Component & Element API

Vaadin 14

vaadin}>

Agenda

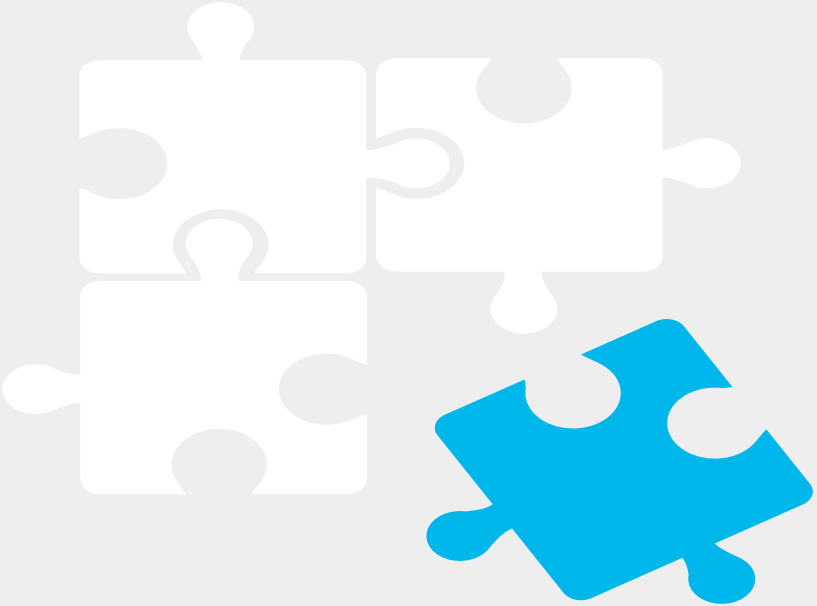
- Component API
- Event Handling
- Shortcuts
- Exercise 1
- DOM manipulation
- Communication
- Exercise 2

vaadin}>

Components

Vaadin uses a component-oriented approach in building UIs.

By combining components, we build an application.

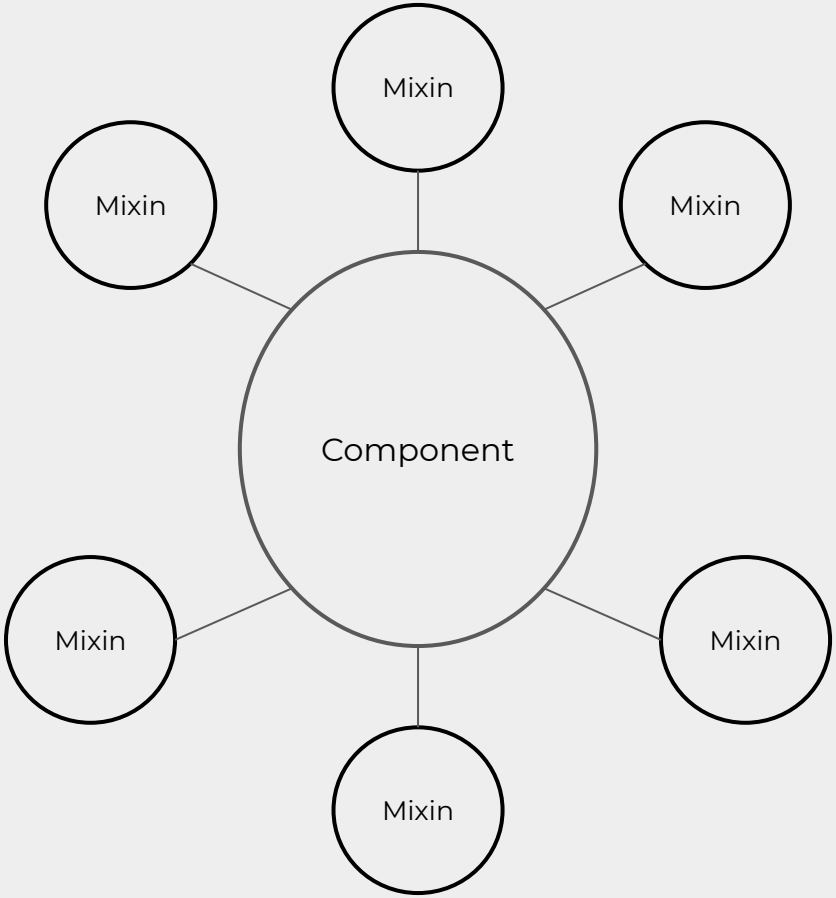


Mixins

Vaadin provides the **Component** abstract class for essential features.

Additional features are provided through mixins.

A mixin refers here to a Java interface with default methods.



