Creating CRUD UI with Vaadin

This guide walks you through the process of building an application that uses a Vaadin-based UI on a Spring Data JPA based backend.

What You Will build

You will build a Vaadin UI for a simple JPA repository. What you will get is an application with full CRUD (Create, Read, Update, and Delete) functionality and a filtering example that uses a custom repository method.

You can follow either of two different paths:

- Starting from the `initial` project that is already in the project.
- Making a fresh start.

The differences are discussed later in this document.

What You Need

- About 15 minutes
- A favorite text editor or IDE
- Java 17 or later

- Gradle 7.5+ or Maven 3.5+
- You can also import the code straight into your IDE:
    - Spring Tool Suite (STS)
    - IntelliJ IDEA
    - VSCode

How to complete this guide

Like most Spring Getting Started guides, you can start from scratch and complete each step or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to Starting with Spring Initializr.

To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git: `git clone https://github.com/spring-guides/gs-crud-with-vaadin.git`
- cd into `gs-crud-with-vaadin/initial`
- Jump ahead to Create the Backend Services.

When you finish, you can check your results against the code in `gs-crud-with-vaadin/complete`.

Starting with Spring Initializr

You can use this pre-initialized project and click Generate to download a ZIP file. This project is configured to fit the examples in this tutorial.

Manual Initialization (optional)

You can also fork the project from Github and open it in your IDE or other editor.

If you want to initialize the project manually rather than use the links shown earlier, follow the steps given below:

1. Navigate to https://start.spring.io. This service pulls in all the dependencies you need for an application and does most of the setup for you.
2. Choose either Gradle or Maven and the language you want to use. This guide assumes that you chose Java.
3. Click **Dependencies** and select **Vaadin**, **Spring Data JPA** and **H2 Database**.
4. Click **Generate**.

5. Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.

Create the Backend Services

If your IDE has the Spring Initializr integration, you can complete this process from your IDE.

This guide is a continuation from Accessing Data with JPA. The only differences are that the entity class has getters and setters and the custom search method in the repository is a bit more graceful for end users. You need not read that guide to walk through this one, but you can if you wish.

If you started with a fresh project, you need to add entity and repository objects. If you started from the `initial` project, these object already exist.

The following listing (from `src/main/java/com/example/crudwithvaadin/Customer.java`) defines the customer entity:

```
package com.example.crudwithvaadin;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;

@Entity
public class Customer {

        @Id
        @GeneratedValue
        private Long id;

        private String firstName;

        private String lastName;

        protected Customer() {
        }

        public Customer(String firstName, String lastName) {
                this.firstName = firstName;
                this.lastName = lastName;
        }

        public Long getId() {
                return id;
        }
```

```
        public String getFirstName() {
                return firstName;
        }

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

        public String getLastName() {
                return lastName;
        }

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

        @Override
        public String toString() {
                return String.format("Customer[id=%d, firstName='%s',
lastName='%s']", id,
                                firstName, lastName);
        }

}
```

The following listing (from `src/main/java/com/example/crudwithvaadin/CustomerRepository.java`) defines the customer repository:

```
package com.example.crudwithvaadin;

import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface CustomerRepository extends JpaRepository<Customer, Long> {

        List<Customer> findByLastNameStartsWithIgnoreCase(String lastName);
}
```

The following listing (from `src/main/java/com/example/crudwithvaadin/CrudWithVaadinApplication.java`) shows the application class, which creates some data for you:

```
package com.example.crudwithvaadin;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class CrudWithVaadinApplication {

	private static final Logger log =
LoggerFactory.getLogger(CrudWithVaadinApplication.class);

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

	@Bean
	public CommandLineRunner loadData(CustomerRepository repository) {
		return (args) -> {
			// save a couple of customers
			repository.save(new Customer("Jack", "Bauer"));
			repository.save(new Customer("Chloe", "O'Brian"));
			repository.save(new Customer("Kim", "Bauer"));
			repository.save(new Customer("David", "Palmer"));
			repository.save(new Customer("Michelle",
"Dessler"));

			// fetch all customers
			log.info("Customers found with findAll():");
			log.info("-------------------------------");
			for (Customer customer : repository.findAll()) {
				log.info(customer.toString());
			}
			log.info("");

			// fetch an individual customer by ID
			Customer customer = repository.findById(1L).get();
			log.info("Customer found with findOne(1L):");
			log.info("--------------------------------");
			log.info(customer.toString());
```

```
                        log.info("");

                        // fetch customers by last name
                        log.info("Customer found with
findByLastNameStartsWithIgnoreCase('Bauer'):");
                        log.info("----------------------------------------
---");
                        for (Customer bauer : repository

.findByLastNameStartsWithIgnoreCase("Bauer")) {
                                log.info(bauer.toString());
                        }
                        log.info("");
                };
        }

}
```

