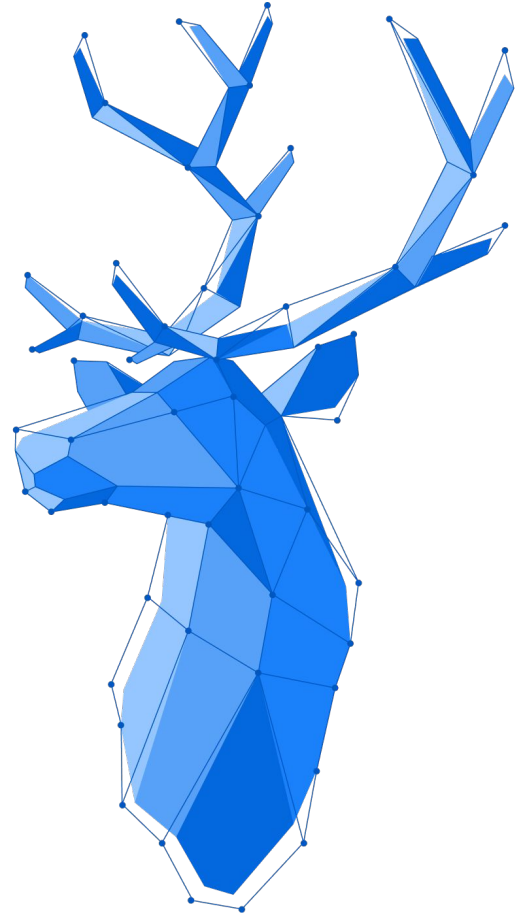


vaadin}>

Theming and Styling Applications

Vaadin Flow



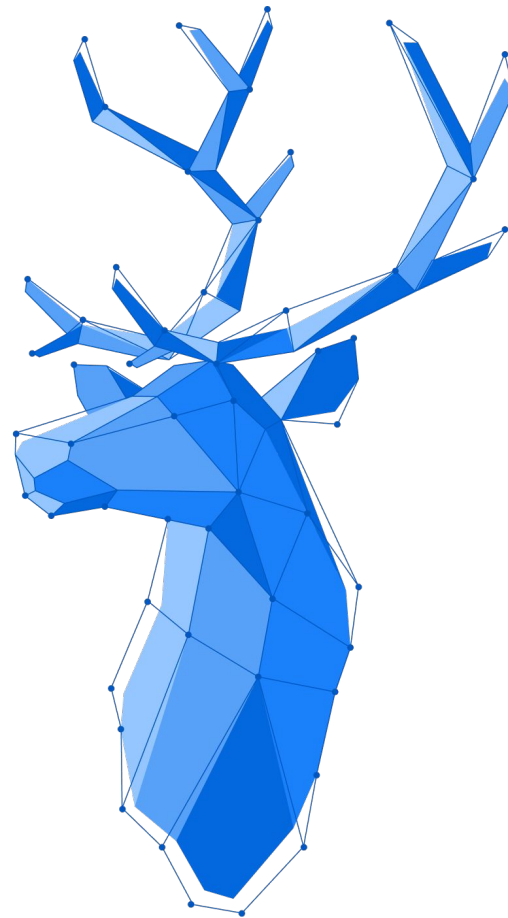
Vaadin training set

Vaadin Foundation

- Introduction
- Layouting
- Creating Forms
- Data Lists with Grid
- Routing and Navigation
- **Theming and Styling Applications**

Agenda

- Part 1
 - Styling Vaadin Components
 - Exercise 1
- Part 2
 - More advanced styling options
 - Styling Other UI Elements
 - Exercise 2
- Part 3
 - Application Theme
- Part 4
 - Grid dynamic styling
 - Exercise 3
- Part 5
 - Case study: custom font



Theming and Styling Applications, Part 1

Styling Vaadin Components

Styling Vaadin Components

There are several ways how to change style of Vaadin components.

When styling the Vaadin Components, we recommend progressing from the simplest techniques to more advanced ones.

I.e. try to use Lumo CSS Variables custom values as far as possible and only after their power is exhausted, then try to style the components using parts and states in Shadow DOM.

Styling Vaadin Components

The order of the styling techniques to try should be the following:

1. Theme variants
2. Lumo CSS Variables customization
3. Styling using component parts and states selectors
4. Applying styles to elements using `LumoUtility` predefined classes
5. Styling using custom CSS classes and variables (`addClassName(s)()` method)
6. Styling using directly setting style on an element (`getStyle().set()` method)

Styling Vaadin Components

We also recommend carefully studying all available Vaadin components - how they look and behave - prior to any styling customization attempts to prevent forcefully trying to style a component that might already exist as a different component elsewhere.

Note: Vaadin provides an official [Figma library](#), so pixel-perfect UI designs and prototypes can be easily created and shared with developers. The components in the libraries use the default Lumo styling, and you can adapt them to suit your branding and vision.

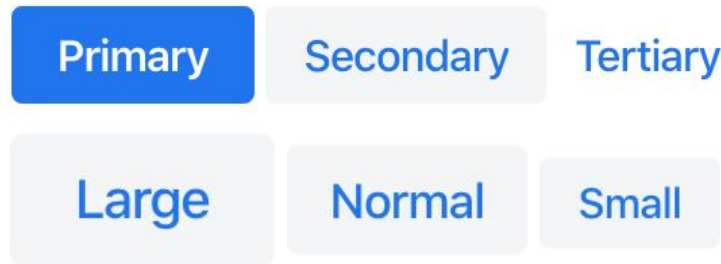
Theme Variants

Many Vaadin components have predefined variants, which allow to change a component's look and feel quickly with

```
component.addThemeVariants()
```

E.g.

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



Theme Variants

Theme variants can affect colors, sizes, alignment, and other visual properties of the components.

Theme variants are described in the [Vaadin Component Documentation](#) for each component.

Variants are also available as Java enumerations:

`ButtonVariant`, `AvatarVariant`, `TextFieldVariant` etc.

CSS variables

CSS variables allow you to define custom properties and their values, which you can later reference in other style sheets or JavaScript. They reduce copy-paste, ensure consistency, and make it easier to read and understand style sheets.

Traditional code

```
.container {  
  background-color: #ffffff;  
  color: #1e90ff;  
}  
  
button {  
  background-color: #ffffff;  
  color: #1e90ff;  
  border: 1px solid #1e90ff;  
}
```

CSS variables

```
html {  
  --app-background-color: #ffffff;  
  --app-primary-color: #1e90ff;  
}  
  
.container {  
  background-color: var(--app-background-color);  
  color: var(--app-primary-color);  
}  
  
button {  
  background-color: var(--app-background-color);  
  color: var(--app-primary-color);  
  border: 1px solid var(--app-primary-color);  
}
```

Lumo Variables

Lumo, the default Vaadin Theme, is based on CSS style properties.

Lumo defines global CSS variables that you can use to adjust 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;  
}
```

Customizing Style Properties

Style properties are recommended as the primary approach for both Vaadin component style customization and custom CSS. They make it easier to achieve a consistent look and feel across the application.



--lumo-primary-color

Aa

--lumo-font-family



--lumo-border-radius-l



Checkbox

Style property scopes

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. You can work with the following scopes:

- Global
- Component type specific
- Component instance

Style property scopes

Global style properties

A global style property affects the entire application. It's applied to the html selector.

```
/* Global style values */  
html {  
  --lumo-primary-color: green;  
  --lumo-font-family: "Roboto";  
}
```

Style property scopes

Component type properties

Component type-specific style properties affect every component of the given type in the application. It's applied to the component's root element selector

```
/* Scoped to a type of component */  
vaadin-button {  
  --lumo-font-family: "Courier";  
}
```

Style property scopes

Component instance properties

Component instance properties are specific to one or more component instances – to which a particular CSS class name has been applied (as shown below)

Java

```
Button specialButton = new Button("I'm special");  
specialButton.addClassName("special");
```

CSS

```
/* Scoped to instances with a particular CSS class name */  
vaadin-button.special {  
  --lumo-primary-color: cyan;  
}
```


Example customization

Lumo Variables

```
html {  
  
}
```

New order

Due

9/12/2018



🕒 4:00 PM



@



Customer •



Phone number •



Additional Details

Products



1



\$0.00

Total

Cancel

Review order →

Example customization

Lumo Variables

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

New order

Due

9/12/2018



4:00 PM



@



Customer •



Phone number •



Additional Details

Products



1



\$0.00

Total

Cancel

Review order →

Example customization

Lumo Variables

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

New order

Due

9/12/2018



🕒 4:00 PM



@



Customer •



Phone number •



Additional Details

Products



1



\$0.00

Total

Cancel

Review order →

Example customization

Lumo Variables

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

New order

Due

9/12/2018



🕒 4:00 PM



@



Customer •



Phone number •



Additional Details

Products



1



\$0.00

Total

Cancel

Review order →

Example customization

Lumo Variables

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

New order

Due

9/12/2018



🕒 4:00 PM



@



Customer •



Phone number •



Additional Details

Products



1



\$0.00

Total

Cancel

Review order →

Example customization

Lumo Variables

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

New order

Due

9/12/2018



🕒 4:00 PM



@



Customer •



Phone number •



Additional Details

Products



Total

Cancel

Review order →

Example customization

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;  
}  
  
.mybutton {  
  --lumo-border-radius: 5px;  
}
```

New order

Due

9/12/2018



🕒 4:00 PM



@



Customer *



Phone number *



Additional Details

Products



Total

Cancel

Review order →

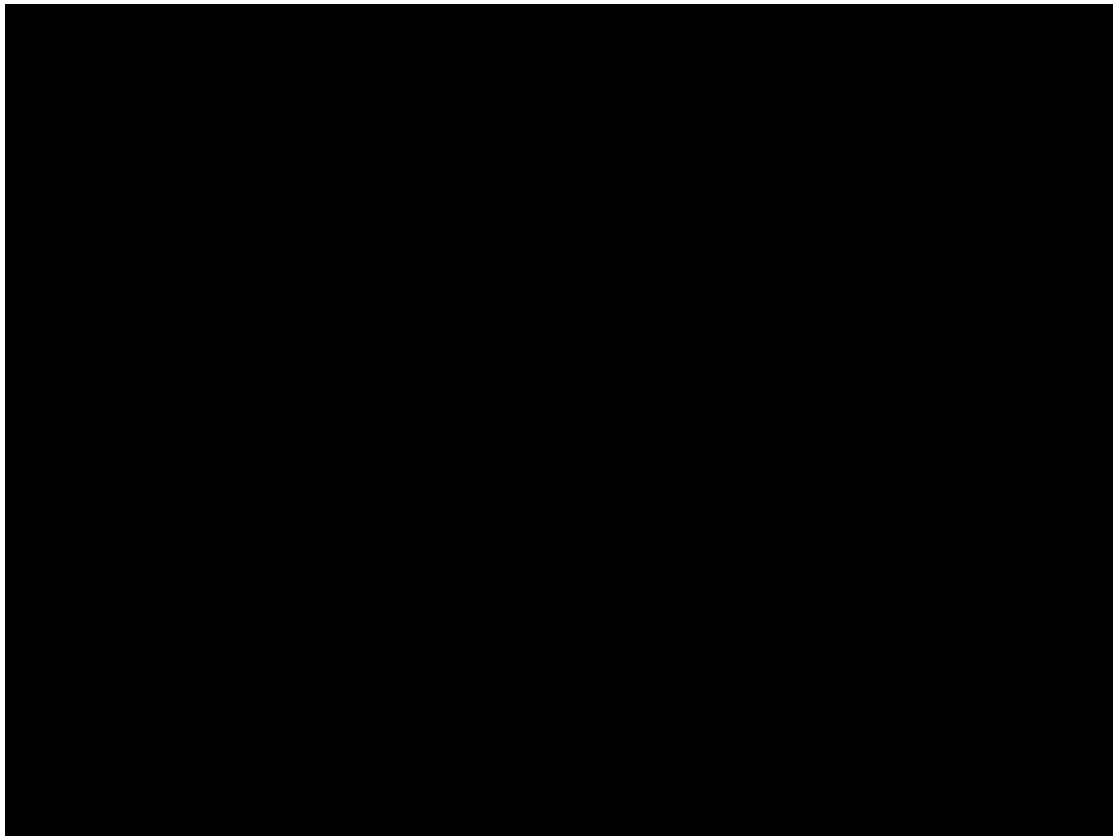
Lumo Variables

Explore all the Lumo variables at

<https://vaadin.com/docs/latest/styling/lumo/lumo-style-properties>

Lumo Editor

<https://demo.vaadin.com/lumo-editor/>



Lumo Editor

Note: the output of the Lumo editor is an HTML file, so it's not directly applicable in the newest versions of Vaadin.

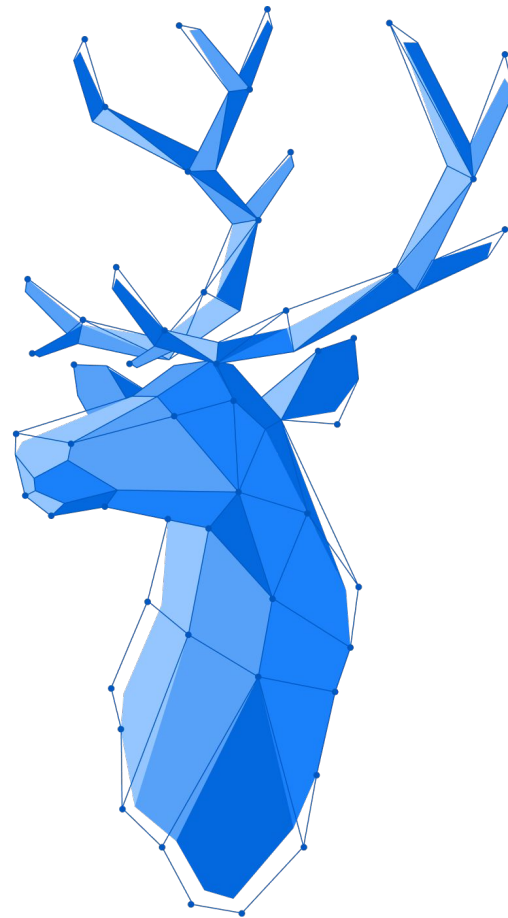
The contents inside the `<style>` block are directly usable in a CSS file, however. Just copy the content of the `<style>` block to your master style sheet.