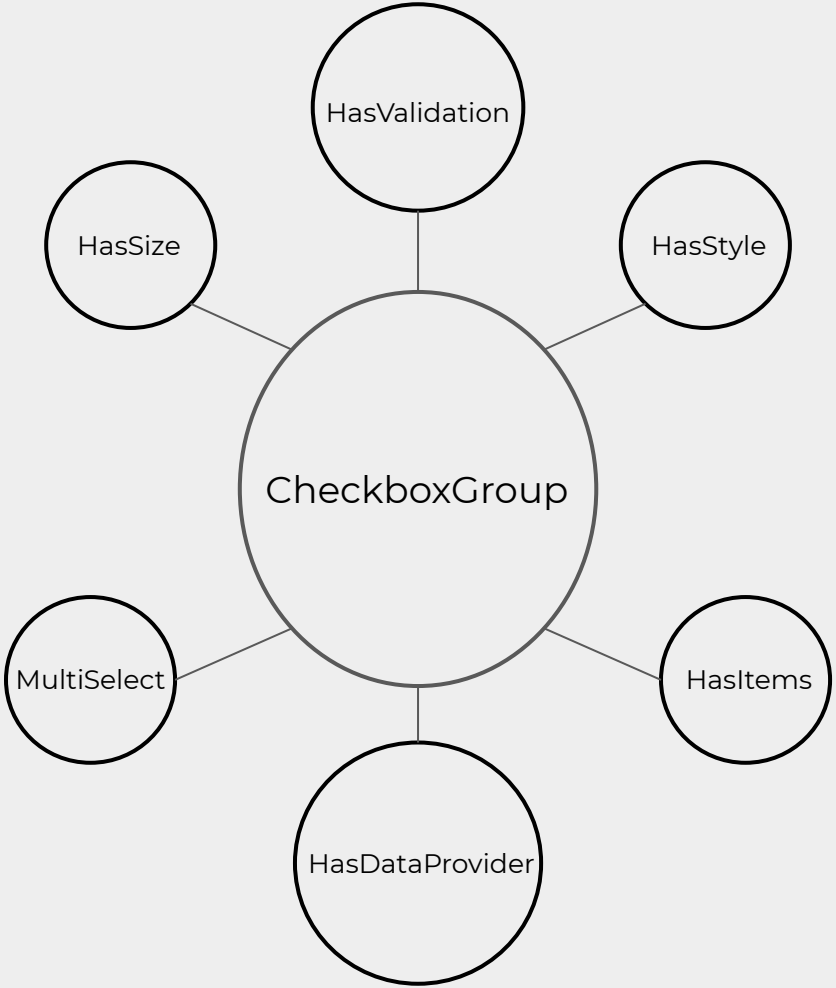
Mixins

Vaadin provides the **Component** abstract class for essential features.

Additional features are provided through mixins.

A mixin refers here to a Java interface with default methods.

For example, CheckboxGroup implements these mixins (and more!).

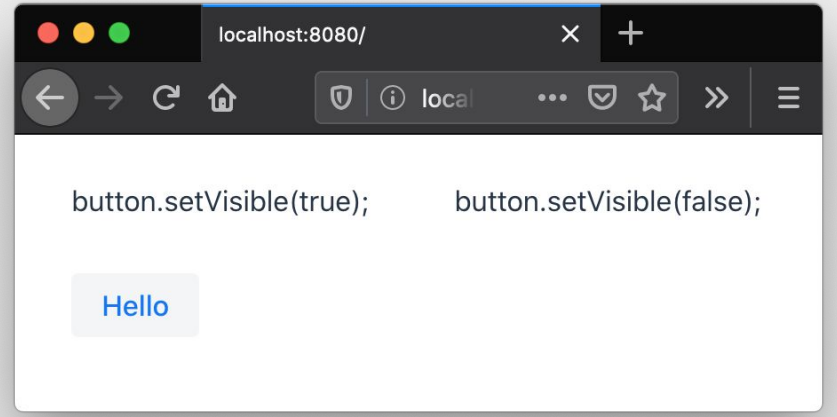


Basic features of Components

Visibility

```
component.isVisible()
```

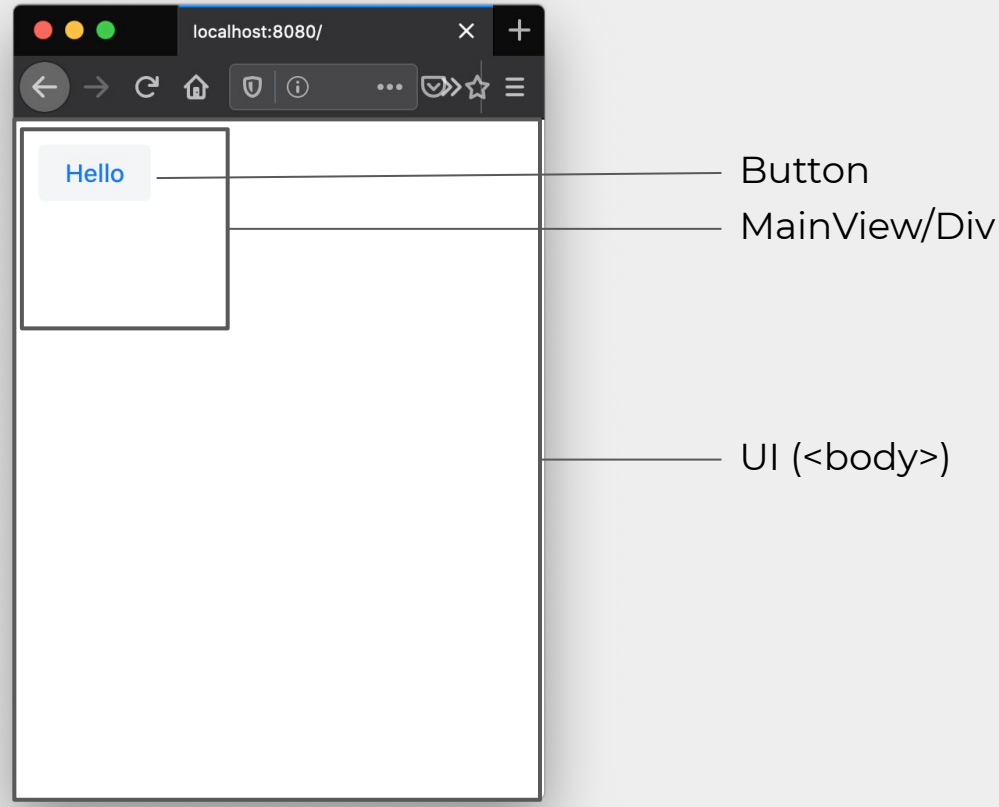
```
component.setVisible(true/false)
```



`setVisible(false)` will remove the component from the DOM and block any updates from the client

Hierarchy

```
public class MainView extends Div {

  public MainView() {
    Button button = new Button("Hello");
    add(button);
    setHeight("200px");
    setWidth("200px");
  }
}
```



