Theming

Vaadin 14

vaadin}>

Agenda

- Theming with Java
- Exercise 1
- Theming with CSS
- Shadow DOM Styling
- Exercise 2
- Grid Dynamic Styling
- Exercise 3

vaadin}>

Theming with Java

@Theme

Add @Theme to the root layout to select a theme.

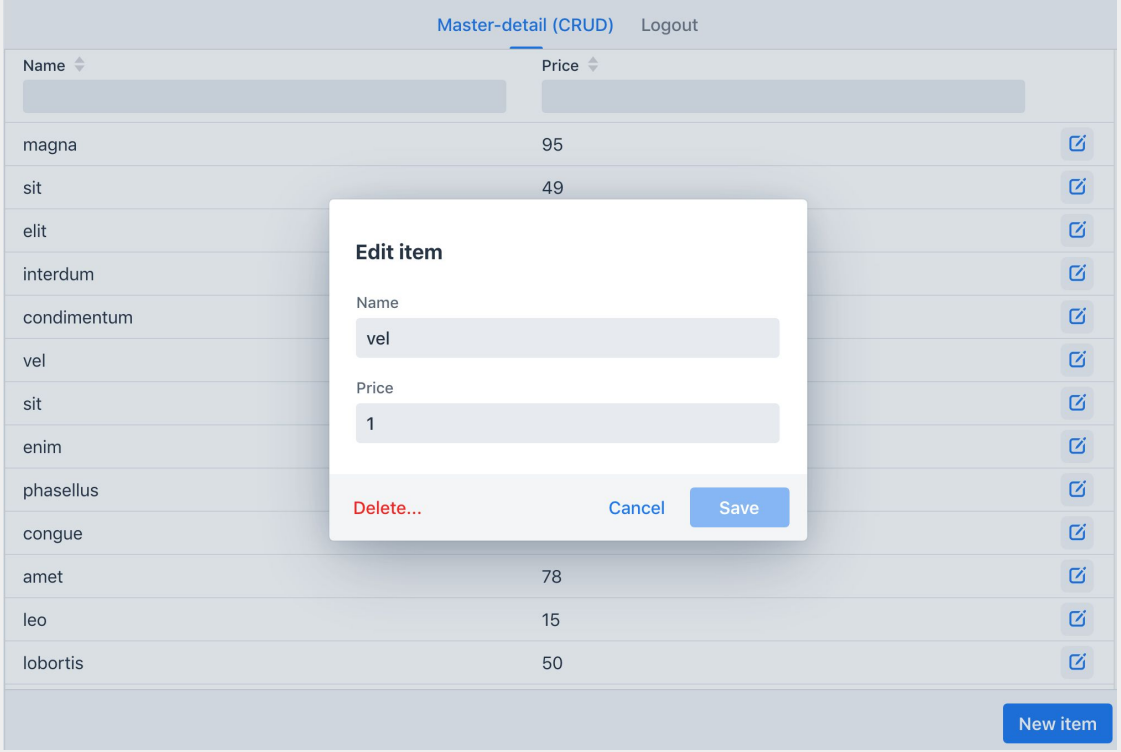
There are two predefined themes: **Lumo** and **Material**

Lumo is the **default** theme if nothing is specified

Lumo

Default theme

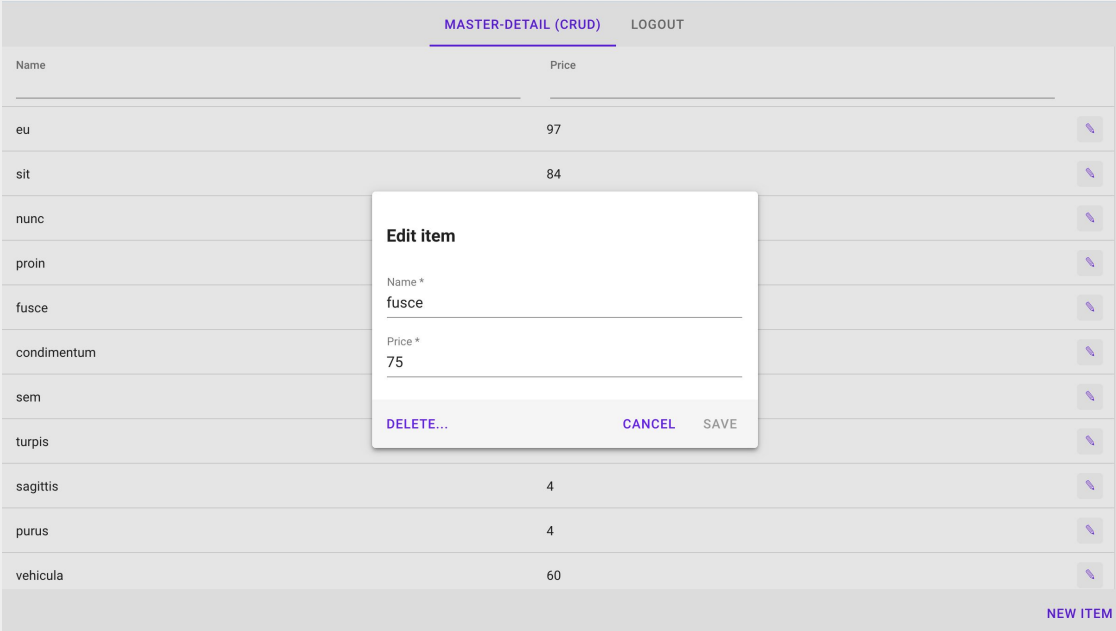
```
@Theme(Lumo.class)
public class MainLayout
```



Material

Inspired by Google's Material Design guidelines

```
@Theme(Material.class)
public class MainLayout
```

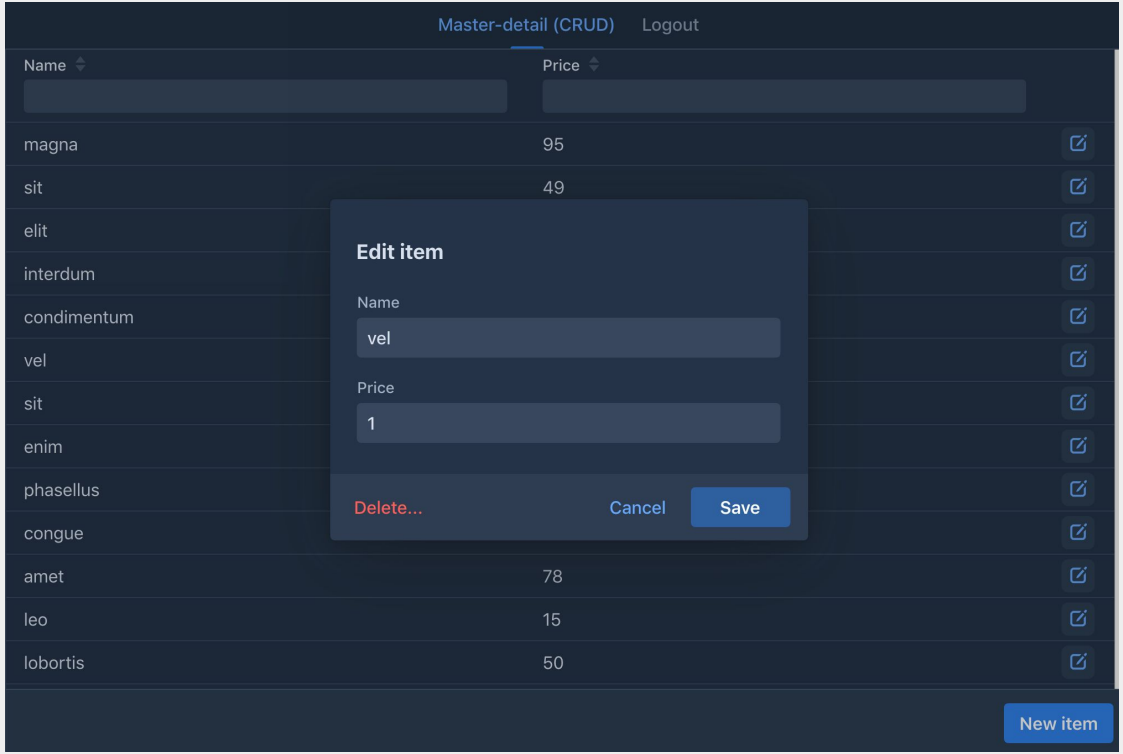


Theme Variants

Both Lumo and Material themes have **light** and **dark** variants.

