BUILDING MODERN WEB APPS

WITH

Spring Boot & Vaadin

A practical introduction to web application development using Java

vaadin}>

Table of Contents

A practical guide to Spring Boot and Vaadin

Vaadin Flow Project Setup

This part of this tutorial covers downloading a Vaadin application starter, and importing a Vaadin Maven project in IntelliJ. Plus, it explains how to configure IntelliJ for productive development.

Download Vaadin Application Starter

This tutorial uses a pre-configured starter from Vaadin Start. The starter application includes a few essential items:

- JPA data model consisting of `Contact`, `Company`, and `Status` JPA entities;
- Spring Data repositories for persisting and retrieving the entities from an embedded H2 database;
- `data.sql` file containing some test data;
- Single, empty view; and a
- Dockerfile.

To begin, download the starter application, which is a zip file. You can find it here:

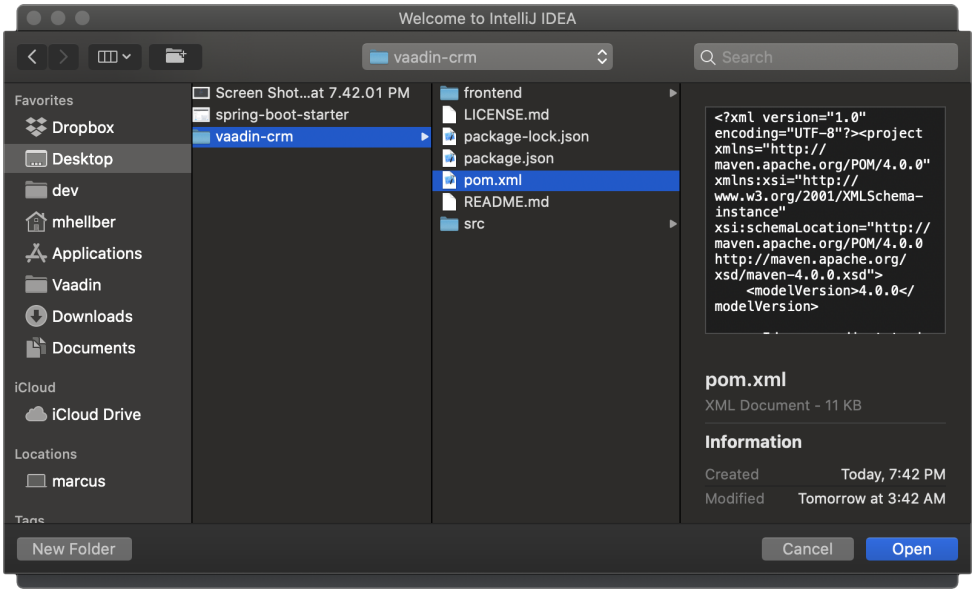
https://start.vaadin.com/dl?preset=flow-crm-tutorial&preset=partial-prerelease

Import a Maven Project into IntelliJ

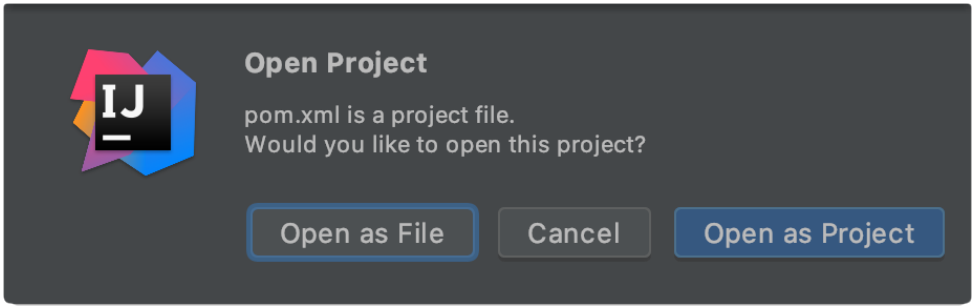
Having downloaded the zip archive file, you'll first have to unzip it somewhere. Any directory is fine: just don't unzip it to the download folder since you might unintentionally delete your project later when clearing out old downloads.

In IntelliJ, select **Open** in the Welcome screen or **File** menu, as you can see in the screenshot here:

Next, from the directory tree that is displayed, find the folder where you extracted the files. Select the `pom.xml` file and click the **Open** button.

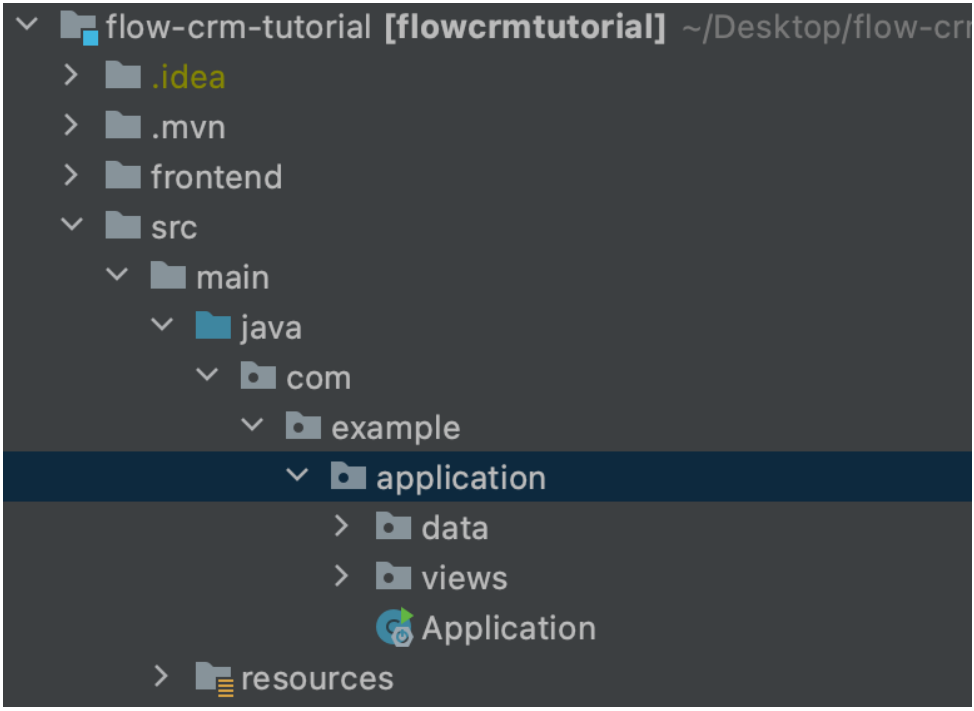


Select **Open as Project** from the dialog, as seen in the following screenshot:



This imports the project based on the `pom.xml` file. IntelliJ then imports the project and downloads all necessary dependencies. It can take several minutes, depending on the internet connection speed.

When the import is complete, your project structure should look as seen in the screenshot here:



Notice that the Java source files are in the `src/main/java` folder.

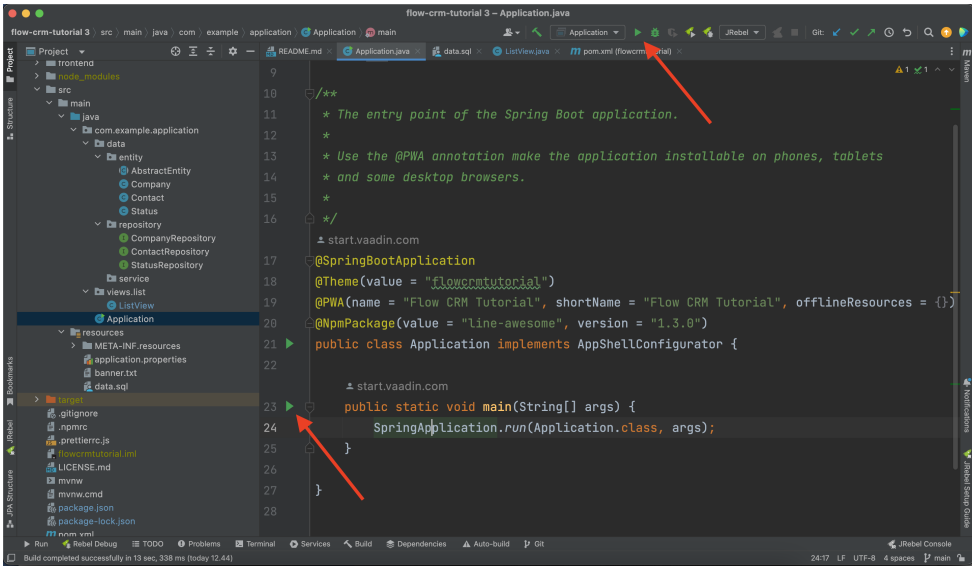
Run a Spring Boot Project in IntelliJ

Spring Boot makes it easier to run a Java web application because it handles starting and configuring the server.

To run your application, run the Application class that contains the `main()` method that starts Spring Boot. IntelliJ detects automatically that you have a class with a `main()` method and displays it in the **run configurations** drop-down.

To start your application, open `Application.java` and click the play button next to the code line containing the `main()` method.

After you've run the application once from the `main()` method, it will appear in the **run configurations** drop-down in the main toolbar (see screenshot). On subsequent runs, you can run the application from there.

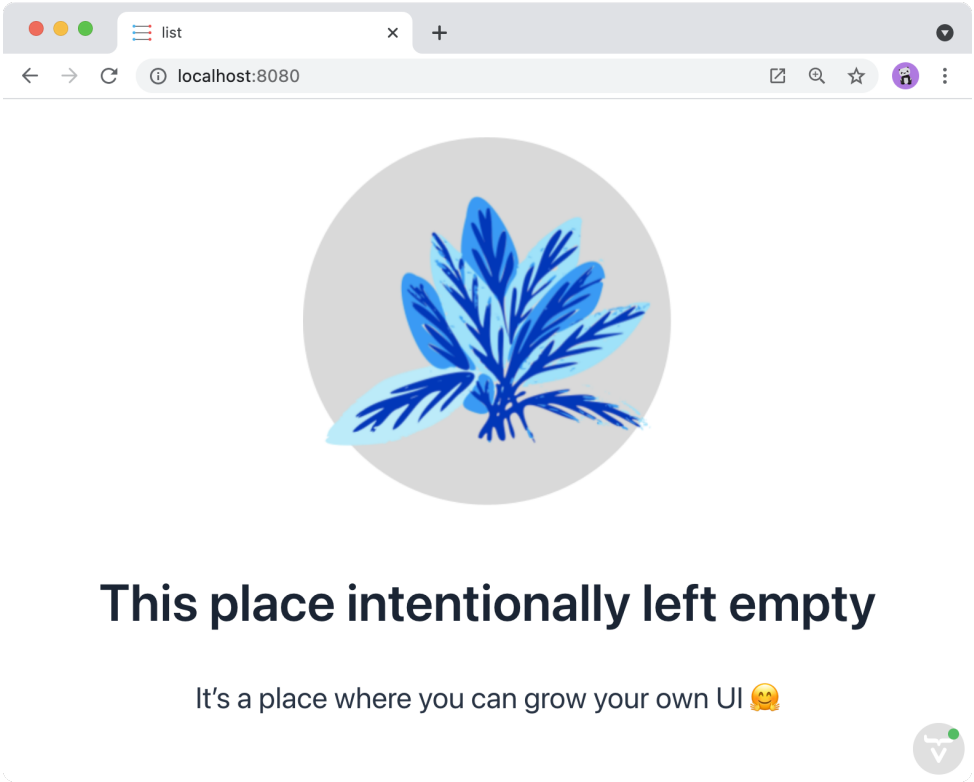


The first time you start a Vaadin application, it downloads front-end dependencies and builds a JavaScript bundle. IntelliJ indexes all the added dependencies. It won't need to do that when run subsequently.

You'll know that your application has started when you see output in the console similar to what you see here:

```
Tomcat started on port(s): 8080 (http) with context path ''
```

The development mode in Vaadin also opens a browser window for you automatically. You'll see a content placeholder and image similar to the screenshot here:

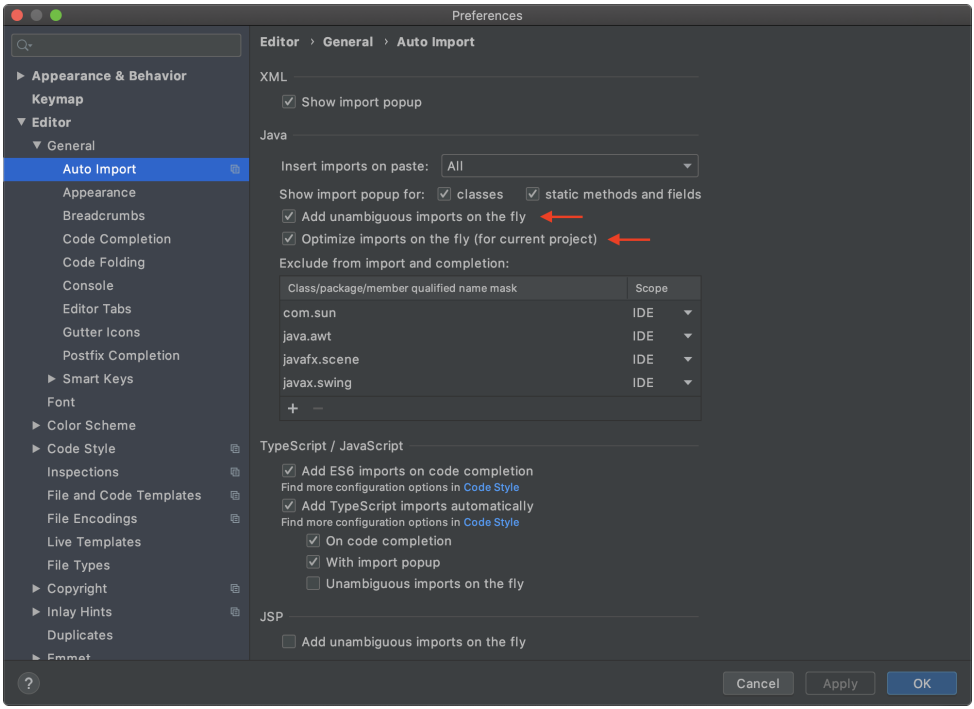


Enable Auto Import in IntelliJ

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

To enable auto import in IntelliJ, open the Preferences/Settings window and navigate to Editor → General → Auto Import.