Exercise 1

Use Theme Variants to style some Vaadin components

Summary, Part 1

- Theme variants
- Css variables
- Lumo variables

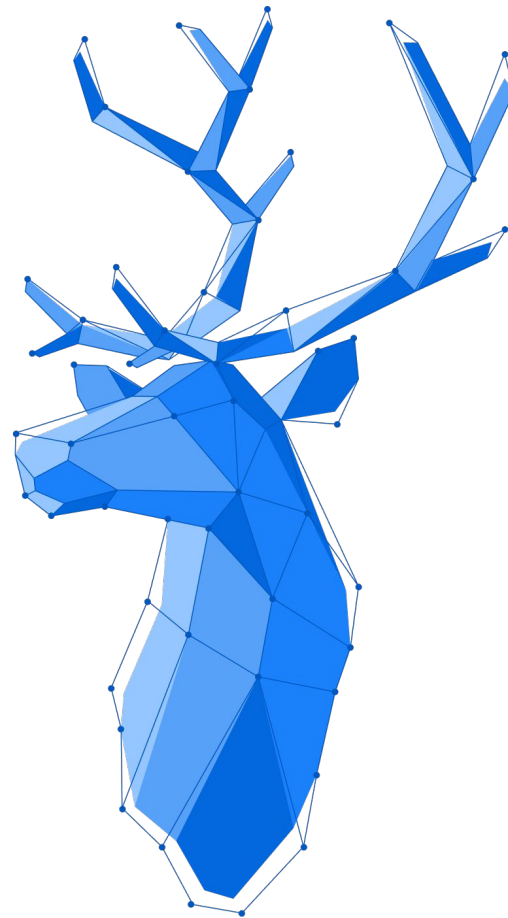


Theming and Styling Applications, Part 2

More advanced styling options

Recap, part 1

- Theme variants
- Css variables
- Lumo variables



Shadow DOM

Vaadin components use a native HTML feature called [Shadow DOM](#) which isolates their JavaScript and CSS from the surrounding page. This helps simplify the internal scripting and styling of the components.

```
▼ <div>
  ▼ <vaadin-text-field clear-button-visible="" has-value="" has-label=""> event inline-flex custom...
    ▼ #shadow-root (open)
      ▶ <style> ... </style>
      ▼ <div class="vaadin-field-container"> flex
        ▶ <div part="label"> ... </div>
        ▶ <vaadin-input-container part="input-field"> ... </vaadin-input-container> event flex custom...
        ▶ <div part="helper-text"> ... </div>
        ▶ <div part="error-message"> ... </div>
      </div>
      <slot name="tooltip"></slot> event contents
      ▶ <style> ... </style>
      ::before
      ▶ <vaadin-icon icon="vaadin:map-marker" slot="prefix"> ... </vaadin-icon> event flex custom...
      <label id="label-vaadin-text-field-0" slot="label" for="input-vaadin-text-field-3">
        Street Address</label> event
      <div id="error-message-vaadin-text-field-2" slot="error-message" hidden=""></div>
      <input id="input-vaadin-text-field-3" slot="input" value="Ruukinkatu 2" type="text" aria-
        labelledby="label-vaadin-text-field-0"> event
      </vaadin-text-field>
    </div>
```

Applying styles to Shadow DOM

Vaadin components are more complex than native HTML elements due to the additional features they provide. Shadow DOM helps to achieve and encapsulate this rich functionality.

As a consequence of the encapsulation, even seemingly simple components like Text Fields and Buttons cannot be styled directly using native HTML element selectors like `input {...}` and `button {...}`.

Applying styles to Shadow DOM

Each Vaadin component exposes a number of **parts** and **states** that can be targeted with specific selectors in CSS style blocks.

These parts and states form the styling API of Vaadin components.

`::part(label)` — Phone number

`::part(input-field)` — 555-123 123 123

`::part(helper)` — Include area code

[disabled] Phone number
555-123 123 123

[focused] Phone number
555-123 123 123

Applying styles to Shadow DOM

To affect the style of a particular part, use the appropriate selector to target it in the CSS stylesheet.

`frontend/themes/my-theme/styles.css`

```
vaadin-text-field::part(input-field) {  
  border: 1px solid gray;  
}  
  
vaadin-text-field[focused]::part(input-field) {  
  border-color: blue;  
}
```

Default

555-123 123 123

After applying CSS

555-123 123 123

Focused

555-123 123 123

Applying styles to Shadow DOM

Themable **parts** and **states** for each Vaadin component is listed in the in the **Styling tab** on [component documentation pages](#).

An example list of parts and states for TextField component can be found here:

<https://vaadin.com/docs/latest/components/text-field/styling>

Styling specific components

If you don't want to style every component in your application, you can specify a specific CSS **class** value with `setClassName/addClassName` for the component(s) you want to target.

Java

```
TextField nameField = new TextField("This is a warning");  
nameField.addClassName("special");
```

frontend/themes/my-theme/styles.css

```
vaadin-text-field.special {  
  color: orange;  
}
```

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.

