Creating Rich Web Applications with Jakarta EE and Vaadin

The Payara® Platform - Production-Ready,
Cloud Native and Aggressively Compatible.

User Guide

Contents

Guide Updated: **July 2023**

Modern web applications have become the de facto way by which organisations deliver their products and services to their intended users. Almost every one of us uses one web application or the other, whether on our personal computers or mobile phones. Gmail, Twitter, Instagram, Facebook, Canva, YouTube are all examples of web applications that have become part of our daily lives. Companies like Google and Amazon pioneered the use of massive compute power to create highly customised user experiences in their web applications. Over time, users have come to expect a certain standard and experience when consuming web applications.

Jakarta EE, as a general-purpose development platform, has had excellent support for developing web applications from the beginning. With its Jakarta Faces specification, you can develop web applications for a wide variety of domains. However, Faces UI components are mostly created in XML. For Java developers that want to get things done faster, this can be daunting.

Vaadin is an open-source UI development framework that has a pure Java API for development of modern, rich web applications. As a component-based framework, you create your user interfaces by composing individual components into your desired look using a Java API, in a similar way to how Swing/JavaFX development is done.

In this guide, we look at building rich web applications with Vaadin on the Jakarta EE Platform. We start by defining what a rich web application is, what Vaadin is and isn't, the structure of a Vaadin application, and how to create a sample app using the Vaadin and Jakarta EE. By the end of this guide, you will have an overview of how to get started developing your own rich web applications on the Jakarta EE Platform using Vaadin. You can take a look at the sample app for this guide currently deployed to the cloud here.

Rich Web Applications

Rich web applications, sometimes also called rich internet applications, can be defined as web applications that deliver the experiences of desktop applications in a browser. Desktop applications have traditionally been characterised by instant results, dynamic changes and overall snappy feel. Rich web applications are apps that deliver similar experiences over HTTP.

A typical example of a rich web application is Google's Gmail. It allows you to read and compose mails simultaneously without needing to refresh the browser window. All actions in the app happen without reloading the full page. Another rich web application is Canva, a creative web application that allows you to do photo and video editing along with other creative work in the browser.

Rich web applications differ from traditional web applications by the experiences they offer. Whereas traditional web applications have mostly been characterised by server-side processing and static pages, rich web applications use a combination of server and client-side processing to offer a much fluid and smoother user experience. Some characteristics of rich web applications are:

Characteristics

There are some features that differentiate rich web applications from traditional applications. These include:

Interactivity

Rich web applications allow users to interact with the application in real-time, providing a responsive and fluid user experience. They can perform actions like drag and drop, instant form validation, auto complete, and live data updates without needing to reload the entire page.

Client-Side Processing

Unlike traditional web applications where most processing occurs on the server, rich web applications shift some processing tasks to the client-side using JavaScript and other client-side technologies. This reduces the need for server roundtrips, improving performance and allowing for more interactive features.

Dynamic Content

RWAs can dynamically update the content on the page without requiring a full-page refresh. This enables smoother transitions, real-time data updates, and asynchronous processing, resulting in a more engaging and seamless user experience.

Cross-Platform Compatibility

Rich web applications are designed to work seamlessly across different platforms and devices, including desktops, laptops, tablets, and mobile devices. They often utilise responsive design principles to adapt the layout and functionality to various screen sizes.

Rich User Interface (UI)

RWAs often incorporate visually appealing and interactive UI components, including advanced forms, charts, graphs, interactive maps, sliders, and multimedia elements. These components enhance the usability and attractiveness of the application.

Offline Capabilities

Some RWAs incorporate offline capabilities, allowing users to continue using the application even when an internet connection is not available. Offline data synchronisation ensures that changes made while offline are synchronised with the server once a connection is reestablished.

Vaadin

Vaadin is an open source web application development framework that simplifies the development of rich business web applications using Java. You mostly build fully functioning rich web applications without needing HTML, CSS and JavaScript expertise. Using a server-side programming model, the UI components and the business logic reside on the server, allowing you to focus on creating your UI in pure Java. This in turn abstracts away the need for client-side scripting. Vaadin handles the communication between the server and the client automatically, ensuring a seamless, smooth, and rich user experience.

Key Features of Vaadin

Component-Based Architecture

Vaadin provides a wide range of core UI components, such as buttons, tables, forms, charts, and layouts. These components are highly customizable and can be easily assembled to create complex and visually appealing user interfaces.

Data Binding and Validation

Vaadin provides powerful data binding capabilities, allowing you to bind UI components directly to data sources, such as Java objects or backend services. It also includes built-in validation mechanisms to ensure data integrity and consistency. It has tight integration with the Jakarta Bean Validation API for automatic validation of user input.