From there, you can enable the following two options: **Add unambiguous imports on the fly**; and **Optimize imports on the fly**. You can see the checkboxes for these choices in the screenshot here:



Vaadin shares many class names (e.g., Button) with Swing, Java Abstract Window Toolkit (AWT), and JavaFX.

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, you're ready to start building a web application.

`3C607714-1A52-49F0-9CB6-809F7A59F608`

Create a Vaadin Flow View with Components

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

On this part, you'll learn core Vaadin concepts and scaffold the first view of the Custom Relationship Management (CRM) application. It covers Vaadin component basics and constructing a view with the Vaadin Flow Java API.

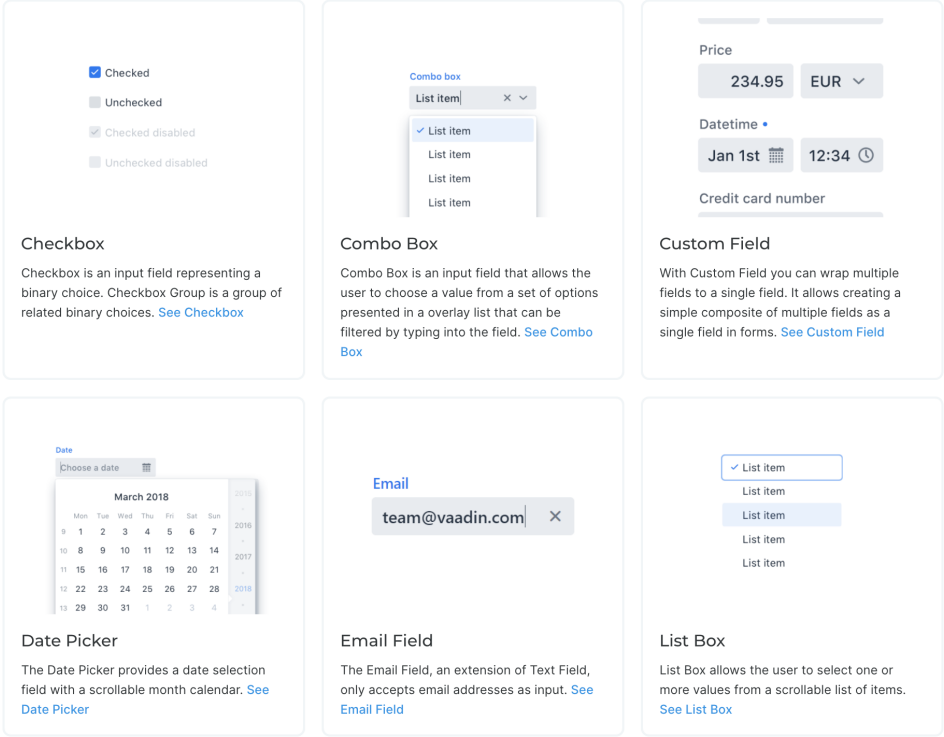
Basic Elements

Before getting into the details of creating a view, it's important to understand the basic elements that you'll use.

Vaadin UI Components

Vaadin includes over forty UI components to help you build applications faster. By using ready-made building blocks, you're able to focus primarily on building end-user functionality.

Vaadin components are custom HTML elements that are registered with the browser. They are based on W3C web component standards. The components have light and dark themes that can be customized with CSS variables to fit your brand.



You can create a new component by initializing a Java object. For instance, to create a `Button`, you would write something like this:

Creating a Button

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

Layouts

Layouts decide how components are positioned 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, whereas `Div` lets you control the positioning with CSS.

`HorizontalLayout` and `VerticalLayout` have methods to align items on both the primary and the cross axis. For example, 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`.

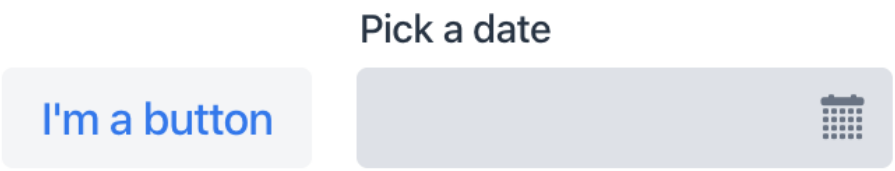
Using this example, and the `add()` method to add components to layouts, you would do something like this:

Setting layout alignment

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

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

The result of the above is what you see in the screenshot here:



UI Events

You can add functionality to your application by listening to events. Events can include button clicks and value changes from select components.

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

Listening to click events

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

HTML

One unique Vaadin Flow feature is that you can build web applications entirely in Java, eliminating the need to write common HTML. This higher level of abstraction makes development more productive and debugging easier.

Vaadin does support HTML templates and customizing the code that runs in the browser. However, you don't usually have to worry about this.

The Contact List View

The first view is the Contact list view. You can see how it looks in the screenshot here. It lists all contacts. Users can search, add, edit, and delete contacts in this view.

You'll focus initially only on the list view. You'll add a layout containing the header and sidebar later on the "Navigation & App Layout" part.

On this part and the next, you'll create the required layouts and components for the view. Then, on the part that follows, you'll create a service class for accessing the backend and populating the view with data.

Start by locating the `ListView.java` class under `src/main/java`. Then replace the contents of the file with the following:

ListView.java

```
package com.example.application.views.list;

import com.example.application.data.entity.Contact;
import com.vaadin.flow.component.Component;
import com.vaadin.flow.component.button.Button;
import com.vaadin.flow.component.grid.Grid;
import com.vaadin.flow.component.orderedlayout.HorizontalLayout;
import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.component.textfield.TextField;
import com.vaadin.flow.data.value.ValueChangeMode;
import com.vaadin.flow.router.PageTitle;
import com.vaadin.flow.router.Route;

@Route(value = "")
@PageTitle("Contacts | Vaadin CRM")
public class ListView extends VerticalLayout { ①
    Grid<Contact> grid = new Grid<>(Contact.class); ②
    TextField filterText = new TextField();

    public ListView() {
        addClassName("list-view"); ⑩
        setSizeFull();
        configureGrid(); ③

        add(getToolbar(), grid); ④
    }

    private void configureGrid() {
        grid.addClassNames("contact-grid"); ⑩
        grid.setSizeFull();
        grid.setColumns("firstName", "lastName", "email"); ⑤
        grid.addColumn(contact -> contact.getStatus().getName()).setHeader("Status"
); ⑥
        grid.addColumn(contact -> contact.getCompany().getName()).setHeader("
Company");
        grid.getColumns().forEach(col -> col.setAutoWidth(true)); ⑦
    }

    private HorizontalLayout getToolbar() {
        filterText.setPlaceholder("Filter by name...");
        filterText.setClearButtonVisible(true);
        filterText.setValueChangeMode(ValueChangeMode.LAZY); ⑧

        Button addContactButton = new Button("Add contact");

        var toolbar = new HorizontalLayout(filterText, addContactButton); ⑨
        toolbar.addClassName("toolbar"); ⑩
        return toolbar;
    }
}
```

The following are comments about the code before. The numbers reference the circled

numbers at the end of various lines of code.

① The view extends `VerticalLayout`, which places all child components vertically.

② The Grid component is typed with `Contact`.

③ The grid configuration is extracted to a separate method to keep the constructor easier to read.

④ Add the toolbar and grid to the `VerticalLayout`.

⑤ Define which properties of `Contact` the grid should show.

⑥ Define custom columns for nested objects.

⑦ Configure the columns to adjust automatically their size to fit their content.

⑧ Configure the search field to fire value-change events only when the user stops typing. This way you avoid unnecessary database calls, but the listener is still fired without the user leaving the focus from the field.

⑨ The toolbar uses a `HorizontalLayout` to place the `TextField` and `Button` next to each other.

⑩ Adding some class names to components makes it easier to style the application later using CSS.

If your application is still running from the previous step, you only need to perform a build, either with the kbd:[Command+F9]/kbd:[Ctrl+F9] keyboard shortcut, or by pressing the "hammer" icon in the toolbar (see cropped screenshot). Vaadin automatically reloads your browser to display the changes.

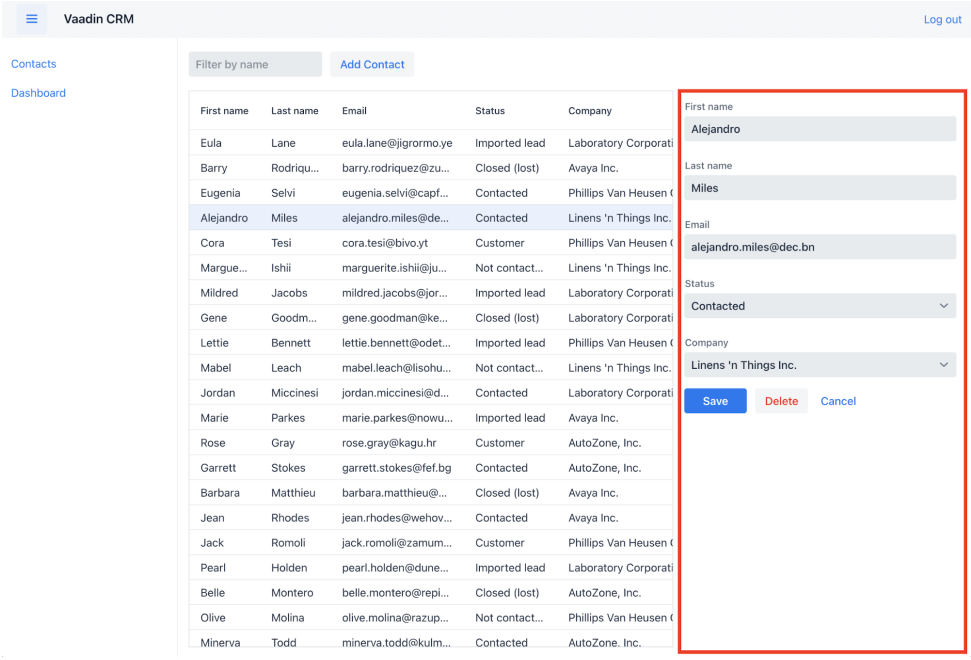
Incidentally, you can keep the server running throughout this tutorial. You only need to restart the server in a couple of instances. These are highlighted in the instructions.

You should now see the empty view structure in the browser window. On the next part, you'll build a component for the form that's used for editing contacts.

`79C51513-862E-47EC-829D-9A149C06F7A0`

Create a Form Component for Editing Contacts

The list view now has a grid to display `Contact` objects. To complete the view, you need to create a form for editing contacts — like the one in the screenshot here:



This part covers creating a new component, as well as importing and using a custom component.

Components Using Composition

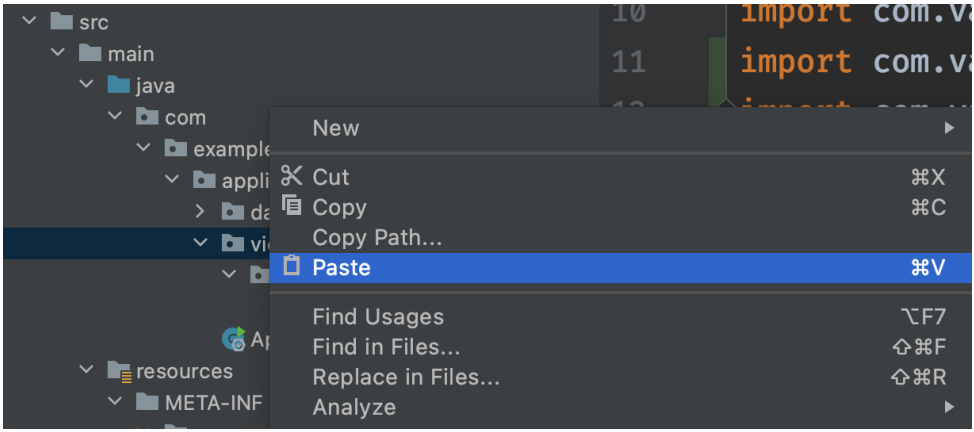
Vaadin Flow is a component-based framework. In the previous parts here, you worked with several components, like `Grid`, `TextField`, and `VerticalLayout`. However, the real power of the component-based architecture is in the ability to create your own components.

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.

Form Component

The form component you'll create needs text fields for the first and last name, an email field, and two select fields: one to select the company, and another to select the contact status.

Start by creating a new file, `ContactForm.java`, in the `com.example.application.views.list` package. If you're using IntelliJ, copy the code below and paste it into the `views` package. IntelliJ automatically creates the file.