Vaadin Dependencies

If you checked out the `initial` project, or you created your project with initializr, you have all necessary dependencies already set up. However, the rest of this section describes how to add Vaadin support to a fresh Spring project. Spring's Vaadin integration contains a Spring Boot starter dependency collection, so you need add only the following Maven snippet (or a corresponding Gradle configuration):

```
<dependency>
        <groupId>com.vaadin</groupId>
        <artifactId>vaadin-spring-boot-starter</artifactId>
</dependency>
```

The example uses a newer version of Vaadin than the default one brought in by the starter module. To use a newer version, define the Vaadin Bill of Materials (BOM) as follows:

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

In developer mode, the dependency is enough, but When you are building for production, you need to enable your app for production builds.

> By default, Gradle does not support BOMs, but there is a handy plugin for that. Check out the `build.gradle` build file for an example of how to accomplish the same thing.

Define the Main View class

The main view class (called `MainView` in this guide) is the entry point for Vaadin's UI logic. In Spring Boot applications, if you annotate it with `@Route`, it is automatically picked up and shown at the root of your web application. You can customize the URL where the view is shown by giving a parameter to the `@Route` annotation. The following listing (from the `initial` project at `src/main/java/com/example/crudwithvaadin/MainView.java`) shows a simple "Hello, World" view:

```
package com.example.crudwithvaadin;

import com.vaadin.flow.component.button.Button;
import com.vaadin.flow.component.notification.Notification;
import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.router.Route;

@Route
public class MainView extends VerticalLayout {

        public MainView() {
                add(new Button("Click me", e -> Notification.show("Hello,
Spring+Vaadin user!")));
        }
}
```

List Entities in a Data Grid

For a nice layout, you can use the `Grid` component. You can pass the list of entities from a constructor-injected `CustomerRepository` to the `Grid` by using the `setItems` method. The body of your `MainView` would then be as follows:

```
@Route
public class MainView extends VerticalLayout {

        private final CustomerRepository repo;
        final Grid<Customer> grid;

        public MainView(CustomerRepository repo) {
                this.repo = repo;
```

```
                this.grid = new Grid<>(Customer.class);
                add(grid);
                listCustomers();
        }

        private void listCustomers() {
                grid.setItems(repo.findAll());
        }

}
```

If you have large tables or lots of concurrent users, you most likely do not want to bind the whole dataset to your UI components.
Although Vaadin Grid lazy loads the data from the server to the browser, the preceding approach keeps the whole list of data in the server memory. To save some memory, you could show only the topmost results by employing paging or using lazy loading, for examply using the `grid.setItems(VaadinSpringDataHelpers.fromPagingRepository(repo))` method.

Filtering the Data

Before the large data set becomes a problem to your server, it is likely to cause a headache for your users as they try to find the relevant row to edit. You can use a `TextField` component to create a filter entry. To do so, first modify the `listCustomer()` method to support filtering. The following example (from the `complete` project in `src/main/java/com/example/crudwithvaadin/MainView.java`) shows how to do so:

```
void listCustomers(String filterText) {
        if (StringUtils.hasText(filterText)) {

grid.setItems(repo.findByLastNameStartsWithIgnoreCase(filterText));
        } else {
                grid.setItems(repo.findAll());
        }
}
```