Hierarchy

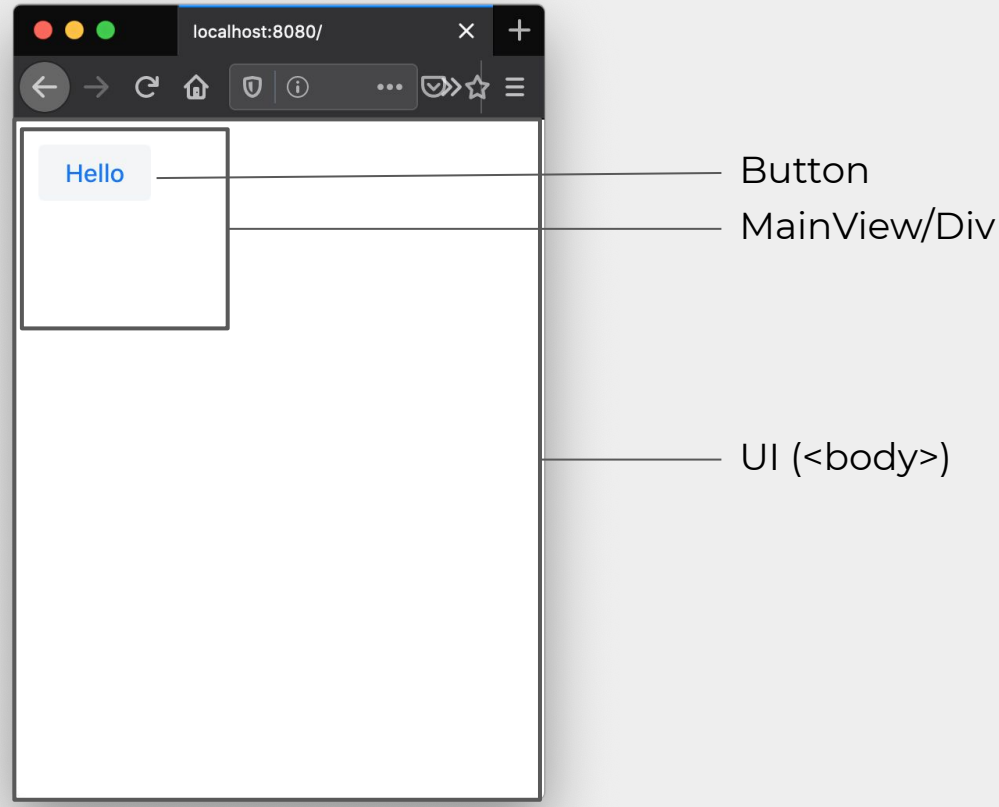
Component method	Returned type
getParent()	Optional<Component>
getChildren()	Stream<Component>
getUI()	Optional<UI>

You can access the parent of any **attached** component through `getParent()`. If the component hasn't yet been added to a parent, the method will return an empty `Optional`.

You can access the UI object

- through `component.getUI()` of any attached component
- `getParent()` of a top level component (like `MainView` here).
- through `UI.getCurrent()`
- From the `attach` event (use `addAttachListener`)

Calling `getParent()` on a UI object will return an empty Optional.



Additional component features

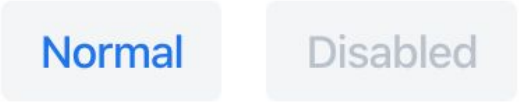
Additional component features

Many Vaadin components have some additional features through mixin interfaces, like **HasEnabled**, **HasSize**, **HasStyle**, **HasComponents**.

HasEnabled

Many components implement this interface. You can use it to put a component into a disabled state or enable it again.

```
button.setEnabled(true);
button.setEnabled(false);
button.isEnabled();
```



`setEnabled(false)` will block any updates from the client from reaching the server.

HasSize

setHeight and **setWidth** accept String values like “60%”, “200px”, “2em” etc.

setHeightFull/**setWidthFull** gives a component 100% height/width.

setSizeFull gives a component both 100% width and height.

setMaxHeight/**setMaxWidth** and **setMinHeight**/**setMinWidth** allow setting maximum and minimum dimensions.

HasSize

- getHeight(): String
- getMaxHeight(): String
- getMaxWidth(): String
- getMinHeight(): String
- getMinWidth(): String
- getWidth(): String
- setHeight(String): void
- setHeightFull(): void
- setMaxHeight(String): void
- setMaxWidth(String): void
- setMinHeight(String): void
- setMinWidth(String): void
- setSizeFull(): void
- setSizeUndefined(): void
- setWidth(String): void
- setWidthFull(): void

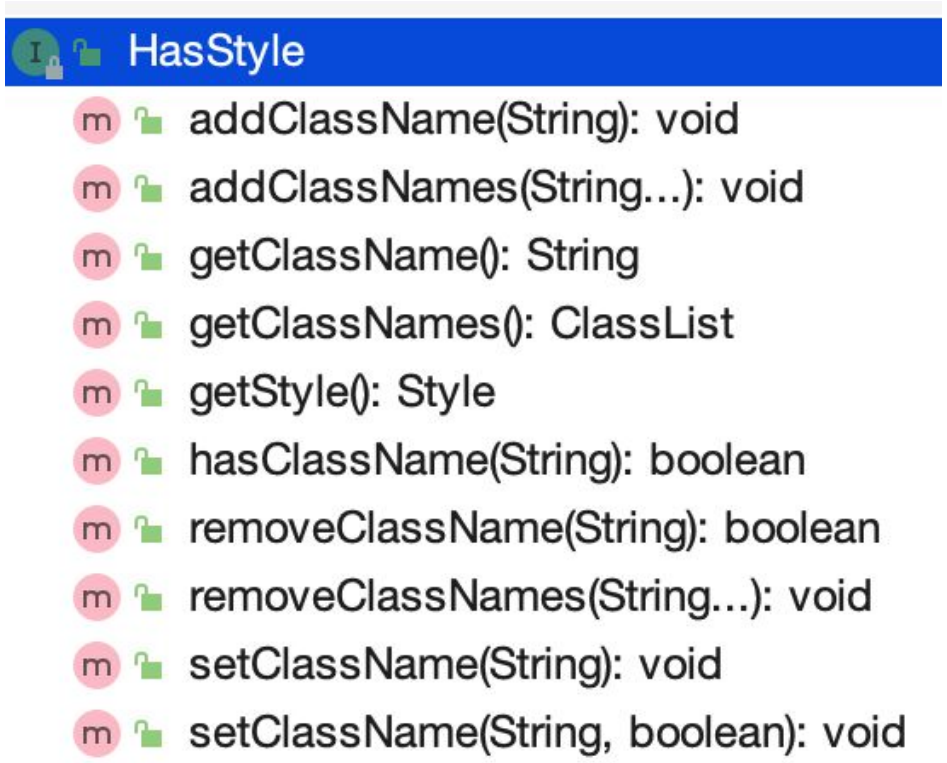
HasStyle

You can use HasStyle's methods to add and remove CSS class names.

```
component.addClassName("hello");
component.removeClassName("hello");
```