ContactForm.java

```
package com.example.application.views.list;

import com.example.application.data.entity.Company;
import com.example.application.data.entity.Status;
import com.vaadin.flow.component.Key;
import com.vaadin.flow.component.button.Button;
import com.vaadin.flow.component.button.ButtonVariant;
import com.vaadin.flow.component.combobox.ComboBox;
import com.vaadin.flow.component.formlayout.FormLayout;
import com.vaadin.flow.component.orderedlayout.HorizontalLayout;
import com.vaadin.flow.component.textfield.EmailField;
import com.vaadin.flow.component.textfield.TextField;

import java.util.List;

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<Status> status = new ComboBox<>("Status");
  ComboBox<Company> company = new ComboBox<>("Company");

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

  public ContactForm(List<Company> companies, List<Status> statuses) {
    addClassName("contact-form"); ③

    company.setItems(companies);
    company.setItemLabelGenerator(Company::getName);
    status.setItems(statuses);
    status.setItemLabelGenerator(Status::getName);

    add(firstName, ④
        lastName,
        email,
        company,
        status,
        createButtonsLayout());
  }

  private HorizontalLayout createButtonsLayout() {
    save.addThemeVariants(ButtonVariant.LUMO_PRIMARY); ⑤
    delete.addThemeVariants(ButtonVariant.LUMO_ERROR);
    close.addThemeVariants(ButtonVariant.LUMO_TERTIARY);

    save.addClickShortcut(Key.ENTER); ⑥
    close.addClickShortcut(Key.ESCAPE);

    return new HorizontalLayout(save, delete, close); ⑦
  }
}
```

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

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

③ Gives the component a CSS class name, so you can style it later.

④ Adds all the UI components to the layout. The buttons require a bit of extra configuration. Create and call a new method, `createButtonsLayout()`.

⑤ Makes the buttons visually distinct from each other using built-in theme variants.

⑥ Defines keyboard shortcuts: `Enter` to save and `Escape` to close the editor.

⑦ Returns a `HorizontalLayout` containing the buttons to place them next to each other.

Add Form to Main View

The next step is to add the form you created to the main view. To do this, change `ListView` as follows:

ListView.java

```
public class ListView extends VerticalLayout {
    Grid<Contact> grid = new Grid<>(Contact.class);
    TextField filterText = new TextField();
    ContactForm form; ①

    public ListView() {
        addClassName("list-view");
        setSizeFull();
        configureGrid();
        configureForm(); ②

        add(getToolbar(), getContent()); ③
    }

    private Component getContent() {
        HorizontalLayout content = new HorizontalLayout(grid, form);
        content.setFlexGrow(2, grid); ④
        content.setFlexGrow(1, form);
        content.addClassNames("content");
        content.setSizeFull();
        return content;
    }

    private void configureForm() {
        form = new ContactForm(Collections.emptyList(), Collections.emptyList()); ⑤
        form.setWidth("25em");
    }

    // Remaining methods omitted
}
```

① Creates a reference to the form so you have access to it from other methods.

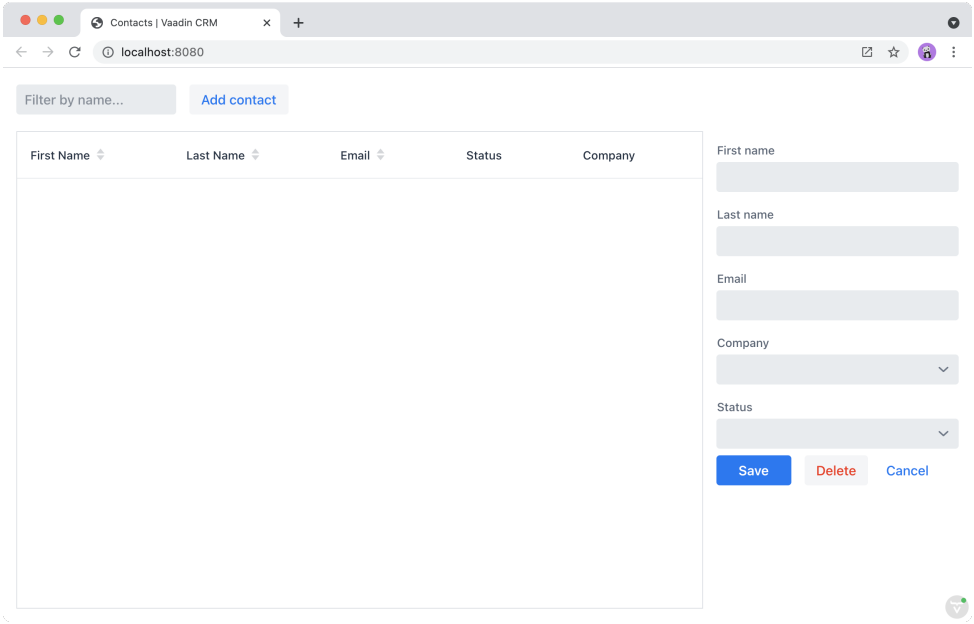
② Create a method for initializing the form.

③ Change the `add()` method to call `getContent()`. The method returns a `HorizontalLayout` that wraps the form and the grid, showing them next to each other.

④ Use `setFlexGrow()` to specify that the Grid should have twice the space of the form.

⑤ Initialize the form with empty company and status lists: you'll add these on the next part.

You can now build the project to reload the browser. You should see the form on the right side of the grid.



Now that you have the view built, it's time to connect it to the backend.

2B0A44E7-14EA-4FF5-ACC0-983F03C27AC4

Connect a View to the Backend

On the previous part of this tutorial, you created a view using Vaadin components and layouts. On this part, you'll connect the view to the backend to display and update data.

You can find the back-end code in the `src/main/java` directory.

This part of this tutorial covers three main aspects of this:

- Spring Boot introduction;
- Spring Service Interface to the backend; and
- Accessing a service from a view.

Introduction to Spring Boot

Vaadin uses Spring Boot on the server. Spring Boot is an opinionated, convention-over-configuration approach to creating Spring applications. It automates much of the required configuration and manages an embedded Tomcat server. So, you don't need to deploy the application to a separate server.

This tutorial uses the following features that are configured by Spring Boot:

- Spring Data for accessing the database through JPA and Hibernate;
- An embedded H2 Database for development (easy to replace with e.g. PostgreSQL for production);
- Spring Boot DevTools for automatic code reload;
- Embedded Tomcat server for deployment; and
- Spring Security for authenticating users.

Backend Overview

The starter you downloaded contains the *entities* and *repositories* you need. It also contains sample data loaded using `src/main/resources/data.sql` file.

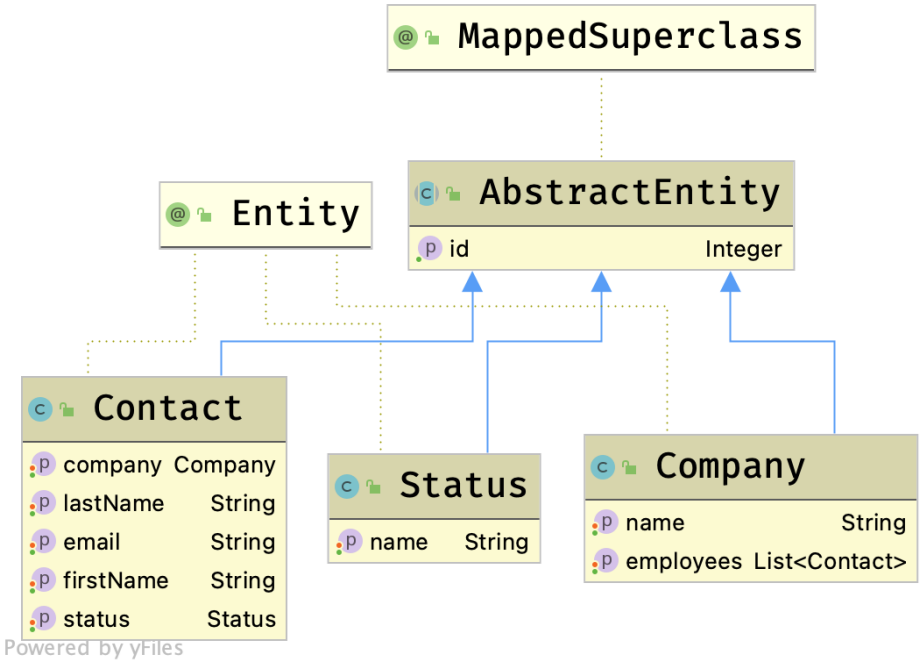
Domain Model: Entities

<!-- vale Vaadin.Abbr = NO -->

The Vaadin CRM application has three JPA entities that make up its domain model: `Contact`, `Company`, and `Status`. A contact belongs to a company and has a status.

<!-- vale Vaadin.Abbr = YES -->

You can find the entities in the `com.example.application.data.entity` package.



Database Access: Repositories

The application uses Spring Data JPA repositories for database access. Spring Data provides implementations of basic create, read, update, and delete (i.e., CRUD) database operations when you extend from the `JpaRepository` interface.

You can find the repositories in the `com.example.application.data.repository` package.

Sample Data

The `src/main/resources/data.sql` file contains sample data that Spring Boot populates to the database on startup.

Create a Service for Database Access

Instead of accessing the database directly from the view, you would create a Spring Service. The service class handles the application's business logic and, in larger applications, it often transforms database entities into Data-Transfer Objects (DTO) for views. This tutorial shows how to create a single service that provides all of the methods you need.

First, create a new class, `CrmService.java`, in the `data.service` package with the following content:

`CrmService.java`

```
package com.example.application.data.service;

import com.example.application.data.entity.Company;
import com.example.application.data.entity.Contact;
import com.example.application.data.entity.Status;
import com.example.application.data.repository.CompanyRepository;
import com.example.application.data.repository.ContactRepository;
import com.example.application.data.repository.StatusRepository;
import org.springframework.stereotype.Service;

import java.util.List;

@Service ①
public class CrmService {

    private final ContactRepository contactRepository;
    private final CompanyRepository companyRepository;
    private final StatusRepository statusRepository;

    public CrmService(ContactRepository contactRepository,
                      CompanyRepository companyRepository,
                      StatusRepository statusRepository) { ②
        this.contactRepository = contactRepository;
        this.companyRepository = companyRepository;
        this.statusRepository = statusRepository;
    }

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

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

    public void deleteContact(Contact contact) {
```

```
        contactRepository.delete(contact);
    }

    public void saveContact(Contact contact) {
        if (contact == null) { ④
            System.err.println("Contact is null. Are you sure you have connected
your form to the application?");
            return;
        }
        contactRepository.save(contact);
    }

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

    public List<Status> findAllStatuses(){
        return statusRepository.findAll();
    }
}
```

① The `@Service` annotation makes this a Spring-managed service that you can inject into your view.

② Use Spring constructor injection to autowire the database repositories.

③ Check if there's an active filter: return either all contacts, or use the repository to filter based on the string.

④ Service classes often include validation and other business rules before persisting data. You check here that you aren't trying to save a `null` object.

Implement Filtering in the Repository

Add the `search()` method to the contacts repository so as to provide the service class with the required method for filtering contacts.

`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
}
```

This example uses the `@Query` annotation to define a custom query (see annotation 1). 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.

You don't need to implement the method. Spring Data provides the implementation based on the query.

Using Back-End Service

You can now inject the `CrmService` into the list view to access the backend.

`ListView.java`

```
package com.example.application.views.list;

import com.example.application.data.entity.Contact;
import com.example.application.data.service.CrmService;
import com.vaadin.flow.component.Component;
import com.vaadin.flow.component.button.Button;
import com.vaadin.flow.component.grid.Grid;
import com.vaadin.flow.component.orderedlayout.HorizontalLayout;
import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.component.textfield.TextField;
import com.vaadin.flow.data.value.ValueChangeMode;
import com.vaadin.flow.router.PageTitle;
import com.vaadin.flow.router.Route;

@Route(value = "")
@PageTitle("Contacts | Vaadin CRM")
public class ListView extends VerticalLayout {
    Grid<Contact> grid = new Grid<>(Contact.class);
    TextField filterText = new TextField();
    ContactForm form;
    CrmService service;

    public ListView(CrmService service) { ①
        this.service = service;
        addClassName("list-view");
        setSizeFull();
        configureGrid();
        configureForm();

        add(getToolbar(), getContent());
        updateList(); ②
    }

    private Component getContent() {
        HorizontalLayout content = new HorizontalLayout(grid, form);
        content.setFlexGrow(2, grid);
        content.setFlexGrow(1, form);
        content.addClassNames("content");
        content.setSizeFull();
        return content;
```

```
    }

    private void configureForm() {
        form = new ContactForm(service.findAllCompanies(), service.findAllStatuses(
)); ③
        form.setWidth("25em");
    }

    private void configureGrid() {
        grid.addClassNames("contact-grid");
        grid.setSizeFull();
        grid.setColumns("firstName", "lastName", "email");
        grid.addColumn(contact -> contact.getStatus().getName()).setHeader("Status"
);
        grid.addColumn(contact -> contact.getCompany().getName()).setHeader("
Company");
        grid.getColumns().forEach(col -> col.setAutoWidth(true));
    }

    private HorizontalLayout getToolbar() {
        filterText.setPlaceholder("Filter by name...");
        filterText.setClearButtonVisible(true);
        filterText.setValueChangeMode(ValueChangeMode.LAZY);
        filterText.addValueChangeListener(e -> updateList()); ④

        Button addContactButton = new Button("Add contact");