You can hook a listener to the `TextField` component and plug its value into that filter method. The `ValueChangeListener` is called automatically as the user types into the field. Using `ValueChangeMode.LAZY` on the filter text field. The following example shows how to set up such a listener:

> This is where Spring Data's declarative queries come in handy. Writing a single line definition in the `CustomerRepository` interface.

```
TextField filter = new TextField();
filter.setPlaceholder("Filter by last name");
filter.setValueChangeMode(ValueChangeMode.LAZY);
filter.addValueChangeListener(e -> listCustomers(e.getValue()));
add(filter, grid);
```

Define the Editor Component

As Vaadin UIs are plain Java code, you can write re-usable code from the beginning. To do so, define an editor component for your `Customer` entity. You can make it be a Spring-managed bean so that you can directly inject the `CustomerRepository` into the editor and tackle the Create, Update, and Delete parts or your CRUD functionality. The following example (from `src/main/java/com/example/crudwithvaadin/CustomerEditor.java`) shows how to do so:

```
package com.example.crudwithvaadin;

import com.vaadin.flow.component.Key;
import com.vaadin.flow.component.KeyNotifier;
import com.vaadin.flow.component.button.Button;
import com.vaadin.flow.component.button.ButtonVariant;
import com.vaadin.flow.component.icon.VaadinIcon;
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.binder.Binder;
import com.vaadin.flow.spring.annotation.SpringComponent;
import com.vaadin.flow.spring.annotation.UIScope;
import org.springframework.beans.factory.annotation.Autowired;

/**
 * A simple example to introduce building forms. As your real application
is probably much
 * more complicated than this example, you could re-use this form in
multiple places. This
 * example component is only used in MainView.
 * <p>
 * In a real world application you'll most likely using a common super
class for all your
 * forms - less code, better UX.
 */
@SpringComponent
@UIScope
public class CustomerEditor extends VerticalLayout implements KeyNotifier {

        private final CustomerRepository repository;

        /**
         * The currently edited customer
         */
        private Customer customer;
```

```
        /* Fields to edit properties in Customer entity */
        TextField firstName = new TextField("First name");
        TextField lastName = new TextField("Last name");

        /* Action buttons */
        Button save = new Button("Save", VaadinIcon.CHECK.create());
        Button cancel = new Button("Cancel");
        Button delete = new Button("Delete", VaadinIcon.TRASH.create());
        HorizontalLayout actions = new HorizontalLayout(save, cancel,
delete);

        Binder<Customer> binder = new Binder<>(Customer.class);
        private ChangeHandler changeHandler;

        @Autowired
        public CustomerEditor(CustomerRepository repository) {
                this.repository = repository;

                add(firstName, lastName, actions);

                // bind using naming convention
                binder.bindInstanceFields(this);

                // Configure and style components
                setSpacing(true);

                save.addThemeVariants(ButtonVariant.LUMO_PRIMARY);
                delete.addThemeVariants(ButtonVariant.LUMO_ERROR);

                addKeyPressListener(Key.ENTER, e -> save());

                // wire action buttons to save, delete and reset
                save.addClickListener(e -> save());
                delete.addClickListener(e -> delete());
                cancel.addClickListener(e -> editCustomer(customer));
                setVisible(false);
        }

        void delete() {
                repository.delete(customer);
                changeHandler.onChange();
        }
```

```
        void save() {
                repository.save(customer);
                changeHandler.onChange();
        }

        public interface ChangeHandler {
                void onChange();
        }

        public final void editCustomer(Customer c) {
                if (c == null) {
                        setVisible(false);
                        return;
                }
                final boolean persisted = c.getId() != null;
                if (persisted) {
                        // Find fresh entity for editing
                        // In a more complex app, you might want to load
                        // the entity/DTO with lazy loaded relations for
editing
                        customer = repository.findById(c.getId()).get();
                }
                else {
                        customer = c;
                }
                cancel.setVisible(persisted);

                // Bind customer properties to similarly named fields
                // Could also use annotation or "manual binding" or
programmatically
                // moving values from fields to entities before saving
                binder.setBean(customer);

                setVisible(true);

                // Focus first name initially
                firstName.focus();
        }

        public void setChangeHandler(ChangeHandler h) {
                // ChangeHandler is notified when either save or delete
                // is clicked
                changeHandler = h;
        }
```

```
}
```

In a larger application, you could then use this editor component in multiple places. Also note that, in large applications, you might want to apply some common patterns (such as MVP) to structure your UI code.