You can also manipulate the Style object to do inline styling.

```
component.getStyle().set("color", "red");
```



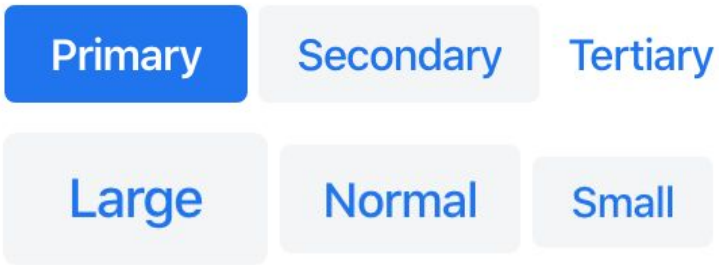
Theme Variants

Some components have predefined variants, which allows to change a component's look and feel quickly with

```
component.addThemeVariants
```

E.g.

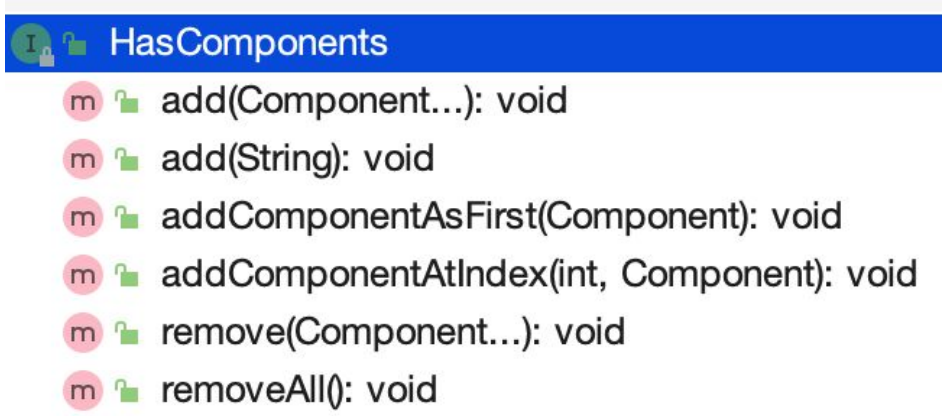
```
button.addThemeVariants(ButtonVariant.LUMO_PRIMARY,
ButtonVariant.LUMO_LARGE);
```



HasComponents

Many container components implement this interface, e.g. HorizontalLayout, VerticalLayout, FlexLayout, Div, Tabs, Board, etc.

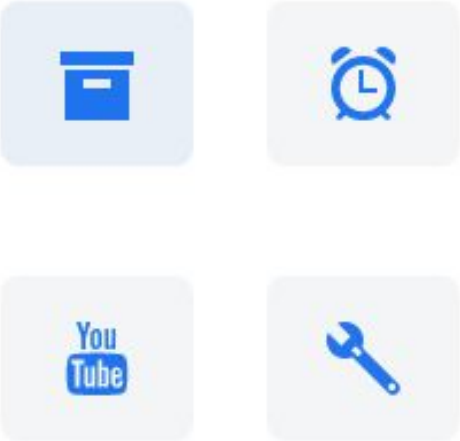
```
layout.add(component);
layout.remove(component);
```



Use of Icons

Vaadin provides a set of icons through the VaadinIcon enum.

```
new Button(VaadinIcon.ARCHIVE.create());
new Button(VaadinIcon.ALARM.create());
new Button(VaadinIcon.YOUTUBE.create());
new Button(VaadinIcon.WRENCH.create());
```



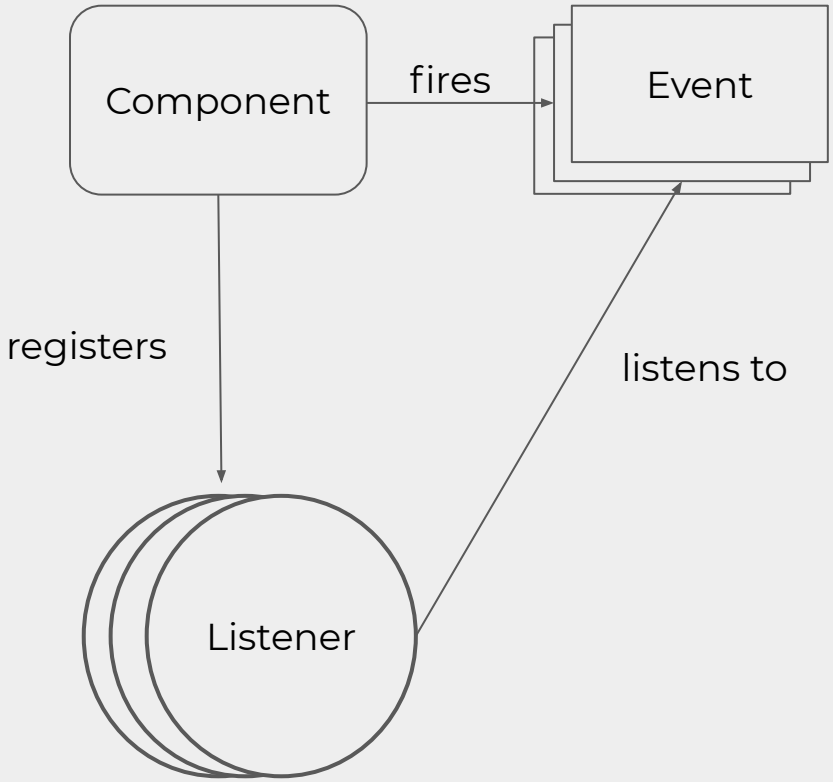
Event Handling

Event Handling

A component allows adding of listeners.

A component can fire certain types events when something interesting happens.

Listeners listen to specific types of events.



Event Handling

There are many types of events.

You can add a listeners for specific events by calling

```
component.add[x]Listener()
```

ClickEvent

FocusEvent

KeyboardEvent

...

ColumnResizeEvent

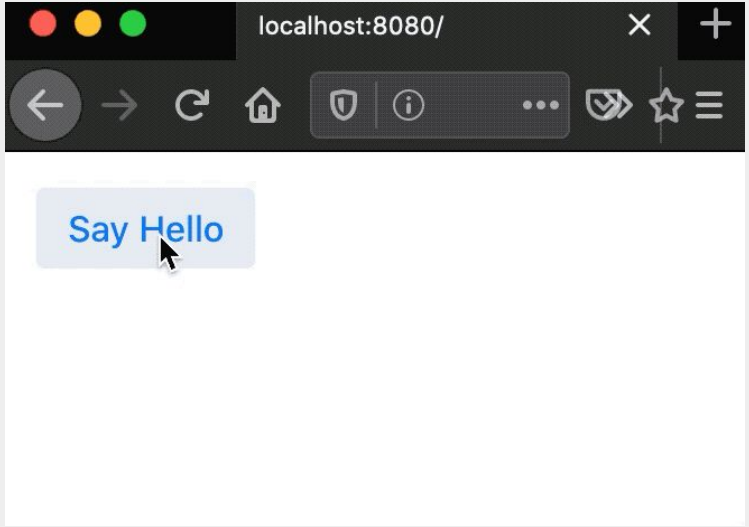
ValueChangeEvent

DragStartEvent

Event Handling

A very common use case is to add a click listener.

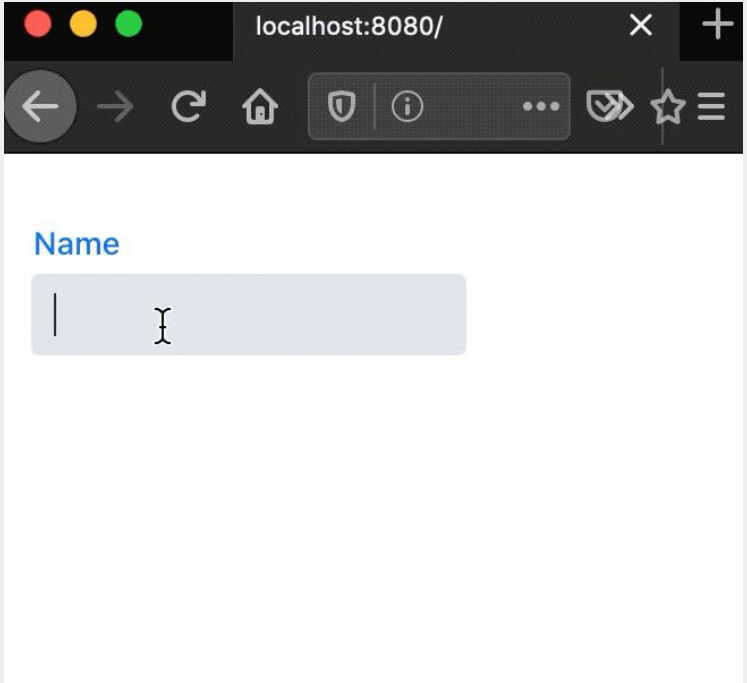
```
Button button = new Button("Say Hello");
button.addClickListener(
       e -> Notification.show("Hello!"));
```



Event Handling

Another very common use case is to listen for the value change event.

```
TextField name = new TextField("Name");
name.addValueChangeListener(e ->
   Notification.show("Hello "+ e.getValue())
);
```



Event Handling

You can also create your own Event types. Extend from ComponentEvent<T>, where T is the type of the source component which fires these events.

Fire events with `component.fireEvent(event)`

You can add listeners to your event with the `component.addListener` by specifying the event's type as the first parameter.

```
public class ConstructorCompletedEvent extends
                        ComponentEvent<MyCustomComponent> {
// ...


public class MyCustomComponent extends Component {

   public MyCustomComponent() {
      ConstructorCompletedEvent event =
             new  ConstructorCompletedEvent(this, true);
      fireEvent(event);
   }
// ...
```

Shortcuts

Shortcuts

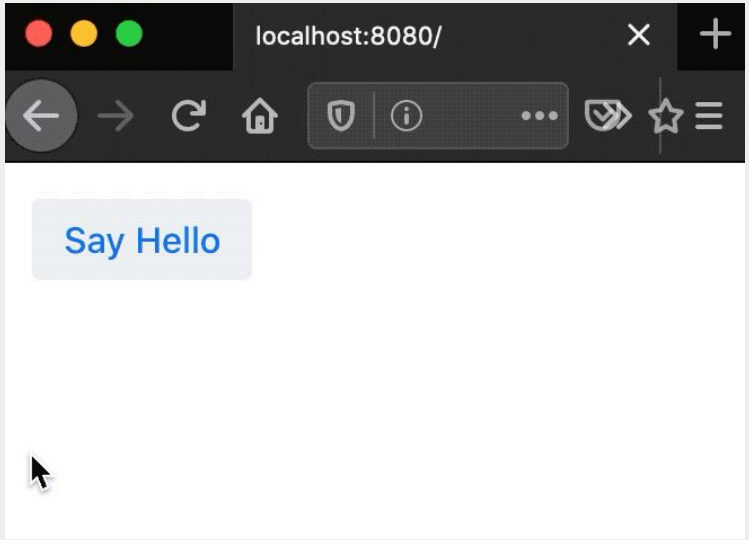
Shortcuts are keyboard key event listeners that are especially useful in applications that have complicated or long forms.

There are several varieties of shortcuts you can use. Some more specific types of shortcuts are built into components, but there are also more generic ones that are more flexible .

Click Shortcut

A common use case is to add a shortcut key for button clicking.