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

    private void updateList() { ⑤
        grid.setItems(service.findAllContacts(filterText.getValue()));
    }
}
```

① Autowire `CrmService` through the constructor. Save it in a field, so you can access it from other methods.

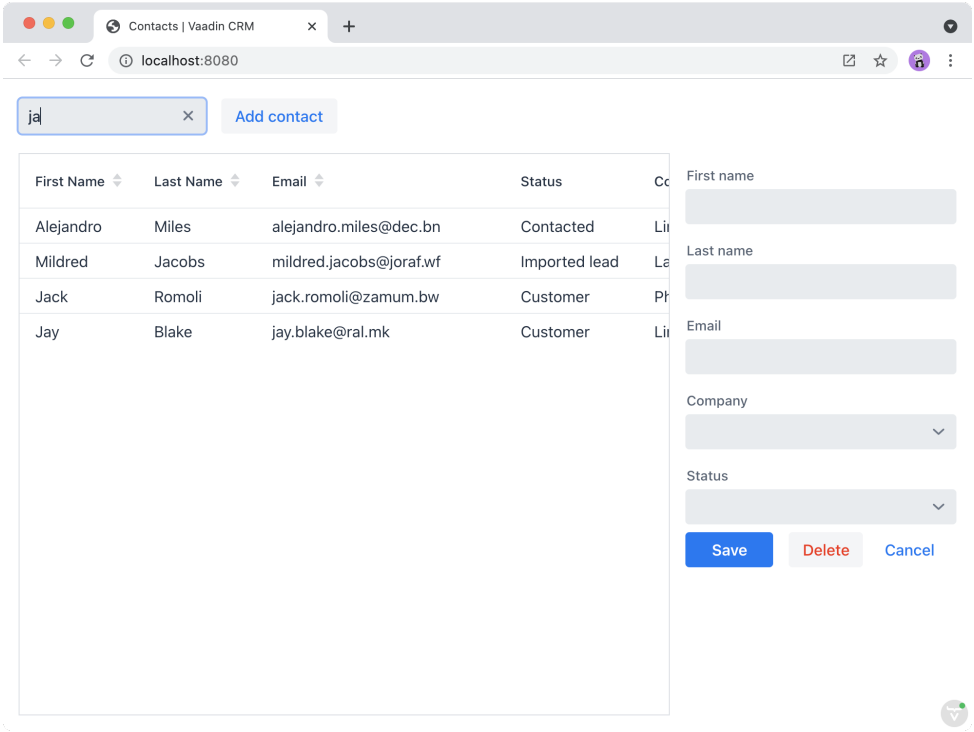
② Call `updateList()` once you have constructed the view.

③ Use the service to fetch companies and statuses.

④ Call `updateList()` any time the filter changes.

⑤ `updateList()` sets the grid items by calling the service with the value from the filter text field.

Now build the project, refresh the browser, and verify that you can now see contacts in the grid. It should look like the screenshot here. Try filtering the contents by typing in the filter text field.



1EA60808-40B7-4F0C-8B71-C0CB905299D2

Vaadin Forms: Data Binding & Validation

On the Create a Component part of this tutorial, you created the input fields and buttons that you need for editing contacts. On this part, you'll bind those inputs to a `Contact` object to create a fully functional form with validation.

This part covers creating a Vaadin Binder, binding input fields, and field validation.

Use Vaadin Binder to Create a Form & Validate Input

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 filled with input values from the UI.

Vaadin Binder 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. Vaadin uses the `Binder` class to build forms.

NOTE

Advanced Binder API

Binder also supports an advanced API where you can configure data conversions and additional validation rules. For this application, though, 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 both in the browser and before saving to the database, without duplicating code.

Bean Validation Rules in Java

You can define data validation rules as Java Bean Validation annotations on the Java class. You can see all of the applied validation rules by inspecting `Contact.java`. The validations are placed above the field declarations like this:

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

Create the Binder

Instantiate a Binder and use it to bind the input fields like this:

ContactForm.java

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

public ContactForm(List<Company> companies, List<Status> statuses) {
    addClassName("contact-form");
    binder.bindInstanceFields(this); ②
    // Rest of constructor omitted
}
```

① `BeanValidationBinder` is a `Binder` that's aware of bean validation annotations. By passing it in the `Contact.class`, you define the type of object to which you're binding.

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

With these two lines of code, you've prepared the UI fields to be connected to a contact, which is the next step.

Set the Contact

You're ready now to create a setter for the `contact`. Unlike the companies and statuses, it can change over time as a user browses through the contacts.

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

ContactForm.java

```
public class ContactForm extends FormLayout {

    public void setContact(Contact contact) {
        binder.setBean(contact); ①
    }
}
```

① Calls `binder.setBean()` to bind the values from the contact to the UI fields. The method also adds value change listeners to update changes in the UI back to the domain object.

Set Up Component Events

Vaadin comes with an event-handling system for components. You've already used it to listen to value-change events from the filter Text Field in the main view. The form component should have a similar way of informing parent components of events.

A few events can be fired: `SaveEvent`; `DeleteEvent`; and `CloseEvent`. To define new events,

add the following code at the end of the `ContactForm` class:

`ContactForm.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 Registration addDeleteListener(ComponentEventListener<DeleteEvent> listener)
{ ②
  return addListener(DeleteEvent.class, listener);
}

public Registration addSaveListener(ComponentEventListener<SaveEvent> listener) {
  return addListener(SaveEvent.class, listener);
}
public Registration addCloseListener(ComponentEventListener<CloseEvent> listener) {
  return addListener(CloseEvent.class, listener);
}
```

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

② The `add*Listener()` methods that passes the well-typed event type to Vaadin's event bus to register the custom event types. Select the `com.vaadin` import for `Registration` if

IntelliJ asks.

Save, Delete, & Close the Form

With the event types defined, you 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`

```
private Component createButtonsLayout() {
  save.addThemeVariants(ButtonVariant.LUMO_PRIMARY);
  delete.addThemeVariants(ButtonVariant.LUMO_ERROR);
  close.addThemeVariants(ButtonVariant.LUMO_TERTIARY);

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

  save.addClickListener(event -> validateAndSave()); ①
  delete.addClickListener(event -> fireEvent(new DeleteEvent(this, binder.getBean()
))); 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() {
  if(binder.isValid()) {
    fireEvent(new SaveEvent(this, binder.getBean())); ⑤
  }
}
```

① The **save** button calls the `validateAndSave()` method.

② The **delete** button triggers a delete event and passes the active contact.

③ The **cancel** button fires a close event.

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

⑤ Fire a save event, so the parent component can handle the action.

<!-- vale Vaadin.ThereIs = NO -->

Now, build the project and verify that it compiles. There won't be, though, any visible changes yet.

On the next part of this tutorial, you'll connect the form to the list view to complete the first

view.

<!-- vale Vaadin.ThereIs = YES -->

D788B762-1531-4C0C-A207-BB01672A413F

Passing Data & Events among Vaadin Components

On the previous part of this tutorial, you created a reusable form component to edit contacts. Now you'll connect it to the rest of the view and manage the view state.

The form shows the selected contact in the grid. It's hidden when no contact is selected. It also saves and deletes contacts in the database.

Show Selected Contact in Form

The first step is to show the selected grid row in the form. To do this, update `ListView` as follows:

`ListView.java`

```
package com.example.application.views.list;

import com.example.application.data.entity.Contact;
import com.example.application.data.service.CrmService;
import com.vaadin.flow.component.Component;
import com.vaadin.flow.component.button.Button;
import com.vaadin.flow.component.grid.Grid;
import com.vaadin.flow.component.orderedlayout.HorizontalLayout;
import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.component.textfield.TextField;
import com.vaadin.flow.data.value.ValueChangeMode;
import com.vaadin.flow.router.PageTitle;
import com.vaadin.flow.router.Route;

@Route(value = "")
@PageTitle("Contacts | Vaadin CRM")
public class ListView extends VerticalLayout {
    Grid<Contact> grid = new Grid<>(Contact.class);
    TextField filterText = new TextField();
    ContactForm form;
    CrmService service;

    public ListView(CrmService service) {
        this.service = service;
        addClassName("list-view");
        setSizeFull();
        configureGrid();
        configureForm();

        add(getToolbar(), getContent());
        updateList();
        closeEditor(); ①
```

```
    }

    private HorizontalLayout getContent() {
        HorizontalLayout content = new HorizontalLayout(grid, form);
        content.setFlexGrow(2, grid);
        content.setFlexGrow(1, form);
        content.addClassNames("content");
        content.setSizeFull();
        return content;
    }

    private void configureForm() {
        form = new ContactForm(service.findAllCompanies(), service.findAllStatuses(
));
        form.setWidth("25em");
    }

    private void configureGrid() {
        grid.addClassNames("contact-grid");
        grid.setSizeFull();
        grid.setColumns("firstName", "lastName", "email");
        grid.addColumn(contact -> contact.getStatus().getName()).setHeader("Status"
);
        grid.addColumn(contact -> contact.getCompany().getName()).setHeader("
Company");
        grid.getColumns().forEach(col -> col.setAutoWidth(true));

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

    private Component getToolbar() {
        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()); ③

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

    public void editContact(Contact contact) { ④
        if (contact == null) {
            closeEditor();
        } else {
            form.setContact(contact);
            form.setVisible(true);
            addClassName("editing");
        }
    }
```

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

    private void addContact() { ⑤
        grid.asSingleSelect().clear();
        editContact(new Contact());
    }


    private void updateList() {
        grid.setItems(service.findAllContacts(filterText.getValue()));
    }
}
```

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.

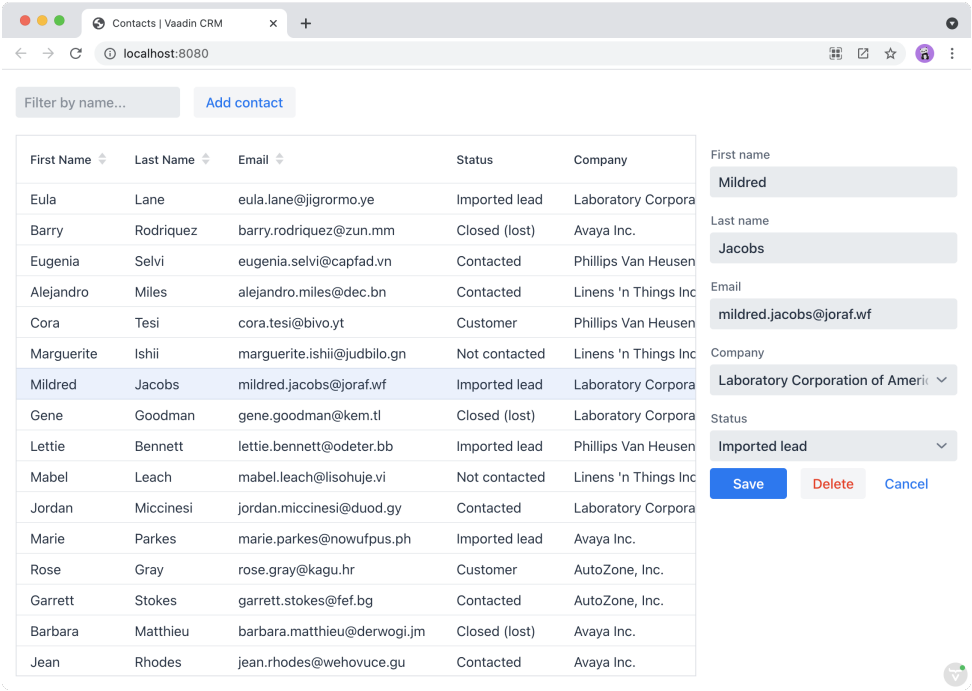
② `addValueChangeListener()` adds a listener to the grid. The `Grid` component supports multi- and single-selection modes. You only need to select a single `Contact`, so you can use the `asSingleSelect()` method. The `getValue()` method returns the `Contact` in the selected row, or null if there is no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

③ Call `addContact()` when the user clicks on the "Add contact" button.

④ `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.

⑤ `addContact()` clears the grid selection and creates a new `Contact`.

Next, you'll build the application. You should be able to select contacts in the grid and see them in the form. However, none of the buttons work yet.



Form Events

<!-- vale Vaadin.So = NO -->

The `ContactForm` API is designed to be reusable; it's configurable through properties and it fires the necessary events. So far, you've passed a list of companies, the status, and the contact to the form. However, you need the application to listen for the events to complete the integration.

<!-- vale Vaadin.So = YES -->

To handle event listeners, update `configureForm()` and add `saveContact()` and `deleteContact()` methods.

`ListView.java`

```
private void configureForm() {
    form = new ContactForm(service.findAllCompanies(), service.findAllStatuses());
    form.setWidth("25em");
    form.addSaveListener(this::saveContact); ①
    form.addDeleteListener(this::deleteContact); ②
    form.addCloseListener(e -> closeEditor()); ③
}

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

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

① The save event listener calls `saveContact()`. It does a few things:

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

b. Updates the list; and

c. Closes the editor.

② The delete event listener calls `deleteContact()`. In the process, it also does a few things:

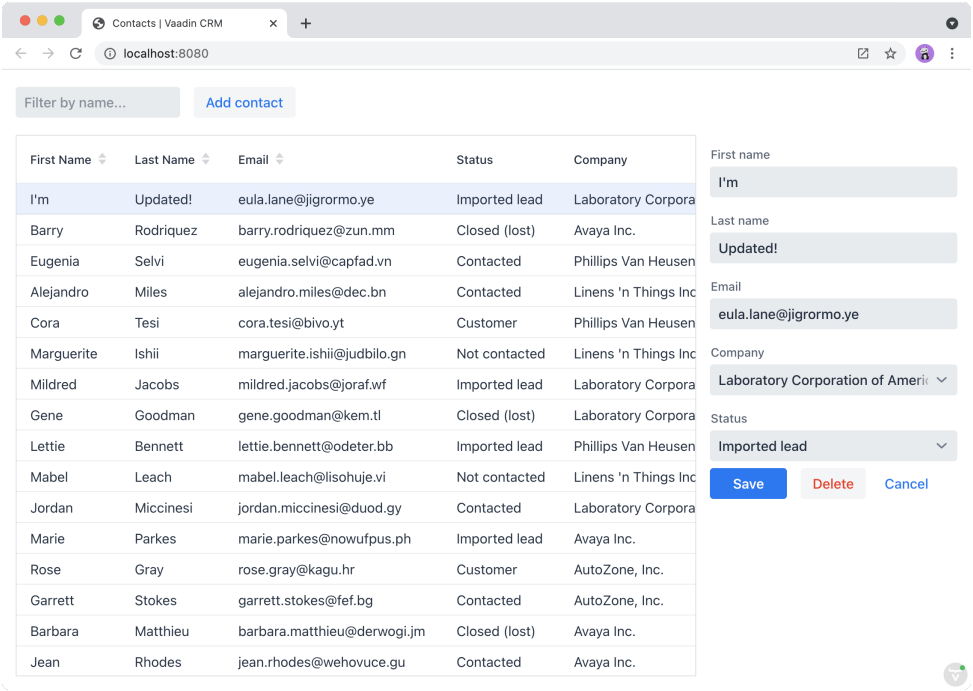
a. Uses `contactService` to delete the contact from the database;

b. Updates the list; and

c. Closes the editor.

③ The close event listener closes the editor.

Build the application now and verify that you're able to select, add, update, and delete contacts.



Making the Layout Responsive

Now if you try the UI with a mobile device or make your desktop browser really narrow, you see that the UI is not currently well optimized for small screens. It usually makes sense to hide certain UI elements for smaller screens. You can accomplish this using Java code or with CSS media queries.

This example uses CSS and utilize the class names previously assigned to the components. Add the following CSS to `frontend/themes/flowcrmtutorial/styles.css`:

`styles.css`

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

The CSS media query hides the grid and the toolbar when you are editing contacts on a narrow screen.

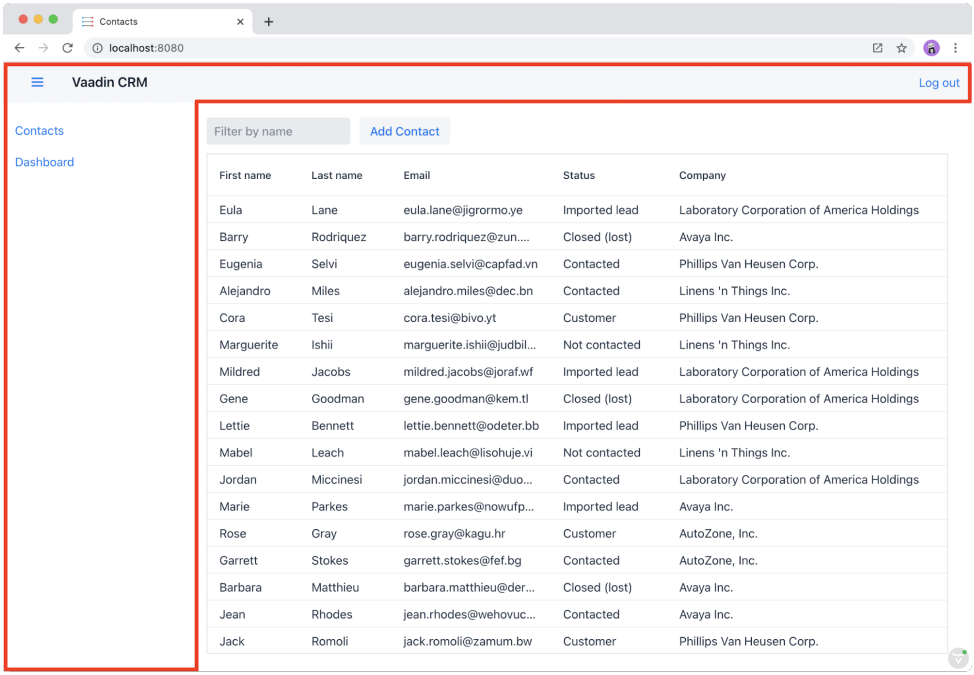
7ADFAE2F-44BD-4EE2-A8E1-E8B49581856B

Navigating among Views in Vaadin

<!-- vale Vaadin.So = NO -->

So far in this tutorial series, you've built a Customer Relationship Management (CRM) application for listing and editing contacts. Now, you'll add a dashboard view to the application. You'll also add a responsive application layout, with a header and a navigation sidebar that can be toggled on small screens (see the screenshot here).

<!-- vale Vaadin.So = YES -->



View Routes

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

Parent Layout

The application should have a shared parent layout with two child views. The first, `MainLayout`: App Layout should have a header and navigation: `ListView`: the default view, mapped to `""`; and `DashboardView`: mapped to `"dashboard"`. The second child should have a responsive application layout and navigation links.

Begin by creating a new Java class named `MainLayout` in the `views` package with the following content. This is the shared parent layout of both views in the application.

MainLayout.java

```
package com.example.application.views;

import com.example.application.views.list.ListView;
import com.vaadin.flow.component.applayout.AppLayout;
import com.vaadin.flow.component.applayout.DrawerToggle;
import com.vaadin.flow.component.html.H1;
import com.vaadin.flow.component.orderedlayout.FlexComponent;
import com.vaadin.flow.component.orderedlayout.HorizontalLayout;
import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.router.HighlightConditions;
import com.vaadin.flow.router.RouterLink;
import com.vaadin.flow.theme.lumo.LumoUtility;

public class MainLayout extends AppLayout { ①

    public MainLayout() {
        createHeader();
        createDrawer();
    }

    private void createHeader() {
        H1 logo = new H1("Vaadin CRM");
        logo.addClassNames(
            LumoUtility.FontSize.LARGE, ②
            LumoUtility.Margin.MEDIUM);

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

        header.setDefaultVerticalComponentAlignment(FlexComponent.Alignment.CENTER);
④
        header.setWidthFull();
        header.addClassNames(
            LumoUtility.Padding.Vertical.NONE,
            LumoUtility.Padding.Horizontal.MEDIUM);

        addToNavbar(header); ⑤

    }

    private void createDrawer() {
        addToDrawer(new VerticalLayout( ⑥
                new RouterLink("List", ListView.class)  ⑦
        ));
    }
}
```

① `AppLayout` is a Vaadin layout with a header and a responsive drawer.

② Instead of styling the text with raw CSS, use Lumo Utility Classes shipped with the default theme.

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 application layout's nav bar, the section at the top of the screen.

⑥ Wraps the router link in a `VerticalLayout` and adds it to the `AppLayout` drawer.

⑦ Creates a `RouterLink` with the text "List" and `ListView.class` as the destination view.

RouterLink automatically maintains the `highlight` attribute currently active in the element, but there is no default styling for it. Add the following CSS to `frontend/themes/flowcrmtutorial/styles.css` to highlight the selected link.

`styles.css`

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

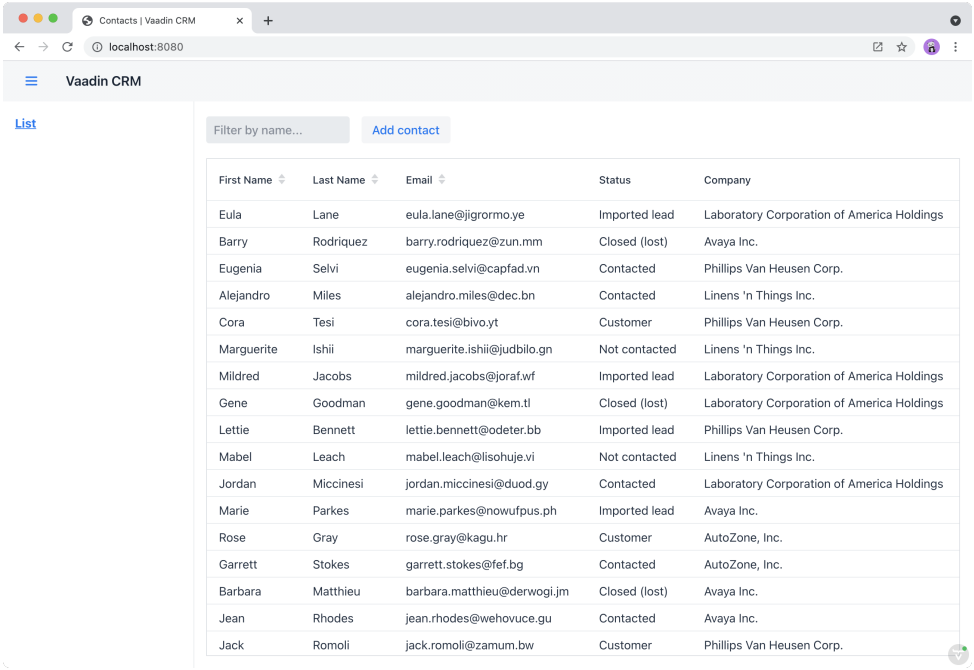
Lastly, in `ListView`, update the `@Route` mapping to use the new `MainLayout` like so:

`ListView.java`

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

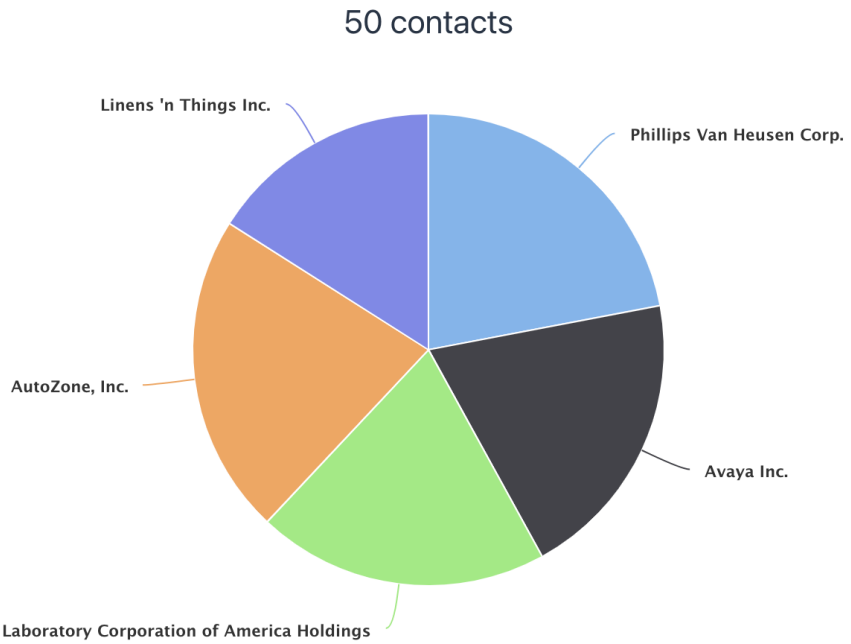
① `ListView` still matches the empty path, but now uses `MainLayout` as its parent.

Now you're ready to run the application. When you do, you should now see a header and a sidebar on the list view.



Dashboard View

Next, you'll create a new dashboard view. It'll show some basic statistics: the number of contacts in the system, and a pie chart of the number of contacts per company.



Now create a new Java class named `DashboardView` in the `views` package with the following content:

`DashboardView.java`

```
package com.example.application.views;

import com.example.application.data.service.CrmService;
import com.vaadin.flow.component.Component;
import com.vaadin.flow.component.charts.Chart;
import com.vaadin.flow.component.charts.model.ChartType;
import com.vaadin.flow.component.charts.model.DataSeries;
import com.vaadin.flow.component.charts.model.DataSeriesItem;
import com.vaadin.flow.component.html.Span;
import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.router.PageTitle;
import com.vaadin.flow.router.Route;
import com.vaadin.flow.theme.lumo.LumoUtility;

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

    public DashboardView(CrmService service) { ②
        this.service = service;
        addClassName("dashboard-view");
        setDefaultHorizontalComponentAlignment(Alignment.CENTER); ③
        add(getContactStats(), getCompaniesChart());
    }

    private Component getContactStats() {
        Span stats = new Span(service.countContacts() + " contacts"); ④
        stats.addClassNames(
            LumoUtility.FontSize.XLARGE,
            LumoUtility.Margin.Top.MEDIUM);
        return stats;
    }

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

        DataSeries dataSeries = new DataSeries();
        service.findAllCompanies().forEach(company ->
            dataSeries.add(new DataSeriesItem(company.getName(), company
.getEmployeeCount()))); ⑤
        chart.getConfiguration().setSeries(dataSeries);
        return chart;
    }
}
```

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

② Takes `CrmService` as a constructor parameter and saves it as a field.

③ Centers the contents of the layout.

④ Calls the service to get the number of contacts.

⑤ Calls the service to get all companies, then creates a `DataSeriesItem` for each, containing the company name and employee count. Don't worry about the compilation error, the missing method is added in the next step.

> **NOTE**
>
> *Vaadin Charts is a Commercial Component Set*
>
> Vaadin Charts is a collection of data visualization components that's part of the 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. For an open source alternative for Vaadin Charts, check out the wide selection of community extensions via Vaadin Directory.

Open `Company.java` and add the following field and getter to get the employee count without having to fetch all of the entities.

`Company.java`

```
@Formula("(select count(c.id) from Contact c where c.company_id = id)") ①
private int employeeCount;

public int getEmployeeCount(){
    return employeeCount;
}
```

① The Formula is a Hibernate feature that allows you to specify SQL snippets to fetch special fields. The query gets the count of employees without needing to fetch all of the employees. Note that in a larger application you'll probably want to do this in some alternative way since all `Company` entity loads now triggers an additional SQL query, even though the `employeeCount` field is only needed in this `DashboardView` class.

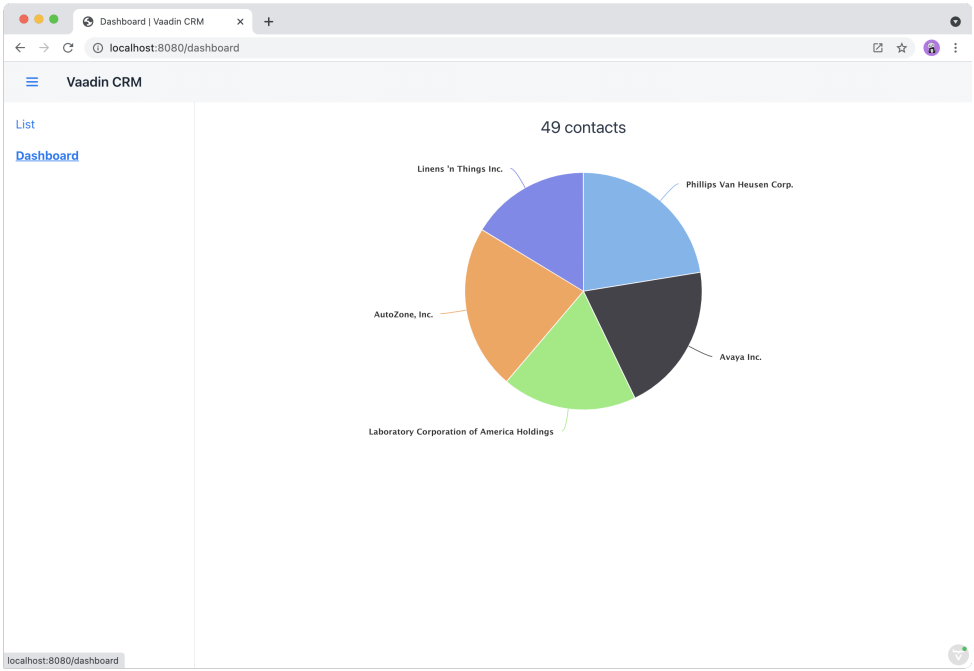
Dashboard View in Main Layout Sidebar

To include a dashboard view in the main layout side base, add a navigation link to `DashboardView` in the `MainLayout` drawer:

`MainLayout.java`