Esta debería ser la última opción a elegir a la hora de implementar estilos

Changing Lumo properties in Java

You can also set Lumo property values through `getStyle()`:

```
textField.getStyle().set("--lumo-contrast-10pct", "lightgreen")
```

Name •

Muy poderoso para cambiar estilos en tiempo de ejecución

Element API

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

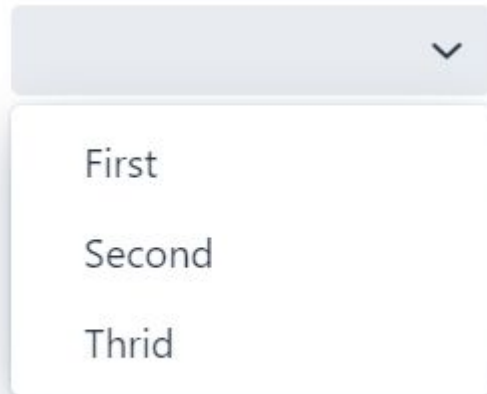
1. Inline styling with `component.getElement().getStyle().set("name", "value")`
2. Add class name(s) with `component.getElement().getClassList().add("class_name")`
3. Remove class name(s) with `component.getElement().getClassList().remove("class_name")`

Applying styles to overlays

Several Vaadin components use an “overlay” part. The overlay is a visual element that “pops over” other content on the page. Examples include:

- ComboBox - overlay with options is shown when the user clicks the drop-down button
- ContextMenu - overlay is shown after right click, showing menu items
- DatePicker - overlay shows the calendar where you can select a date

Overlays are not a part of the root element, so they can't be targeted with CSS rules that target the component itself. Instead, another approach is needed.



Applying styles to overlays

You can style the overlay part of the component using `setOverlayClassName` or `setClassName` method of the given depending on the component you use.

Java

```
ComboBox<String> comboBox = new ComboBox<>();  
comboBox.setItems("First", "Second", "Thrid");  
comboBox.setOverlayClassName("red-text-options");
```

CSS

```
vaadin-combo-box-overlay.red-text-options [role="option"] {  
  color: red;  
}
```



Additional styling options

Theme attribute

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

The theme attribute is used for Theme Variants, but you can also create custom themes.

One big difference between the theme attribute and the traditional class name is that the **theme attribute** of the root element **will propagate to the shadow DOM and any child elements**, which might be useful when you're working with add-ons. This feature is brought by implementing the ThemableMixin interface in TypeScript.

Actualmente no se recomienda usar el atributo theme a otros usos. En anteriores versiones de Vaadin si se usaba mucho. Pero no está deprecated y es muy útil para componentes personalizados.

Additional styling options

Theme attribute

Most Vaadin 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()`

Styling other UI Elements

Applying CSS to other UI Elements

Although Vaadin application UIs are built primarily using Vaadin components, native HTML elements, like `` and `<div>`, are also often used for layout and custom UI structures. These can be styled with custom CSS and utility classes that bundle predefined styles as easy-to-use constants.

Applying CSS to HTML Elements

Custom CSS is applied to native HTML elements, similarly to Vaadin components, by placing it in a stylesheet in the application theme folder.

Java






```
Span warning = new Span("This is interesting");  
warning.addClassName("interest");
```

frontend/themes/my-theme/styles.css

```
span.interest {  
    color: blue;  
}
```

Lumo Utility Classes

Lumo Utility Classes are predefined CSS class names and stylesheets that can be used to style HTML elements and layouts without writing CSS yourself. Each utility class applies a particular style to the element, such as background color, borders, fonts, sizing, or spacing.

	<code>LumoUtility.Background.BASE</code>
	<code>LumoUtility.TextColor.ERROR</code>
	<code>LumoUtility.Padding.SMALL</code>
	<code>LumoUtility.BoxShadow.XSMALL</code>
	<code>LumoUtility.BorderRadius.LARGE</code>

Lumo Utility Classes

The LumoUtility collection in Flow provides constants for each utility class. They are applied using the same addClassNames API as is used for custom CSS class names.

```
Span errorMsg = new Span("Error");  
errorMsg.addClassNames(  
    LumoUtility.TextColor.ERROR,  
    LumoUtility.Padding.SMALL,  
    LumoUtility.Background.BASE,  
    LumoUtility.BoxShadow.XSMALL,  
    LumoUtility.BorderRadius.LARGE  
);
```

Error



Lumo Utility Classes in shadow DOM

Lumo Utility classes cannot be used to override Shadow DOM styles of the component. The Shadow DOM styling always takes precedence.

The Lumo utility classes are primarily designed to be used with native HTML elements, Vaadin layout components, and custom UI structures. Although some of them do work as expected on some Vaadin components, this is not their intended use.

Lumo Utility Classes

Lumo Utility classes can be used to style the following areas:

- Backgrounds
- Borders
- Box Shadow
- Sizing
- Spacing
- Typography
- Flexbox & Grid
- Layout
- Accessibility

The utility classes contain helpful presets for display categories, such as:

- Contrast
- **Primary**
- **Error**
- **Warning**
- **Success**



CSS custom properties and variables

As in the default Lumo theme, we recommend creating your custom styles using CSS properties and variables.