The default is light.

```
@Theme(value = Lumo.class, variant = Lumo.DARK)
public class MainLayout
```



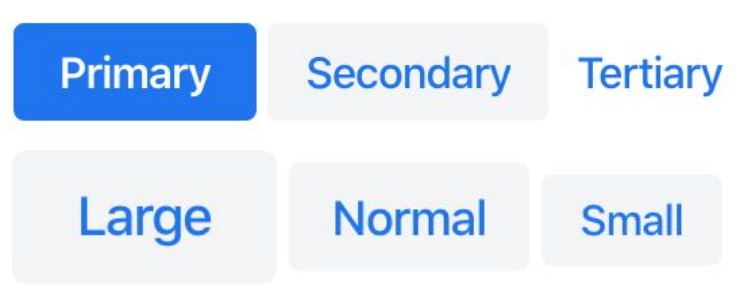
Theme Variants

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);
```



HasStyle

Most components implement the **HasStyle** interface, which allows you to:

1. Inline styling with getStyle().set(), e.g. component.getStyle().set("background-color", "gray")
2. Add CSS class name(s) with addClassName(s)("class_name");
3. Remove CSS class name(s) with removeClassName(s)("class_name");

It modifies the **style** and/or **class** attribute respectively

Element API

For components that are **NOT** implementing the **HasStyle** interface, you can still use **Element** API

1. Inline styling with `component.getElement().getStyle().set("name", "value")`
2. Add class name(s) with `component.getElement().getClassList().add("class_name")`
3. Remove class name(s) with `component.getElement().getClassList().remove("class_name")`

Exercise 1

Use Theme Variants to style some Vaadin components

Theming with CSS

Theming with CSS

Global styles: traditional CSS styles, usually for styling the application.

Local styles: CSS styles for shadow DOM, usually for styling components.

Global styles with @CssImport

CSS file

Should be located under project's **frontend** folder, e.g. frontend/styles/shared-styles.css

+

@CssImport

Annotation should be put on the top most parent layout

Example

frontend/styles/shared-styles.css

```
.main-layout {
   display: flex;
   flex-direction: column;
   width: 100%;
   height: 100%;
   min-height: 100vh;
   max-width: 960px;
   margin: 0 auto;
}
```

MainLayout.java

```
@CssImport("styles/shared-styles.css")
public class MainLayout extends Div implements RouterLayout
```

vaadin}>

Lumo Variables

Lumo defines global CSS variables that you can use to make adjustments to the styles. E.g You can use --lumo-border-radius to change the border radius of components on the page.

```
html {
    /*All the components will have 2px border radius*/
    --lumo-border-radius: 2px;
}
```

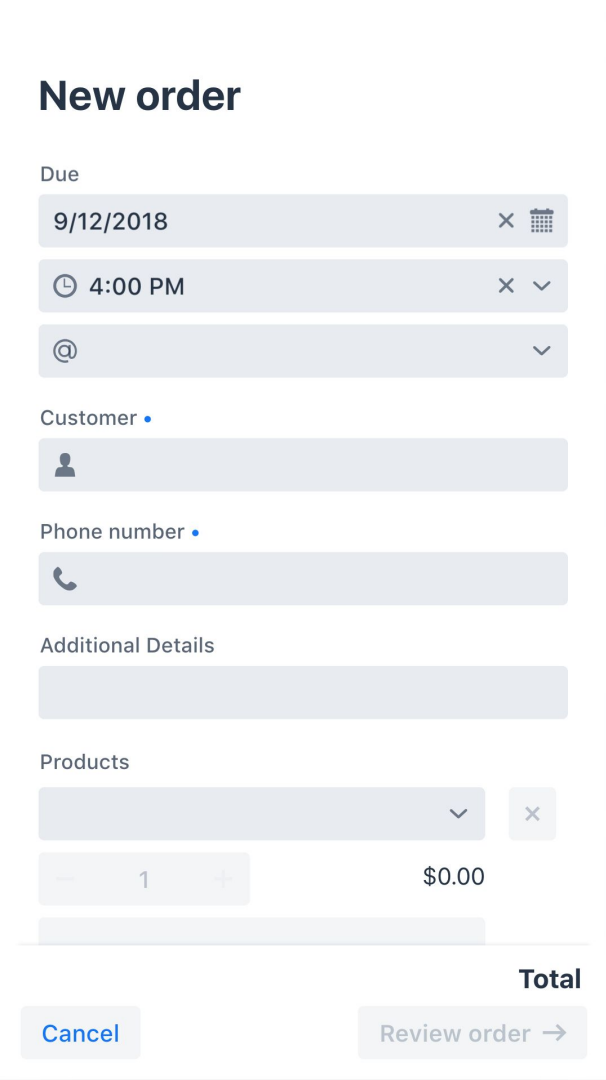
Lumo Variables

One of the really cool features of Lumo variables is that you can **scope** them, i.e. you can specify different values for some variables within some selector, and those values will override the global values within that scope.

```
html {
    /*All the components except Vaadin buttons will have 2px border radius*/
    --lumo-border-radius: 2px;
}

vaadin-button {
    /*Vaadin buttons will have 5px border radius*/
    --lumo-border-radius: 5px;
}
```

Lumo variables

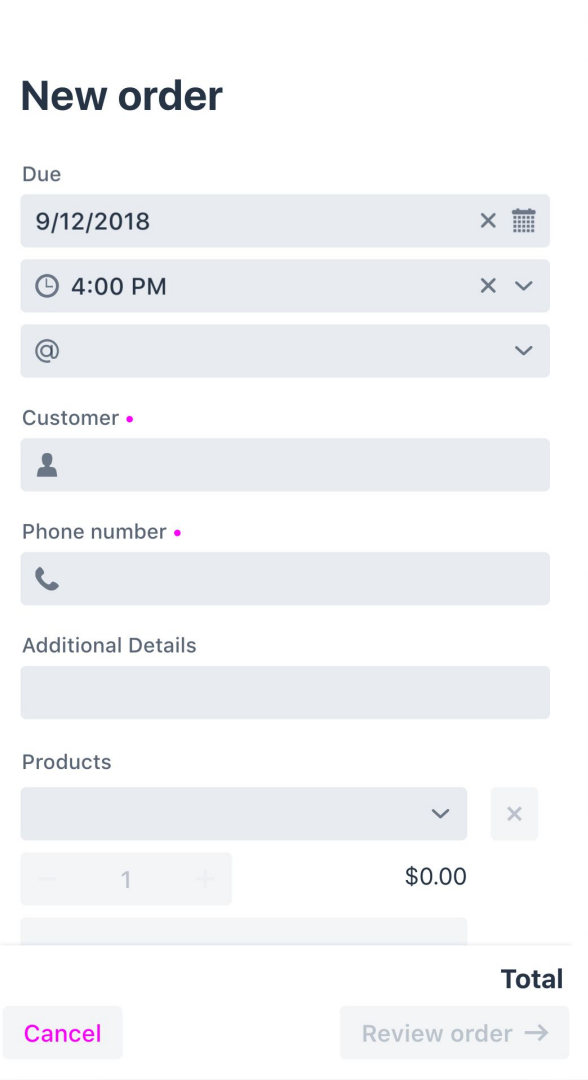
```
html {

}
```



vaadin}>

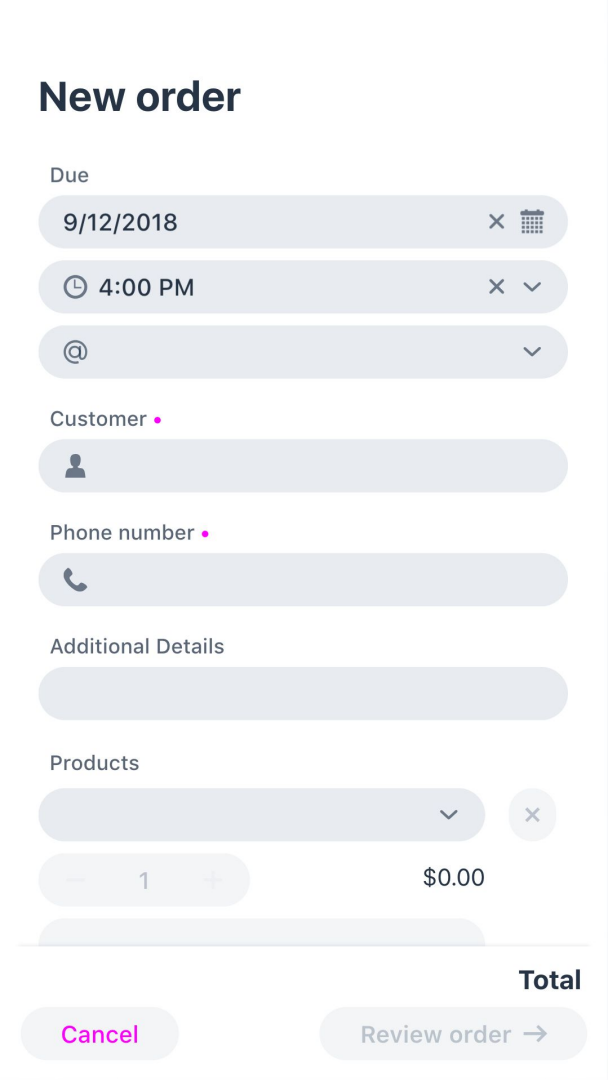
Lumo variables

```
html {
    --lumo-primary-color: magenta;
}
```



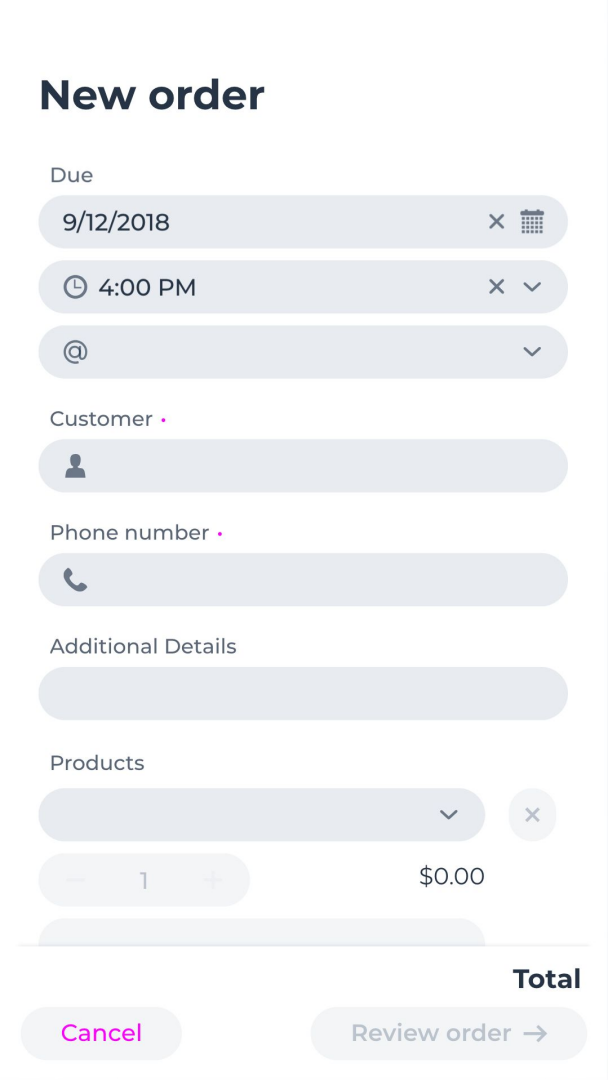
Lumo variables

```
html {
    --lumo-primary-color: magenta;
    --lumo-border-radius: 30px;
}
```



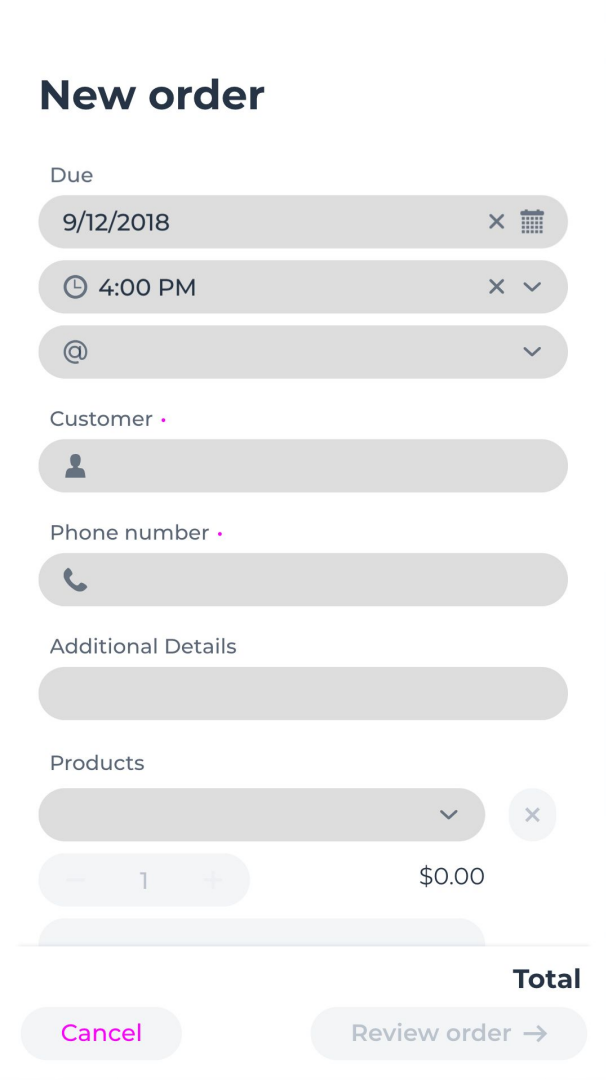
Lumo variables

```
html {
    --lumo-primary-color: magenta;
    --lumo-border-radius: 30px;
    --lumo-font-family: Montserrat;
}
```



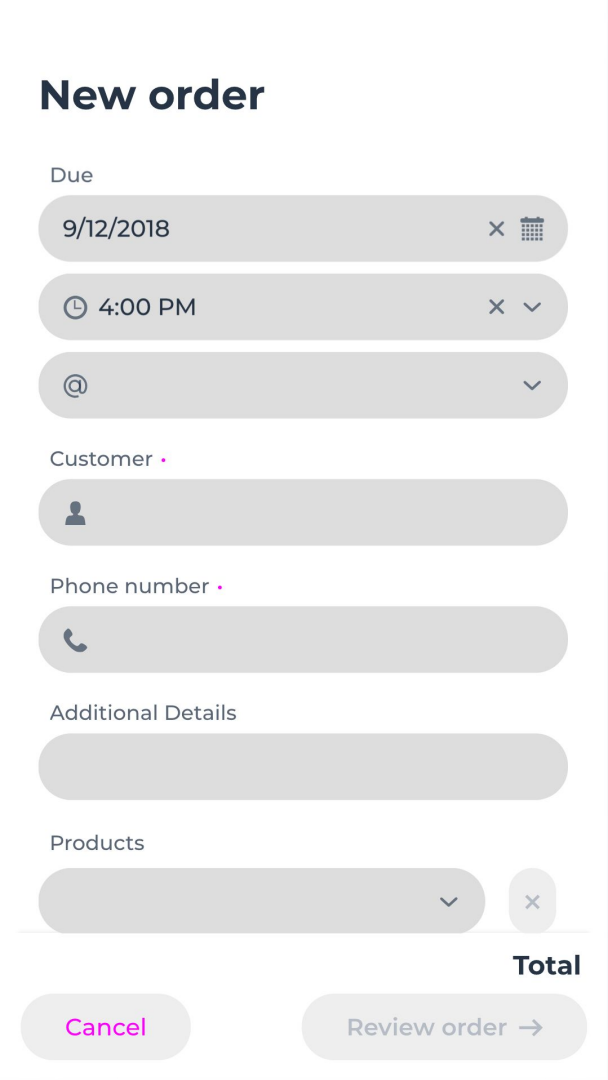
Lumo variables

```
html {
    --lumo-primary-color: magenta;
    --lumo-border-radius: 30px;
    --lumo-font-family: Montserrat;
    --lumo-contrast-10pct: #ddd;
}
```



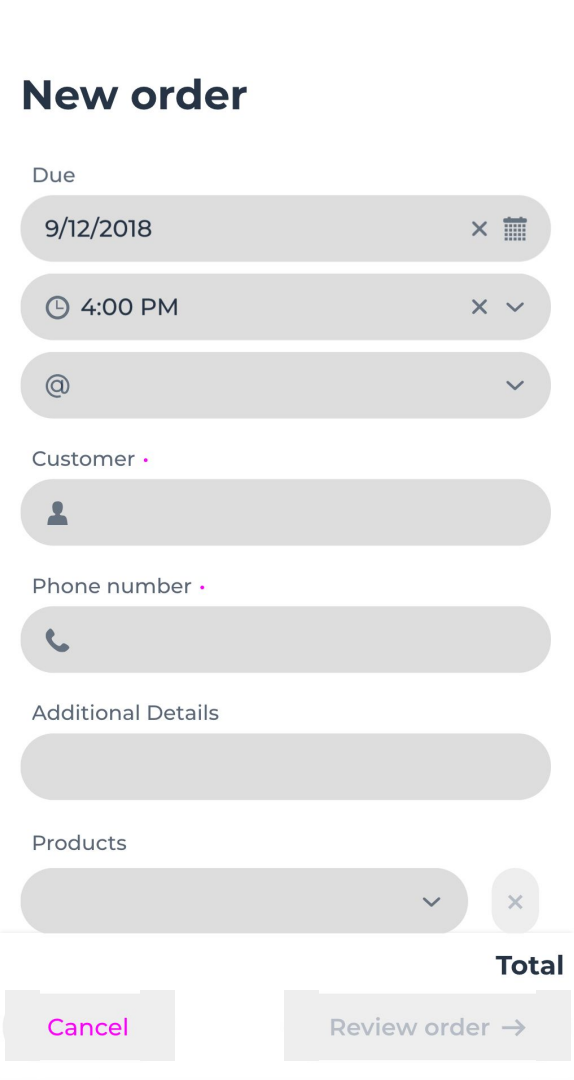
Lumo variables

```
html {
    --lumo-primary-color: magenta;
    --lumo-border-radius: 30px;
    --lumo-font-family: Montserrat;
    --lumo-contrast-10pct: #ddd;
    --lumo-size-m: 45px;
}
```



Lumo variables

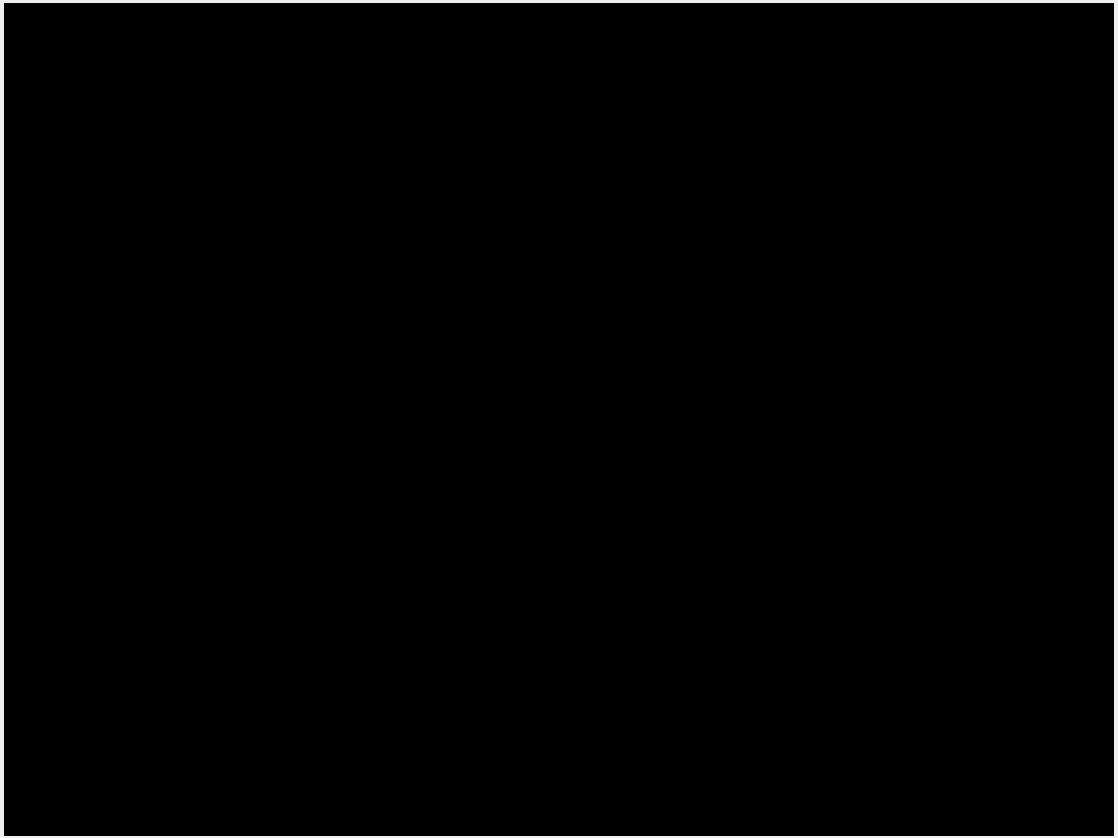
```
html {
    --lumo-primary-color: magenta;
    --lumo-border-radius: 30px;
    --lumo-font-family: Montserrat;
    --lumo-contrast-10pct: #ddd;
    --lumo-size-m: 45px;
}
vaadin-button {
  --lumo-border-radius: 0px;
}
```



Lumo Variables

Explore all the Lumo variables at vaadin.com/themes/lumo

Lumo Editor



Lumo Editor

Note: the output of Lumo editor is currently a HTML file, so it's not directly applicable for CssImport.

Check the guides how to convert the styles from HTML to CSS.

Static Resources

Quite often, you need to use some static resources, e.g. an image file for background, some font files for custom fonts.

Static Resources

For projects with **war** packaging, the resources should be located under the **src/main/webapp** directory.

```
src/main/webapp/img/logo.jpg
```

Java

```
new Image("img/logo.jpg", "The logo")
```

CSS

```
div {
   background-image: url(img/logo.jpg);
}
```

Static Resources

For projects with **jar** packaging(Spring project, Addon project), the resources should be located under the **src/main/resources/META-INF/resources** directory

```
src/main/resources/META-INF/resources/img/logo.jpg
```

Java

```
new Image("img/logo.jpg", "The logo")
```

CSS

```
div {
   background-image: url(img/logo.jpg);
}
```

vaadin}>

Global styles with @StyleSheet

It’s also possible to use @StyleSheet to import a CSS file. The usage is quite similar as @CssImport

```
@StyleSheet("./styles/my-styles.css"")
public class MainLayout
```

@CssImport vs @StyleSheet

The main differences between @CssImport and @StyleSheet are:

- CSS file location
- Webpack Bundle
- External CSS files
- Polyfill
- Relative path

CSS file location

@CssImport loads a CSS file from the **frontend** folder under the **project's root** directory.

@StyleSheet by **default** loads a CSS file from **src/main/webapp/frontend** folder.

It's good to use **context://** prefix to load a CSS file relative to the context root.

E.g., `@StyleSheet("context://styles/global.css")`, which the CSS file should be located at /src/main/webapp/styles/global.css for a war packaging project
or
/src/main/resources/META-INF/resources/styles/global.css for a jar packaging project

vaadin}>

Webpack Bundle

@CssImport files are processed by Webpack and inlined into the frontend bundle.

The content of the CSS file will be inlined, thus no separate HTTP request to the CSS file.

@StyleSheet files are loaded as-is by the browser.

There is a HTTP request for the CSS file.

vaadin}>

External URL

@CssImport is only intended for local files since the contents are inlined during the build.

@StyleSheet can be used for importing CSS files from external URL.

vaadin}>

Polyfill

@CssImport adds polyfill to prevent style leaking into shadow DOM in the browsers without native support, such as IE 11.

@StyleSheet files are loaded as-is by the browser.

No Polyfill.

vaadin}>

Relative path in url()

With @CssImport, the relative path in the url() function is relative to the document/app URL.

Given an image file located at src/main/webapp/img/login-bg.jpg

A CSS file in the project's frontend folder

```
@CssImport("styles/login-styles.css")
public class LoginView

.login-screen {
   /*relative to the context root*/
   background-image: url(img/login-bg.jpg);
}
```

With @StyleSheet, the relative path in the url() function is relative to the CSS file URL.

Given an image file located at src/main/webapp/img/login-bg.jpg

A CSS file in the src/main/webapp/styles folder

```
@StyleSheet("context://styles/login-styles.css")
public class LoginView

.login-screen {
   /*relative to the CSS file location*/
   background-image: url(../img/login-bg.jpg);
}
```

vaadin}>

@CssImport vs @StyleSheet

Normally should just use @CssImport.

Use @StyleSheet in case

- Need to separate caching for CSS files that change very rarely
- Need to load styles from an external URL, e.g. a CDN serving web fonts.
- Don’t need to support IE 11

Case study: custom font

Use a custom font is not an uncommon requirement, especially for enterprise applications.

Font files are also just static resources, referencing a font file is quite similar as referencing an image.

Custom font: Step 1

Get your font files ready.

Suppose you have your **.ttf** or **.oft** font file ready.

If not, you can download one from the internet, e.g. the Google Chilanka font

Custom font: Step 2

Generate a Web Font Kit.

Upload your .ttf or .otf file to the Webfont Generator to generate a Web Font Kit.

The Web Font Kit is a zip file that contains a .woff and a .woff2 font file, a stylesheet.css file. The CSS file contains the @font-faces. Other files are for demo purposes and can be ignored.

Custom font: Step 3

Put the font files into the right place.

Create a folder for the font under src/main/webapp directory, for Spring projects, it should be under src/main/resources/META-INF/resources. E.g. src/main/webapp/fonts/Chilanka

Put the .woff, .woff2 and stylesheet.css file under the Chilanka folder.

Custom font: Step 4

Import the stylesheet.css by adding @StyleSheet("context://fonts/Chilanka/stylesheet.css") onto your root layout class.

```
@StyleSheet("context://fonts/Chilanka/stylesheet.css")
public class MainLayout
```

Custom font: Step 4

Alternatively, could also copy the content of the stylesheet.css file to your glocal style file, e.g. shared-styles.css. But need to modify the url() in the CSS.

```
@font-face {
   ...
   src: url('chilanka-regular-webfont.woff2') format('woff2'),
        url('chilanka-regular-webfont.woff') format('woff');
}
```

↓

```
@font-face {
   ...
   src: url('fonts/Chilanka/chilanka-regular-webfont.woff2') format('woff2'),
        url('fonts/Chilanka/chilanka-regular-webfont.woff') format('woff');
}
```

vaadin}>

Custom font: Step 5

Use the font with --lumo-font-family variable in your global style file, e.g. shared-styles.css file.

```
html {
 --lumo-font-family: 'chilankaregular';
}
```

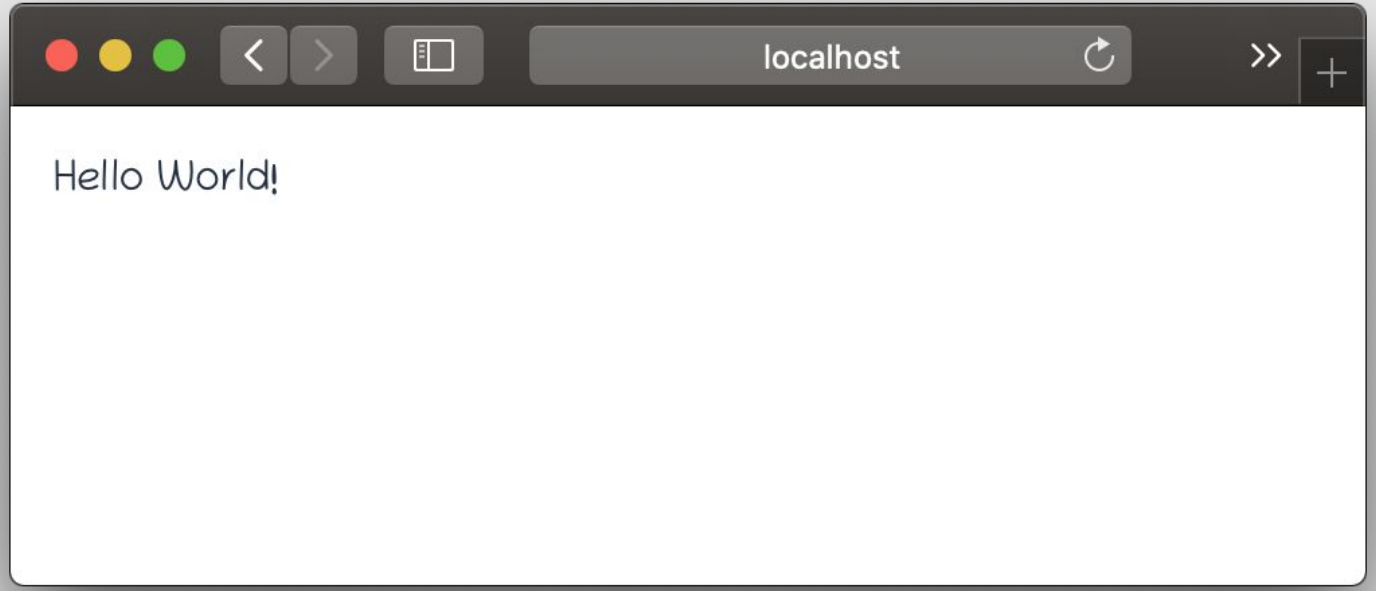
vaadin}>

Custom font: Step 5

It’s also a good practice to define at least one fallback font in the front stack, for cases where the web font fails to load.

```
html {
 --lumo-font-family: 'chilankaregular', sans-serif;
}
```

The result



Style Module

When using @CssImport, you can specify an id to make it a style module.

frontend/styles/common-styles.css

```
.my-outline-style {
   outline: 1px solid green;
}
```

MainLayout.java

```
@CssImport(value = "./styles/common-styles.css",
           id = "common-styles")
public class MainLayout extends Div
```

Style Module

Include style modules with **include** attribute, the value of the include attribute is the id of the style module to be included.

frontend/styles/specific-styles.css

```
.my-border-style {
   border: 2px solid grey;
}
```

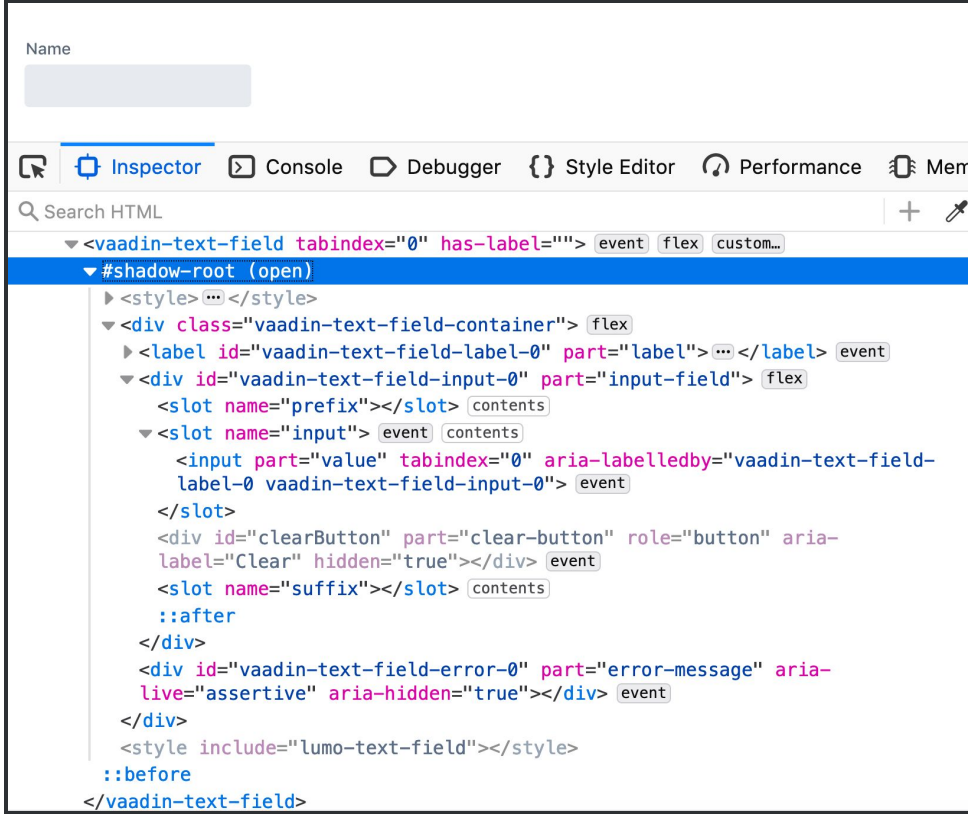
MainLayout.java

```
@CssImport(value = "./styles/my-view.css",
          include = "common-styles")
public class MyView extends Div
```

Local Styles

Internal implementation of a web component is encapsulated inside shadow DOM.

Local styles allows to style the elements inside the shadow DOM.



vaadin}>

Example

How to make the background of the text field to be yellow?

vaadin}>

Example

The relevant element is the div highlighted on the right side screenshot.

But how to write the CSS?

```
▼<vaadin-text-field tabindex="0" has-label>
  ▼#shadow-root (open)
    ▶<style>…</style>
    ▼<div class="vaadin-text-field-container">
      ▶<label part="label" id="vaadin-text-field-label-0">…
      </label>
      ▼<div part="input-field" id="vaadin-text-field-input-
      0"> == $0
        ▶<slot name="prefix">…</slot>
        ▶<slot name="input">…</slot>
         <div part="clear-button" id="clearButton" role=
         "button" aria-label="Clear" hidden="true"></div>
        ▶<slot name="suffix">…</slot>
         ::after
        </div>
       <div part="error-message" aria-live="assertive"
       aria-hidden="true" id="vaadin-text-field-error-0">
       </div>
      </div>
      <style include="lumo-text-field"></style>
      <style include="flow_css_mod_0"></style>
  ::before
</vaadin-text-field>
```

vaadin}>

Example

frontend/styles/shared-styles.css

```
#vaadin-text-field-input-0{
    background: yellow;
}
```

MainLayout.java

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

```
▼<vaadin-text-field tabindex="0" has-label>
  ▼#shadow-root (open)
    ▶<style>…</style>
    ▼<div class="vaadin-text-field-container">
      ▶<label part="label" id="vaadin-text-field-label-0">…
      </label>
      ▼<div part="input-field" id="vaadin-text-field-input-
      0"> == $0
        ▶<slot name="prefix">…</slot>
        ▶<slot name="input">…</slot>
         <div part="clear-button" id="clearButton" role=
         "button" aria-label="Clear" hidden="true"></div>
        ▶<slot name="suffix">…</slot>
         ::after
        </div>
       <div part="error-message" aria-live="assertive"
       aria-hidden="true" id="vaadin-text-field-error-0">
       </div>
      </div>
      <style include="lumo-text-field"></style>
      <style include="flow_css_mod_0"></style>
    ::before
  </vaadin-text-field>
```

vaadin}>

Example

Add a “themeFor” attribute to the @CssImpor will make it work.

frontend/styles/shared-styles.css

```
#vaadin-text-field-input-0{
   background: yellow;
}
```

MainLayout.java

```
@CssImport(value = "styles/shared-styles.css", themeFor
= "vaadin-text-field")
public class MainView
```

```
▼<vaadin-text-field tabindex="0" has-label>
  ▼#shadow-root (open)
    ▶<style>…</style>
    ▼<div class="vaadin-text-field-container">
      ▶<label part="label" id="vaadin-text-field-label-0">…
      </label>
      ▼<div part="input-field" id="vaadin-text-field-input-
      0"> == $0
        ▶<slot name="prefix">…</slot>
        ▶<slot name="input">…</slot>
         <div part="clear-button" id="clearButton" role=
         "button" aria-label="Clear" hidden="true"></div>
        ▶<slot name="suffix">…</slot>
         ::after
        </div>
       <div part="error-message" aria-live="assertive"
       aria-hidden="true" id="vaadin-text-field-error-0">
       </div>
      </div>
      <style include="lumo-text-field"></style>
      <style include="flow_css_mod_0"></style>
   ::before
</vaadin-text-field>
```

vaadin}>

Theme module

How it works is that with the **themeFor** attribute, Vaadin will generate a **<dom-module>**, which allows styling the inner elements of a shadow DOM. Note that an id value is created if necessary when using the themeFor attribute. You could also specify the id value manually with the id attribute.