```
private void createDrawer() {
    private void createDrawer() {
        addToDrawer(new VerticalLayout(
                new RouterLink("List", ListView.class),
                new RouterLink("Dashboard", DashboardView.class)
        ));
    }
}
```

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



On the next part of this tutorial, you'll secure the application by adding a log-in screen.

52AFFD31-EA40-4AEF-B60F-E3BB6E5A8379

Add a Login Screen to an Application

On this part of this tutorial, you'll secure the Customer Relationship Management (CRM) application by setting up Spring Security and adding a login screen to limit access to logged-in users.

Login View

Start by creating a new view, `LoginView`, in the `views` package. You would do that like so:

LoginView.java

```
package com.example.application.views;

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 com.vaadin.flow.server.auth.AnonymousAllowed;

@Route("login") ①
@PageTitle("Login | Vaadin CRM")
@AnonymousAllowed
public class LoginView extends VerticalLayout implements BeforeEnterObserver {

    private final LoginForm login = new LoginForm(); ②

    public LoginView(){
        addClassName("login-view");
        setSizeFull(); ③
        setAlignItems(Alignment.CENTER);
        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 Map the view to the `"login"` path. `LoginView` should encompass the entire browser window, so don't use `MainLayout` as the parent.

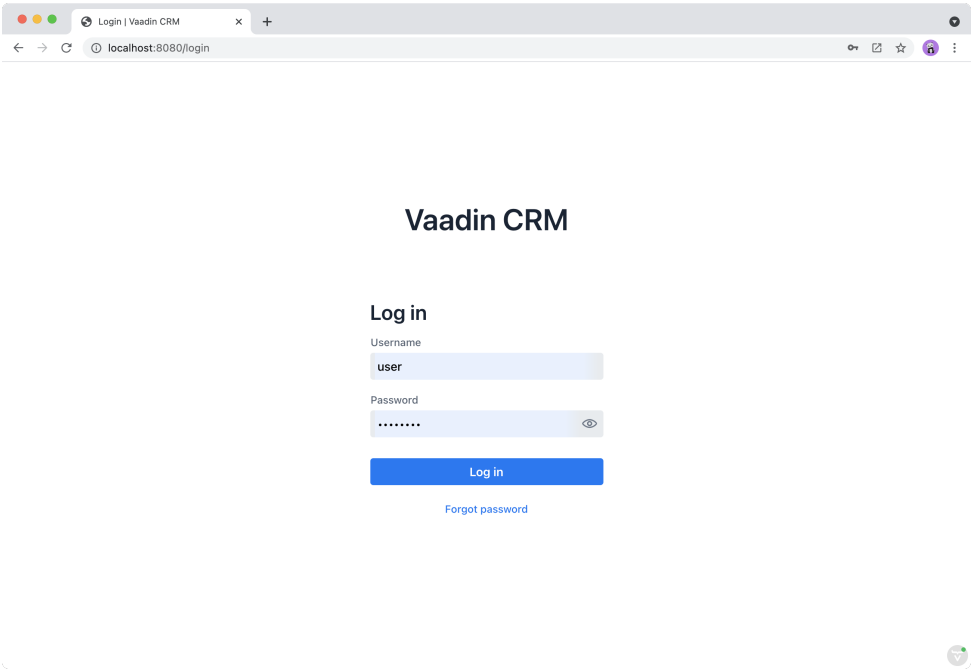
② Instantiate a `LoginForm` component to capture username and password.

③ Make `LoginView` full size and center its content — both horizontally and vertically — by calling `setAlignItems(`Alignment.CENTER)` and `setJustifyContentMode(`JustifyContentMode.CENTER)`.

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

⑤ Read query parameters and show an error if a login attempt fails.

Build the application and navigate to http://localhost:8080/login. You should see a centered login form like the one in the screenshot here:



Set Spring Security to Handle Logins

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

Installing Spring Security Dependencies

Add the Spring Security dependency in `pom.xml` like so:

`pom.xml`

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Confirm that the dependency is downloaded. If you're unsure, run `./mvnw install` from the command line to download the dependency.

Configure Spring Security

Create a new package, `com.example.application.security` for classes related to security.

> **TIP**
>
> *Create Classes Automatically*
>
> Paste the class code into the `security` package to have IntelliJ automatically create the class for you.

Enable and configure Spring Security with a new class, `SecurityConfig.java` like this:

`SecurityConfig.java`

```
package com.example.application.security;

import com.example.application.views.LoginView;
import com.vaadin.flow.spring.security.VaadinWebSecurity;
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.configuration.EnableWebSecurity;
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
public class SecurityConfig extends VaadinWebSecurity { ②

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
                .requestMatchers("/images/*.png").permitAll();  ③
        super.configure(http);
        setLoginView(http, LoginView.class); ④
    }

    @Bean
    public UserDetailsService users() {
        UserDetails user = User.builder()
                .username("user")
                // password = password with this hash, don't tell anybody :-)
                .password(
"{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
                .roles("USER")
                .build();
        UserDetails admin = User.builder()
                .username("admin")
                .password(
"{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
                .roles("USER", "ADMIN")
                .build();
        return new InMemoryUserDetailsManager(user, admin); ⑤
    }
}
```

① Enable Spring Security.

② Extend the `VaadinWebSecurity` class to configure Spring Security for Vaadin.

③ Allow public access to the image directory.

④ Allow access to `LoginView`.

⑤ Configure an in-memory users for testing (see note below).

WARNING	*Never use hard-coded credentials in production.* Don't use hard-coded credentials in real applications. You can change the Spring Security configuration to use an authentication provider for Lightweight Directory Access Protocol (LDAP), Java Authentication and Authorization Service (JAAS), and other real-world sources. Read more about Spring Security authentication providers.

Next, in the same package, create a service for accessing information on the logged-in user and for logging out the user.

`SecurityService.java`

```
package com.example.application.security;

import com.vaadin.flow.spring.security.AuthenticationContext;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;

@Component
public class SecurityService {

    private final AuthenticationContext authenticationContext;

    public SecurityService(AuthenticationContext authenticationContext) {
        this.authenticationContext = authenticationContext;
    }

    public UserDetails getAuthenticatedUser() {
        return authenticationContext.getAuthenticatedUser(UserDetails.class).get();
    }

    public void logout() {
        authenticationContext.logout();
    }
}
```

Finally, add `@PermitAll` annotations to both views to allow all logged-in users to access them.

`ListView.java`

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

DashboardView.java

```
@PermitAll
@Route(value = "dashboard", layout = MainLayout.class)
@PageTitle("Dashboard | Vaadin CRM")
public class DashboardView extends VerticalLayout {
    // omitted
}
```

Add a Logout Button

You can now log in to the application. The last item to add is a logout button in the application header.

In `MainLayout`, add a link to the header like this:

MainLayout.java

```
package com.example.application.views;

import com.example.application.security.SecurityService;
import com.example.application.views.list.ListView;
import com.vaadin.flow.component.applayout.AppLayout;
import com.vaadin.flow.component.applayout.DrawerToggle;
import com.vaadin.flow.component.button.Button;
import com.vaadin.flow.component.html.H1;
import com.vaadin.flow.component.orderedlayout.FlexComponent;
import com.vaadin.flow.component.orderedlayout.HorizontalLayout;
import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.router.RouterLink;
import com.vaadin.flow.theme.lumo.LumoUtility;

public class MainLayout extends AppLayout {
    private final SecurityService securityService;

    public MainLayout(SecurityService securityService) {  ①
        this.securityService = securityService;
        createHeader();
        createDrawer();
    }

    private void createHeader() {
        H1 logo = new H1("Vaadin CRM");
        logo.addClassNames(
            LumoUtility.FontSize.LARGE,
            LumoUtility.Margin.MEDIUM);

        String u = securityService.getAuthenticatedUser().getUsername();
        Button logout = new Button("Log out " + u, e -> securityService.logout());
②

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

        header.setDefaultVerticalComponentAlignment(FlexComponent.Alignment.CENTER);
        header.expand(logo); ④
        header.setWidthFull();
        header.addClassNames(
            LumoUtility.Padding.Vertical.NONE,
            LumoUtility.Padding.Horizontal.MEDIUM);

        addToNavbar(header);

    }

    private void createDrawer() {
        addToDrawer(new VerticalLayout(
                new RouterLink("List", ListView.class),
                new RouterLink("Dashboard", DashboardView.class)
        ));
    }
}
```

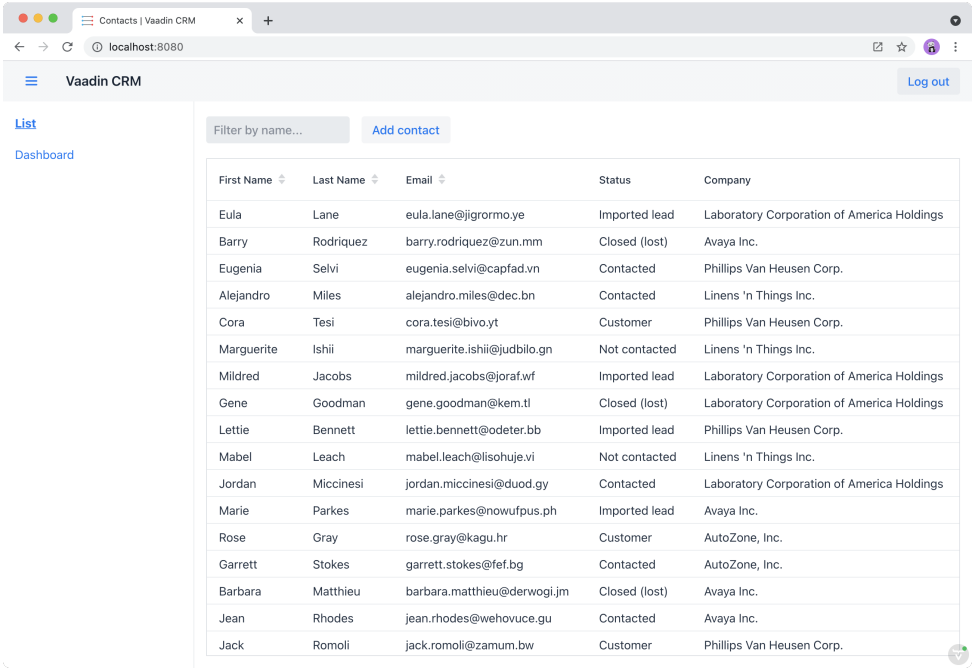
① Autowire the `SecurityService` and save it in a field.

② Create a logout button that calls the `logout()` method in the service.

③ Add the button to the header layout.

④ Call `header.expand(logo)` to make the logo take up all of the extra space in the layout. This can push the logout button to the far right.

Stop and restart the server to get the new Maven dependencies. You should now be able to log in and out of the application. Verify that you can't access http://localhost:8080/dashboard without being logged in. You can log in with the username, `user`, and the password, `password`.



You have now built a full-stack CRM application with navigation and authentication. On the next part of this tutorial, you'll learn how to turn it into a PWA to make it installable on mobile and desktop platforms.

234932EC-C4B0-4FA5-A22E-DE6E5A070007

Make a Vaadin Flow Application an Installable PWA

On this part of this tutorial, you'll turn the completed Customer Relationship Management (CRM) application into a Progressive Web Application (PWA), so that users can install it.

Understanding PWAs

The term PWA is used to describe modern web applications that offer a user experience similar to a native application. PWA technologies make applications faster, more reliable, and more engaging.

PWAs can be installed on most mobile devices and on desktops 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 PWA configuration documentation.

Two main components enable PWA technologies:

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

PWA Resources

Vaadin provides the `@PWA` annotation, which automatically generates the required PWA resources. Add the `@PWA` annotation on `Application.java` as follows:

Application.java

```
@SpringBootApplication
@Theme(value = "flowcrmtutorial")
@PWA( ①
    name = "Vaadin CRM", ②
    shortName = "CRM" ③
)
public class Application extends SpringBootServletInitializer implements
AppShellConfigurator {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

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 shouldn't exceed 12 characters.

Application Icon

You can override the default icon by replacing <code>src/main/resources/META-INF/resources/icons/icon.png</code> with another 512px × 512px PNG icon.

You can use your own icon, or save a sample image, by right-clicking the link and selecting **Save Link As**. Be sure to have the file in PNG format.

Customize Offline Page

Vaadin creates a generic offline fallback page that displays when the application is launched offline. You can make your application appear more polished by replacing this default page with a custom page that follows your own design guidelines.

Use the code below to create `offline.html` in the `src/main/resources/META-INF/resources` 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>
    <style>
        body {
            display: flex;
            flex-direction: column;
            align-items: center;
            font-family: sans-serif;
            color: #555;
        }

        .content {
            width: 80%;
        }

        .offline-image {
            width: 100%;
            margin: 4em 0px;
        }
    </style>
</head>
<body>

<div class="content">
    <img src="./images/offline.png" alt="VaadinCRM is offline" class="offline-image
"/>
    <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', () => window.location.reload()); ①
</script>
</body>
</html>
```

① The JavaScript snippet reloads the page if the browser detects that it's back online.