Custom properties need to start with a double-dash to distinguish them from standard properties and avoid naming collisions with future standards.

```
html {  
  --custom-theme-color: orange;  
}
```

Defining custom properties

Custom properties can be defined inside any CSS selector, scoping them to that particular selector.

```
html {  
  --custom-theme-color: orange;  
}
```

You can override the value of a custom property with a higher priority selector.

```
.purple-theme {  
  --custom-theme-color: purple;  
}
```

Using custom properties

You can use the `var()` function to get the value of a custom property.

```
.link {  
  color: var(--custom-theme-color);  
}
```

You can also define a fallback value:

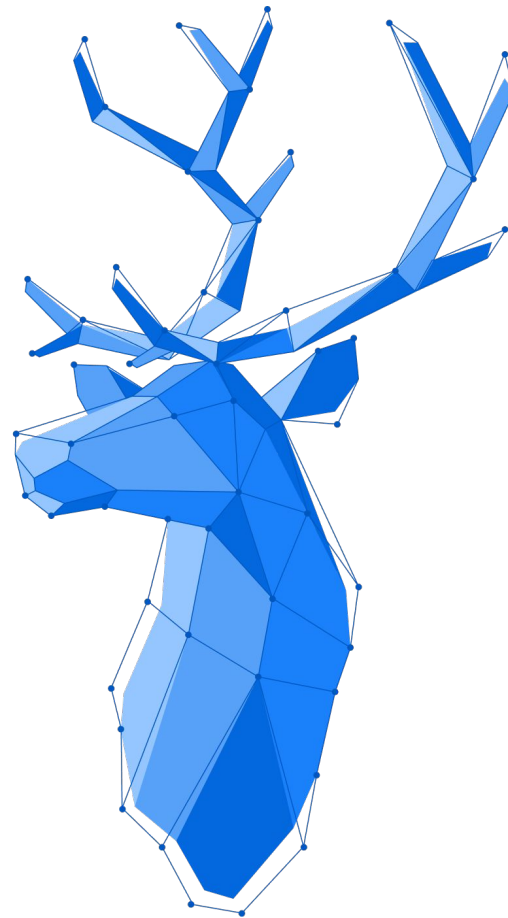
```
link {  
  color: var(--custom-theme-color, blue);  
}
```

Exercise 2

Styling component parts and states

Summary, Part 2

- Applying styles to Shadow DOM
- Applying styles to overlays
- Styling other UI elements
- Lumo Utility Classes
- Using custom properties

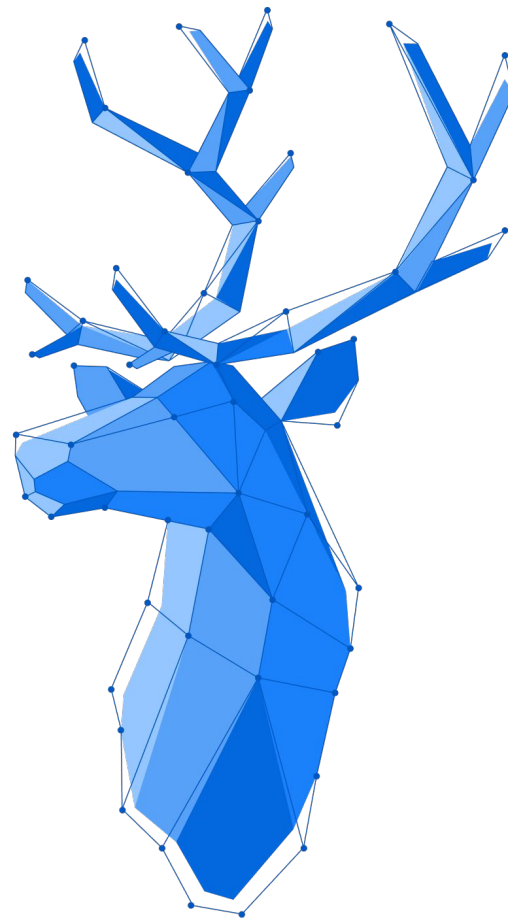


Theming and Styling Applications, Part 3

Application Theme

Recap, parts 1-2

- Theme variants
- Css variables
- Applying styles to Shadow DOM
- Applying styles to overlays
- Styling other UI elements
- Lumo Utility Classes



@Theme

The application theme is defined using the **@Theme** annotation. It can be either a predefined theme or a custom theme.

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

Lumo

Default theme

```
@Theme(themeClass = Lumo.class)
public class AppShell implements
AppShellConfigurator {
```

Se añade la anotación en la clase que implementa la interface AppShellConfigurator

The screenshot displays a web application interface with a table of items and an 'Edit item' modal dialog. The table has columns for 'Name' and 'Price'. The modal dialog is open, showing the 'Name' field with the value 'vel' and the 'Price' field with the value '1'. The dialog also includes a 'Delete...' button, a 'Cancel' button, and a 'Save' button.

Name	Price	
magna	95	✎
sit	49	✎
elit		✎
interdum		✎
condimentum		✎
vel		✎
sit		✎
enim		✎
phasellus		✎
congue		✎
amet	78	✎
leo	15	✎
lobortis	50	✎

Master-detail (CRUD) Logout

Name Price

magna 95

sit 49

elit

interdum

condimentum

vel

sit

enim

phasellus

congue

amet 78

leo 15

lobortis 50

New item

Edit item

Name

vel

Price

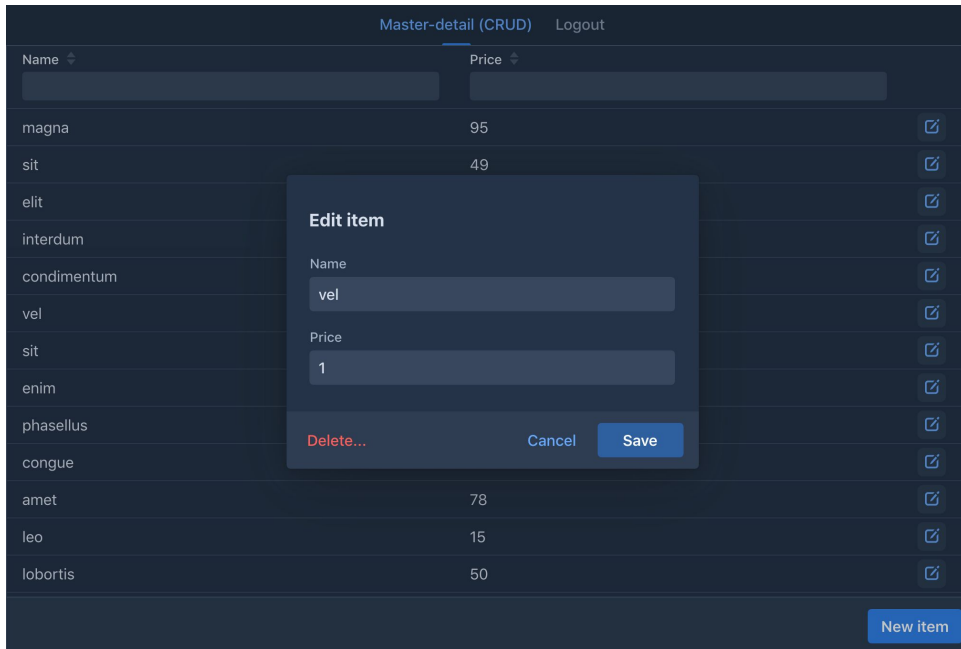
1

Delete... Cancel Save

Lumo dark

Lumo has light and dark variants.
The default is light.

```
@Theme(themeClass = Lumo.class, variant  
= Lumo.DARK)  
public class AppShell implements  
AppShellConfigurator {
```



@NoTheme

If needed, you can disable the Lumo theme for your application using the **@NoTheme** annotation. However, this isn't recommended for most applications, as it's easier to extend the Lumo theme than to create a brand new theme.

Custom theme

A custom theme is the easiest way to provide a custom, consistent look and feel for your entire application.

Style property customizations and custom CSS of the theme are both placed in CSS stylesheets, typically in the application's theme folder. The theme folder is specified using the **@Theme** annotation.

```
@Theme("my-theme")
public class Application implements AppShellConfigurator {
    ...
}
```

Custom theme

- The CSS in a custom theme is always applied on top of the default Lumo theme.
- It can be packaged as a dependency for reuse in multiple applications.
- Application projects generated with Vaadin Start have a theme folder applied by default

Custom theme

Folder structure

For use in a single application, a custom theme is implemented as a folder inside `frontend/themes`, with the following minimal structure:

`frontend`

└─ `themes`

└─ `my-theme`

└─ `theme.json`

└─ `styles.css`

Custom theme folder structure

Themes subfolder

The “themes” folder uses a special naming syntax and can contain multiple custom themes (but only one can be applied to the application at a time)

frontend

└─ themes

└─ my-theme

└─ theme.json

└─ styles.css

Custom theme folder structure

Your theme folder

Each theme is in its own sub-folder. The name of this folder is provided as a parameter to the @Theme annotation to apply the theme to the application. In this case, it would be @Theme("my-theme")

frontend

└─ themes

└─ my-theme

└─ theme.json

└─ styles.css

Custom theme folder structure

Your theme folder

The theme configuration file, `theme.json`, can be used to configure various theme-related features.

frontend

└─ themes

└─ my-theme

└─ theme.json

└─ styles.css

Custom theme folder structure

Master style sheet

A mandatory part of a theme is a **styles.css** file, which is the theme's master style sheet. The master stylesheet is automatically loaded when the theme is applied.

frontend

└─ themes

└─ my-theme

└─ theme.json

└─ **styles.css**

Master style sheet

The master style sheet typically contains the following:

- Imports of other global style sheets within the theme folder
- Overrides of default Lumo properties
- Declarations of custom CSS properties
- Styles that are applied to UI elements through class names and other CSS selectors.