```
<dom-module id="flow_css_mod_x" theme-for="vaadin-text-field">
   <template>
       <style>
          #vaadin-text-field-input-0 {
            background-color: yellow;
          }
       </style>
   </template>
</dom-module>
```

Style module

To improve the previous example:

1. Each style module should be in a separate CSS file.
2. shared-styles.css is usually for global styles, thus yellow-bg-text-field.css would be a better name.
3. Use **part attribute selector** rather than id since the part attribute is considered the ONLY public styling API. CSS classes, id attributes, and even the elements are considered implementation details and they can change.

```
▼<vaadin-text-field tabindex="0" has-label>
  ▼#shadow-root (open)
    ▶<style>…</style>
    ▼<div class="vaadin-text-field-container">
      ▶<label part="label" id="vaadin-text-field-label-0">…
      </label>
      ▼<div part="input-field" id="vaadin-text-field-input-
      0"> == $0
        ▶<slot name="prefix">…</slot>
        ▶<slot name="input">…</slot>
         <div part="clear-button" id="clearButton" role=
         "button" aria-label="Clear" hidden="true"></div>
        ▶<slot name="suffix">…</slot>
         ::after
        </div>
       <div part="error-message" aria-live="assertive"
       aria-hidden="true" id="vaadin-text-field-error-0">
       </div>
      </div>
      <style include="lumo-text-field"></style>
      <style include="flow_css_mod_0"></style>
    ::before
  </vaadin-text-field>
```

vaadin}>

Example

frontend/styles/yellow-bg-text-field.css

```
[part="input-field"] {
   background: yellow;
}
```

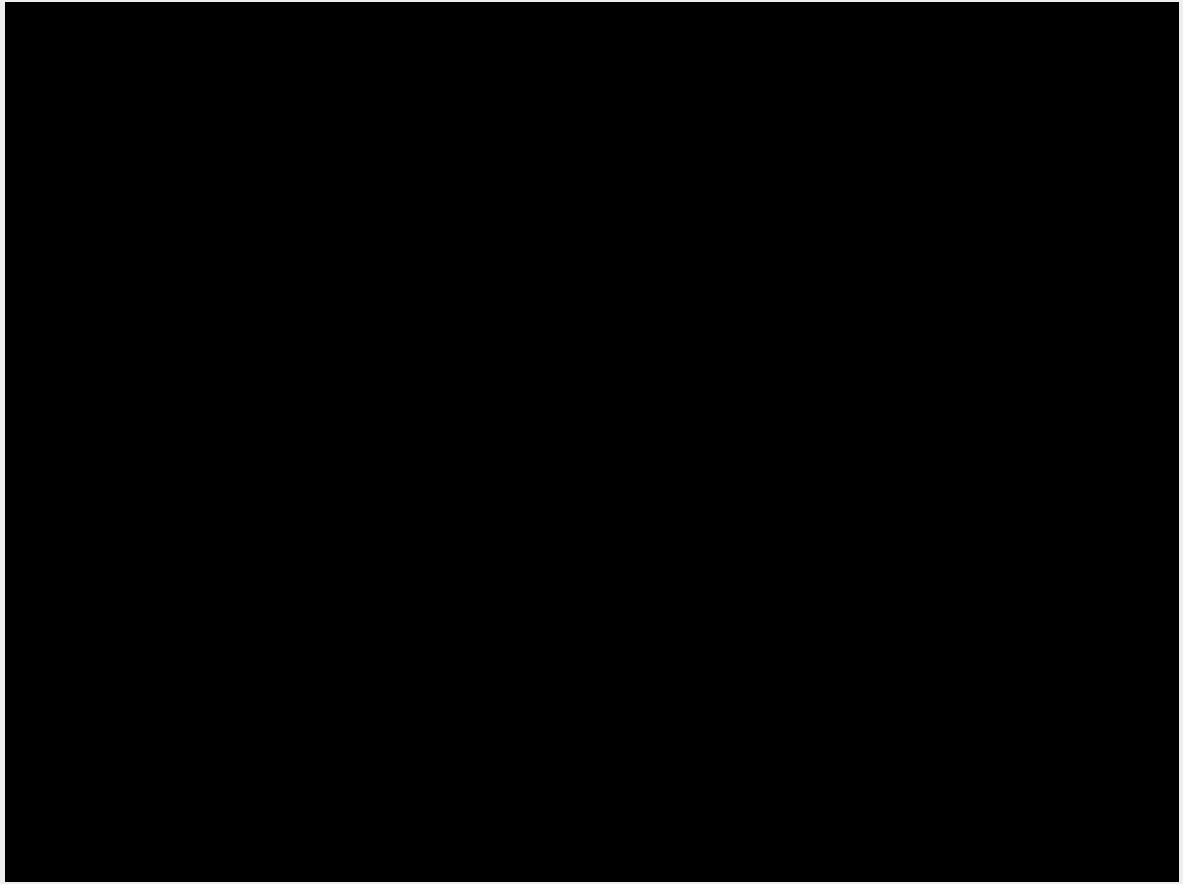
MainLayout.java

```
@CssImport(value = "styles/yellow-bg-text-field.css",
themeFor = "vaadin-text-field")
public class MainView
```

```
▼<vaadin-text-field tabindex="0" has-label>
  ▼#shadow-root (open)
    ▶<style>…</style>
    ▼<div class="vaadin-text-field-container">
      ▶<label part="label" id="vaadin-text-field-label-0">…
      </label>
      ▼<div part="input-field" id="vaadin-text-field-input-
      0"> == $0
        ▶<slot name="prefix">…</slot>
        ▶<slot name="input">…</slot>
         <div part="clear-button" id="clearButton" role=
         "button" aria-label="Clear" hidden="true"></div>
        ▶<slot name="suffix">…</slot>
         ::after
        </div>
       <div part="error-message" aria-live="assertive"
       aria-hidden="true" id="vaadin-text-field-error-0">
       </div>
      </div>
      <style include="lumo-text-field"></style>
      <style include="flow_css_mod_0"></style>
    ::before
  </vaadin-text-field>
```

vaadin}>

Find all the styleable parts of a component



Special selectors for shadow DOM

:host selector for targeting the root element

```
/* Selects a shadow root host */
:host {
    …
}

/* Selects a shadow root host, only if it is matched by the selector argument */
:host(.some-class-name) {
    …
}
```

Example

```
/* All the <vaadin-text-field> will be yellow */
[part="input-field"] {
   background: yellow;
}


/* Only the <vaadin-text-field> with a custom class
will be yellow*/
:host(.custom) [part="input-field"] {
   background: yellow;
}
```

```
▼<vaadin-text-field tabindex="0" has-label>
  ▼#shadow-root (open)
    ▶<style>…</style>
    ▼<div class="vaadin-text-field-container">
      ▶<label part="label" id="vaadin-text-field-label-0">…
      </label>
      ▼<div part="input-field" id="vaadin-text-field-input-
      0"> == $0
        ▶<slot name="prefix">…</slot>
        ▶<slot name="input">…</slot>
         <div part="clear-button" id="clearButton" role=
         "button" aria-label="Clear" hidden="true"></div>
        ▶<slot name="suffix">…</slot>
         ::after
        </div>
       <div part="error-message" aria-live="assertive"
       aria-hidden="true" id="vaadin-text-field-error-0">
       </div>
      </div>
      <style include="lumo-text-field"></style>
      <style include="flow_css_mod_0"></style>
  ::before
</vaadin-text-field>
```

vaadin}>

Special selectors for shadow DOM

::slotted selector for targeting the slotted children

```
/* Selects any element placed inside a slot */
::slotted(*) {
   font-weight: bold;
}


/* Selects any <span> placed inside a slot */
::slotted(span) {
   font-weight: bold;
}
```

vaadin}>

Style module limitation

It only works with web components that implement Vaadin Themable mixin, mostly just Vaadin components.

Other web components mostly use **CSS Variables**.

CSS Variables

- It allows defining variables for CSS
- A variable name has **--** prefix
- Variables are evaluated with the var() function

```
/* An element using CSS variables for styling */
element {
   background-color: var(--bg-color);
}

-------------------------------------------------

/* Define the value for the variable */
html {
   --bg-color: brown;
}
```

vaadin}>

CSS Variables

Lumo variables are CSS variables. So could also use some Lumo variables when defining your own CSS.

```
div {
   background-color: var(--lumo-contrast-10pct);
}
```

vaadin}>

Example

Check how to style the Polymer <paper-toggle-button> element.

```
<paper-toggle-button></paper-toggle-button>
```

vaadin}>

Styling API

Styling

The following custom properties and mixins are available for styling:

Custom property	Description	Default
`--paper-toggle-button-unchecked-bar-color`	Slider color when the input is not checked	`#000000`
`--paper-toggle-button-unchecked-button-color`	Button color when the input is not checked	`--paper-grey-50`
`--paper-toggle-button-unchecked-ink-color`	Selected/focus ripple color when the input is not checked	`--dark-primary-color`
`--paper-toggle-button-checked-bar-color`	Slider button color when the input is checked	`--primary-color`
`--paper-toggle-button-checked-button-color`	Button color when the input is checked	`--primary-color`
`--paper-toggle-button-checked-ink-color`	Selected/focus ripple color when the input is checked	`--primary-color`
`--paper-toggle-button-invalid-bar-color`	Slider button color when the input is invalid	`--error-color`
`--paper-toggle-button-invalid-button-color`	Button color when the input is invalid	`--error-color`
`--paper-toggle-button-invalid-ink-color`	Selected/focus ripple color when the input is invalid	`--error-color`

Example

Tweak a few css variables and now the toggle button becomes green.

```
--paper-toggle-button-checked-bar-color :
var(--paper-green-500);
--paper-toggle-button-checked-button-color :
var(--paper-green-500);
--paper-toggle-button-checked-ink-color:
var(--paper-green-500);
--paper-toggle-button-unchecked-bar-color :
var(--paper-teal-500);
--paper-toggle-button-unchecked-button-color :
var(--paper-teal-500);
--paper-toggle-button-unchecked-ink-color:
var(--paper-teal-500);
```

vaadin}>

Combination

Theme modules and CSS variables are **not** exclusive.

Combine them can make compelling use cases, e.g., expose new variables.

Example

In the previous TextField example, what if the background color should be determined at runtime, e.g., according to some value in the DB?

vaadin}>

Example

frontend/styles/custom-bg-text-field.css

```
[part="input-field"] {
   background: var(--bg-color);
}
```

MainLayout.java

```
@CssImport(value = "styles/custom-bg-text-field.css",
themeFor = "vaadin-text-field")
public class MainView extends Div {
   public MainView() {
       TextField name = new TextField("Name");
       //Change only this TextField
       name.getStyle().set("--bg-color",
                   service.getColor());
       add(name);
   }
}
```

vaadin}>

Example

frontend/styles/custom-bg-text-field.css

```
[part="input-field"] {
   background: var(--bg-color);
}
```

MainLayout.java

```
@CssImport(value = "styles/custom-bg-text-field.css",
themeFor = "vaadin-text-field")
public class MainView extends Div {
   public MainView() {
        //Change all the TextFields
        //by defining the variable to the HTML <body> element
        UI.getCurrent().getElement().getStyle().set(
           "--bg-color", service.getColor());
   }
}
```

vaadin}>

Theme attribute

You might have noticed that there is a **theme** attribute for Vaadin components.

One big difference between theme attribute and the traditional class name is that the **theme attribute** of the root element will **propagate to the shadow DOM**.

Theme attribute

Some components implement the **HasTheme** interface, which allows you to modify the theme attribute by:

1. Adding theme name(s) with addThemeName(s)(“theme_name”)
2. Setting theme name(s) with setThemeName(s)(“theme_name”)
3. Removing theme name(s) with removeThemeName(s)(“theme_name”)
4. Getting all the theme names with getThemeNames()

Theme attribute

For components that **NOT** implementing the **HasTheme** interface, you can still use **Element API**:

1. Add theme name(s) with `component.getElememt().getThemeList().add("theme_name")`
2. Remove theme name(s) with `component.getElememt().getThemeList().remove("theme_name")`
3. Get all the theme names with `component.getElememt().getThemeList()`
4. Or manipulate on the theme attribute directly, `component.getElememt().getAttribute("theme")`, `component.getElememt().setAttribute("theme", "value")`

Example

```
ComboBox<String> comboBox = new ComboBox<>();
comboBox.getElement().getThemeList().add("custom");
```

After setting the theme attribute to the <vaadin-combo-box>, the <vaadin-text-field> in the shadow DOM also got the same attribute.

```
▼<vaadin-combo-box theme="custom" tabindex="0">
  ▼#shadow-root (open)
    ▶<style>…</style>
    ▶<vaadin-text-field part="text-field" id="input"
    autocomplete="off" autocapitalize="none" theme="custom"
    tabindex="0">…</vaadin-text-field>
    ▶<vaadin-combo-box-dropdown-wrapper id="overlay">…
    </vaadin-combo-box-dropdown-wrapper>
      <style include="lumo-combo-box"></style>
  </vaadin-combo-box>
```

vaadin}>

Example

How to make a yellow background combobox?

Make the text field inside the combobox have a yellow background.

vaadin}>

Example

frontend/styles/yellow-bg-text-field.css

```
:host([theme~="custom"]) [part="input-field"] {
   background: yellow;
}
```

MainLayout.java

```
@CssImport(value = "styles/yellow-bg-text-field.css",
themeFor = "vaadin-text-field")
public class MainView extends Div {

   public MainView() {
       ComboBox<String> comboBox = new ComboBox<>();
       comboBox.getElement().getThemeList().add("custom");
       add(comboBox);
   }
}
```

vaadin}>

Exercise 2

Style the internal elements of a Combo Box

Grid dynamic styling

Style a Grid cell/column based on its content

Month	Expenses
January	451
February	544
March	369
April	747
May	744
June	482
July	387
August	660
September	436
October	422
November	615
December	396

setClassNameGenerator()

To style the whole **row**, call setClassNameGenerator on the **Grid**, which applies the class name on all the cells in the row.

```
grid.setClassNameGenerator(this::getClassName);

private String getClassName(MonthlyExpense expense){
   if(expense.getSpending() > LIMIT){
       return "warn";
   }else{
       return null;
   }
}
```

vaadin}>

setClassNameGenerator()

To style the a **cell**, call setClassNameGenerator on a **Column**.

```
grid.getColumnByKey("mycolumn").setClassNameGenerator(this::getClassName);

private String getClassName(MonthlyExpense expense){
   if(expense.getSpending() > LIMIT){
       return "warn";
   }else{
       return null;
   }
}
```

vaadin}>

setClassNameGenerator()

Then add a style module for Grid

frontend/styles/my-grid.css

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

MainLayout.java

```
@CssImport(value = "styles/my-grid.css", themeFor = "vaadin-grid")
public class MainLayout
```

Exercise 3

Style a Grid cell/column based on its content

Summary

- Theming with Java
- Theming with CSS
- Shadow DOM styling
- Grid dynamic styling

vaadin}>

Feedback

bit.ly/vaadin-training