Server-Side Rendering

Vaadin renders the UI on the server-side, generating HTML and JavaScript that is sent to the client browser. This approach enables efficient data handling, automatic state management, and reduces the amount of code that needs to be written.

Responsive Design

Vaadin supports responsive design principles, allowing your applications to adapt their layout and behaviour based on the screen size and device type. This allows you to develop applications that are optimised for desktop, tablet, and mobile devices.

Event-Driven Programming Model

Vaadin uses an event-driven programming model, where user interactions trigger events on the server. You can respond to these events by implementing event listeners in Java, enabling dynamic and responsive user interfaces.

Integrations

Vaadin seamlessly integrates with other Java frameworks, libraries, and backend technologies. Using the Vaadin CDI add on, it has tight integration with the Jakarta EE Platform, allowing you to use the best of both.

What Vaadin Is Not

As discussed in the last section, Vaadin is for developing rich web applications, mostly in a business context. Of course, you can use Vaadin to develop any web app, it is, however, more suited to developing applications that, at their core, take user data, carry out some kind of processing on the data and then give back some kind of response. For example, the app for this guide is one that takes input from the user, makes a specific request to the OpenAI GPT API based on the user input, then shows the response.

Even though it is a great framework for getting applications out fast, it may not necessarily be the right tool for all web applications. For example, you may want to create a blogging application. You would be better off picking a much more suitable tool like Wordpress than using Vaadin. However, if you want to create for instance a banking app, or some internal number crunching application that requires dashboards and data visualisations among others, then Vaadin is a great choice.

Setup

Before we start the technical discussion of using Vaadin, let us first set it up using Maven. The following dependency shows the addition of Vaadin version 24.1.0 to a typical Jakarta EE 10/MicroProfile 6 based application.

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

  <dependencies>
      <dependency>
          <groupId>com.vaadin</groupId>
          <artifactId>vaadin-core</artifactId>
      </dependency>
      <dependency>
          <groupId>com.vaadin</groupId>
          <artifactId>vaadin-cdi</artifactId>
      </dependency>
      <dependency>
          <groupId>jakarta.platform</groupId>
          <artifactId>jakarta.jakartaee-api</artifactId>
          <version>10.0.0</version>
          <scope>provided</scope>
      </dependency>
      <dependency>
          <groupId>org.eclipse.microprofile</groupId>
          <artifactId>microprofile</artifactId>
          <version>6.0</version>
          <type>pom</type>
          <scope>provided</scope>
      </dependency>
  </dependencies>
```

With these in place, we are ready to start using Vaadin on top of Jakarta EE 10. Let's go!

The Structure of A Vaadin Application

A typical Vaadin application will comprise custom components composed from the core components of the framework. For example, the image shown below is the UI for the Jakarta GPT app of this guide. It features the traditional forms and tables found in all web applications in one form or the other.

This UI is a component that comprises buttons, text fields, an image field, a grid/table and labels. A typical Vaadin application will be made up of the UI components, navigation between those components, validation of user input and client server communication. With the exception of the creation of the UI components that represent the various aspects of your application, the Vaadin framework does the heavy lifting for all the others, giving you simple constructs for navigation, data validation and client-server communication. The rest of this guide breaks down these aspects of a Vaadin application, starting with UI components.

UI Components

Vaadin application user interfaces are composed from the components provided by the framework. A typical component will consist of different input and output components arranged in desired order using provided layouts. The full component that renders the user interface shown in Image 1-1 is shown below.

```
public class PointsOfInterestView extends VerticalLayout {
    private Grid<PointOfInterest> grid;
    private Binder<SearchCriteria> binder;
    private Button searchButton;
    private Button resetButton;
    private DynamicFileDownloader pdfDownload;
    private HorizontalLayout userInputLayout;
    private TextField totalTextField;
    private SearchCriteria searchCriteria;
    private PointsOfInterestResponse response;
    private ComboBox<String> currencyField;
    private static final Map<String, Locale> currencies = new HashMap<>();

    static {
        currencies.put("US Dollar ($)", Locale.US);
        currencies.put("British Pound (\u00a3)", Locale.UK);
        currencies.put("Euro (\u20ac)", Locale.GERMANY);
    }
```

```
    @PostConstruct

    private void init() {


        Component logoLayout = ComponentUtil.generateTitleComponent("images/
trip_on_budget.png","BudgetJourney");

        // Create the binder for the search criteria
        binder = new Binder<>(SearchCriteria.class);

        userInputLayout = new HorizontalLayout();
        userInputLayout.setDefaultVerticalComponentAlignment(Alignment.END);

        // Create the text fields for the search criteria
        TextField cityField = new TextField("Your next destination:");
        cityField.setWidth("300px");
        cityField.getStyle().set("margin-right", "10px");
        cityField.setPlaceholder("city name");

        binder.forField(cityField)
                .asRequired("City name is required")
                .bind(SearchCriteria::getCity, SearchCriteria::setCity);


        var budgetField = new NumberField("Your Budget")
                .withMin(1)
                .withPlaceholder("Your budget")
                .withStyle("margin-right", "10px")
                .withWidth("300px")
                .withRequired(true)
                .withTitle("Enter your budget:");

        binder.forField(budgetField)
                .asRequired("Budget is required")
                .withValidator(budget -> budget > 0, "Budget must be greater
than zero")
                .withConverter(new DoubleToBigDecimalConverter())
                .bind(SearchCriteria::getBudget, SearchCriteria::setBudget);

        // Create the search button
        searchButton = new Button("Go!");
```

```
        searchButton.getStyle().set("margin-top", "10px");
        searchButton.addClickListener(e -> searchPointsOfInterest());
        searchButton.setDisableOnClick(true);

        resetButton = new Button()
                .withIcon(VaadinIcon.TRASH.create())
                .withStyle("margin-top", "10px")
                .withClickListener(b -> resetFields());

        currencyField = new ComboBox<>();
        currencyField.setItems(currencies.keySet());
        currencyField.setPlaceholder("Pick a currency");
        userInputLayout.add(cityField, budgetField, currencyField, resetButton,
searchButton);

        add(logoLayout, userInputLayout);
        totalTextField = new TextField();
        totalTextField.setVisible(false);
        // Create the grid to display the points of interest
        grid = new Grid<>();
        grid.addColumn(PointOfInterest::getName).setHeader("Place").
setFlexGrow(1).setSortable(true);
        grid.addColumn(PointOfInterest::getInfo).setHeader("Info").
setFlexGrow(2);
        Grid.Column<PointOfInterest> costColumn = grid.addColumn(v ->
renderCost(v.getCost()));
        costColumn.setFooter(totalTextField);
        costColumn.setHeader("Cost").setFlexGrow(0).setSortable(true).
setTextAlign(ColumnTextAlign.END);
        grid.setSelectionMode(Grid.SelectionMode.NONE);
        add(grid);

        pdfDownload = new DynamicFileDownloader();
        pdfDownload.setText("PDF");
        pdfDownload.setFileName("itinerary_" + LocalDateTime.now(ZoneOffset.
UTC) + ".pdf");


    }
```

Class `PointsOfInterestView` extends `com.vaadin.flow.component.orderedlayout.VerticalLayout`, making it a layout that vertically stacks its children, one atop the other. It begins with the declaration of instance variables comprising text fields, buttons, a grid, combo box and some custom objects. All these objects are then instantiated in an initialization method annotated with `@jakarta.annotation.PostConstruct`. This method will be invoked after an instance of this bean is fully instantiated by the Jakarta CDI runtime. Remember this application declares a dependency on Vaadin CDI addon, so we can freely use Jakarta CDI to manage our dependencies and objects.

A typical Vaadin component is instantiated like any other Java object. For instance the grid object is instantiated as `grid = new Grid<>()`, as are all the other built-in Vaadin components. Thanks to its rich Java API, we are able to customise the various components by calling different methods on them. For instance the searchButton object gets a "margin-top" CSS style set by calling `getStyle()` method on it. All the other components are customised by calling relevant methods on them.

Arranging With Layouts

Class `PointsOfInterestView`, by extending `VerticalLayout`, is itself a component capable of accepting and arranging other components in a vertical hierarchy. In Vaadin applications, you will use a myriad combination of different layouts to achieve the exact arrangement of UI elements your application requires. Within the class is another layout of type `HorizontalLayout`. A `HorizontalLayout` is a one that arranges its components, or children in a horizontal manner. It is the counterpart to `VerticalLayout`.

The `cityField`, `budgetField`, `currencyField`, `resetButton` and `searchButton` are all placed together in a horizontal layout. This places them side by side horizontally, an arrangement that our application requires. Referring back to Image 1-1, this results in the input and button fields above the grid. Both `HorizontalLayout` and `VerticalLayout` offer a very convenient way of placing your app in an exact position as your application desires.

Displaying Data With Grid

Almost every nontrivial application will need to display data in one form or the other. Vaadin ships with a Grid component that allows you to display tabular data by attaching it to a source. The source of the data should be a Collection of any type you want to display in the table.

Class `PointsOfInterestView` uses the `com.vaadin.flow.component.grid.Grid` component to display data returned from the OpenAI GPT API. The grid is instantiated and the various columns are set to match the fields of the Java class whose instances are going to be displayed in the grid. Remember as the grid is typesafe, it is bound to a type through its setItems() method. You can customise all aspects of the grid including the columns rendering labels. The grid component is one of the most versatile components in the Vaadin library.

Creating Dynamism With Events

Vaadin components send information from the client to the server through the firing of events. An event is any occurrence on the component that your application might be interested in. For example you can listen for mouse clicks on a given button and take some form of action. For example the `searchButton` object, when clicked, should cause the user entered data to be relayed to the server for processing. We attach a click listener to it by calling the `addClickListener()` method, passing in a lambda.

In our code, we call the `searchPointsOfInterest()`. You can listen for any event that a given component can emit. Using events allows you to deliver rich experiences based on user interaction anywhere in your Vaadin application.

Binding Data

You can bind properties in your Java classes to UI components in Vaadin using a binder. A binder is a collection of bindings, each representing the mapping of a single field, through converters and validators, to a backing property. A binder instance can be bound to a single bean instance at a time but can be rebound as needed. Class `PointsOfInterestView` uses a `com.vaadin.flow.data.binder.Binder` binder instance to bind the fields in class `SearchCriteria`, making any given instance of `SearchCriteria` passed to the binder get bound to values held by the bound fields.

Data Validation

Making sure user entered data is valid is an important part of web application development. As the application is opened to the world, there is always the risk of having users entering invalid data, whether by accident or intentionally. To prevent that, you should validate user entered data to ensure it meets some minimum application requirement before processing. Vaadin has different ways of validating user entered data. You can set constraints on input components themselves or use the binder to validate. On the Jakarta EE platform, you can use the Bean Validation API and instead of Binder, use the Bean Validation aware `com.vaadin.flow.data.binder.BeanValidationBinder` sub class.

Class `PointsOfInterestView` manually adds validations to the binder using its fluent API. For the citiField text field, the requirement is that it's not null or empty, and if that constraint fails, then the message "City name is required" is shown. For the `budgetField`, which is a number field bound type `BigDecimal` in the Java code (`SearchCriteria` class), the constraint is that a value greater than zero must be set, if not then the message "Budget must be greater than zero" is shown.

With our validations in place, a call to the `writeBeanIfValid` method on the binder will cause validations to take place. The `searchPointsOfInterst()` method in which the binder is bound to an instance of its bean is shown below.

```
private void searchPointsOfInterest() {
        // Bind the search criteria to the binder
        searchCriteria = new SearchCriteria();

        if (binder.writeBeanIfValid(searchCriteria)) {
            // Call the suggestPointsOfInterest method and update the grid with
the results

            response = tripsAdvisorService
                    .suggestPointsOfInterest(searchCriteria.getCity(),
searchCriteria.getBudget());

            if (response.getError() != null) {
                showErrorMessage(String.format("Failed loading data from OpenAI
GPT: %n%s", response.getError()));
            } else {

                grid.setItems(response.getPointsOfInterest());
                totalTextField.setVisible(true);
                totalTextField.setValue(renderCost(response.getTotalCost()));
                downloadAsPDF();


            }

            searchButton.setEnabled(true);
        }
    }
```

The method creates an instance of the `SearchCriteria` class and passes it to the `writeBeanIfValid` method of the binder. This method will write the values of the bound instances to the passed bean instance only if all declared validations pass. If the user entered data is valid, then a call is made to the backend with the city and budget. The returned response is used to populate the grid by calling its `setItems()` method. If there is any constraint violation, Vaadin automatically shows the various messages assigned to each field. All this happens automatically on your behalf.

Navigation With Routers

Even though Vaadin can be used to develop single page applications, you can also navigate between components through the use of the `com.vaadin.flow.router.Route`. Class

`PointsOfInterestView` is a route that is mapped to the path 'budget-journey', meaning when the user navigates to that path, which is relative to the application context path, Vaadin will instantiate an instance of `PointsOfInterestView`. To declare the class as mapped to the route 'budget-journey', we annotate it as follows:

```
@Route(value = "budget-journey", layout = ParentAppLayout.class)
public class PointsOfInterestView extends VerticalLayout {
}
```

The `com.vaadin.flow.router.Route` annotation takes a value that is the URL path to which this view should be mapped. The second parameter of the annotation as shown above is layout.

Consistent Look With App Layouts

Most web applications have a consistent look and feel. To achieve the same with Vaadin, you can use `com.vaadin.flow.component.applayout.AppLayout`. An app layout is a component for building common application layouts. The layout consists of three sections: a horizontal navigation bar (navbar), a collapsible navigation drawer (drawer) and a content area. An application's main navigation blocks should be positioned in the navbar and/or drawer while views are rendered in the content area. App Layout is responsive and adjusts automatically to fit desktop, tablet, and mobile screen sizes. The `ParentLayout` used in the application is shown below.

```
public class ParentAppLayout extends AppLayout {

@PostConstruct
private void init() {
UI.getCurrent().getElement().setAttribute("theme", Lumo.LIGHT);

createHeader();
createDrawer();
}


private void createHeader() {
H1 logo = new H1("Jakarta EE GPT");
logo.addClassNames(
LumoUtility.FontSize.LARGE,
LumoUtility.Margin.MEDIUM);
```

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

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

addToNavbar(header);

}

private void createDrawer() {
var verticalLayout = new VerticalLayout();
var pointOfViewLink = generateComponent("Trip", VaadinIcon.AIRPLANE.create(),
PointsOfInterestView.class);
var imageGen =generateComponent("AI Image", VaadinIcon.CAMERA.create(),
GptImageGenerator.class);
var home =generateComponent("Home", VaadinIcon.HOME_O.create(), HomePage.
class);
verticalLayout.add(home, pointOfViewLink, imageGen);
addToDrawer(verticalLayout);
}

}
```

The `ParentLayout` extends `AppLayout` and sets up the header and drawer parts. The drawer is similar to what you find on mobile devices. It can be oriented vertically or horizontally. This app uses the vertical drawer and can be seen in the first image shown above. Passing this layout class to view components causes them to be rendered consistently.

PWA With AppShellConfigurator

You can make your Vaadin application a progressive web app by implementing the `com.vaadin.flow.component.page.AppShellConfigurator` interface as shown below. This interface, coupled with the `com.vaadin.flow.server.PWA` annotation makes our app installable as an app on the user's machine. The `AppShellConfigurator` implementation is shown below.

```
@PWA(name = "Budget Journey With ChatGPT", shortName = "budgetGPT",
description = "A Jakarta EE/Vaadin app that takes a city/country "
+
"and a budget amount then suggests places to visit based on the budget")
public class AppShell implements AppShellConfigurator {
}
```

With this in place, Vaadin will make the install icon available to the user in the address bar. Installing the app makes it available alongside other installed apps of the user.

Business Logic

As a server side framework, both the UI and business components are side by side. PointsOfInterestView class depends on two business components which are injected using Jakarta CDI's `@Inject` annotation. The class is reproduced below, showing its dependency declaration.

```
@Route(value = "budget-journey", layout = ParentAppLayout.class)
public class PointsOfInterestView extends VVerticalLayout {

    @Inject
    private TripsAdvisorService tripsAdvisorService;
    @Inject
    private ReportService reportService;
}
```

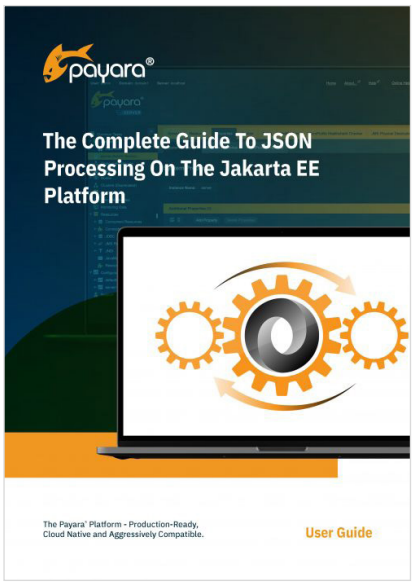
Because Vaadin is a server-side framework, both the UI and the business components are side by side. This greatly simplifies app development because you as a developer do not have to context switch between client-server side. Everything is automatically handled for you.

There's More!

In this brief guide, we have looked at Jakarta EE application development with Vaadin. There is so much more you can do with Vaadin and Jakarta EE. As stated at the beginning of this guide, the app can be accessed from here. The full code is also available on GitHub for your reference. You should check out the Vaadin tutorials for how to fulfil common application requirements like database access, user login, handling state and testing.

Found this useful? Try more of our guides:

A Developer Guide To WebSocket Development On The Jakarta EE Platform



The Complete Guide To JSON Processing On the Jakarta EE Platform

The Complete Guide to Testing on the Jakarta EE Platform