Wire the Editor

In the previous steps, you have already seen some basics of component-based programming. By using a `Button` and adding a selection listener to `Grid`, you can fully integrate your editor into the main view. The following listing (from `src/main/java/com/example/crudwithvaadin/MainView.java`) shows the final version of the `MainView` class:

```
package com.example.crudwithvaadin;

import com.vaadin.flow.component.button.Button;
import com.vaadin.flow.component.grid.Grid;
import com.vaadin.flow.component.icon.VaadinIcon;
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.Route;
import org.springframework.util.StringUtils;

@Route
public class MainView extends VerticalLayout {

        private final CustomerRepository repo;

        private final CustomerEditor editor;

        final Grid<Customer> grid;

        final TextField filter;

        private final Button addNewBtn;

        public MainView(CustomerRepository repo, CustomerEditor editor) {
                this.repo = repo;
                this.editor = editor;
                this.grid = new Grid<>(Customer.class);
                this.filter = new TextField();
                this.addNewBtn = new Button("New customer",
```

```
VaadinIcon.PLUS.create());

                // build layout
                HorizontalLayout actions = new HorizontalLayout(filter,
addNewBtn);
                add(actions, grid, editor);

                grid.setHeight("300px");
                grid.setColumns("id", "firstName", "lastName");
                grid.getColumnByKey("id").setWidth("50px").setFlexGrow(0);

                filter.setPlaceholder("Filter by last name");

                // Hook logic to components

                // Replace listing with filtered content when user changes
filter
                filter.setValueChangeMode(ValueChangeMode.LAZY);
                filter.addValueChangeListener(e ->
listCustomers(e.getValue()));

                // Connect selected Customer to editor or hide if none is
selected
                grid.asSingleSelect().addValueChangeListener(e -> {
                        editor.editCustomer(e.getValue());
                });

                // Instantiate and edit new Customer the new button is
clicked
                addNewBtn.addClickListener(e -> editor.editCustomer(new
Customer("", "")));

                // Listen changes made by the editor, refresh data from
backend
                editor.setChangeHandler(() -> {
                        editor.setVisible(false);
                        listCustomers(filter.getValue());
                });

                // Initialize listing
                listCustomers(null);
        }

        // tag::listCustomers[]
```

```
        void listCustomers(String filterText) {
                if (StringUtils.hasText(filterText)) {

grid.setItems(repo.findByLastNameStartsWithIgnoreCase(filterText));
                } else {
                        grid.setItems(repo.findAll());
                }
        }
        // end::listCustomers[]

}
```

Build an executable JAR

You can run the application from the command line with Gradle or Maven. You can also build a single executable JAR file that contains all the necessary dependencies, classes, and resources and run that. Building an executable jar makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you use Gradle, you can run the application by using `./gradlew bootRun`. Alternatively, you can build the JAR file by using `./gradlew build` and then run the JAR file, as follows:

```
java -jar build/libs/gs-crud-with-vaadin-0.1.0.jar
```

If you use Maven, you can run the application by using `./mvnw spring-boot:run`. Alternatively, you can build the JAR file with `./mvnw clean package` and then run the JAR file, as follows:

```
java -jar target/gs-crud-with-vaadin-0.1.0.jar
```

You can see your Vaadin application running at http://localhost:8080. The steps described here create a runnable JAR. You can also build a classic WAR file.

Summary

Congratulations! You have written a full-featured CRUD UI application by using Spring Data JPA for persistence. And you did it without exposing any REST services or having to write a single line of JavaScript or HTML.

See Also

The following guides may also be helpful:

- Building an Application with Spring Boot
- Accessing Data with JPA
- Accessing Data with MongoDB
- Accessing Data with Neo4j

- Accessing data with MySQL

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines.