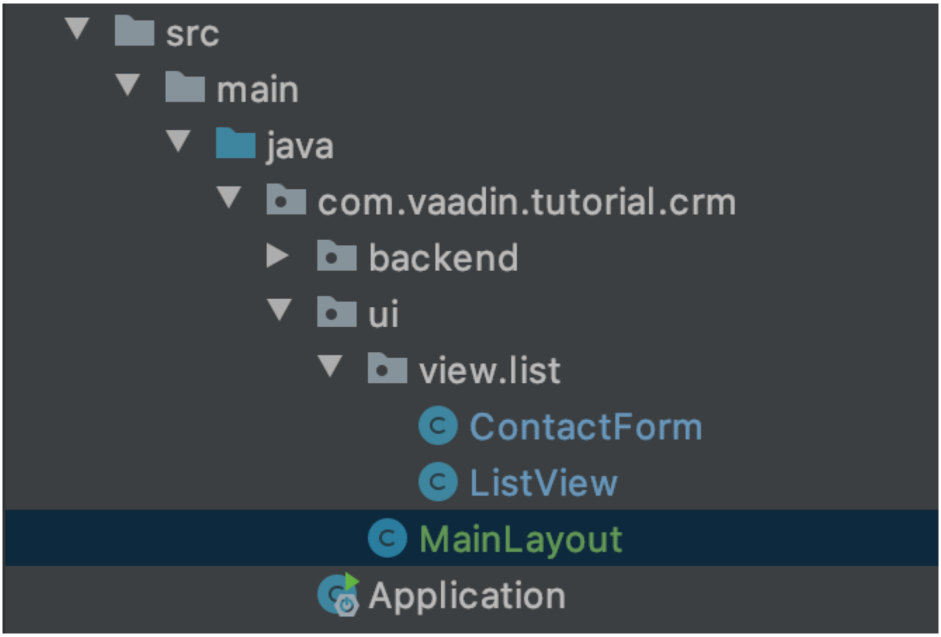
shared-styles.css

```
/* Main layout */

a[highlight] {
    font-weight: bold;
    text-decoration: underline;
}

.header {
    padding: 0 var(--lumo-space-m);
}

.header h1.logo {
    font-size: 1em;
    margin: var(--lumo-space-m);
}
```

4. Create a new package for the list view, `com.vaadin.tutorial.crm.ui.view.list`. Separate packages for each view makes it easier to keep our project organized.



5. Move `ListView` and `ContactForm` into the new package. The resulting package structure should look like this:



6. Finally, in `ListView` update the `@Route` mapping to use the new `MainLayout` and delete the `@CSSImport` annotation.

`ListView.java`

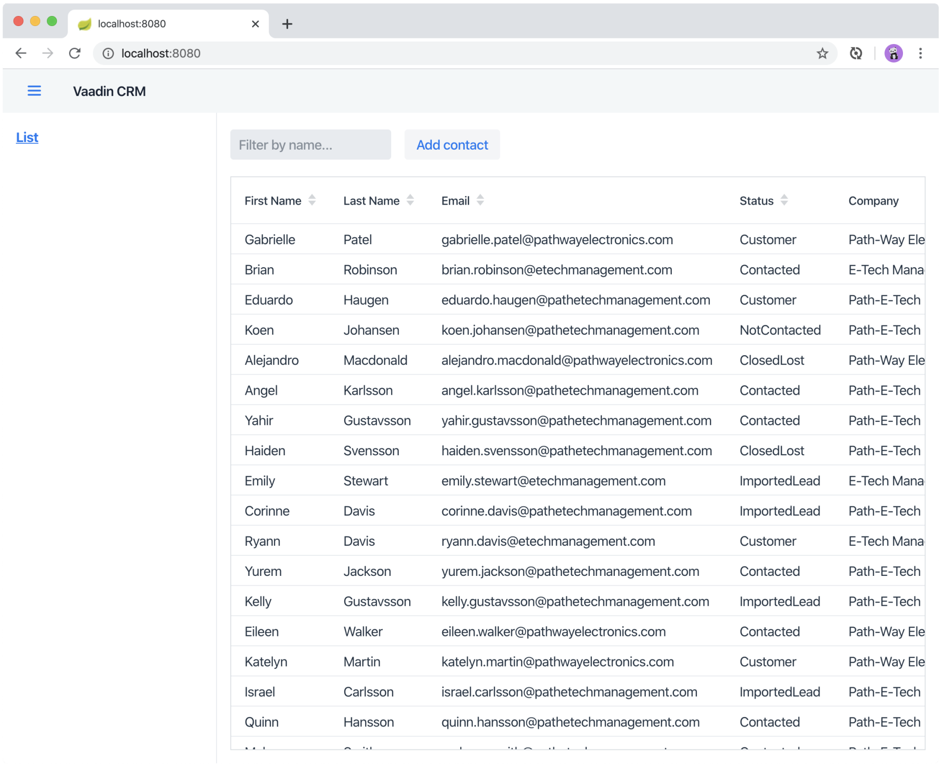
```
@Route(value="", layout = MainLayout.class) ①
②
@PageTitle("Contacts | Vaadin CRM") ③
public class ListView extends VerticalLayout {
    ...
}
```

① `ListView` still matches the empty path, but now uses `MainLayout` as its parent.

② The @CSSImport annotation is now removed, as it is now on `MainLayout` instead.

③ Adds a title to the page.

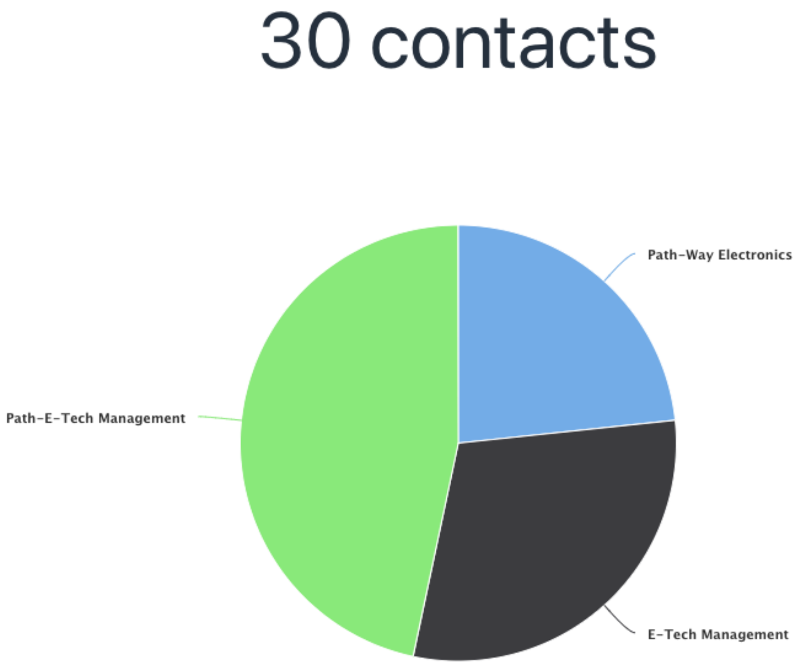
7. Run the application. You should now see a header and a sidebar on the list view.



First Name	Last Name	Email	Status	Company
Gabrielle	Patel	gabrielle.patel@pathwayelectronics.com	Customer	Path-Way Ele
Brian	Robinson	brian.robinson@etechmanagement.com	Contacted	E-Tech Mana
Eduardo	Haugen	eduardo.haugen@pathetechmanagement.com	Customer	Path-E-Tech
Koen	Johansen	koen.johansen@pathetechmanagement.com	NotContacted	Path-E-Tech
Alejandro	Macdonald	alejandro.macdonald@pathwayelectronics.com	ClosedLost	Path-Way Ele
Angel	Karlsson	angel.karlsson@pathetechmanagement.com	Contacted	Path-E-Tech
Yahir	Gustavsson	yahir.gustavsson@pathetechmanagement.com	Contacted	Path-E-Tech
Haiden	Svensson	haiden.svensson@pathetechmanagement.com	ClosedLost	Path-E-Tech
Emily	Stewart	emily.stewart@etechmanagement.com	ImportedLead	E-Tech Mana
Corinne	Davis	corinne.davis@pathetechmanagement.com	ImportedLead	Path-E-Tech
Ryann	Davis	ryann.davis@etechmanagement.com	Customer	E-Tech Mana
Yurem	Jackson	yurem.jackson@pathetechmanagement.com	Contacted	Path-E-Tech
Kelly	Gustavsson	kelly.gustavsson@pathetechmanagement.com	ImportedLead	Path-E-Tech
Eileen	Walker	eileen.walker@pathwayelectronics.com	Contacted	Path-Way Ele
Katelyn	Martin	katelyn.martin@pathwayelectronics.com	Customer	Path-Way Ele
Israel	Carlsson	israel.carlsson@pathetechmanagement.com	ImportedLead	Path-E-Tech
Quinn	Hansson	quinn.hansson@pathetechmanagement.com	Contacted	Path-E-Tech

Creating the dashboard view

Now let's create a new dashboard view. This view will show stats: the number of contacts in the system and a pie chart of the number of contacts per company.



1. Create a new package `com.vaadin.tutorial.crm.ui.view.dashboard` by right clicking the list package and selecting **New > Package**.
2. In the new package, create a new Java class named `DashboardView`.

DashboardView.java

```java
package com.vaadin.tutorial.crm.ui.view.dashboard;

import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.router.Route;
import com.vaadin.tutorial.crm.backend.service.CompanyService;
import com.vaadin.tutorial.crm.backend.service.ContactService;
import com.vaadin.tutorial.crm.ui.MainLayout;

@Route(value = "dashboard", layout = MainLayout.class) ①
@PageTitle("Dashboard | Vaadin CRM") ②
public class DashboardView extends VerticalLayout {

    private ContactService contactService;
    private CompanyService companyService;

    public DashboardView(ContactService contactService, CompanyService
companyService) { ③
        this.contactService = contactService;
        this.companyService = companyService;
        addClassName("dashboard-view");
        setDefaultHorizontalComponentAlignment(Alignment.CENTER); ④
    }
}
```

① `DashboardView` is mapped to the `"dashboard"` path and uses `MainLayout` as a parent layout.

② Sets the page title.

③ Takes both `ContactService` and `CompanyService` as constructor parameters and save them as fields.

④ Centers the contents of the layout.

3. Create a method to display the number of contacts in the system.

DashboardView.java

```java
private Component getContactStats() {
    Span stats = new Span(contactService.count() + " contacts"); ①
    stats.addClassName("contact-stats");
    return stats;
}
```

① contactService.count() gives us the number of contacts in the database. It returns a `Span` with the count and a text explanation.

4. Add the following CSS to `frontend/styles/shared-styles.css`

shared-styles.css

```css
/* Dashboard view */

.dashboard-view .contact-stats {
    font-size: 4em;
    margin: 1em 0;
}
```

5. In `CompanyService`, add the following method to create the pie chart. As an alternative, you could calculate the number of employees per company right in the view, but it's better to move this logic into `CompanyService` so it can be reused later in other views.

> **NOTE** Vaadin charts is a collection of data visualization components that is a part of the Vaadin Vaadin Pro subscription. Vaadin charts comes with a free trial that you can activate in the browser. All Vaadin Pro tools and components are free for students through the GitHub Student Developer Pack. You can skip the chart if you only want to use free components.

CompanyService.java

```java
public Map<String, Integer> getStats() {
  HashMap<String, Integer> stats = new HashMap<>();
  findAll().forEach(company -> stats.put(company.getName(), company.getEmployees
().size())); ①
  return stats;
}
```

① Loops through each company and returns a `Map` containing the company name and number of employees.

6. In `DashboardView`, create a method to construct the chart:

DashboardView.java

```java
private Chart getCompaniesChart() {
    Chart chart = new Chart(ChartType.PIE); ①

    DataSeries dataSeries = new DataSeries(); ②
    Map<String, Integer> companies = companyService.getStats();
    companies.forEach((company, employees) ->
        dataSeries.add(new DataSeriesItem(company, employees))); ③
    chart.getConfiguration().setSeries(dataSeries); ④
    return chart;
}
```

① Creates a new pie chart.

② Charts use a DataSeries for data.

③ Adds a DataSeriesItem, containing the company name and number of employees, for each company.

④ Sets the data series to the chart configuration.

7. Add both components to the `DashboadView` in the constructor to display the company stats.

   `DashboardView.java`

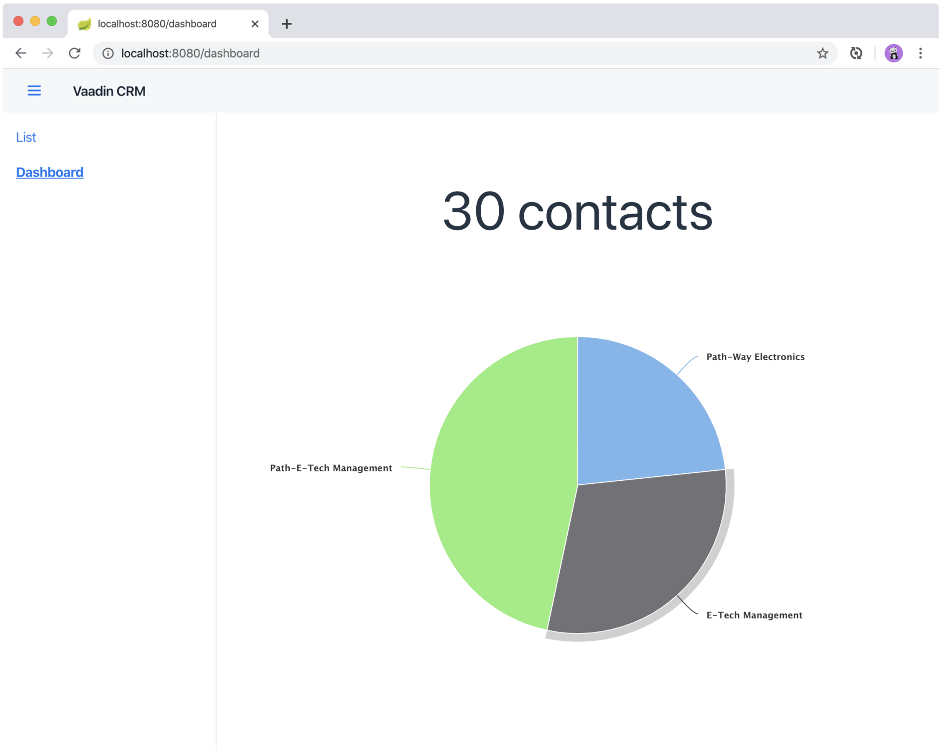
```
public DashboardView(ContactService contactService, CompanyService
companyService) {
    this.contactService = contactService;
    this.companyService = companyService;

    add(getContactStats(), getCompaniesChart());
}
```

8. Add a navigation link to `DashboardView` in the `MainLayout` drawer:

   `MainLayout.java`

```
private void createDrawer() {
    RouterLink listLink = new RouterLink("List", ListView.class);
    listLink.setHighlightCondition(HighlightConditions.sameLocation());

    addToDrawer(new VerticalLayout(
        listLink,
        new RouterLink("Dashboard", DashboardView.class)
    ));
}
```

9. Build and run the application. You should now be able to navigate to the dashboard view and see stats on your CRM contacts. If you want to, go ahead and add or remove contacts in the list view to see that the dashboard reflects your changes.



In the next tutorial, we'll secure the application by adding a login screen.

You can find the completed source code for this tutorial on GitHub.

Adding a login screen to a Vaadin app with Spring Security

So far in this tutorial series, we've built a CRM application that has one view for listing and editing contacts, and a dashboard view for showing stats.

In this tutorial we set up Spring Security and add a login screen to limit access to logged in users.

You can download the completed source code at the bottom. The code from the previous tutorial chapter can be found here, if you want to jump directly into this chapter.

Creating a login view

1. Start by creating a new package `com.vaadin.tutorial.crm.ui.view.login`.
2. Create a new class, `LoginView`, in the new package.

LoginView.java

```java
package com.vaadin.tutorial.crm.ui.view.login;

import com.vaadin.flow.component.html.H1;
import com.vaadin.flow.component.login.LoginForm;
import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.router.BeforeEnterEvent;
import com.vaadin.flow.router.BeforeEnterObserver;
import com.vaadin.flow.router.PageTitle;
import com.vaadin.flow.router.Route;

import java.util.Collections;

@Route("login") ①
@PageTitle("Login | Vaadin CRM")

public class LoginView extends VerticalLayout implements BeforeEnterObserver {

    private LoginForm login = new LoginForm(); ②

    public LoginView(){
        addClassName("login-view");
        setSizeFull();
        setAlignItems(Alignment.CENTER); ③
        setJustifyContentMode(JustifyContentMode.CENTER);

        login.setAction("login");  ④

        add(new H1("Vaadin CRM"), login);
    }

    @Override
    public void beforeEnter(BeforeEnterEvent beforeEnterEvent) {
        // inform the user about an authentication error
        if(beforeEnterEvent.getLocation() ⑤
        .getQueryParameters()
        .getParameters()
        .containsKey("error")) {
            login.setError(true);
        }
    }
}
```

① Maps the view to the `"login"` path. `LoginView` should take up the whole browser window, so we don't use `MainLayout` as the parent.

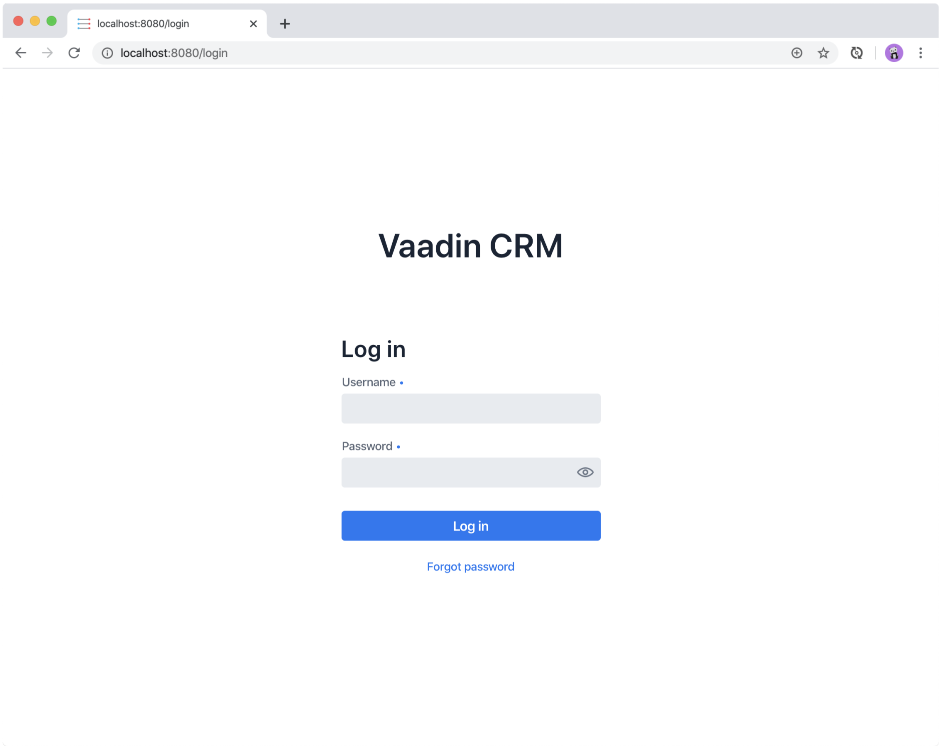
② Instantiates a `LoginForm` component to capture username and password.

③ Makes `LoginView` full size and centers its content both horizontally and vertically, by calling setAlignItems(Alignment.CENTER) and setJustifyContentMode(JustifyContentMode.CENTER).

④ Sets the `LoginForm` action to `"login"` to post the login form to Spring Security.

⑤ Reads query parameters and shows an error if a login attempt fails.

3. Build the application and navigate to http://localhost/login. You should see a centered login form.



Setting up Spring Security to handle logins

With the login screen in place, we now need to configure Spring Security to perform the authentication and to prevent unauthorized users from accessing views.

Installing Spring Security dependencies

1. Start by adding the following 2 dependencies in your `pom.xml`:

`pom.xml`

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
</dependency>
```

2. Check that the dependencies are downloaded. If you enabled automatic downloads in an earlier tutorial, you're all set. If you didn't, or are unsure, run `mvn install` from the command line to download the dependencies.

3. Next, disable Spring MVC auto configuration on the `Application` class, as this interferes with how Vaadin works and can cause strange reloading behavior.

`Application.class`

```
@SpringBootApplication(exclude = ErrorMvcAutoConfiguration.class)
public class Application extends SpringBootServletInitializer {
    ...
}
```

Configuring Spring Security

1. Create a new package `com.vaadin.tutorial.crm.security` for classes related to security.

2. In the new package create the following classes using the code detailed below:

   - `SecurityUtils`: Utility methods.
   - `CustomRequestCache`: A cache to keep track of unauthenticated requests.
   - `SecurityConfiguration`: Spring Security configuration.

TIP	Paste the class code into the package and IntelliJ will automatically create the class for you.

   a. SecurityUtils

SecurityUtils.java

```
package com.vaadin.tutorial.crm.security;

import com.vaadin.flow.server.ServletHelper;
import com.vaadin.flow.shared.ApplicationConstants;
import
org.springframework.security.authentication.AnonymousAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;

import javax.servlet.http.HttpServletRequest;
import java.util.stream.Stream;

public final class SecurityUtils {

    private SecurityUtils() {
        // Util methods only
    }

    static boolean isFrameworkInternalRequest(HttpServletRequest request) {
①
        final String parameterValue = request.getParameter
(ApplicationConstants.REQUEST_TYPE_PARAMETER);
        return parameterValue != null
            && Stream.of(ServletHelper.RequestType.values())
            .anyMatch(r -> r.getIdentifier().equals(parameterValue));
    }

    static boolean isUserLoggedIn() { ②
        Authentication authentication = SecurityContextHolder.getContext()
.getAuthentication();
        return authentication != null
            && !(authentication instanceof AnonymousAuthenticationToken)
            && authentication.isAuthenticated();
    }
}
```

① `isFrameworkInternalRequest` determines if a request is internal to Vaadin.

② `isUserLoggedIn` checks if the current user is logged in.

b. CustomRequestCache

CustomRequestCache.java

```
package com.vaadin.tutorial.crm.security;

import
org.springframework.security.web.savedrequest.HttpSessionRequestCache;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

class CustomRequestCache extends HttpSessionRequestCache {

    @Override
    public void saveRequest(HttpServletRequest request, HttpServletResponse
response) { ①
        if (!SecurityUtils.isFrameworkInternalRequest(request)) {
            super.saveRequest(request, response);
        }
    }

}
```

① Saves unauthenticated requests so we can redirect the user to the page they were trying to access once they're logged in.

c. SecurityConfiguration

SecurityConfiguration.java

```
package com.vaadin.tutorial.crm.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.builders.WebSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebS
ecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurit
yConfigurerAdapter;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.provisioning.InMemoryUserDetailsManager;


@EnableWebSecurity ①
@Configuration ②
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    private static final String LOGIN_PROCESSING_URL = "/login";
    private static final String LOGIN_FAILURE_URL = "/login?error";
    private static final String LOGIN_URL = "/login";
    private static final String LOGOUT_SUCCESS_URL = "/login";

}
```

① `@EnableWebSecurity` turns on Spring Security for the application.

② `@Configuration` tells Spring Boot to use this class for configuring security.

3. Add a method to block unauthenticated requests to all pages, except the login page.

SecurityConfiguration.java

```java
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable()  ①
        .requestCache().requestCache(new CustomRequestCache()) ②
        .and().authorizeRequests() ③
        .requestMatchers(SecurityUtils::isFrameworkInternalRequest).permitAll()
④

        .anyRequest().authenticated()  ⑤

        .and().formLogin()  ⑥
        .loginPage(LOGIN_URL).permitAll()
        .loginProcessingUrl(LOGIN_PROCESSING_URL)  ⑦
        .failureUrl(LOGIN_FAILURE_URL)
        .and().logout().logoutSuccessUrl(LOGOUT_SUCCESS_URL); ⑧
}
```

① Disables cross-site request forgery (CSRF) protection, as Vaadin already has CSRF protection.

② Uses `CustomRequestCache` to track unauthorized requests so that users are redirected appropriately after login.

③ Turns on authorization.

④ Allows all internal traffic from the Vaadin framework.

⑤ Allows all authenticated traffic.

⑥ Enables form-based login and permits unauthenticated access to it.

⑦ Configures the login page URLs.

⑧ Configures the logout URL.

4. Add another method to configure test users.

SecurityConfiguration.java

```java
@Bean
@Override
public UserDetailsService userDetailsService() {
    UserDetails user =
        User.withUsername("user")
            .password("{noop}password")
            .roles("USER")
            .build();

    return new InMemoryUserDetailsManager(user);
}
```

- Defines a single user with the username `"user"` and password `"password"` in an in-memory `DetailsManager`.

> WARNING We do not recommend that you configure users directly in the code for applications in production. You can easily change this Spring Security configuration to use an authentication provider for LDAP, JAAS, and other real world sources. Read more about Spring Security authentication providers.

5. Finally, exclude Vaadin-framework communication and static assets from Spring Security.

SecuirtyConfiguration.java

```
@Override
public void configure(WebSecurity web) {
    web.ignoring().antMatchers(
        "/VAADIN/**",
        "/favicon.ico",
        "/robots.txt",
        "/manifest.webmanifest",
        "/sw.js",
        "/offline.html",
        "/icons/**",
        "/images/**",
        "/styles/**",
        "/h2-console/**");
}
```

Restricting access to Vaadin views

Spring Security restricts access to content based on paths. Vaadin applications are single-page applications. This means that they do not trigger a full browser refresh when you navigate between views, even though the path does change.To secure a Vaadin application, we need to wire Spring Security to the Vaadin navigation system.

To do this, create a new class in the `security` package, `ConfigureUIServiceInitListener`.

ConfigureUIServiceInitListener.java

```java
package com.vaadin.tutorial.crm.security;

import com.vaadin.flow.component.UI;
import com.vaadin.flow.router.BeforeEnterEvent;
import com.vaadin.flow.server.ServiceInitEvent;
import com.vaadin.flow.server.VaadinServiceInitListener;
import com.vaadin.tutorial.crm.ui.view.login.LoginView;
import org.springframework.stereotype.Component;

@Component ①
public class ConfigureUIServiceInitListener implements VaadinServiceInitListener {

    @Override
    public void serviceInit(ServiceInitEvent event) {
        event.getSource().addUIInitListener(uiEvent -> { ②
            final UI ui = uiEvent.getUI();
            ui.addBeforeEnterListener(this::authenticateNavigation);
        });
    }

    private void authenticateNavigation(BeforeEnterEvent event) {
        if (!LoginView.class.equals(event.getNavigationTarget())
            && !SecurityUtils.isUserLoggedIn()) { ③
            event.rerouteTo(LoginView.class);
        }
    }
}
```

① The `@Component` annotation registers the listener. Vaadin will pick it up on startup.

② In `serviceInit`, we listen for the initialization of the UI (the internal root component in Vaadin) and then add a listener before every view transition.

③ In authenticateNavigation, we reroute all requests to the login, if the user is not logged in

TIP	You can read more about fine-grained access control in the Spring Security tutorial series.

Adding a logout link

You can now log in to the application. The final thing we need to do is add a logout link to the application header.

1. In `MainLayout`, add a link to the header:

`MainLayout.java`

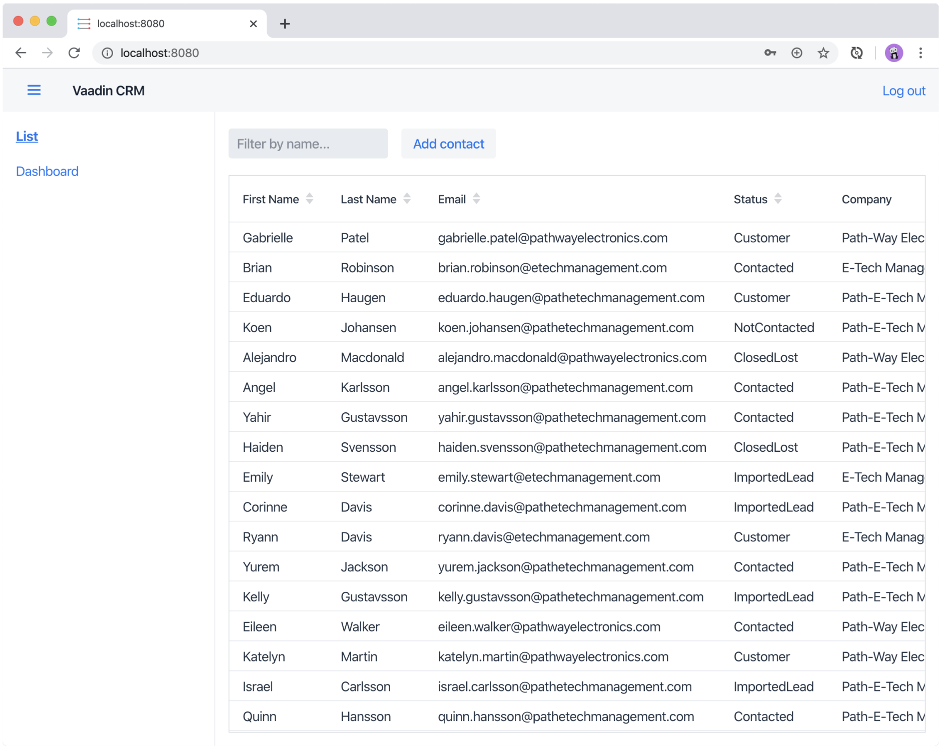
```java
private void createHeader() {
    H1 logo = new H1("Vaadin CRM");
    logo.addClassName("logo");

    Anchor logout = new Anchor("logout", "Log out"); ①

    HorizontalLayout header = new HorizontalLayout(new DrawerToggle(), logo,
logout); ②
    header.expand(logo); ③
    header.setDefaultVerticalComponentAlignment(FlexComponent.Alignment.CENTER);
    header.setWidth("100%");
    header.addClassName("header");

    addToNavbar(header);
}
```

① Creates a new `Anchor` (`<a>` tag) that links to `/logout`.

② Adds the link last in the header layout.

③ Calls `header.expand(logo)` to make the logo take up all the extra space in the layout. This pushes the logout button to the far right.

2. Stop and restart the server to pick up the new Maven dependencies. You should now be able to log in and out of the app. Verify that you can't access http://localhost/dashboard without being logged in.



First Name	Last Name	Email	Status	Company
Gabrielle	Patel	gabrielle.patel@pathwayelectronics.com	Customer	Path-Way Elec
Brian	Robinson	brian.robinson@etechmanagement.com	Contacted	E-Tech Manag
Eduardo	Haugen	eduardo.haugen@pathetechmanagement.com	Customer	Path-E-Tech M
Koen	Johansen	koen.johansen@pathetechmanagement.com	NotContacted	Path-E-Tech M
Alejandro	Macdonald	alejandro.macdonald@pathwayelectronics.com	ClosedLost	Path-Way Elec
Angel	Karlsson	angel.karlsson@pathetechmanagement.com	Contacted	Path-E-Tech M
Yahir	Gustavsson	yahir.gustavsson@pathetechmanagement.com	Contacted	Path-E-Tech M
Haiden	Svensson	haiden.svensson@pathetechmanagement.com	ClosedLost	Path-E-Tech M
Emily	Stewart	emily.stewart@etechmanagement.com	ImportedLead	E-Tech Manag
Corinne	Davis	corinne.davis@pathetechmanagement.com	ImportedLead	Path-E-Tech M
Ryann	Davis	ryann.davis@etechmanagement.com	Customer	E-Tech Manag
Yurem	Jackson	yurem.jackson@pathetechmanagement.com	Contacted	Path-E-Tech M
Kelly	Gustavsson	kelly.gustavsson@pathetechmanagement.com	ImportedLead	Path-E-Tech M
Eileen	Walker	eileen.walker@pathwayelectronics.com	Contacted	Path-Way Elec
Katelyn	Martin	katelyn.martin@pathwayelectronics.com	Customer	Path-Way Elec
Israel	Carlsson	israel.carlsson@pathetechmanagement.com	ImportedLead	Path-E-Tech M
Quinn	Hansson	quinn.hansson@pathetechmanagement.com	Contacted	Path-E-Tech M

You have now built a full-stack CRM application with navigation and authentication. In the next tutorial, you'll learn how to make the application installable on mobile and desktop.

You can find the completed source code for this tutorial on GitHub.

Turning a Vaadin app into an installable PWA

In this chapter, we turn the completed CRM application into a progressive web application (PWA) so that users can install it.

You can download the completed source code at the bottom. The code from the previous tutorial chapter can be found here, if you want to jump directly into this chapter.

What is a PWA?

The term PWA is used to describe modern web applications that offer a native app-like user experience. PWA technologies make applications faster, more reliable, and more engaging. PWAs can be installed on most mobile devices and on desktop when using supported browsers. They can even be listed in the Microsoft Store and Google Play Store. You can learn more about the underlying technologies and features in the Vaadin PWA documentation.

Two main components enable PWA technologies:

- `ServiceWorker`: A JavaScript worker file that controls network traffic and enables custom cache control.
- Web app manifest: A JSON file that identifies the web app as an installable app.

Generating PWA resources

Vaadin provides the `@PWA` annotation that automatically generates the required PWA resources.

Add the `@PWA` annotation to `MainLayout` as follows:

`MainLayout.java`

```
@CssImport("./styles/shared-styles.css")
@PWA( ①
    name = "VaadinCRM", ②
    shortName = "CRM")  ③
public class MainLayout extends AppLayout {
     ...
}
```

① The `@PWA` annotation tells Vaadin to create a `ServiceWorker` and a manifest file.

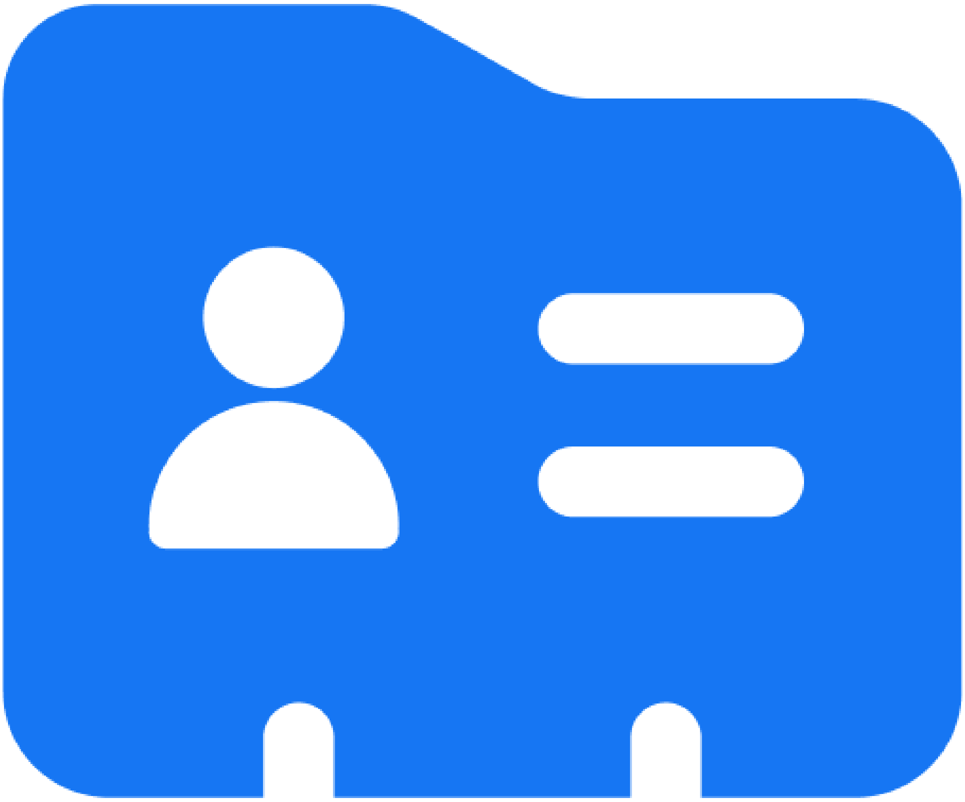
② `name` is the full name of the application for the manifest file.

③ `shortName` should be short enough to fit under an icon when installed, and should not exceed 12 characters.

Customizing the app icon

1. Start by creating a new folder `src/main/webapp`.
2. Create a new subfolder `src/main/webapp/icons` and add your own icon image named `icon.png`. The image resolution should be 512 px x 512 px. This overrides the default image in the starter.

You can use your own icon, or save the image below, by right clicking and selecting **Save Image**.



Customizing the offline page

Vaadin creates a generic offline fallback page that displays when the application is launched offline. Replacing this default page with a custom page that follows your own design guidelines makes your app more polished.

1. Use the code below to create `offline.html` in the new `src/main/webapp` folder:

   `offline.html`

   ```
   <!DOCTYPE html>
   <html lang="en">
   <head>
     <meta charset="UTF-8"/>
     <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
     <meta http-equiv="X-UA-Compatible" content="ie=edge"/>
     <title>Offline | Vaadin CRM</title>
     <link rel="stylesheet" href="./styles/offline.css"> ①
   </head>
   <body>

   <div class="content">
     <img src="./images/offline.png" alt="VaadinCRM is offline" class="offline-
   image"/> ②
     <h1>Oh deer, you're offline</h1>
     <p>Your internet connection is offline. Get back online to continue using
   Vaadin CRM.</p>
   </div>
   <script>
     window.addEventListener('online', location.reload);
   </script> ③
   </body>
   </html>
   ```

   ① The page loads a CSS file, offline.css.

   ② The page displays an image, offline.png.

   ③ The JavaScript snippet reloads the page if the browser detects that it’s back online.

2. Create two new folders, `src/main/webapp/styles` and `src/main/webapp/images`.

3. In the `styles` folder, create `offline.css` and add the following styles:

offline.css

```
body {
    display: flex; ①
    flex-direction: column;
    align-items: center;
    font-family: sans-serif;
    color: #555;
}

.content {
    width: 80%;
}

.offline-image {
    width: 100%;
    margin: 4em 0px;
}
```

① Makes the page a flexbox that centers content horizontally.

4. Add the following image (or use one of your own) to the `images` folder and name it `offline.png`.

5. Make the files available offline by adding them to the `@PWA` annotation in `MainLayout` as follows:

MainLayout.java

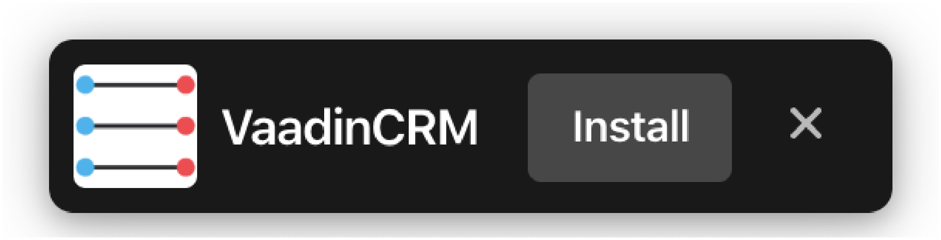
```
@CssImport("./styles/shared-styles.css")
@PWA(
    name = "VaadinCRM",
    shortName = "VaadinCRM",
    offlineResources = { ①
        "./styles/offline.css",
        "./images/offline.png"})
public class MainLayout extends AppLayout {
    ...
}
```

① `offlineResources` is a list of files that Vaadin will make available offline through the `ServiceWorker`.

WARNING

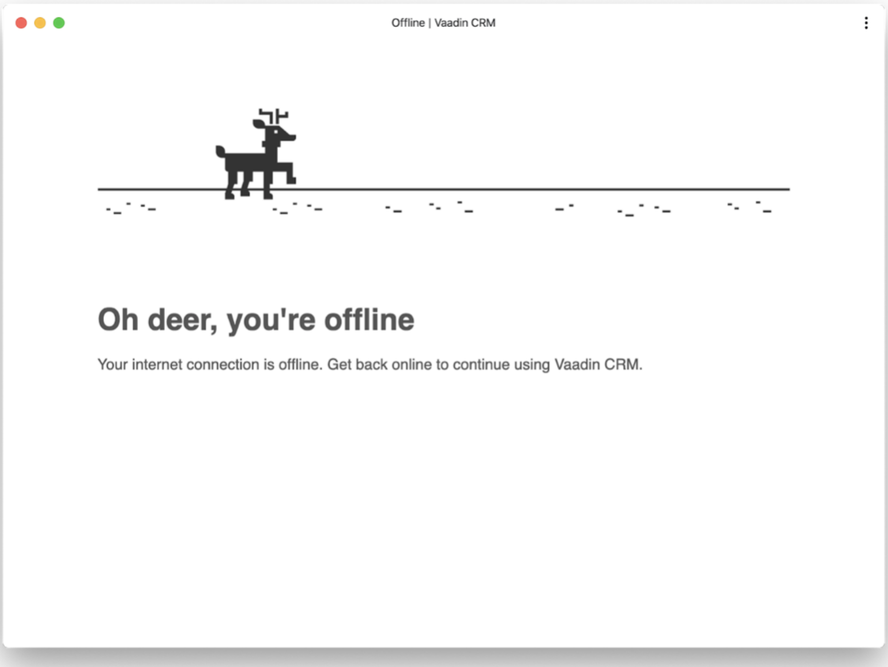
Even though the paths for the CSS files is identical in the Java file, `shared-styles.css` is loaded from `frontend/styles/shared-styles.css`, whereas `offline.css` is loaded from `src/main/java/webapp/styles/offline.css`. If you have trouble accessing files while offline, check that these files are in the correct folders.

6. Restart the app. On supported browsers, your will now see an install prompt that you can use to install the application:



Testing the offline page

Shut down the server in IntelliJ and refresh the browser (or launch the installed app). You should now see the custom offline page.


Offline | Vaadin CRM

Oh deer, you're offline

Your internet connection is offline. Get back online to continue using Vaadin CRM.

In the next chapter, we cover testing the application: both unit tests and in-browser tests.

You can find the completed source code for this tutorial on GitHub.

Testing Spring Boot apps with unit and integration tests

It is a common best practice to test as little code as possible in a single test. In this way, when things go wrong, only relevant tests fail. For UI testing, there are three main approaches:

- Unit tests: for simple UI logic.
- Integration tests: for more advanced UI logic.
- End-to-end tests: to test what the user sees.

Unit and integration tests can be run standalone, that is, without any external dependencies, such as a running server or database.

End-to-end tests require the application to be deployed and are run in a browser window to simulate an actual user.

In this chapter, we write and run unit and integration tests. We cover end-to-end tests in the next chapter.

You can download the completed source code at the bottom. The code from the previous tutorial chapter can be found here, if you want to jump directly into this chapter.

Creating and running unit tests for simple UI logic

The most minimalistic way of testing is to create a plain Java unit test. This only works with UI classes with no dependencies, no auto wiring etc. For the `ContactForm`, you can create a unit test to verify that the form fields are correctly populated, based on the given bean.

CAUTION	All test classes should go in the test folder, `src/test/java`. Pay special attention to the package names. We use package-access to class fields. If the test isn't in the same package as the class that's being tested, you will get errors.

1. Start by deleting the tests included with the starter: AbstractViewTest.java` and `LoginViewIT.java`.
2. In the `test` folder, create a new package, `com.vaadin.tutorial.crm.ui.view.list`, and add a new `ContactFormTest.java` file with the code below. When prompted, select the `org.junit` import of `@Before`.

ContactFormTest.java

```java
public class ContactFormTest {
    private List<Company> companies;
    private Contact marcUsher;
    private Company company1;
    private Company company2;

    @Before
    public void setupData() {
        companies = new ArrayList<>();
        company1 = new Company("Vaadin Ltd");
        company2 = new Company("IT Mill");
        companies.add(company1);
        companies.add(company2);

        marcUsher = new Contact();
        marcUsher.setFirstName("Marc");
        marcUsher.setLastName("Usher");
        marcUsher.setEmail("marc@usher.com");
        marcUsher.setStatus(Contact.Status.NotContacted);
        marcUsher.setCompany(company2);
    }
}
```

- The `@Before` annotation adds dummy data that is used for testing. This method is executed before each `@Test` method.

3. Now, add a test method that uses `ContactForm`:

ContactFormTest.java

```java
@Test
public void formFieldsPopulated() {
    ContactForm form = new ContactForm(companies);
    form.setContact(marcUsher); ①
    Assert.assertEquals("Marc", form.firstName.getValue());
    Assert.assertEquals("Usher", form.lastName.getValue());
    Assert.assertEquals("marc@usher.com", form.email.getValue());
    Assert.assertEquals(company2, form.company.getValue());
    Assert.assertEquals(Contact.Status.NotContacted, form.status.getValue()); ②
}
```

① Validates that the fields are populated correctly, by first initializing the contact form with some companies, and then setting a contact bean for the form.

② Uses standard JUnit `assertEquals` methods to compare the values from the fields available through the `ContactForm` instance:

4. Similarly, test the save functionality of `ContactForm`:

   a. First, initialize a `ContactForm` with an empty contact:

`ContactFormTest.java`

```
@Test
public void saveEventHasCorrectValues() {
    ContactForm form = new ContactForm(companies);
    Contact contact = new Contact();
    form.setContact(contact);
}
```

b. Continue the method by populating values into the form:

`saveEventHasCorrectValues()`

```
form.firstName.setValue("John");
form.lastName.setValue("Doe");
form.company.setValue(company1);
form.email.setValue("john@doe.com");
form.status.setValue(Contact.Status.Customer);
```

c. Finally, add the following code to the end of `saveEventHasCorrectValues()` to click the save button and assert that the values from the fields end up in the bean:

`saveEventHasCorrectValues()`

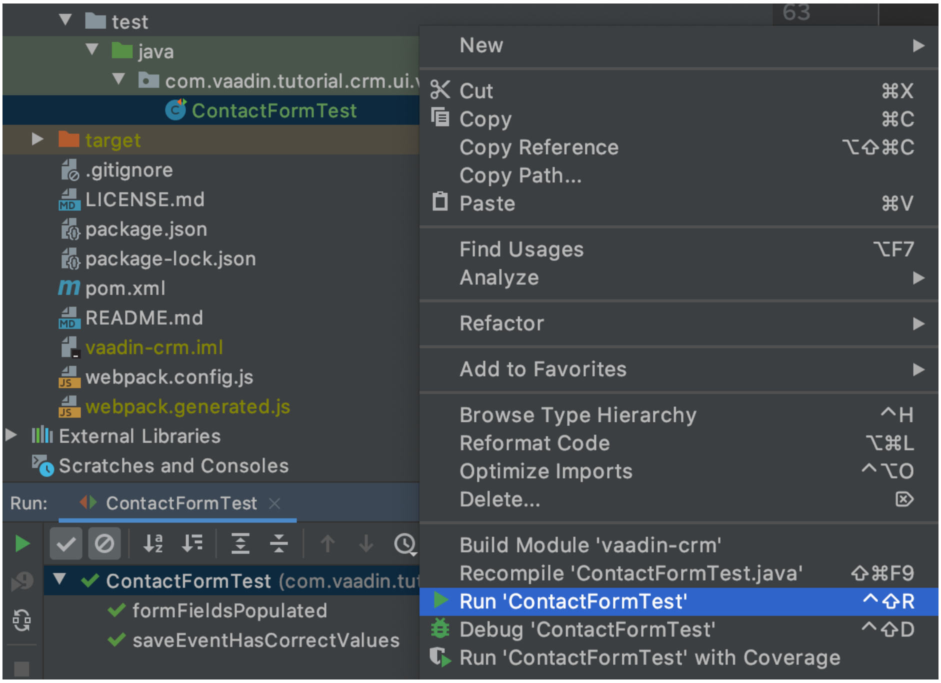
```
AtomicReference<Contact> savedContactRef = new AtomicReference<>(null);
form.addListener(ContactForm.SaveEvent.class, e -> {
    savedContactRef.set(e.getContact()); ①
});
form.save.click();
Contact savedContact = savedContactRef.get(); ②

Assert.assertEquals("John", savedContact.getFirstName());
Assert.assertEquals("Doe", savedContact.getLastName());
Assert.assertEquals("john@doe.com", savedContact.getEmail());
Assert.assertEquals(company1, savedContact.getCompany());
Assert.assertEquals(Contact.Status.Customer, savedContact.getStatus()); ③
```

① As `ContactForm` fires an event on save and the event data is needed for the test, an `AtomicReference` is used to store the event data, without using a class field.

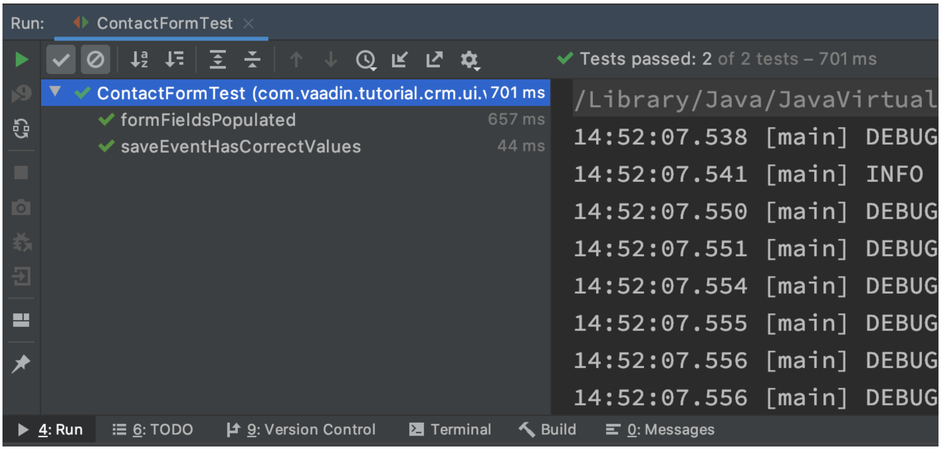
② Clicks the save button and asserts that the values from the fields end up in the bean.

③ Once the event data is available, you can use standard `assertEquals` calls to verify that the bean contains the expected values.

5. To run the unit test, right click `ContactFormTest` and Select **Run 'ContactFormTest'**.



6. When the test finishes, you will see the results at the bottom of the IDE window in the test runner panel. As you can see, both tests passed.


Run: ContactFormTest
Tests passed: 2 of 2 tests – 701 ms
ContactFormTest (com.vaadin.tutorial.crm.ui. 701 ms
formFieldsPopulated 657 ms
saveEventHasCorrectValues 44 ms
4: Run
6: TODO
9: Version Control
Terminal
Build
0: Messages

```
/Library/Java/JavaVirtual
14:52:07.538 [main] DEBUG
14:52:07.541 [main] INFO
14:52:07.550 [main] DEBUG
14:52:07.551 [main] DEBUG
14:52:07.554 [main] DEBUG
14:52:07.555 [main] DEBUG
14:52:07.556 [main] DEBUG
14:52:07.556 [main] DEBUG
```

Creating and running integration tests for more advanced UI logic

To test a class that uses `@Autowire`, a database, or any other feature provided by Spring Boot, you can no longer use plain JUnit tests. Instead, you can use the Spring Boot test runner. This does add a little overhead, but it makes more features available to your test.

1. First, add the `spring-boot-starter-test` dependency to the project's `pom.xml` to be able to use the features:

   `pom.xml`

   ```
   <dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-test</artifactId>
     <scope>test</scope>
     <exclusions>
       <exclusion>
         <groupId>org.junit.vintage</groupId>
         <artifactId>junit-vintage-engine</artifactId>
       </exclusion>
     </exclusions>
   </dependency>
   ```

2. To set up a unit test for `ListView`, create a new file, `ListViewTest`, in the `com.vaadin.tutorial.crm.ui.views.list` package:

   `ListViewTest.java`

   ```
   @RunWith(SpringRunner.class)
   @SpringBootTest
   public class ListViewTest {

       @Autowired
       private ListView listView;