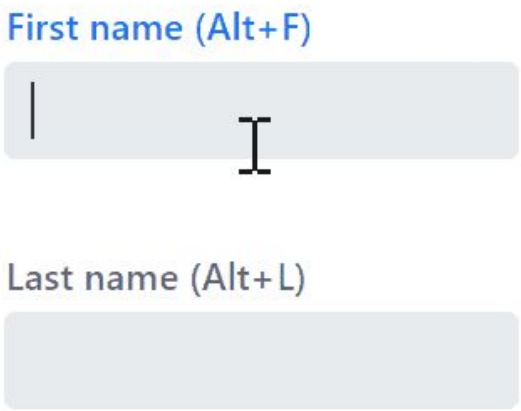
```
Button button = new Button("Say Hello");
button.addClickListener(
       e -> Notification.show("Hello!"));
button.addClickShortcut(Key.ENTER);
```



Focus Shortcut

Focus shortcuts are helpful for mouseless data input in applications.

```
TextField firstName = new TextField("First name (Alt+F)");

firstName.addFocusShortcut(Key.KEY_F, KeyModifier.ALT);

TextField lastName = new TextField("Last name (Alt+L)");

lastName.addFocusShortcut(Key.KEY_L, KeyModifier.ALT);
```



Command Shortcut

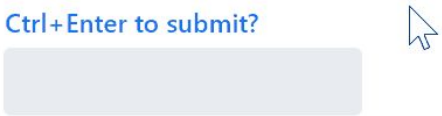
It’s also possible to register a generic command to a shortcut. You need to specify a lifecycle owner - the shortcut listener will be unregistered when the lifecycle owner is detached.

```
Shortcuts.addShortcutListener(UI.getCurrent(), // lifecycle bound to current UI

        () -> {

          // do something

        },

        Key.KEY_P, KeyModifier.CONTROL, KeyModifier.ALT);
```

Command Shortcut

When you react to a shortcut listener, keep in mind that it doesn't automatically trigger other client side events. For example, by default a TextField needs to lose focus (blur event) to update its value property.

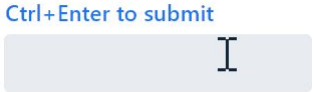
```
TextField textField = new TextField("Ctrl+Enter to submit?");

Shortcuts.addShortcutListener(UI.getCurrent(), () -> {
      Notification.show("Ctrl+Enter pressed! Value of TextField on the server: '"
           + textField.getValue() + "'");
   },
   Key.ENTER, KeyModifier.CONTROL);
```



Command Shortcut

To get around the issue, you can use TextField's `setValueChangeMode(ValueChangeMode.EAGER)` to update the value more often, or you can trigger the value change by telling the TextField to blur itself.

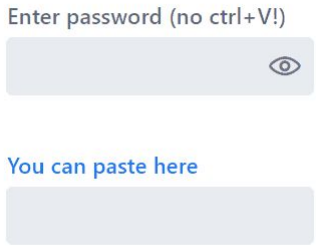
```
TextField textField = new TextField("Ctrl+Enter to submit");
Shortcuts.addShortcutListener(UI.getCurrent(), () -> {
   textField.blur(); // make sure TextField updates its value
}, Key.ENTER, KeyModifier.CONTROL);

textField.addValueChangeListener(e -> {
   Notification.show("Submitted! Value of TextField on the server: '"
           + textField.getValue() + "'");
}
```



Command Shortcut

By default, shortcuts apply to the entire app. The first parameter only specifies how long the shortcut is valid. You can limit the scope of a shortcut by specifying the component to listen on. The ShortcutRegistration also enables you to disable the default browser action and event propagation.

```
PasswordField passwordField = new PasswordField("Enter password (no ctrl+V!)");
ShortcutRegistration shortcutRegistration = Shortcuts.addShortcutListener(passwordField,
   () -> {
       Notification.show("You can't paste text with ctrl+V here! Note: pasting from
context menu is still possible");
   }, Key.KEY_V, KeyModifier.CONTROL);
// limit shortcut to passwordField only
shortcutRegistration.listenOn(passwordField);
// disable ctrl+v paste from clipboard
shortcutRegistration.setBrowserDefaultAllowed(false);

TextField textField = new TextField("You can paste here");
```



Exercise 1

vaadin}>

DOM manipulation

With Java

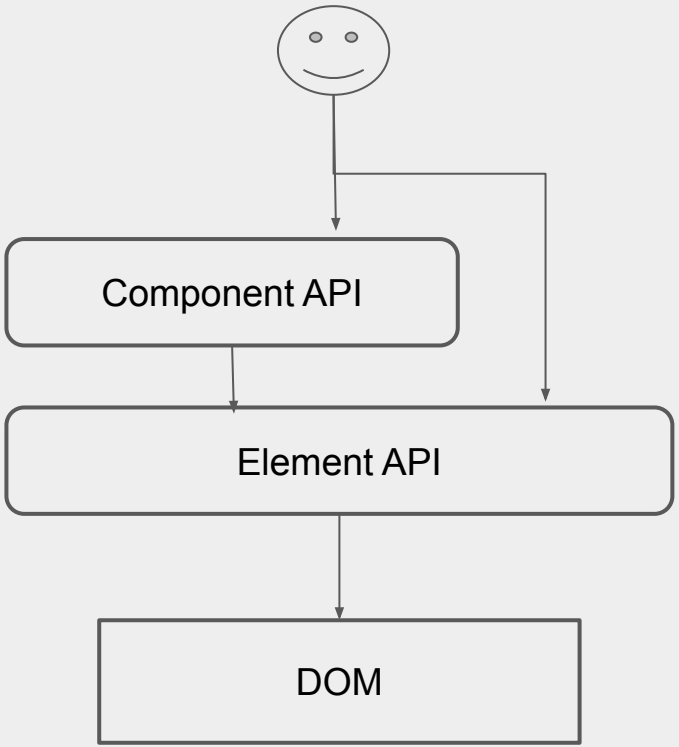
vaadin}>

Component vs Element API

The Element API provides a low-level DOM API, which is very powerful.

The Component API provides a higher abstraction level over Element API, which is more convenient to use.

The rule of thumb is to start with Component API and resort to Element API if something is missing.



Component -> Element

The Component class implements the **HasElement** interface, which allows you to access the underlying Element

```
//get the underlying element
Element element = component.getElement();
```

Attributes

An element may contain any number of “attributes”, which are mainly used for initial configuration of the element.

Attributes can be Boolean (on/off) or they can have a value. Attribute values are always stored as Strings.

Attributes

Date

Pick a date

attribute name

attribute value (always a String)

```
<vaadin-date-picker label="Date" placeholder="Pick a date">
</vaadin-date-picker>
```

Disabled input

8/14/1980

Readonly input

8/14/1980

```
<vaadin-date-picker disabled label="Disabled input" value="1980-08-14"></vaadin-date-picker>
<vaadin-date-picker readonly label="Readonly input" value="1980-08-14"></vaadin-date-picker>
```

Boolean (valueless) attribute

vaadin}>

Attributes

You can get, set, and remove attributes with the Element API

```
Element element = component.getElement();
```

Attributes

You can get, set, and remove attributes with the Element API

```
Element element = component.getElement();

element.setAttribute("placeholder", "someValue");
element.setAttribute("booleanAttribute", true);
```

Attributes

You can get, set, and remove attributes with the Element API

```
Element element = component.getElement();

element.setAttribute("placeholder", "someValue");
element.setAttribute("booleanAttribute", true);

String placeholder = element.getAttribute("placeholder"); // "someValue"
```

Attributes

You can get, set, and remove attributes with the Element API

```
Element element = component.getElement();

element.setAttribute("placeholder", "someValue");
element.setAttribute("booleanAttribute", true);

String placeholder = element.getAttribute("placeholder"); // "someValue"

element.hasAttribute("placeholder"); // true

element.getAttributeNames().toArray(); // ["placeholder", "booleanAttribute"]
```

Attributes

You can get, set, and remove attributes with the Element API

```
Element element = component.getElement();

element.setAttribute("placeholder", "someValue");
element.setAttribute("booleanAttribute", true);

String placeholder = element.getAttribute("placeholder"); // "someValue"

element.hasAttribute("placeholder"); // true

element.getAttributeNames().toArray(); // ["placeholder", "booleanAttribute"]

element.removeAttribute("placeholder");

element.getAttributeNames().toArray(); // [“booleanAttribute”]
```

Properties

An element’s properties are like attributes, but meant for dynamic uses - a common example is the **`value`** property of many fields which can change after user interaction.

Property values can be typed as **strings**, **booleans**, **floating-point numbers** or **JsonValue**s.

Properties

You can get and set properties and listen to property value changes.

```
element.setProperty("progress", "42.2");

// by default, a property is read as String -> "42.2"
String value = element.getProperty("progress");
```

Properties

You can get and set properties and listen to property value changes. The second parameter of getProperty is a “type hint” that defines both the default value and the type the value is read as.

```
element.setProperty("progress", "42.2");

// by default, a property is read as String -> "42.2"
String value = element.getProperty("progress");

// read “42.2” as a Boolean: any non-empty string is True in JavaScript -> valueBoolean is set to true
boolean valueBoolean = element.getProperty("progress", true);
```

Properties

You can get and set properties and listen to property value changes. The second parameter of getProperty is a “type hint” that defines both the default value and the type the value is read as.

```
element.setProperty("progress", "42.2");

// by default, a property is read as String -> "42.2"
String value = element.getProperty("progress");

// read “42.2” as a Boolean: any non-empty string is True in JavaScript -> valueBoolean is set to true
boolean valueBoolean = element.getProperty("progress", true);

// read “42.2” as an int: the string “42.2” is parsed to a JS number and then truncated to int -> valueInt is set to 42
int valueInt = element.getProperty("progress", 0);

// Undefined property will be read as the specified default -> otherInt is set to 0
int otherInt = element.getProperty("nonExistingProperty", 0);
```

Properties

Property values can be synchronized to the server when some specific event is fired on the client.

```
element.setProperty("progress", "42.2");

// Listen to changes to the property “progress”, synchronize to server when the “change” DOM event occurs
element.addPropertyChangeListener("progress","change",  e -> {
       Serializable value = e.getValue();
       Serializable oldValue = e.getOldValue();
       String propertyName = e.getPropertyName();
       Element source = e.getSource();
   }
```

Attribute vs Property

In many cases attributes and properties work interchangeably.

Sometimes an attribute only works for initialization and a property only works after initialization.

Styling

It’s easy to do inline styling with the help of the **getStyle()** method. It affects the **style** attribute of an element.

```
element.getStyle().set("color", "red");
element.getStyle().remove("background-color");
element.getStyle().has("cursor");
element.getStyle().get("name");
```

```
<input id="nameField" style="color: red;" placeholder="John Doe">
```

Styling

It can become tedious if you have to set many styles with Java.

```
element.getStyle().set(“prop1”, “value1”);
element.getStyle().set(“prop2”, “value2”);
element.getStyle().set(“prop3”, “value3”);
element.getStyle().set(“prop4”, “value4”);
element.getStyle().set(“prop5”, “value5”);
element.getStyle().set(“prop6”, “value6”);
element.getStyle().set(“prop7”, “value7”);
element.getStyle().set(“prop8”, “value8”);
…
```

vaadin}>

Styling

When the styles start to grow, it's usually better to create a separate CSS file for styling.

frontend/shared-styles.css

```
.red {
      color: red;
      ...
}
```

MainView.java

```
@CssImport("./shared-styles.css")
public class MainView {

}
```

vaadin}>

Styling

Use the **getClassList()** method to add/remove CSS class names. It affects the **class** attribute.

```
element.getClassList().add("red");
element.getClassList().remove("red");
```

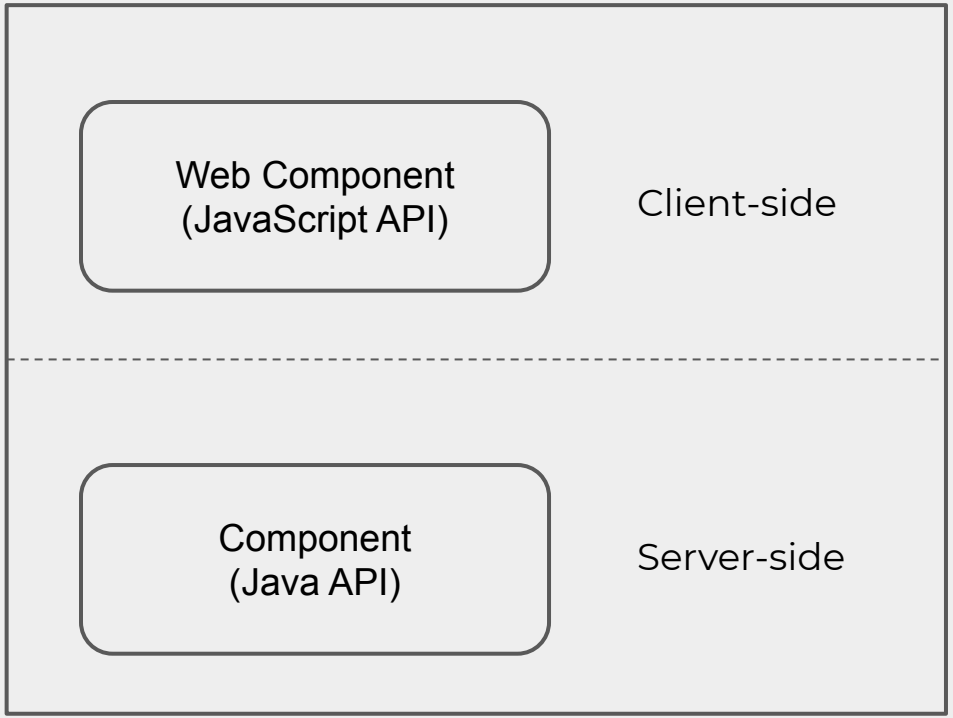
```
<input id="nameField" class="red" placeholder="John Doe">
```

Communication

Vaadin Component

A Vaadin Component has client-side JavaScript API and server-side Java API.

It might happen that there is no server-side Java API for your need but there is already a client-side JavaScript API which you could use.



Creating an Element

You can create plain HTML elements and use them in your application along with Vaadin components.

```
// With ElementFactory
Element nameField = ElementFactory.createInput();

// With the new keyword
Element nameField = new Element("input");
```

Creating an Element

You can't use the **add** method of a Layout to add a plain Element, but the Element API will help you there too - you can append elements inside containers.

```
VerticalLayout verticalLayout = new VerticalLayout();

Element nameField = new Element("input");

verticalLayout.add(nameField);// Compile error!

verticalLayout.getElement().appendChild(nameField);
```

Event handling with Element API

You can specify a server-side listener to any event happening on the client side through the Element API even when there isn't an explicit Component API listener method available.

```
TextField textField = new TextField("You can click this TextField");
textField.getElement().addEventListener("click", event -> {
   Notification.show("Textfield was clicked");
});
```

Event handling with Element API

When you're creating an event listener, you can add extra data to the event to synchronise to the server when the event fires.

```
inputElement.addEventListener("click", event -> {
   Notification.show("The offset width of clicked input: " +
           event.getEventData().getNumber("element.offsetWidth"));

}).addEventData("element.offsetWidth");
```

Event handling with Element API

You can add multiple pieces of event and element data to the listener. The event's data is returned in a JsonObject. Find default event properties from MDN: https://developer.mozilla.org/en-US/docs/Web/API/Event

```
Element buttonElement = ElementFactory.createButton();

buttonElement.addEventListener("click", this::handleClick)
       .addEventData("event.shiftKey")
       .addEventData("element.offsetWidth");


private void handleClick(DomEvent event){
   JsonObject eventData = event.getEventData();
   boolean shiftKeyPressed = eventData.getBoolean("event.shiftKey");
   double offsetWidth = eventData.getNumber("element.offsetWidth");
}
```

JavaScript calls from the server-side

When creating web component integrations, you may want to call JavaScript functions from Java. Using `Element.callJsFunction`, you can call any JavaScript function of an Element.

```
TextField textField = new TextField();
Button button = new Button("Click to scroll textField into view");

button.addClickListener(e -> {
  textField.getElement().callJsFunction("scrollIntoView");
});
```

Note: the function `scrollIntoView` is coming from the JavaScript Element class directly. See MDN for more: https://developer.mozilla.org/en-US/docs/Web/API/Element .

JavaScript calls from the server-side

You can pass additional parameters to `callJsFunction`, too.

```
element.callJsFunction("someFunction", "param1", "param2");
```

JavaScript calls from the server-side

You can execute JavaScript code from an element and access any function or value from the browser and pass them asynchronously to the server. Use **this** to refer to the current element.

```
button.getElement().executeJs("return this.innerHTML.trim();").then(String.class,
    string -> {
        Notification.show(string);
});
```

JavaScript calls from the server-side

You can also execute arbitrary JavaScript code through `Page.executeJs`. You can pass Elements as parameters, too.

```
// focus textField after 3 seconds
UI.getCurrent().getPage().executeJs("setTimeout(e => $0.focus(), 3000);",
        textField.getElement());
```

JavaScript calls to the server-side

If you want to call a method on the server from your JavaScript, annotate it with `@ClientCallable`. Then it can be accessed with `element.$server.methodName()`

```
public class MainView extends VerticalLayout {
   @ClientCallable
   private void someMethod() {
       // called after 3 second delay
   }
   public MainView() {
       Button button = new Button("Make a delayed call to a server method");
       button.addClickListener(e -> {
           UI.getCurrent().getPage().executeJs("setTimeout(e => $0.$server.someMethod(), 3000);",
                   this.getElement());
       });
       add(button);
   }
}
```

vaadin}>

Exercise 2

vaadin}>

Feedback

bit.ly/vaadin-training