Using Templates

Vaadin 14

Suggested trainings before this one:

- Intro
- Router API
- Component & Element API

Agenda

- Intro
- Binding with @Id
- Exercise 1
- DOM
- Exercise 2
- Events
- Data Binding
- Exercise 3

Polymer Templates in Vaadin

Vaadin 14 offers two primary ways of creating UIs: through Java code or with declarative client-side templates. The template approach may be useful for example in the following scenarios:

- Your team has both frontend and backend developers and the frontend developers prefer working directly with HTML and JavaScript
- You want to integrate some third party client-side library or implement some pure HTML and JavaScript functionality
- You want to use Vaadin Designer to visually create and edit your UI

Creating a template component

A template component contains two parts: a **client-side** Polymer element and a **server-side** Java component.

Polymer

A Sugaring Library from Google

- Web Components standards
- Additional features, easier API
- “What jQuery is to JavaScript”
- Created and maintained by the Chrome team
- Current version used in Vaadin 14 is Polymer 3

Vaadin vs Polymer

Compared to a pure Polymer solution, using PolymerTemplates Vaadin has the advantage that you can define a data model and event handlers on the server-side.

Client-side Polymer element - hello-world.js

```
import { html, PolymerElement } from '@polymer/polymer/polymer-element.js';

class HelloWorld extends PolymerElement {

   static get template() {
       return html`
           <b>Hello World!</b>
       `;
   }

   static get is() {
       return 'hello-world';
   }

}

customElements.define(HelloWorld.is, HelloWorld);
```

vaadin}>

Client-side Polymer element - hello-world.js

```
import { html, PolymerElement } from '@polymer/polymer/polymer-element.js';

class HelloWorld extends PolymerElement {

   static get template() {
       return html`
           Hello World!
       `;
   }

   static get is() {
       return 'hello-world';
   }

}

customElements.define(HelloWorld.is, HelloWorld);
```

Naming conventions for custom elements:

- Must start with a letter
- Must be all lowercase
- Must contain at least one hyphen

Ok: `<my-element>`, `<large-image-display>`

Not ok: `<3-column-layout>`, `<text_wrapper>`

vaadin}>

Server-side Java component

A Java class extends from PolymerTemplate

```
public class HelloWorld extends PolymerTemplate {

}
```

Server-side Java component

Use @Tag to match the Java class to a client-side element. The annotation’s value should match the name of the web component - the return value of the `is()` function in the template.

```
@Tag("hello-world")
public class HelloWorld extends PolymerTemplate {

}
```

Server-side Java component

Use @JsModule to import the client-side element file. The file path used in the annotation parameter is relative to the `frontend` folder.

```
@JsModule("./hello-world.js")
@Tag("hello-world")
public class HelloWorld extends PolymerTemplate {

}
```

Use the template component

PolymerTemplate extends from Component so that it can be used like any other Component.

```java
HelloWorld hello = new HelloWorld();
Div layout = new Div();
layout.add(hello);
```

Binding with @Id

@Id

You can use @Id to bind client-side elements to server-side Components, which enables you to manipulate them on the server.

@Id

```
class MainPage extends PolymerElement {

    static get template() {
        return html`
            <div>Main page</div>
            <div id="content"></div>
            <hr>
            <div></div>`;
    }

    static get is() {
        return 'main-page';
    }
}
```

```
@Tag("main-page")
@JsModule("./main-page.js")
public class MainPage extends P...{

    /**
     * Note that just declare the field,
     * then use the @Id annotation,
     * quite similar as injecting with Spring or CDI,
     * DON'T instantiate it manually.
     */
    @Id("content")
    private Div content;

    public void setContent(Component component) {
        this.content.removeAll();
        this.content.add(component);
    }
}
```

@Id

You can also inject a client-side element as an Element on the server-side. It can be useful when using an element where there is no corresponding server-side Component, e,g., a third-party web component.

@Id

```
class MainPage extends PolymerElement {

    static get template() {
        return html`
            <div>Main page</div>
            <paper-card id="content"></paper-card>
            <hr>
            <div></div>`;
    }

    static get is() {
        return 'main-page';
    }
}
```

```
@Tag("main-page")
@JsModule("./main-page.js")
public class MainPage extends P...{

    @Id("content")
    private Element content;

    public void setContent(Component component) {
        this.content.removeAllChildren();
        this.content.appendChild(component.getElement());
    }
}
```

Server-side DOM manipulation

The server-side is NOT automatically in sync with the client-side.

The server-side can only read what has been set on the server.

Neither the hierarchical structure nor attributes/properties on the client side are available on the server side via API.

Server-side DOM manipulation

In practice: you should **either** configure and use the elements purely on the template side...

```
<vaadin-list-box on-change="selectionChanged">
  <vaadin-item>5k</vaadin-item>
  <vaadin-item disabled>10k (sold out)</vaadin-item>
  <vaadin-item>Marathon</vaadin-item>
</vaadin-list-box>
```

Server-side DOM manipulation

…**or** add a plain element with an id in the template, use @Id binding and configure in Java code

```
my-template.js

      <vaadin-list-box id="distanceListBox"></vaadin-list-box>

MyTemplate.java

      @Id("distanceListBox")
      private ListBox<Distance> distanceListBox;

      public MyTemplate() {
            listBox.setDataProvider(DataProvider.ofItems(backendService.getDistances()));
            listBox.setItemEnabledProvider( item -> item.isAvailable() );
            // ...
```

vaadin}>

If you try to do both, you'll end up with unwanted behavior.

```
class MainPage extends PolymerElement {

    static get template() {
        return html`
            <div id="header">
                <!-- header div has child elements -->
                <h1>Header</h1>
                <h2>Subheader</h2>
            </div>
            <div id="content">
                 <!-- content div has text inside -->
                 some text
            </div>
    }

    static get is() {
          return 'main-page';
    }
}
```

```
@Tag("main-page")
@JsModule("./main-page.js")
public class MainPage extends P...{

    @Id("header")
    private Div header;

    @Id("content")
    private Div content;

    public MainPage() {
        //returns empty stream!
        header.getChildren();

        //returns empty string!
        content.getText();
    }
}
```

vaadin}>

Exercise 1

Dealing with DOMs

Local DOM

Shady DOM

Light DOM

Shadow DOM

Composed DOM

Local DOM

The DOM that a custom element creates and manages.

Local DOM is the DOM defined inside the template() method.

```
hello-world.js

import { html, PolymerElement } from
                  '@polymer/polymer/polymer-element.js';

class HelloWorld extends PolymerElement {

   static get template() {
       return html`
           Hello World!
       `;
   }

   static get is() {
       return 'hello-world';
   }

}
```

vaadin}>

Shadow and Shady DOM

Local DOM is implemented with Shadow or Shady DOM

Shadow DOM is one of the Web Components Specifications. Most evergreen browsers already implement full native Shadow DOM.

Shady DOM is a lightweight limited implementation of Shadow DOM - because emulating full Shadow DOM in JavaScript can be performance-heavy

Light DOM

Element’s children are called light DOM (user’s point of view).

The light DOM is visible in the DevTools as a normal subtree.

```
<my-select>
  <option>Volvo</option>
  <option>Saab</option>
  <option>Tesla</option>
  <option>Audi</option>
</my-select>
```

Only Local DOM is rendered

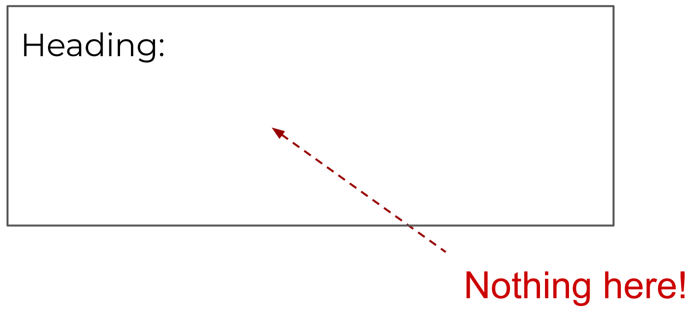
Local Dom:

```
class AppHeading extends PolymerElement {

    static get template() {
        return html`
            Heading:`;
    }

    static get is() {
          return 'app-heading';
    }
}
```

Light DOM:

```
<app-heading>Main page
    <span>Vaadin App</span>
    <span>Other text</span>
</app-heading>
```

Result:



From Light DOM to Local DOM

With <slot> elements, you can set some placeholders in the local DOM which can be filled with content from light DOM.

A <slot> can be either named or unnamed.

```
//named slot
<slot name="query"></slot>

//unnamed slot
<slot></slot>
```

Using named slot

The named slot in the local DOM will be filled with the component in the light DOM with a matching slot attribute.

Local Dom:

```
class AppHeading extends PolymerElement {

    static get template() {
        return html`
            Heading: <slot name="title"></slot>`;
    }

    static get is() {
          return 'app-heading';
    }
}
```

Light DOM:

```
<app-heading>
  Main page
    <span slot="title">Vaadin App</span>
    <span>Other text</span>
</app-heading>
```

Result:

Heading: Vaadin App

Using unnamed slot

The unnamed slot in the local DOM will be filled with all the elements (without a slot attribute) in the light DOM.

Local Dom:

```
class AppHeading extends PolymerElement {

    static get template() {
        return html`
            Heading: <slot></slot>`;
    }

    static get is() {
        return 'app-heading';
    }
}
```

Light DOM:

```
<app-heading>
    <span>Vaadin App</span>
    <span>Other text</span>
</app-heading>
```

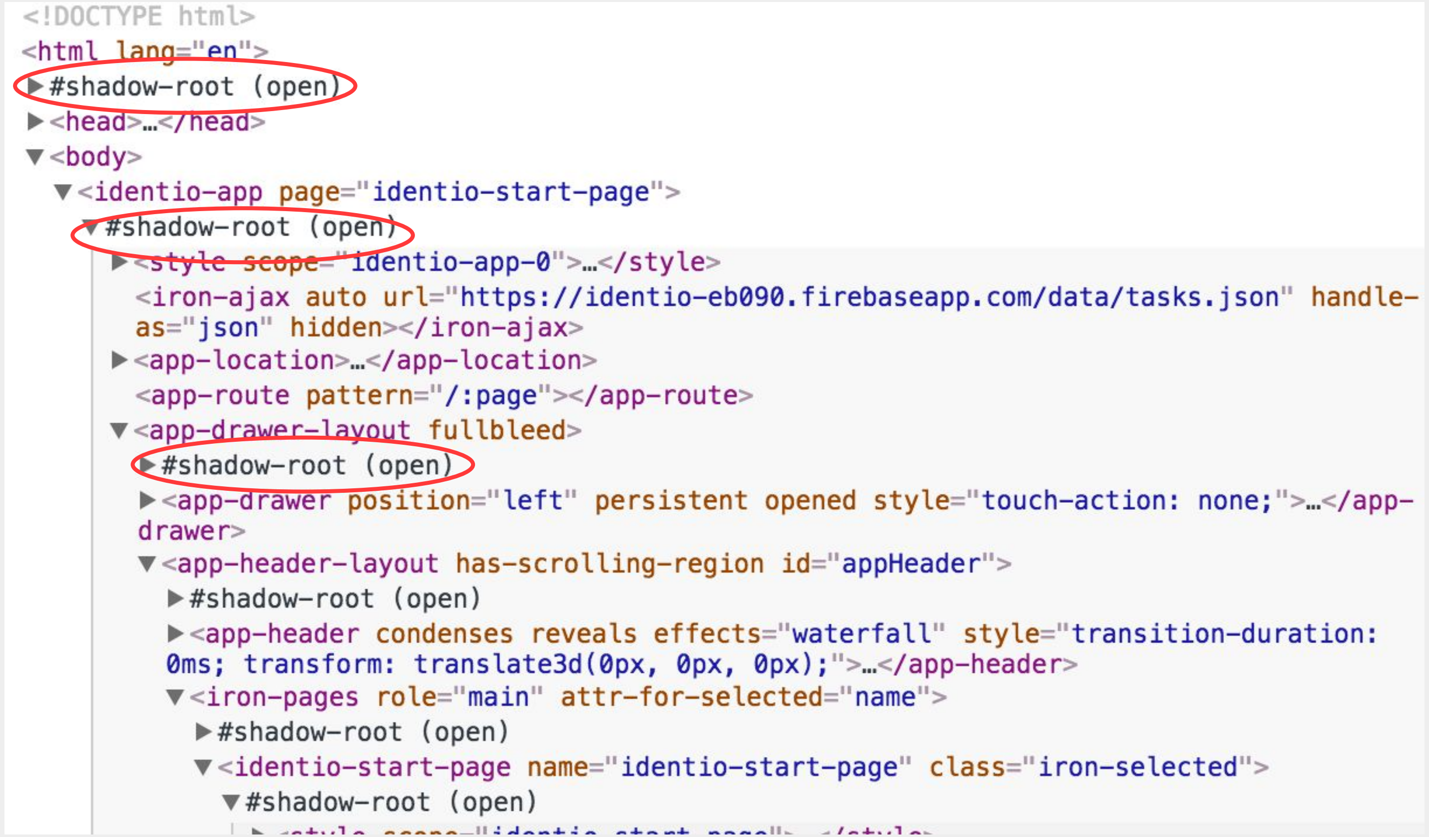
Result:

Heading: Vaadin App Other text

Composed DOM

Composed DOM is what the browser actually renders, it’s a combination of Light and Local DOM.

Shadow DOM in Chrome's DevTools



```
<!DOCTYPE html>
<html lang="en">
▶#shadow-root (open)
▶<head>…</head>
▼<body>
  ▼<identio-app page="identio-start-page">
    ▼#shadow-root (open)
      ▶<style scope="identio-app-0">…</style>
       <iron-ajax auto url="https://identio-eb090.firebaseapp.com/data/tasks.json" handle-
       as="json" hidden></iron-ajax>
      ▶<app-location>…</app-location>
       <app-route pattern="/:page"></app-route>
      ▼<app-drawer-layout fullbleed>
        ▶#shadow-root (open)
        ▶<app-drawer position="left" persistent opened style="touch-action: none;">…</app-
         drawer>
        ▼<app-header-layout has-scrolling-region id="appHeader">
          ▶#shadow-root (open)
          ▶<app-header condenses reveals effects="waterfall" style="transition-duration:
           0ms; transform: translate3d(0px, 0px, 0px);">…</app-header>
          ▼<iron-pages role="main" attr-for-selected="name">
            ▶#shadow-root (open)
            ▼<identio-start-page name="identio-start-page" class="iron-selected">
              ▼#shadow-root (open)
```

RouterLayout Revisit

RouterLayout has a default method showRouterLayoutContent(), which will append the content as its child.

```java
public interface RouterLayout extends HasElement {

    default void showRouterLayoutContent(HasElement content) {
        if (content != null) {
            getElement()
                    .appendChild(Objects.requireNonNull(content.getElement()));
        }
    }

}
```

RouterLayout Revisit

If the layout is implemented with a template, you can leave a <slot> there, so that anything added to the layout will be in the light DOM, thus ends up in the slot.

```
class MainLayout extends PolymerElement {

    static get template() {
        return html`
            <div>Header</div>
            <slot></slot>
            <div>Footer</div>`;
    }

    static get is() {
          return 'main-layout';
    }
}
```

```
@Tag("main-layout")
@JsModule("./main-layout.js")
public class MainLayout extends PolymerTemplate<..>
    implements RouterLayout {



}
```

@Id binding or <slot>?

Many times you can choose either @Id binding or a <slot> for the same results when connecting a server-side Component to a Template. If you're not sure which way you should use, consider the following:

- A <slot> is like a hole in your template that can contain almost anything. If you're not sure which exact element will be shown inside your template at any given time, consider using a <slot>. A typical use case is a top level layout that contains different Views.
- @Id binding connects a server-side component to a specific Element. If you know which Component is going to be used in that place, @Id binding is a safe choice. A typical use case is a ComboBox whose values and value change listeners you want to configure on the server.

Exercise 2

Event handling

With a template-based component, you can choose how to handle events - on the client, on the server, or both.

Client-side event handling

Add on-*event*=”methodName” to an element. The “on-xx” is a special syntax from Polymer for event handler registration in templates.

```
<button on-click="handleClick">Say hello</button>
```

Client-side event handling

Client-side event handling means that an event is handled on the client-side with a JavaScript function.

Client-side event handling

Define a JavaScript method in the template file’s class for the event handling.

```
class MyView extends PolymerElement {
    static get template() {
        return html `
            <button on-click="handleClick">Say hello</button>`;
    }

    handleClick() {
        console.log('Button was clicked.');
    }

    static get is() {
        return 'my-view';
    }
}
```

Server-side event handling

Server-side event handling means that an event is handled on the server-side with a Java method.

Server-side event handling

Instead of defining a JavaScript method in the template file, you define a Java method on the server-side and add the `@EventHandler` annotation to the method.

```
public class MyView extends PolymerTemplate{
    @EventHandler
    private void handleClick() {
        System.out.println("Received a handle click event");
    }
}
```

Connecting to server-side event handling

To connect a client side element to a server side event handler (without using @Id binding), use the same syntax as with client-side event handling, just add `on-xx=”methodName”` to an element inside the template.

```
<button on-click="handleClick">Say hello</button>
```

Hybrid

You can also define an event handler both on the client-side and on the server-side. In such cases, the client-side event handler will be called first.

Data Binding

Data Binding is the core feature of PolymerTemplates.

Model values can be bound to different parts of the element tree defined by the template.

Define a Model on the server-side

A model is a class/interface implements/extends TemplateModel. TemplateModel is the type parameter of the PolymerTemplate interface.

You don’t need to implement the model, an interface with getter/setter methods is enough.

```java
public class PolymerBindingTemplate extends PolymerTemplate<BindingModel> {

}

public interface BindingModel extends TemplateModel {
    void setMyProperty(String propertyValue);
    String getMyProperty();
}
```

Data binding syntax

To bind to a client-side element, use either **[[]]** or **{{}}**

[[]] is for one-way data-binding, meaning data flows only in server-to-client direction.

{{}} is for two-way data-binding, meaning data flows in both server-to-client and client-to-server direction.

```
<div>[[myProperty]]</div>
<my-element my-property="[[myProperty]]"></my-element>
<paper-input value="{{myProperty}}"></paper-input>
```

Data binding syntax

To bind to an **attribute**, you need to use a special syntax, i.e. use the attribute name followed by a $. Most common cases where you need to bind to an attribute instead of property are **style** and **class**.

```
<div style$="[[styleProperty]]"></div>
```

Data binding with Beans

How to bind a POJO to the client-side?

```
public class Person {
    private String firstName, lastName;
    private int age;
    private long id;

    public Person() {
        // Needed for TemplateModel
    }
    /**
     getters and setters
    */
}
```

Data binding with Beans

First, need to define a Model with getter/setter methods.

```
public class Form extends PolymerTemplate<FormModel> {

}

public interface FormModel extends TemplateModel {
    void setPerson(Person person);
    Person getPerson();
}
```

Data binding with Beans

Also need to exclude unsupported types, e.g. long, because JavaScript doesn't support it. Properties that would lead to circular dependencies also need to be excluded. You may also want to restrict some unneeded information from being transmitted to the client as a security measure.

Use @Exclude on the setter method to exclude properties.

```
public class Form extends PolymerTemplate<FormModel> {

}

public interface FormModel extends TemplateModel {
    @Exclude("id")
    void setPerson(Person person);
    Person getPerson();
}
```

Data binding with Beans

Then can bind the bean with sub properties to the client-side element.

```
<vaadin-text-field value="{{person.firstName}}"></vaadin-text-field>
<vaadin-text-field value="{{person.lastName}}"></vaadin-text-field>
<vaadin-text-field value="{{person.age}}"></vaadin-text-field>
```

Data binding with List value

The server-side looks quite similar, just define a Model with getter/setter methods for the POJO list.

```
public class EmployeesTable extends PolymerTemplate<EmployeesModel> {

}

public interface EmployeesModel extends TemplateModel {
    void setEmployees(List<Person> employees);
    List<Person> getEmployees();
}
```

Data binding with List value

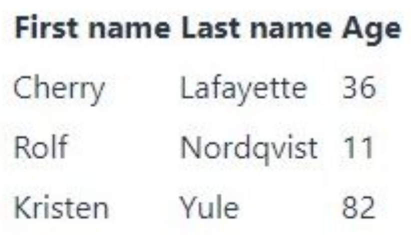
Exclude the unsupported types with @Exclude

```
public class EmployeesTable extends PolymerTemplate<EmployeesModel> {

}

public interface EmployeesModel extends TemplateModel {
    @Exclude("id")
    void setEmployees(List<Person> employees);
    List<Person> getEmployees();
}
```

Data binding with List value

On the client-side, you can use Polymer **`dom-repeat`** template to iterate through the list. Pass the list value to the **items** property; each value in the list is by default referred to as **`item`**.

```
<table>
  <thead><tr><th>First name</th><th>Last name</th><th>Age</th></tr></thead>
  <template is="dom-repeat" items="[[employees]]">
    <tr on-click="handleClick">
      <td>[[item.firstName]]</td>
      <td>[[item.lastName]]</td>
      <td>[[item.age]]</td>
    </tr>
  </template>
</table>
```

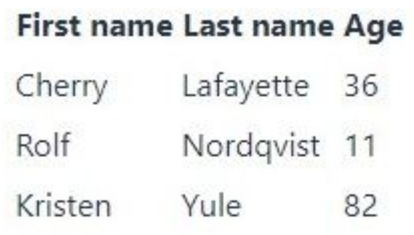


First name	Last name	Age
Cherry	Lafayette	36
Rolf	Nordqvist	11
Kristen	Yule	82

Data binding with List value

Item index can be accessed with @RepeatIndex on the server side method.

```
<table>
 <thead><tr><th>First name</th><th>Last name</th><th>Age</th></tr></thead>
  <template is="dom-repeat" items="[[employees]]">
    <tr on-click="handleClick">
      <td>[[item.firstName]]</td>
      <td>[[item.lastName]]</td>
      <td>[[item.age]]</td>
    </tr>
  </template>
</table>
```



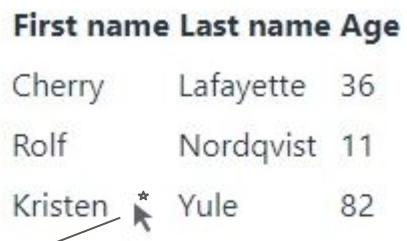
First name	Last name	Age
Cherry	Lafayette	36
Rolf	Nordqvist	11
Kristen	Yule	82

```
@EventHandler
private void handleClick(@RepeatIndex int itemIndex){
  getModel().getEmployees().get(itemIndex).setFirstName("new name");
}
```

Data binding with List value

Item index can be accessed with @RepeatIndex on the server side method.

```
<table>
 <thead><tr><th>First name</th><th>Last name</th><th>Age</th></tr></thead>
  <template is="dom-repeat" items="[[employees]]">
    <tr on-click="handleClick">
      <td>[[item.firstName]]</td>
      <td>[[item.lastName]]</td>
      <td>[[item.age]]</td>
    </tr>
  </template>
</table>
```



First name	Last name	Age
Cherry	Lafayette	36
Rolf	Nordqvist	11
Kristen	Yule	82

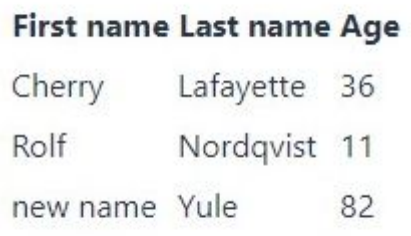
```
handleClick(2);
```

```
@EventHandler
private void handleClick(@RepeatIndex int itemIndex){
  getModel().getEmployees().get(itemIndex).setFirstName("new name");
}
```

Data binding with List value

Item index can be accessed with @RepeatIndex on the server side method.

```
<table>
 <thead><tr><th>First name</th><th>Last name</th><th>Age</th></tr></thead>
  <template is="dom-repeat" items="[[employees]]">
    <tr on-click="handleClick">
      <td>[[item.firstName]]</td>
      <td>[[item.lastName]]</td>
      <td>[[item.age]]</td>
    </tr>
  </template>
</table>


@EventHandler
private void handleClick(@RepeatIndex int itemIndex){
  getModel().getEmployees().get(itemIndex).setFirstName("new name");
}
```



First name	Last name	Age
Cherry	Lafayette	36
Rolf	Nordqvist	11
new name	Yule	82

Exercise 3

Summary

- Intro
- Binding with @Id
- DOM
- Events
- Data Binding