       @Test
       public void formShownWhenContactSelected() {
       }
   }
   ```

   - The `@RunWith(SpringRunner.class)` and `@SpringBootTest` annotations make sure that the Spring Boot application is initialized before the tests are run and allow you to use `@Autowire` in the test.

3. In the `ListView` class:

   a. Add the Spring `@Component` annotation to make it possible to `@Autowire` it. Also add `@Scope("prototype")` to ensure every test run gets a fresh instance.

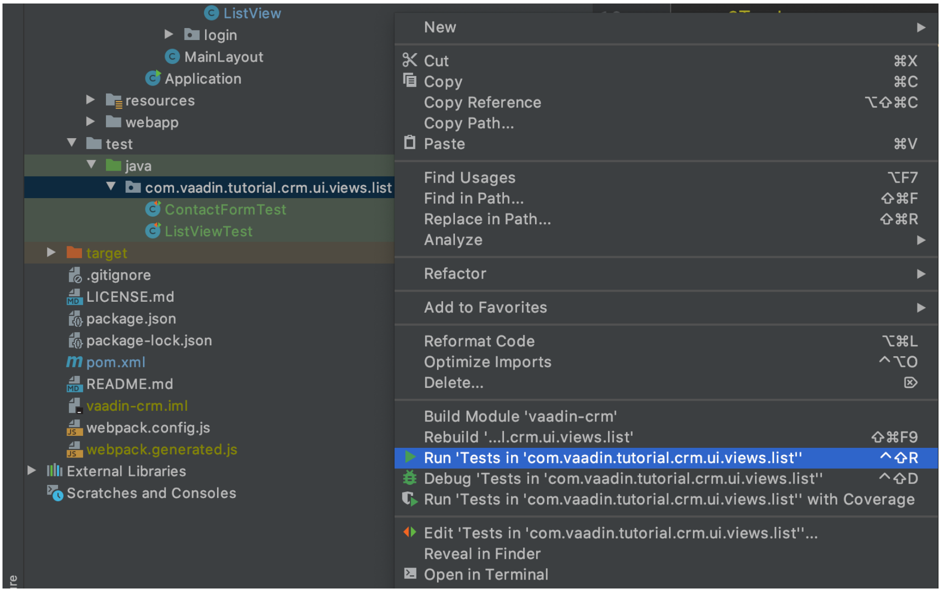
TIP	We didn’t need to add the annotation for normal application usage, as all `@Route` classes are automatically instantiated by Vaadin in a Spring-compatible way.

b. Remove the `private` keyword. This changes the private fields to package private, and allows you to access the grid and form of the `ListView` in your test case.

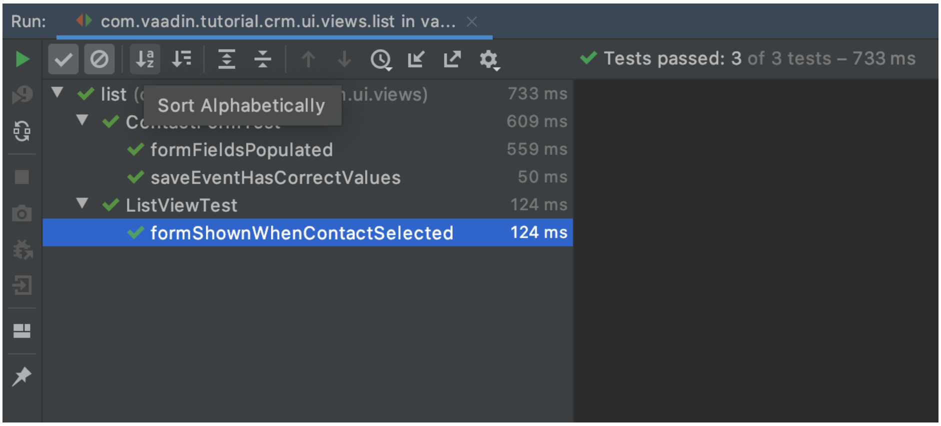
`ListView.java`

```
@Component
@Scope("prototype")
@Route(value = "", layout = MainLayout.class)
@PageTitle("Contacts | Vaadin CRM")
public class ListView extends VerticalLayout {

    ContactForm form;
    Grid<Contact> grid = new Grid<>(Contact.class);
    TextField filterText = new TextField();

    ContactService contactService;

    // rest omitted
}
```

4. Right click the package that contains both tests, and select **Run tests in 'com.vaadin.tutorial.crm.ui.views.list'**.



5. You should see that both test classes run and result in 3 successful tests.


```
Run:  com.vaadin.tutorial.crm.ui.views.list in va...
Tests passed: 3 of 3 tests – 733 ms
▼ ✓ list (c   Sort Alphabetically   .ui.views)   733 ms
  ▼ ✓ Co                                         609 ms
      ✓ formFieldsPopulated                      559 ms
      ✓ saveEventHasCorrectValues                 50 ms
  ▼ ✓ ListViewTest                               124 ms
      ✓ formShownWhenContactSelected             124 ms
```

NOTE

You probably noticed that running the tests the second time took much longer. This is the price of being able to use `@Autowire` and other Spring features and can cost many seconds of startup time.

You can improve the startup time by explicitly listing the needed dependencies in the `@SpringBootTest` annotation using `classes={…}`, mock parts of the application, or using other advanced techniques which are out of scope for this tutorial. Pivotal's Spring Boot Testing Best Practices has tips to speed up your tests.

6. You can now add the actual test implementation, which selects the first row in the grid and validates that this shows the form with the selected `Contact`:

`ListViewTest.java`

```
    @Test
    public void formShownWhenContactSelected() {
        Grid<Contact> grid = listView.grid;
        Contact firstContact = getFirstItem(grid);

        ContactForm form = listView.form;

        Assert.assertFalse(form.isVisible());
        grid.asSingleSelect().setValue(firstContact);
        Assert.assertTrue(form.isVisible());
        Assert.assertEquals(firstContact.getFirstName(), form.firstName.getValue
());
    }
private Contact getFirstItem(Grid<Contact> grid) {
        return( (ListDataProvider<Contact>) grid.getDataProvider()).getItems()
.iterator().next();
    }
```

- The test verifies that the form logic works by:

    - Asserting that the form is initially hidden.
    - Selecting the first item in the grid and verifying that:
        - The form is visible.
        - The form is bound to the correct `Contact` by ensuring the right name is visible in the field.
7. Rerun the tests. They should all pass.

You now know how to test the application logic both in isolation with unit tests and by injecting dependencies to test the integration between several components. In the next chapter, we cover how to test the entire application in the browser.

You can find the completed source code for this tutorial on GitHub.

Testing Vaadin apps in the browser with end-to-end tests

End-to-end (e2e) tests are used to test the entire application. They're far more coarse grained than unit or integration tests. This makes them well suited to check that the application works as a whole, and to catch any regressions that may be missed by more specific tests.

End-to-end tests are executed in a browser window, controlled by a web driver and run on the server where the application is deployed. You need to setup 3 things:

1. Configure Maven to start the Spring Boot server before running tests and to stop it afterwards.
2. Make sure you have Chrome installed and install a web driver manager that will download the needed web driver.
3. Create a base test class that starts a browser and opens the application URL.

You can download the completed source code at the bottom. The code from the previous tutorial chapter can be found here, if you want to jump directly into this chapter.

NOTE	The end-to-end tests use Vaadin TestBench, which is a commercial tool that is a part of the Vaadin Pro Subscription. You can get a free trial at https://vaadin.com/trial?utm_source=github+vaadin. All Vaadin Pro tools and components are free for students through the GitHub Student Developer Pack.

Configuring Maven to start the server

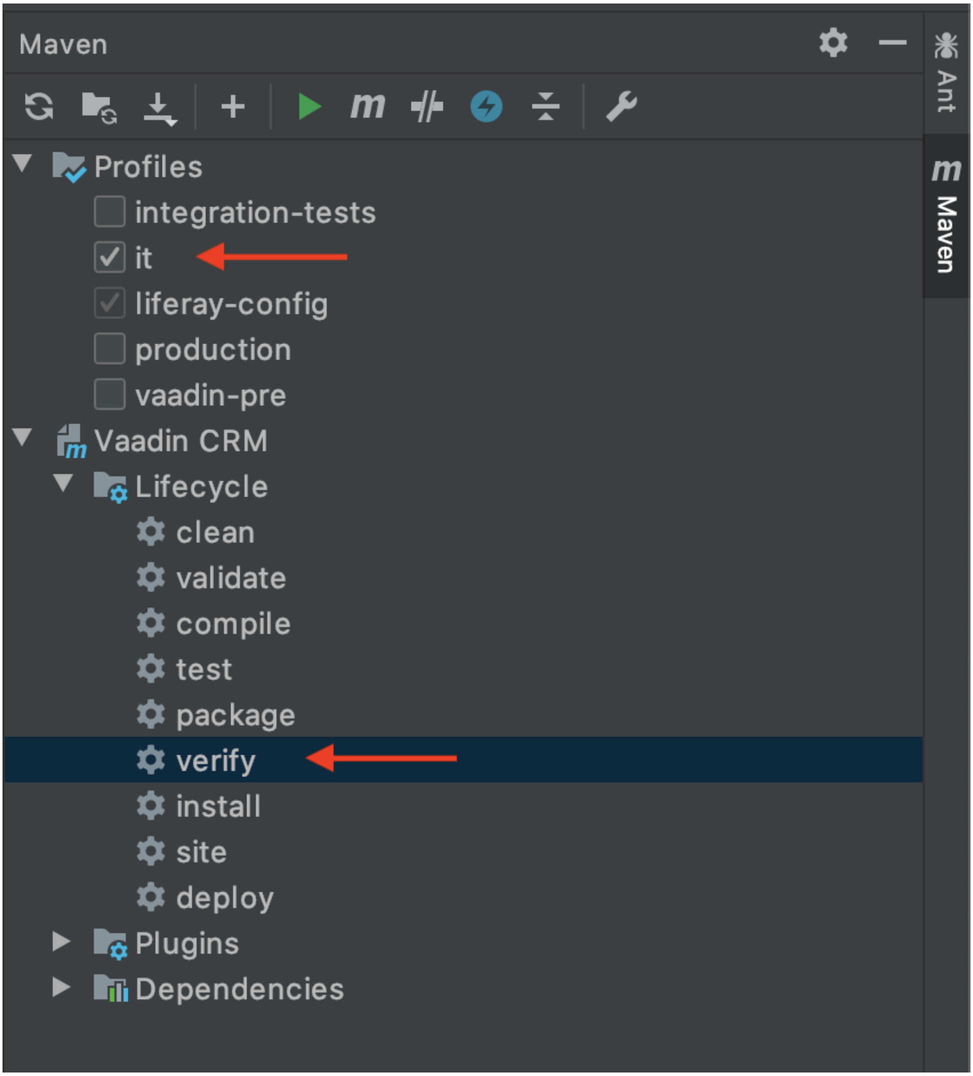
To run integration tests, you need to make sure that the Maven configuration starts and stops Spring Boot at the proper times. If you configure this as a separate profile in your `pom.xml`, you can easily skip running the tests.

In you `pom.xml`, remove the existing `integration-test` profile and add the following profile:

pom.xml

```xml
    <profile>
      <id>it</id>
    <build>
        <plugins>
          <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <executions>
              <execution>
                <id>start-spring-boot</id>
                <phase>pre-integration-test</phase>
                <goals>
                  <goal>start</goal>
                </goals>
              </execution>
              <execution>
                <id>stop-spring-boot</id>
                <phase>post-integration-test</phase>
                <goals>
                  <goal>stop</goal>
                </goals>
              </execution>
            </executions>
          </plugin>
          <!-- Runs the integration tests (*IT) after the server is started -->
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-failsafe-plugin</artifactId>
            <executions>
              <execution>
                <goals>
                  <goal>integration-test</goal>
                  <goal>verify</goal>
                </goals>
              </execution>
            </executions>
            <configuration>
              <trimStackTrace>false</trimStackTrace>
              <enableAssertions>true</enableAssertions>
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>
```

Integration tests are run by executing `mvn -Pit verify` from the command line, or by selecting the `it` profile and running the verify goal from the Maven menu in IntelliJ.



The `pre-integration-test` and `post-integration-test` executions take care of starting and stopping Spring Boot before the `integration test` phase of the build is executed. In the `integration test` phase, the `maven-failsafe-plugin` runs any tests named `*IT.java` found in `src/test/java`. You should set the `trimStackTrace` option to `false` to print full stack traces and ease debugging.

Setting up Chrome and its webdriver to control the browser

For browser tests to work, you need Chrome installed on the machine that runs the tests.

To avoid installing the web driver manually, add a dependency to `webdrivermanager` in your `pom.xml`:

`pom.xml`