Add the following image — or use one of your own — to the `META-INF/resources/images` folder and name it `offline.png`.

You can make the files available offline by adding them to the `@PWA` annotation in `Application` as follows:

`Application.java`

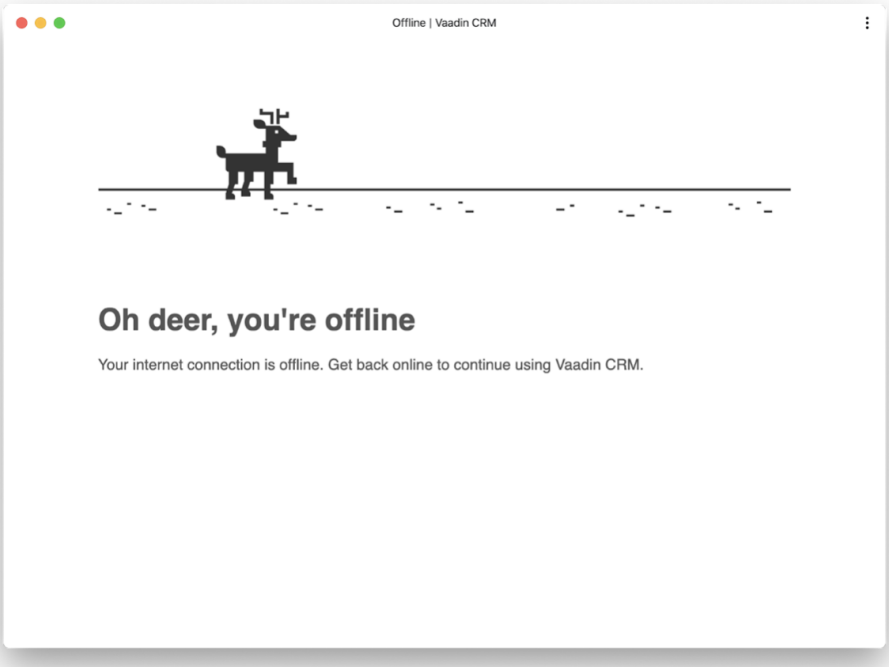
```
@PWA(
    name = "VaadinCRM",
    shortName = "CRM",
    offlinePath="offline.html",
    offlineResources = { "./images/offline.png"} ①
)
```

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

Now, restart the application. You can install it on supported browsers.

Test Offline Page

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



On the next part of this tutorial, you'll add both unit tests and in-browser tests to the application.

2861D7D6-5025-4A8B-A866-38C01AF5FF91

Unit & Integration Tests

It's a common and best practice to test as little code as possible in a single test. This way, when things go wrong, only relevant tests fail. It makes it easier to troubleshoot.

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 check what the user sees.

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

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

On this part of this tutorial, you'll write and run unit and integration tests. End-to-end tests are covered on the next part.

Unit Tests for Simple UI Logic

The most minimal 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.

NOTE

Put tests in the correct folder

All test classes should go in the test folder, `src/test/java`. Pay special attention to the package names. Use package access for class fields. If the test isn't in the same package as the class you're testing, you'll get errors.

Now, create a new folder, <code>src/test/java</code>. In IntelliJ, right-click on the folder and select Mark Directory as → Test Sources Root. In the new folder, create a new package, <code>com.example.application.views.list</code>, and add a new <code class="filename">ContactFormTest.java</code> file with the following code:

ContactFormTest.java

```
package com.example.application.views.list;

import com.example.application.data.entity.Company;
import com.example.application.data.entity.Contact;
import com.example.application.data.entity.Status;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicReference;
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class ContactFormTest {
    private List<Company> companies;
    private List<Status> statuses;
    private Contact marcUsher;
    private Company company1;
    private Company company2;
    private Status status1;
    private Status status2;

    @BeforeEach  ①
    public void setupData() {
        companies = new ArrayList<>();
        company1 = new Company();
        company1.setName("Vaadin Ltd");
        company2 = new Company();
        company2.setName("IT Mill");
        companies.add(company1);
        companies.add(company2);

        statuses = new ArrayList<>();
        status1 = new Status();
        status1.setName("Status 1");
        status2 = new Status();
        status2.setName("Status 2");
        statuses.add(status1);
        statuses.add(status2);

        marcUsher = new Contact();
        marcUsher.setFirstName("Marc");
        marcUsher.setLastName("Usher");
        marcUsher.setEmail("marc@usher.com");
        marcUsher.setStatus(status1);
        marcUsher.setCompany(company2);
    }
}
```

① The `@BeforeEach` annotation adds dummy data that's used for testing. This method is executed before each `@Test` method.

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

`ContactFormTest.java`

```
@Test
public void formFieldsPopulated() {
    ContactForm form = new ContactForm(companies, statuses);
    form.setContact(marcUsher); ①
    assertEquals("Marc", form.firstName.getValue());
    assertEquals("Usher", form.lastName.getValue());
    assertEquals("marc@usher.com", form.email.getValue());
    assertEquals(company2, form.company.getValue());
    assertEquals(status1, 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:

Similarly, you can test the "save" functionality of `ContactForm` like so:

`ContactFormTest.java`

```
@Test
public void saveEventHasCorrectValues() {
    ContactForm form = new ContactForm(companies, statuses);
    Contact contact = new Contact();
    form.setContact(contact); ①
    form.firstName.setValue("John"); ②
    form.lastName.setValue("Doe");
    form.company.setValue(company1);
    form.email.setValue("john@doe.com");
    form.status.setValue(status2);

    AtomicReference<Contact> savedContactRef = new AtomicReference<>(null); ③
    form.addSaveListener(e -> {
        savedContactRef.set(e.getContact());
    });
    form.save.click(); ④
    Contact savedContact = savedContactRef.get();

    assertEquals("John", savedContact.getFirstName()); ⑤
    assertEquals("Doe", savedContact.getLastName());
    assertEquals("john@doe.com", savedContact.getEmail());
    assertEquals(company1, savedContact.getCompany());
    assertEquals(status2, savedContact.getStatus());
}
```

① Initialize the form with an empty `Contact`.

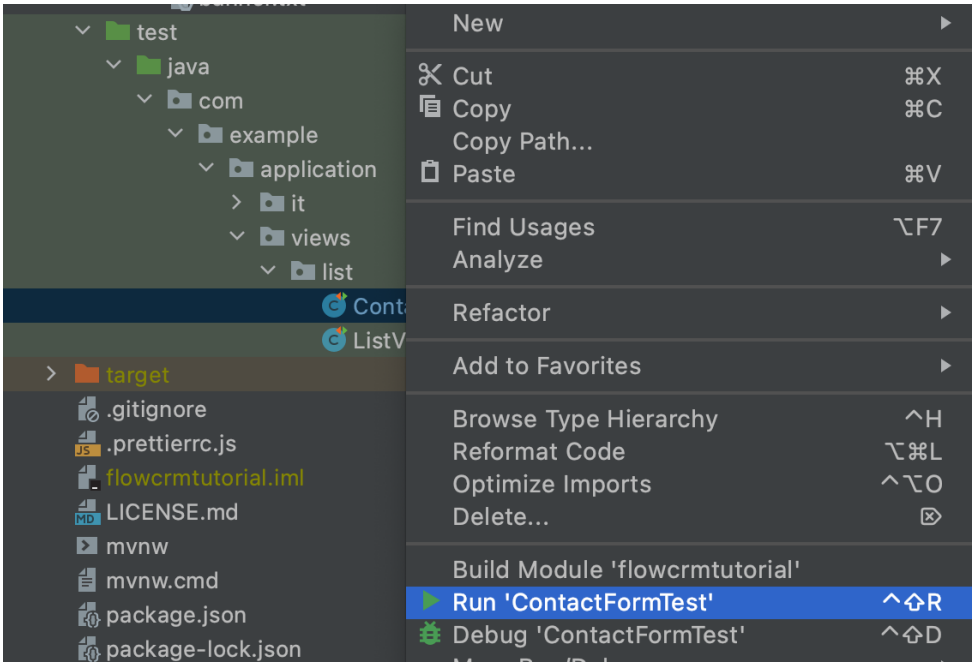
② Populate values into the form.

③ Capture the saved contact into an `AtomicReference`.

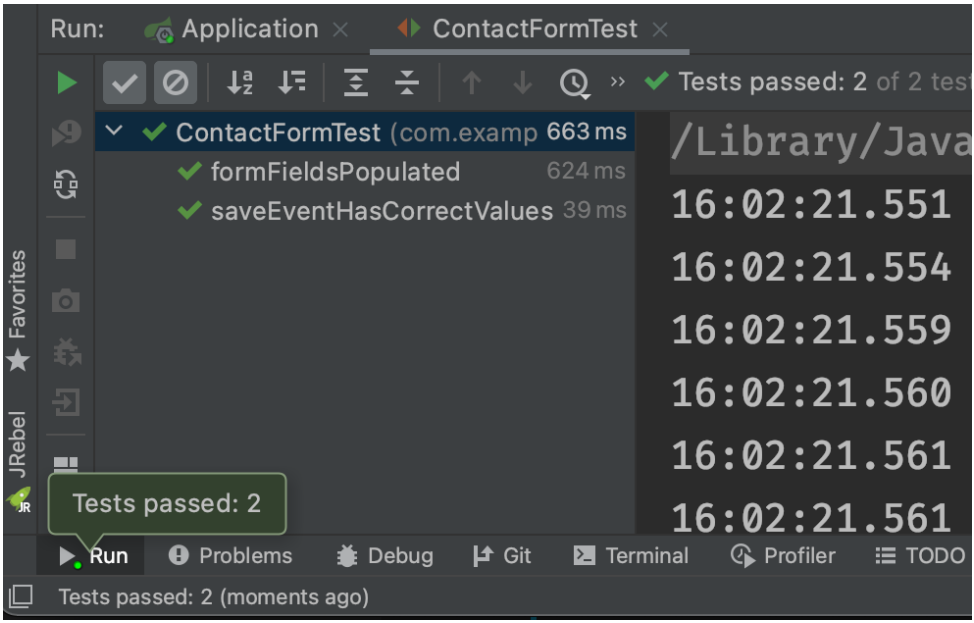
④ Click the **save** button and read the saved contact.

⑤ Once the event data is available, verify that the bean contains the expected values.

To run the unit test, right-click `ContactFormTest` and select **Run 'ContactFormTest'**, as shown in the screenshot here:



When the test finishes, you should see the results at the bottom of the IDE window in the test-runner panel. As shown here, both tests passed.



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, use the Spring Boot test runner. This adds a little overhead, but it makes more features available to your test.

To set up a unit test for `ListView`, create a new file, `ListViewTest`, in the `com.example.application.views.list` package like so:

ListViewTest.java

```
package com.example.application.views.list;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;


@SpringBootTest ①
public class ListViewTest {

    @Autowired
    private ListView listView;

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

① The `@SpringBootTest` annotation makes sure that the Spring Boot application is initialized before the tests are run and allows you to use the `@Autowired` annotation in the test.

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

> **NOTE**
>
> *Annotation isn't needed for normal application runs.*
>
> You don't need to add the annotation for normal application usage since all `@Route` classes are automatically instantiated by Vaadin in a Spring-compatible way.

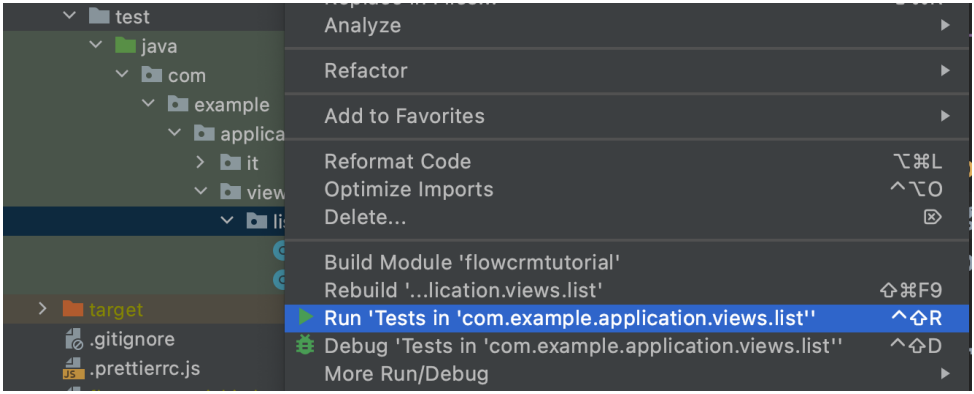
ListView.java

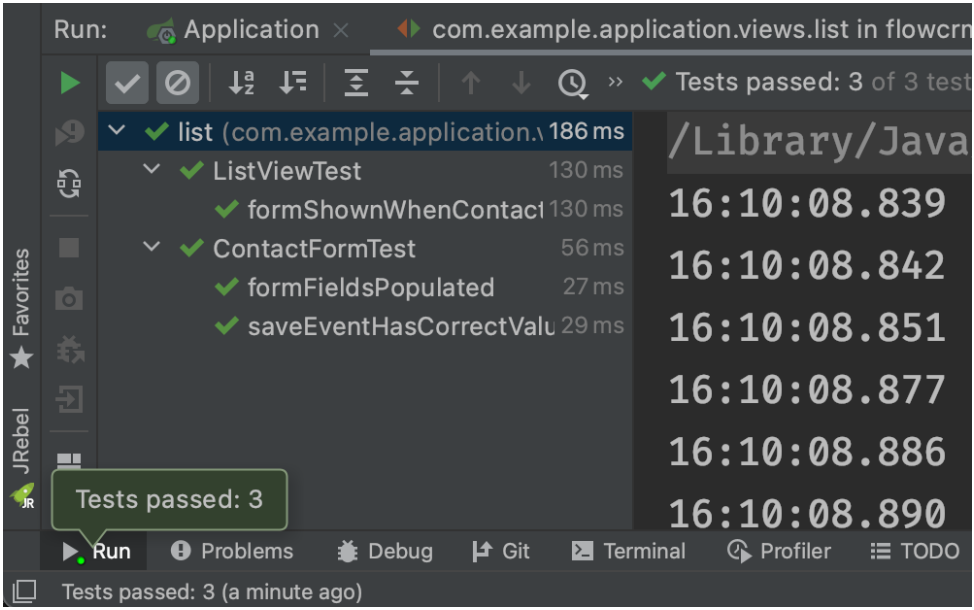
```
@Component
@Scope("prototype")
@Route(value = "", layout = MainLayout.class)
@PageTitle("Contacts | Vaadin CRM")
@PermitAll
public class ListView extends VerticalLayout {
    Grid<Contact> grid = new Grid<>(Contact.class);
    TextField filterText = new TextField();
    ContactForm form;
    CrmService service;

    // rest omitted
}
```

Right-click the package that contains both tests, and select Run tests in

'com.example.application.views.list'. You should see that both test classes run and result in three successful tests.