```
<dependency>
    <groupId>io.github.bonigarcia</groupId>
    <artifactId>webdrivermanager</artifactId>
    <version>3.7.1</version>
    <scope>test</scope>
</dependency>
----
```

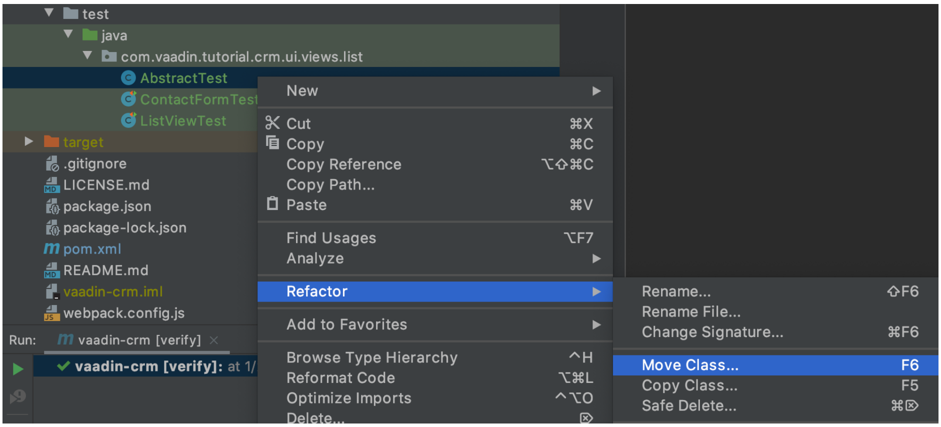
- This is used from the JUnit test class and downloads the correct version of the Chrome web driver, if it is missing.

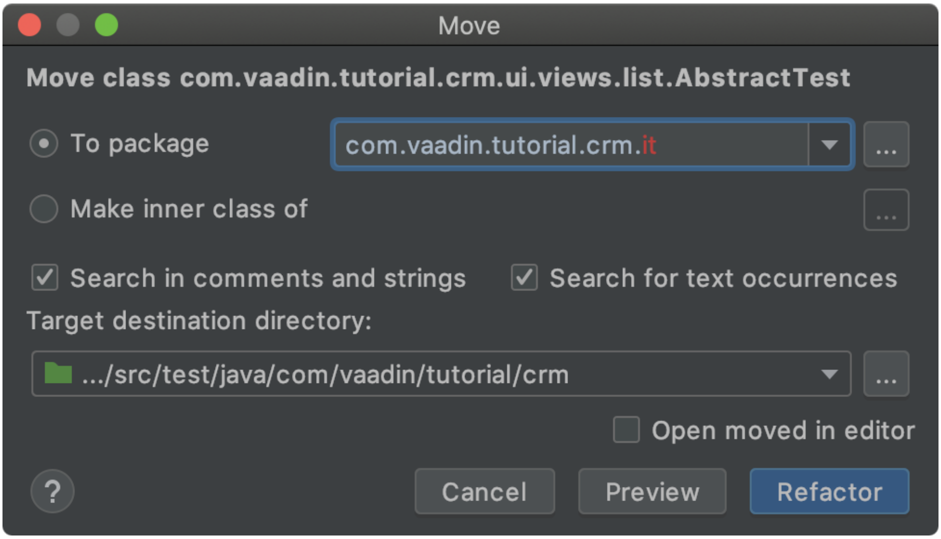
Creating the base test class

To avoid repetition in each test class, it is a good idea to put common logic in an abstract class and have all tests extend this class. Most of the heavy lifting about starting browsers etc. is handled by `ParallelTest` in TestBench, but there are a couple of useful things you can add to the abstract class.

1. Create a new class, `AbstractTest`. Be sure to place it in `src/test/java` and not `src/main/java`.

TIP	IntelliJ collapses empty packages by default, so it's easiest to first create the class in the existing test package, and then move it to the correct package.





Move class com.vaadin.tutorial.crm.ui.views.list.AbstractTest

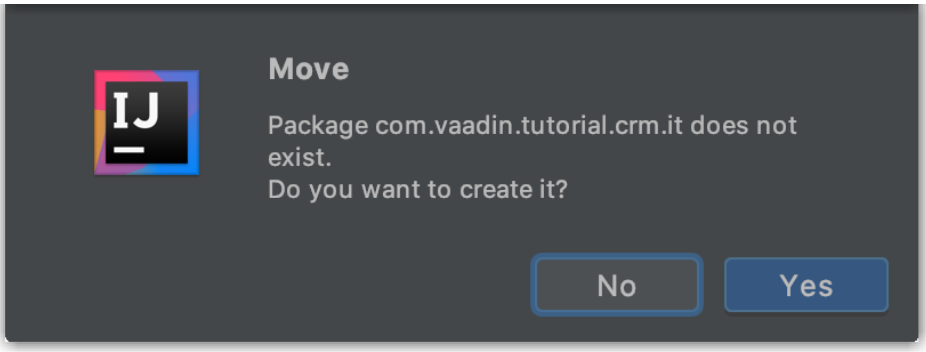
● To package: com.vaadin.tutorial.crm.it

○ Make inner class of

☑ Search in comments and strings ☑ Search for text occurrences

Target destination directory: .../src/test/java/com/vaadin/tutorial/crm

☐ Open moved in editor


Move

Package com.vaadin.tutorial.crm.it does not exist.
Do you want to create it?

AbstractTest.java

```java
package com.vaadin.tutorial.crm.it;

public abstract class AbstractTest extends ParallelTest {
    @BeforeClass
    public static void setupClass() {
        WebDriverManager.chromedriver().setup(); ①
    }

    @Rule
    public ScreenshotOnFailureRule rule = new ScreenshotOnFailureRule(this, true
); ②
}
```

① We start by invoking the `WebDriverManager` before any test method is invoked. TestBench does not invoke the web driver manager.

② `ScreenshotOnFailureRule` tells TestBench to grab a screenshot before exiting, if a test fails. This helps you understand what went wrong when tests do not pass.

2. Next, add the application URL that the tests should open before trying to interact with the application. For this you need the host name where the application runs ("localhost" in development), the port the server uses (set to 8080 in application.properties), and information about the route to start from.

AbstractTest.java

```java
    private static final String SERVER_HOST = IPAddress.findSiteLocalAddress();
    private static final int SERVER_PORT = 8080;
    private final String route;

    @Before
    public void setup() throws Exception {
        super.setup();
        getDriver().get(getURL(route)); // Opens the given URL in the browser
    }

    protected AbstractTest(String route) {
        this.route = route;
    }

    private static String getURL(String route) {
        return String.format("http://%s:%d/%s", SERVER_HOST, SERVER_PORT, route);
    }
```

3. To avoid excessive logging from `WebDriverManager` when running the tests, add the following workaround:

AbstractTest.java

```java
    static {
        // Prevent debug logging from Apache HTTP client
        Logger root = (Logger) LoggerFactory.getLogger(Logger.ROOT_LOGGER_NAME);
        root.setLevel(Level.INFO);
    }
```

4. Select the following Logger dependencies:
    a. `org.slf4j.LoggerFactory`
    b. `ch.qos.logback.classic.Level`
    c. `ch.qos.logback.classic.Logger`

Testing the login view

Now that your setup is complete, you can start developing your first test: ensuring that a user can log in. For this test you need to open the base URL.

1. Create a new class, `LoginIT`, in the same package as `AbstractTest`:

LoginIT.java

```java
package com.vaadin.tutorial.crm.it;

public class LoginIT extends AbstractTest {
    public LoginIT() {
        super("");
    }
}
```

NOTE	The name of the class should end in `IT` for the test runner to pick it up as an integration test. If you name it `LoginTest` instead, it will be run as a unit test and the server will not be started and the test will fail.

2. Add an `@Test` method to validate that you can log in as "user":

LoginIT.java

```java
@Test
public void loginAsValidUserSucceeds() {
    // Find the LoginForm used on the page
    LoginFormElement form = $(LoginFormElement.class).first();
    // Enter the credentials and log in
    form.getUsernameField().setValue("user");
    form.getPasswordField().setValue("password");
    form.getSubmitButton().click();
    // Ensure the login form is no longer visible
    Assert.assertFalse($(LoginFormElement.class).exists());
}
```

TIP While developing tests it is not very efficient to run the tests as `mvn -Pit verify`. Instead, you can start the server manually by launching the `Application` class or with `spring-boot:run`. You can then execute the selected test in your IDE and you do not have to wait for the server to start every time.

3. Start the application normally, then right click `LoginIT.java` and select **Run 'LoginIT'**.

NOTE the first time you run the test, you will be asked to start a trial or validate your existing license. Follow the instructions in the browser window that opens.

Creating a view object

You can now add a second test: validating that you cannot log in with an invalid password.

For this text, you need to write the same code to access the components in the view, as you did for the first test. To make your tests more maintainable, you can create a view object (a.k.a. call page object or element class) for each view. A view object provides a high-level API to interact with the view and hides the implementation details.

1. For the login view, create the `LoginViewElement` class in a new package, `com.vaadin.tutorial.crm.it.elements.login`:

LoginViewElement.java

```java
package com.vaadin.tutorial.crm.it.elements.login;

@Attribute(name = "class", contains = "login-view") ①
public class LoginViewElement extends VerticalLayoutElement {

    public boolean login(String username, String password) {
        LoginFormElement form = $(LoginFormElement.class).first();
        form.getUsernameField().setValue(username);
        form.getPasswordField().setValue(password);
        form.getSubmitButton().click();

        // Return true if we end up on another page
        return !$(LoginViewElement.class).onPage().exists();
    }

}
```

a. Selects the `com.vaadin.testbench.annotations.Attribute` import.

CAUTION	To make the correct functionality available from super classes, the hierarchy of the view object should match the hierarchy of the view (`public class LoginView extends VerticalLayout` vs `public class LoginViewElement extends VerticalLayoutElement`).

b. Adding the `@Attribute(name = "class", contains = "login-view")` annotation allows you to find the `LoginViewElement` using the TestBench query API, for example:

```java
LoginViewElement loginView = $(LoginViewElement.class).onPage().first();
```

The annotation searches for the `login-view` class name, which is set for the login view in the constructor. The `onPage()` call ensures that the whole page is searched. By default a `$` query starts from the active element.

2. Now that the the `LoginViewElement` class is available, you can refactor your `loginAsValidUserSucceeds` test to be:

LoginIT.java

```java
@Test
public void loginAsValidUserSucceeds() {
    LoginViewElement loginView = $(LoginViewElement.class).onPage().first();
    Assert.assertTrue(loginView.login("user", "password"));
}
```

3. Add a test to use an invalid password as follows:

LoginIT.java

```
@Test
public void loginAsInvalidUserFails() {
    LoginViewElement loginView = $(LoginViewElement.class).onPage().first();
    Assert.assertFalse(loginView.login("user", "invalid"));
}
```

4. Continue testing the other views by creating similar view objects and IT classes.

In the next tutorial we cover how to make a production build of the application and deploy it to a cloud platform.

You can find the completed source code for this tutorial on GitHub.

Deploying a Spring Boot app on AWS Elastic Beanstalk

In this final tutorial in the series, we show you how to deploy the Spring Boot application we have built on Amazon Web Services (AWS) with AWS Elastic Beanstalk (EB). EB is a service that orchestrates other AWS services like virtual servers, load balancers, storage, and databases.

TIP You can also deploy your application onto other cloud platforms. Go to the Cloud Deployment tutorials for more options.

So far in this series, the application has used an in-memory H2 database. For the deployed application, we will use a MySQL server for persistent storage instead.

You can download the completed source code at the bottom. The code from the previous tutorial chapter can be found here, if you want to jump directly into this chapter.

Preparing the application for production

Before we can deploy the app, we need to make it ready for production. More specifically, we need to build the app with Vaadin production mode enabled and add a separate configuration file to configure the production database and ports.

When you build a production version of a Vaadin app, the following happens: All the front-end resources are bundled and minified to speed up the app load time. Vaadin runs in production mode to hide debugging and other sensitive information from the browser.

Creating a separate configuration for production

We want to run different databases during development and in production. To support this, we need to create a separate configuration file for the production build.

Create a new file, `application-prod.properties`, in `src/main/resources`.

TIP Do not store sensitive information like passwords in properties files that get committed to a version control system like Git. Instead, use environment variables that can be kept on the server.

`application-prod.properties`

```
server.port=5000 ①
spring.datasource.url=jdbc:mysql://${RDS_HOSTNAME}:${RDS_PORT}/${RDS_DB_NAME} ②
spring.datasource.username=${RDS_USERNAME}
spring.datasource.password=${RDS_PASSWORD}
spring.jpa.hibernate.ddl-auto=create
```

① Elastic Beanstalk maps the internal port 5000 to the external port 80 to expose the application to the internet.

② Elastic Beanstalk will provide environment variables with information about the database so we don't need to store them in the property file.

WARNING `spring.jpa.hibernate.ddl-auto=create` deletes and re-creates the database on every deployment. A more proper solution is to use a database migration tool like Liquibase.

Adding MySQL support

Add MySQL as a dependency to your project's `pom.xml` file.

`pom.xml`

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

Creating a production build of the Vaadin app

Use the Maven `production` profile to create a production-optimized build. Add the `skipTests` parameter to avoid running tests.

```
mvn clean package -Pproduction -DskipTests
```

You now have a production-ready application JAR in `target/vaadin-crm-<version>.jar`. Next, we set up the AWS environment for deploying it.

Creating an AWS account

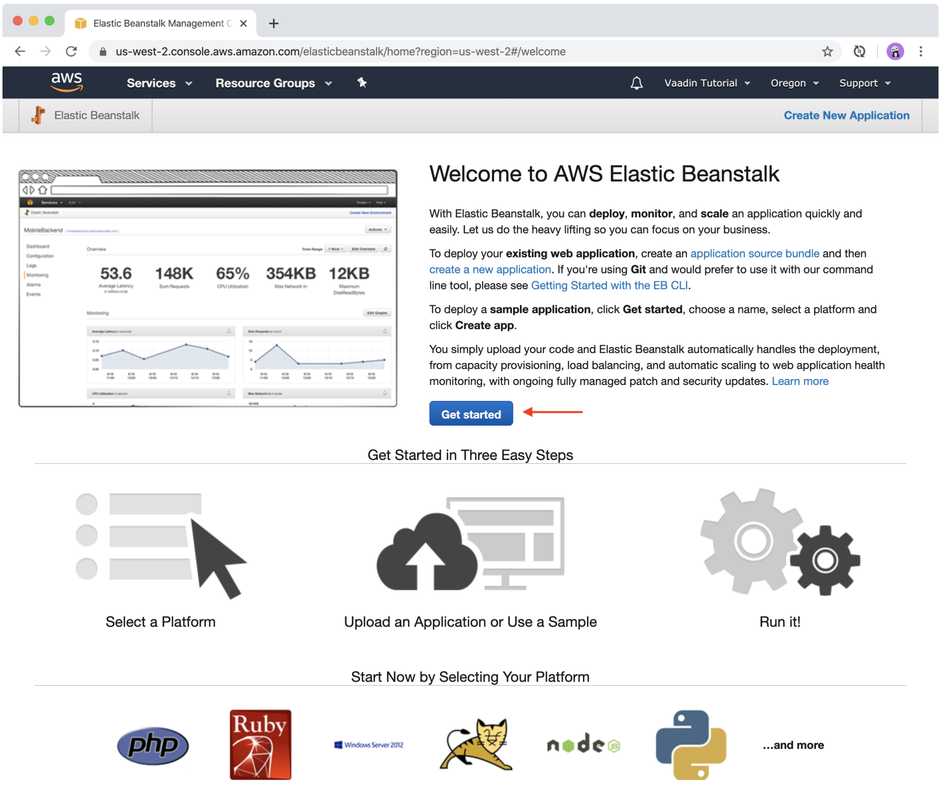
The first thing you need to do is create an AWS account, if you don't already have one. You can sign up for an AWS Free Tier account that provides limited usage of many AWS products.

WARNING	AWS is a paid service and following the instructions below may result in charges. Carefully review the billing costs on AWS to avoid any surprises.

Setting up a Java application on Elastic Beanstalk

First create the app environment on Elastic Beanstalk:

1. Create a new application in the Elastic Beanstalk console.


Elastic Beanstalk Management C
us-west-2.console.aws.amazon.com/elasticbeanstalk/home?region=us-west-2#/welcome
aws
Services
Resource Groups
Vaadin Tutorial
Oregon
Support
Elastic Beanstalk
Create New Application
53.6
148K
65%
354KB
12KB
Get started
Get Started in Three Easy Steps
Select a Platform
Upload an Application or Use a Sample
Run it!
Start Now by Selecting Your Platform
php
Ruby
Windows Server 2012
node
...and more

Welcome to AWS Elastic Beanstalk

With Elastic Beanstalk, you can **deploy**, **monitor**, and **scale** an application quickly and easily. Let us do the heavy lifting so you can focus on your business.

To deploy your **existing web application**, create an application source bundle and then create a new application. If you're using **Git** and would prefer to use it with our command line tool, please see Getting Started with the EB CLI.

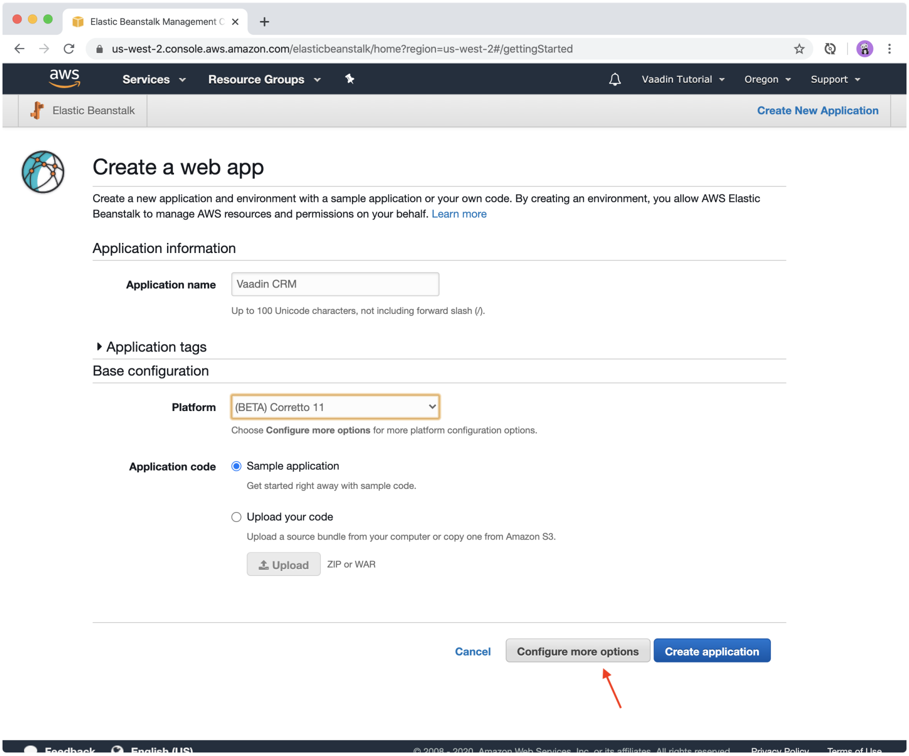
To deploy a **sample application**, click **Get started**, choose a name, select a platform and click **Create app**.

You simply upload your code and Elastic Beanstalk automatically handles the deployment, from capacity provisioning, load balancing, and automatic scaling to web application health monitoring, with ongoing fully managed patch and security updates. Learn more

2. Create an environment for the application with the following configurations, and then click **Configure more options**:

- **Application name**: Vaadin CRM
- **Environment tier**: Web server environment.
- **Platform**: Corretto 11.

- **Application code:** Sample application. You can leave the other fields empty.


Elastic Beanstalk Management C
us-west-2.console.aws.amazon.com/elasticbeanstalk/home?region=us-west-2#/gettingStarted
aws
Services
Resource Groups
Vaadin Tutorial
Oregon
Support
Elastic Beanstalk
Create New Application

Create a web app

Create a new application and environment with a sample application or your own code. By creating an environment, you allow AWS Elastic Beanstalk to manage AWS resources and permissions on your behalf. Learn more

Application information

Application name Vaadin CRM

Up to 100 Unicode characters, not including forward slash (/).

Application tags

Base configuration

Platform (BETA) Corretto 11

Choose **Configure more options** for more platform configuration options.

Application code [✓ Sample application]

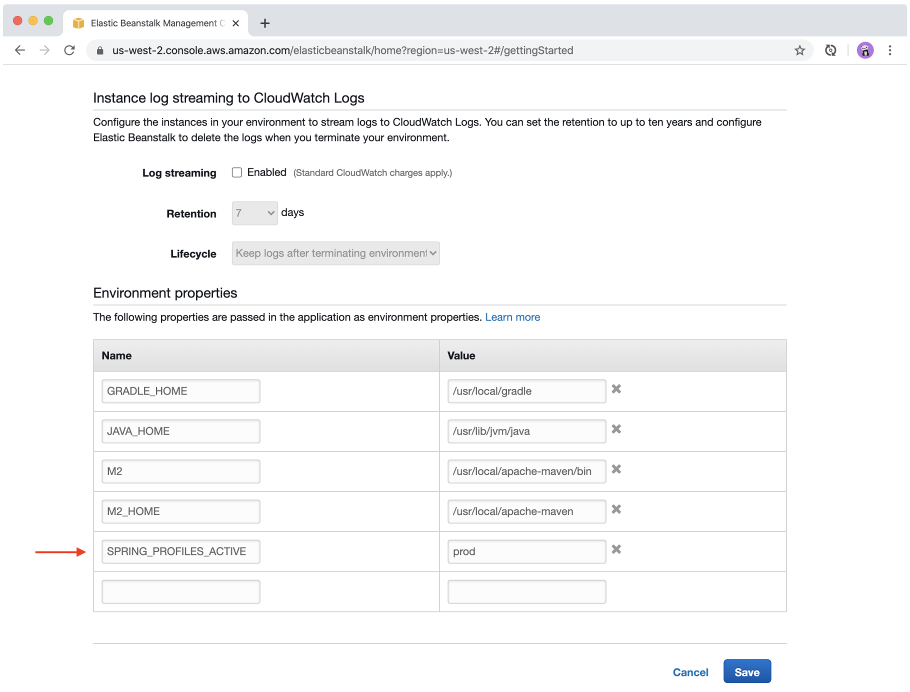
Get started right away with sample code.

Upload your code

Upload a source bundle from your computer or copy one from Amazon S3.

3. Go to **Software** > **Modify** and add an environment property, and then click **Save**:

- **Name:** SPRING_PROFILES_ACTIVE
- **Value:** prod


Elastic Beanstalk Management C
us-west-2.console.aws.amazon.com/elasticbeanstalk/home?region=us-west-2#/gettingStarted

Instance log streaming to CloudWatch Logs

Configure the instances in your environment to stream logs to CloudWatch Logs. You can set the retention to up to ten years and configure Elastic Beanstalk to delete the logs when you terminate your environment.

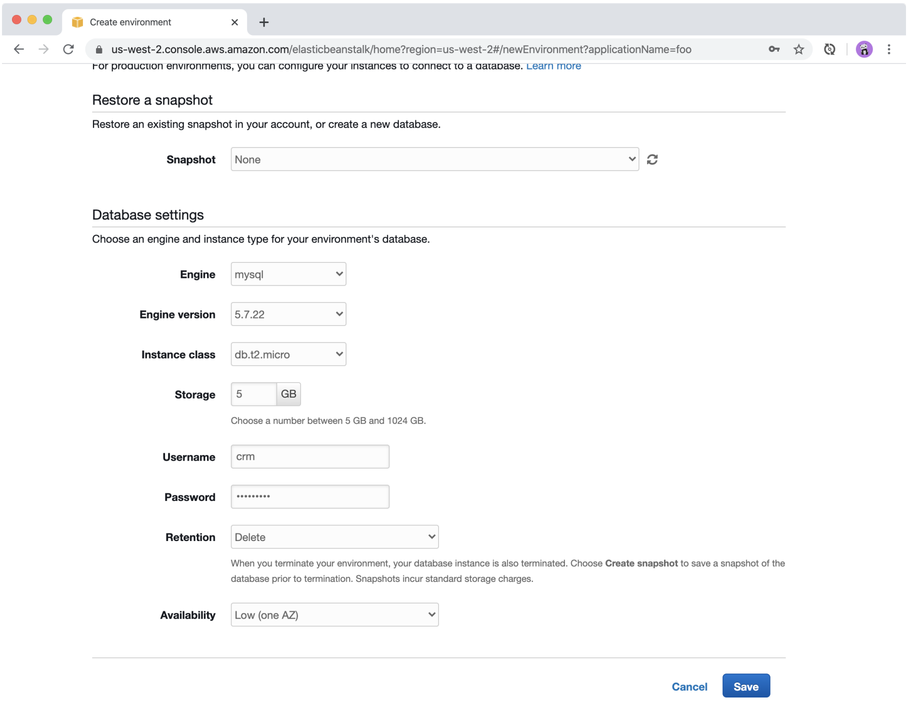
Log streaming	☐ Enabled (Standard CloudWatch charges apply.)
Retention	7 days
Lifecycle	Keep logs after terminating environment

Environment properties

The following properties are passed in the application as environment properties. Learn more

Name	Value
GRADLE_HOME	/usr/local/gradle
JAVA_HOME	/usr/lib/jvm/java
M2	/usr/local/apache-maven/bin
M2_HOME	/usr/local/apache-maven
SPRING_PROFILES_ACTIVE	prod

4. Go to **Database > Modify** and set up an Amazon RDS SQL database with the following configurations, and then click **Save**:
   - The defaults are fine for our purposes.
   - Add a username and password. Elastic Beanstalk will make these available to your application through the environment variables you set up in the properties file.


Create environment

us-west-2.console.aws.amazon.com/elasticbeanstalk/home?region=us-west-2#/newEnvironment?applicationName=foo

For production environments, you can configure your instances to connect to a database. Learn more

Restore a snapshot

Restore an existing snapshot in your account, or create a new database.

Snapshot	None

Database settings

Choose an engine and instance type for your environment's database.

Engine	mysql
Engine version	5.7.22
Instance class	db.t2.micro