NOTE

Integration tests take longer.

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. They can take many seconds to start.

To improve startup time, you can explicitly list the needed dependencies in the `@SpringBootTest` annotation using `classes={…}`. You can also mock up parts of the application. And you can use other advanced techniques, but they're beyond the scope of this tutorial.

See Pivotal's Spring Boot Testing Best Practices for tips on speeding up your tests.

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;

    assertFalse(form.isVisible());
    grid.asSingleSelect().setValue(firstContact);
    assertTrue(form.isVisible());
    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. It also does so by selecting the first item in the grid and verifying that the form is visible and the form is bound to the correct `Contact` by ensuring that the right name is visible in the field.

Now, rerun the tests. They should all pass.

At this point, you should know how to test the application logic both in isolation with unit tests and by injecting dependencies to test the integration between several components. The next part of this tutorial covers how to test the entire application in the browser.

If your components depend on `UI.getCurrent()`, `UI.navigate()`, and similar, you may need to fake or mock the Vaadin environment for those tests to pass. For further information on how to achieve that, look at UI Unit Testing in Vaadin TestBench or the open source Karibu-Testing

project.

76D89DA1-B104-4745-8D51-9589846051C8

Test Vaadin Applications in Browser with End-To-End Tests

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

End-to-end tests are executed in a browser window. Vaadin TestBench controls the browser window using Selenium WebDriver.

NOTE	*Vaadin TestBench is a commercial product.* The end-to-end tests use Vaadin TestBench, which is a commercial tool that's part of the Vaadin Pro Subscription. You can get a free trial at https://vaadin.com/trial. All Vaadin Pro tools and components are free for students through the GitHub Student Developer Pack. For an open source alternative for TestBench, you can get similar results with plain Selenium WebDriver or Playwright.

The Base Test Class

Vaadin TestBench contains handy base classes that you can use as a basis for your e2e tests. The JUnit5 version is called `BrowserTestBase`. It can be used alone, if you orchestrate starting and stopping your server. For example, it can be used with Maven executions in `pre-integration-test` and `post-integration-test`, and to execute the actual test in integration test phase.

In Spring Boot applications, it's easier to use the same @SpringBootTest annotation that you already used in the previous phase to ensure a running server during the browser test execution.

First, create a new class, `LoginE2ETest` in the `com.example.application.it` package. Be sure to place it in `src/test/java` and not `src/main/java`.

LoginE2ETest.java

```
package com.example.application.it;

import ch.qos.logback.classic.Level;
import ch.qos.logback.classic.Logger;
import com.vaadin.testbench.IPAddress;
import com.vaadin.testbench.ScreenshotOnFailureRule;
import com.vaadin.testbench.parallel.ParallelTest;
import io.github.bonigarcia.wdm.WebDriverManager;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Rule;
import org.slf4j.LoggerFactory;

public abstract class LoginE2ETest extends ParallelTest {
    private static final String SERVER_HOST = IPAddress.findSiteLocalAddress();
    private static final int SERVER_PORT = 8080;
    private final String route;

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

    @BeforeClass
    public static void setupClass() {
        WebDriverManager.chromedriver().setup(); ①
    }

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

    @Before
    public void setup() throws Exception {
        super.setup();
        getDriver().get(getURL(route)); ③
    }

    protected LoginE2ETest(String route) {
        this.route = route;
    }

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

① Start by invoking the **Chrome** `WebDriverManager` before any test method is invoked. TestBench doesn't invoke the WebDriver manager.

② `ScreenshotOnFailureRule` tells TestBench to grab a screenshot before exiting, if a test fails. This can help you understand what went wrong when tests don't pass.

③ Open the browser to the correct URL before each test. For this, you need the host name where the application runs (i.e., "localhost" in development), the port the server uses, which is set to 8080 in application.properties, and information about the route from which to start.

Test 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.

Create a new class, `LoginIT`, in the same package as `LoginE2ETest`. The test validates that logging in with the correct user and password succeeds.

`LoginE2ETest.java`

```
package com.example.application.it;

import com.vaadin.flow.component.login.testbench.LoginFormElement;
import com.vaadin.testbench.BrowserTest;
import com.vaadin.testbench.BrowserTestBase;
import com.vaadin.testbench.annotations.RunLocally;
import com.vaadin.testbench.parallel.Browser;
import org.junit.jupiter.api.BeforeEach;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.core.env.Environment;

import static org.junit.jupiter.api.Assertions.assertFalse;

//@RunLocally(Browser.FIREFOX) ①
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class LoginE2ETest extends BrowserTestBase { ②

    @Autowired
    Environment env;

    static {
        // Prevent Vaadin Development mode to launch browser window
        System.setProperty("vaadin.launch-browser", "false");
    }

    @BeforeEach
    void openBrowser() {
        getDriver().get("http://localhost:" +
            env.getProperty("local.server.port") + "/"); ③
    }

    @BrowserTest ④
    public void loginAsValidUserSucceeds() {
        // Find the LoginForm used on the page, using a
        // typed selector API provided by TestBench
```

```
        LoginFormElement form = $(LoginFormElement.class).first();
        // Enter the credentials and log in
        form.getUsernameField().setValue("user");
        form.getPasswordField().setValue("password");
        form.getSubmitButton().click();

        // Behind the scenes TestBench uses lower level WebDriver API
        // Here we can configure it on the fly
        getDriver().manage().timeouts().implicitlyWait(Duration.of(1, ChronoUnit
.SECONDS));
        // Here finding an element on the actual main layout (after login),
        // using pure WebDriver API, BTW. There is also AppLayoutElement for TB
        getDriver().findElement(By.tagName("vaadin-app-layout"));

        // Ensure the login form is no longer visible
        assertFalse($(LoginFormElement.class).exists());
    }

}
```

① This optional annotation specifies the test to be run on the local machine and using Firefox. The default is Chrome.

② The super class `BrowserTestBase` provides handy helper methods and configures TestBench.

③ The `openBrowser` method is annotated to be executed before each actual tests. The URL points to local test server with the random port SpringBootTest has selected. The browser should be automatically redirected to the login screen.

④ BrowserTest annotation is a TestBench extension of the better known `Test` annotation, that is handy if you decide to extend your end-to-end tests to cover multiple browsers at some point.

Right-click `LoginE2ETest.java` and select **Run 'LoginE2ETest'**.

Create a View Object

You can now add a second test, one to validate that you can't log in with an invalid password.

For this test, 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 for each view a view object — otherwise known as a call page object or element class. A view object provides a high-level API to interact with the view and hides the implementation details.

For the login view, create the `LoginViewElement` class in a new package, `com.example.application.it.elements`:

`LoginViewElement.java`

```
package com.example.application.it.elements;

import com.vaadin.flow.component.login.testbench.LoginFormElement;
import com.vaadin.flow.component.orderedlayout.testbench.VerticalLayoutElement;
import com.vaadin.testbench.annotations.Attribute;
import org.openqa.selenium.By;

import java.time.Duration;
import java.time.temporal.ChronoUnit;
import java.util.concurrent.TimeUnit;

@Attribute(name = "class", contains = "login-view")
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();

        try {
            getDriver().manage().timeouts().implicitlyWait(Duration.of(1,
ChronoUnit.SECONDS));
            getDriver().findElement(By.tagName("vaadin-app-layout"));
            return true;
        } catch (Exception e) {
            return false;
        }
    }

}
```

CAUTION

Class hierarchies must match.

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

Adding the `@Attribute(name = "class", contains = "login-view")` annotation allows you to find the `LoginViewElement` using the TestBench query API. The following is an example of this:

Finding a LoginViewElement using the TestBench query API

```
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.

Now that you have the `LoginViewElement` class, you can simplify your `loginAsValidUserSucceeds()` test to be this:

`LoginE2ETest.java`

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

Add a test to use an invalid password as follows:

`LoginE2ETest.java`

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

You can continue testing the other views by creating similar view objects and IT classes.

If you're building a large application, it's probably better to make slower end-to-end tests executed only when requested separately. You can do this by using Maven Failsafe plugin or using the tagging feature in JUnit 5.

The next part covers how to make a production build of the application and deploy it to a cloud platform.

`0DDF0F9E-DCF0-4AEC-9DD4-C241699CC7F7`

Deploy a Vaadin Flow Application on Azure

<!-- vale Vaadin.Terms = NO -->

In this final part of this tutorial, you'll learn how to deploy a Spring Boot application on Azure. You'll use Azure Container Apps, which is a simple way to deploy applications on Azure. For a larger scale deployment and the best possible end-user experience, consider using Azure Kubernetes Service together with Vaadin Kubernetes Kit.

This part covers:

- Vaadin production builds;
- Packaging Vaadin applications as a Docker image;
- Deploying a Docker packaged web application using Azure Container Apps; and
- Tips for production deployment.

> **TIP**
>
> *Vaadin can be deployed on any cloud provider.*
>
> From a cloud provider's point of view, a Vaadin application is a standard Java web application. You can deploy your application onto almost any cloud platform, in many different ways. Read the Cloud Deployment tutorials for more options.

Prepare for Production

It's important to build a separate production-optimized version of the application before deploying it. In development mode, Vaadin has a live-reload widget, debug logging, and uses a quick but unoptimized front-end build that includes source maps for easy debugging that's maintained using npm and Vite. Unoptimized front-end bundles can contain several megabytes of JavaScript and dependencies that aren't needed during production deployment.

The `pom.xml` file includes a `production` profile configuration that prepares an optimized build which is ready for production. Enter the following at the command-line in the project directory to build a production-ready JAR file:

```
mvn install -Pproduction
```

Deployment Using Azure Container Apps

To use Azure Container Apps, you'll need to do the following:

1. Install and run Docker (e.g., using Docker Desktop).
2. Install Azure CLI or make sure you have it up-to-date with `az upgrade`.
3. Log into Azure using a browser and make sure you have an active Azure subscription. Use Start Free Trial, if you don't have an existing one.

From the commmand-line, login using Azure CLI like so:

```
az login
```

TIP	If you're not located in North America, you'll have a better experience with Azure by choosing a nearby region. For example, a Europe based developer can do this with `az config set defaults.location=westeurope` or pick the location per application. Pick a location that already supports Container Apps.

Docker image is a basic building block used by many modern hosting solutions to run applications. The example project contains a simple ready-made `Dockerfile` that essentially describes how the JAR file built in the previous step should be run.

Azure Container Apps contains a handy `up` command, that does much of convention-based setups for Docker-based deployments. With a single command, Azure tooling builds a Docker image, pushes it to a custom project specific Docker registry and creates a single node deployment based on it.

```
az containerapp up -n my-crm-app --source .
```

The `my-crm-app` should be changed to the name for your application in Azure. The last `.` is relevant in the command as it asks Azure to pick the sources from the current directory. The first time you use Azure Container Apps, the CLI asks you to install some new components.

The first deployment can take several minutes, depending on the speed of your computer and network. Once the deployment is finished, you should see the URL to your newly deployed Vaadin application at the end of the command output.

WARNING	*Avoid data loss* This application uses in-memory H2 database by default, which is useful for development. The database is re-created on each deployment and is embedded for each node. For actual usage, you should switch to your preferred database — at least in the production profile — to use `ddl-auto=none` and start to use a database migration tool like Liquibase or Flyway, so you can evolve the database schema without losing data. Check Spring and Azure documentation for more details.

Refer to the Azure Container Apps documentation for more details how to configure your

deployment. The horizontal scaling doesn't work yet with Vaadin applications — session affinity for Container Apps ingress is currently in development. However, you can configure the node to be a larger one to scale up your application. For large scale deployments, you should see Kubernetes based clustering solutions.

Tutorial Conclusion & Next Steps

If you had any problems or were confused by any part of this tutorial, you can contact @vaadin on Twitter or join Vaadin's Discord chat.

If all went correctly, though, you built a full-stack web application in pure Java and deployed it to Azure. You can use it to experiment further or as a foundation for your next project.

Helpful Resources

- Source code GitHub repository
- Vaadin Discord chat
- Stack Overflow
- Flow documentation
- Vaadin components
- Compare Vaadin with React, Angular, and Vue

`01A3D231-9D1B-4D4D-A6BB-CB4D37E01CBE`