Storage	5 GB Choose a number between 5 GB and 1024 GB.
Username	crm
Password	•••••••••
Retention	Delete When you terminate your environment, your database instance is also terminated. Choose **Create snapshot** to save a snapshot of the database prior to termination. Snapshots incur standard storage charges.
Availability	Low (one AZ)

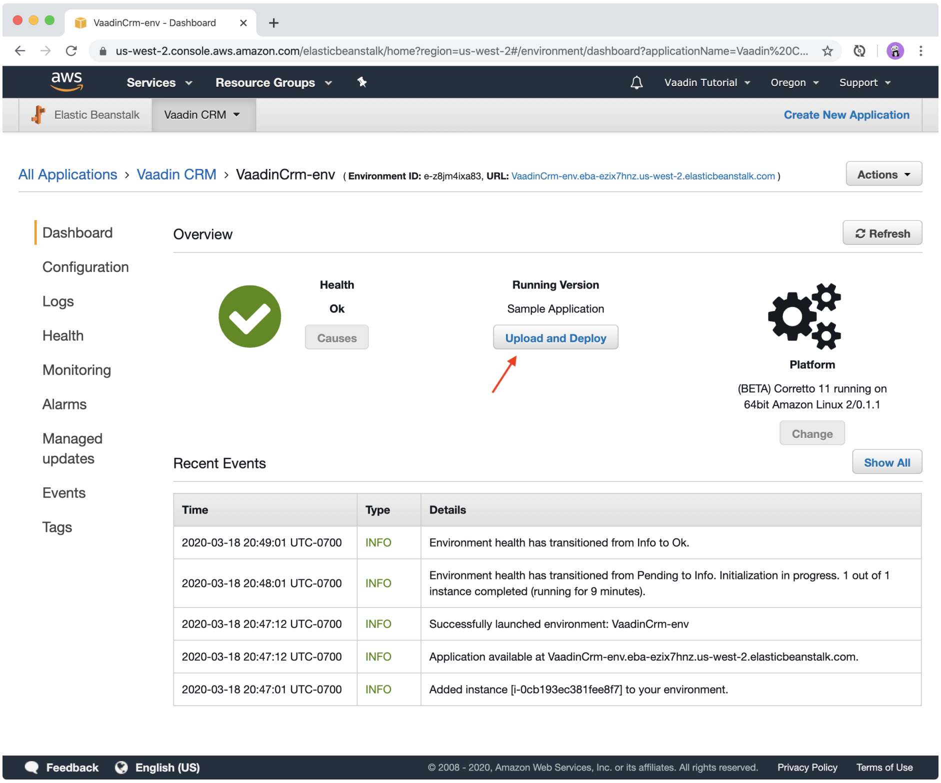
CAUTION	This database setup is suitable for the tutorial, but in a real production application, the database should not be tied to the lifecycle of the environment. Otherwise you may inadvertently delete the database if you remove the server. See Using Elastic Beanstalk with Amazon Relational Database Service.

5. Click Create app.

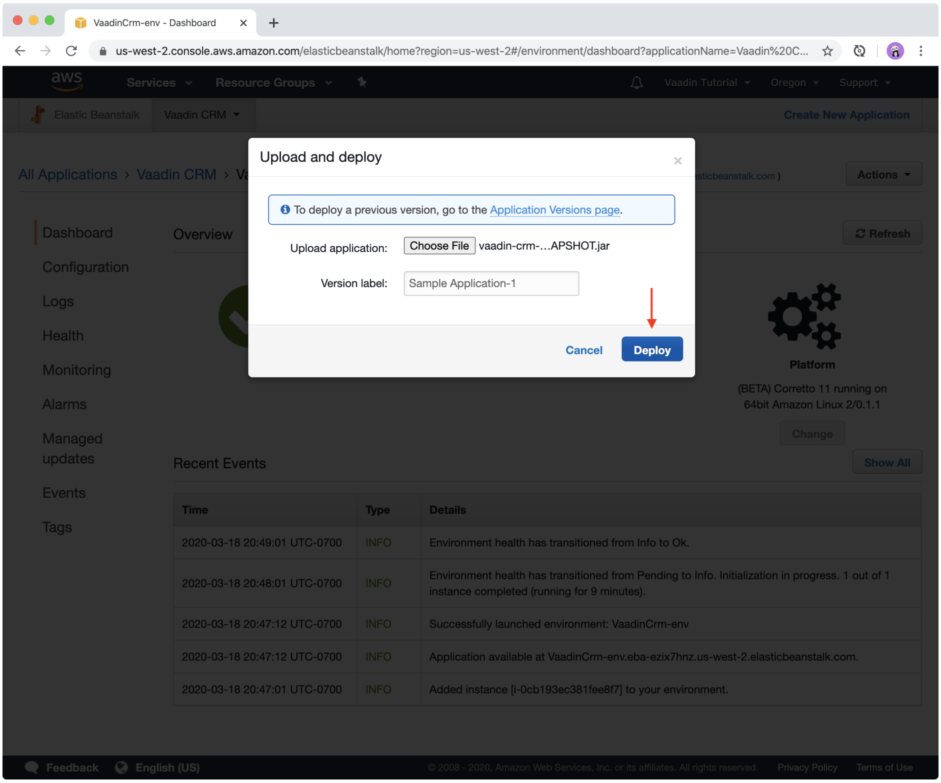
NOTE	Creating the application environment and database can take up to 15 minutes.

Deploying the Elastic Beanstalk app

1. In the EB console Dashboard, click Upload and Deploy and upload your newly-built JAR file, `target/vaadin-crm-<version>.jar`.


VaadinCrm-env - Dashboard
us-west-2.console.aws.amazon.com/elasticbeanstalk/home?region=us-west-2#/environment/dashboard?applicationName=Vaadin%20C...
Services
Resource Groups
Vaadin Tutorial
Oregon
Support
Elastic Beanstalk
Vaadin CRM
Create New Application
All Applications > Vaadin CRM > VaadinCrm-env (Environment ID: e-z8jm4ixa83, URL: VaadinCrm-env.eba-ezix7hnz.us-west-2.elasticbeanstalk.com)
Actions
Dashboard
Configuration
Logs
Health
Monitoring
Alarms
Managed updates
Events
Tags
Overview
Refresh
Health
Ok
Causes
Running Version
Sample Application
Upload and Deploy
Platform
(BETA) Corretto 11 running on 64bit Amazon Linux 2/0.1.1
Change
Recent Events
Show All
Feedback
English (US)
© 2008 - 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Time	Type	Details
2020-03-18 20:49:01 UTC-0700	INFO	Environment health has transitioned from Info to Ok.
2020-03-18 20:48:01 UTC-0700	INFO	Environment health has transitioned from Pending to Info. Initialization in progress. 1 out of 1 instance completed (running for 9 minutes).
2020-03-18 20:47:12 UTC-0700	INFO	Successfully launched environment: VaadinCrm-env
2020-03-18 20:47:12 UTC-0700	INFO	Application available at VaadinCrm-env.eba-ezix7hnz.us-west-2.elasticbeanstalk.com.
2020-03-18 20:47:01 UTC-0700	INFO	Added instance [i-0cb193ec381fee8f7] to your environment.



Upload and deploy

To deploy a previous version, go to the Application Versions page.

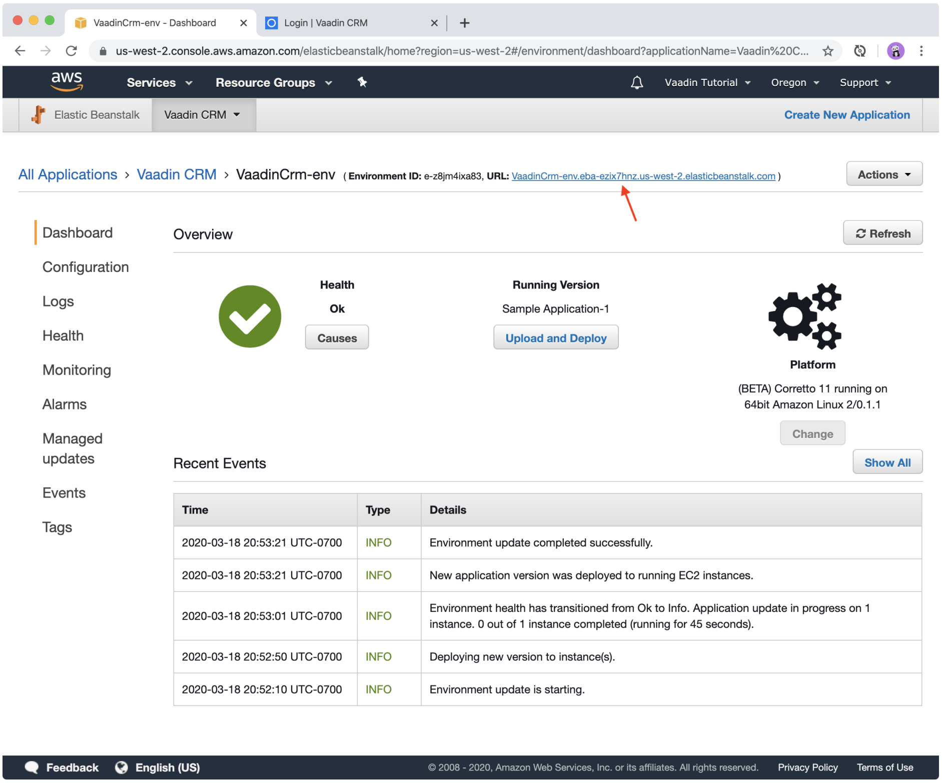
Upload application: Choose File vaadin-crm-…APSHOT.jar

Version label: Sample Application-1

Recent Events

Time	Type	Details
2020-03-18 20:49:01 UTC-0700	INFO	Environment health has transitioned from Info to Ok.
2020-03-18 20:48:01 UTC-0700	INFO	Environment health has transitioned from Pending to Info. Initialization in progress. 1 out of 1 instance completed (running for 9 minutes).
2020-03-18 20:47:12 UTC-0700	INFO	Successfully launched environment: VaadinCrm-env
2020-03-18 20:47:12 UTC-0700	INFO	Application available at VaadinCrm-env.eba-ezix7hnz.us-west-2.elasticbeanstalk.com.
2020-03-18 20:47:01 UTC-0700	INFO	Added instance [i-0cb193ec381fee8f7] to your environment.

2. After the environment has updated (this can take several minutes), the environment Health should indicate as Ok (green tick) and your application should run and be accessible on the web through the link at the top of the dashboard. If the health is not Ok, go to Logs (in the EB console) to troubleshoot the problem.



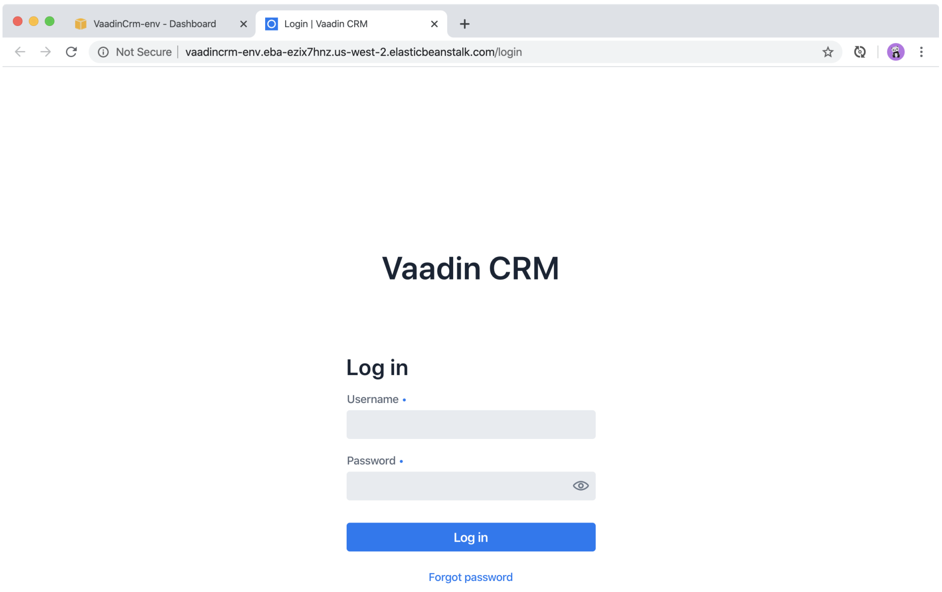
(**Environment ID:** e-z8jm4ixa83, **URL:** VaadinCrm-env.eba-ezix7hnz.us-west-2.elasticbeanstalk.com)

Health
Ok

Running Version
Sample Application-1

Platform
(BETA) Corretto 11 running on 64bit Amazon Linux 2/0.1.1

Time	Type	Details
2020-03-18 20:53:21 UTC-0700	INFO	Environment update completed successfully.
2020-03-18 20:53:21 UTC-0700	INFO	New application version was deployed to running EC2 instances.
2020-03-18 20:53:01 UTC-0700	INFO	Environment health has transitioned from Ok to Info. Application update in progress on 1 instance. 0 out of 1 instance completed (running for 45 seconds).
2020-03-18 20:52:50 UTC-0700	INFO	Deploying new version to instance(s).
2020-03-18 20:52:10 UTC-0700	INFO	Environment update is starting.


VaadinCrm-env - Dashboard
Login | Vaadin CRM
Not Secure
vaadincrm-env.eba-ezix7hnz.us-west-2.elasticbeanstalk.com/login
Vaadin CRM
Log in
Username
Password
Log in
Forgot password

You can find the completed source code for this tutorial on GitHub.

Next steps

Good job on completing the tutorial series! You now have all the skills you need to get started building real-life applications with Spring Boot and Vaadin.

You can find more information about both in the respective frameworks' documentation:

- Spring Boot documentation
- Vaadin documentation

This guide is specifically designed as a practical introduction to web application development using Java. It covers the entire development process, from setup to deployment, following a step-by-step approach. You can replicate each section at your own pace as you follow along.

Learn more on Vaadin at
vaadin.com/learn