Master styles.css example

```
/* Imports other stylesheet files */
@import 'other-styles.css';
@import 'views/admin-view.css';

/* Overrides and custom property declarations */
html, :host {
  --lumo-border-radius-m: 0.5em;
  --my-brand-color: purple;
}

/* Styles with CSS class names */
.application-header {
  background: white;
  border-bottom: 1px solid gray;
}
```

CSS custom properties (for example, for overriding Lumo defaults) are recommended to use the selector “html, :host”, as in the example above, to ensure the styles get applied in embedded Vaadin applications.

Images and other assets

Your theme may include assets like fonts, images, and icons. They can be in the theme root or any subfolder.

frontend

└─ themes

└─ my-theme

└─ logo.png

└─ fonts/

└─ my-font.woff

└─ styles.css

Images and other assets

Assets can be used in the theme's style sheets through URIs relative to the style sheet's location:

`styles.css`

```
@font-face {  
    font-family: "My Font";  
    src: url('../fonts/my-font.woff') format("woff");  
}  
  
.application-logo {  
    background-image: url('../logo.png');  
}
```


External Stylesheets

External stylesheets can be loaded from outside the application by URL using the same @import directive.

`styles.css`

```
@import url('https://example.com/some-external-styles.css');
```

```
html, :host() {
```

```
  ...
```

```
}
```

Cuidado con posibles riesgos de seguridad con estos CSS externos.

Static Resources

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

Static resources

Theme images

Images and icons related to the application theme can be placed in the `frontend/themes/my-theme` folder and referenced from the Java code since the `frontend/themes` folder is automatically copied to the resource folder as part of the build process.

```
[project root]
├── frontend
│   └── themes
│       └── my-theme
│           └── image.png
```

In Java

```
Image img = new Image("themes/my-theme/image.png", "An image in the theme");
```

Static resources

Static Resources – WAR packaging

For projects with **war** packaging, you can also put static resources, including images, 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

Static Resources – JAR packaging

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);  
}
```

Lazy-loading Stylesheets

Stylesheets that are only needed when a particular view or other Flow-based UI class is loaded can be lazy loaded into the UI using a `@StyleSheet` annotation on the class.

When serving a local file, the stylesheet must be placed directly in the static resources folder (not the theme folder).

```
[project root]
├── src
│   ├── main
│   │   ├── resources
│   │   │   ├── META-INF
│   │   │   └── resources
│   │   │       └── lazy-loaded.css
```

```
@StyleSheet("./lazy-loaded.css") // Local file in resource folder
public class MyUI extends Div {
    ...
}
```

Load External Stylesheets from Java

You can also use @StyleSheet annotation to load external stylesheets.

```
@StyleSheet("https://example.com/external-styles.css")  
public class MyUI extends Div {  
    ...  
}
```

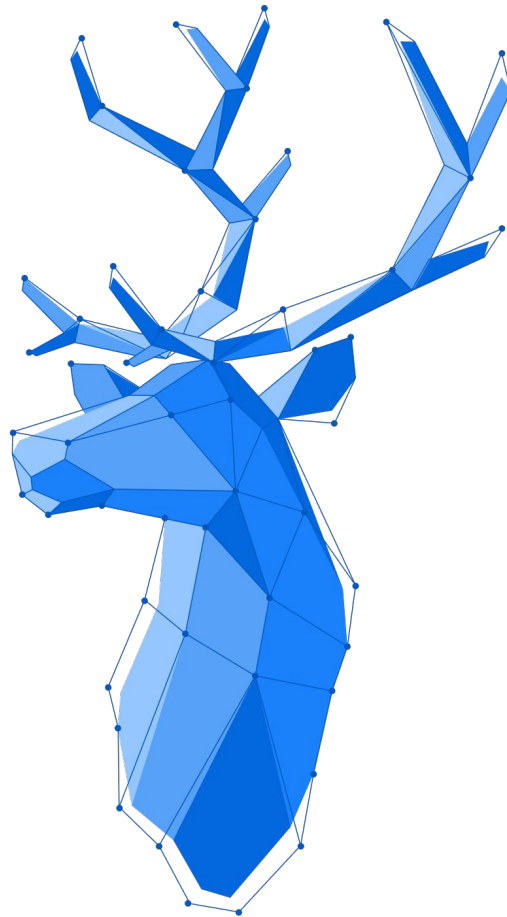
Using Static Resources on Login view

If you have static resources used in your login view and using Spring Boot, i.e. security based on **VaadinWebSecurity** class, the resource needs to be made available for public by configuring:

```
public void configure(WebSecurity web) throws Exception {  
    super.configure(web);  
    // Allow loading background image on login page  
    web.ignoring().antMatchers("/images/*.png");  
    ...  
}
```


Summary, Part 3

- Custom Theme
- Using resources
- Loading stylesheets

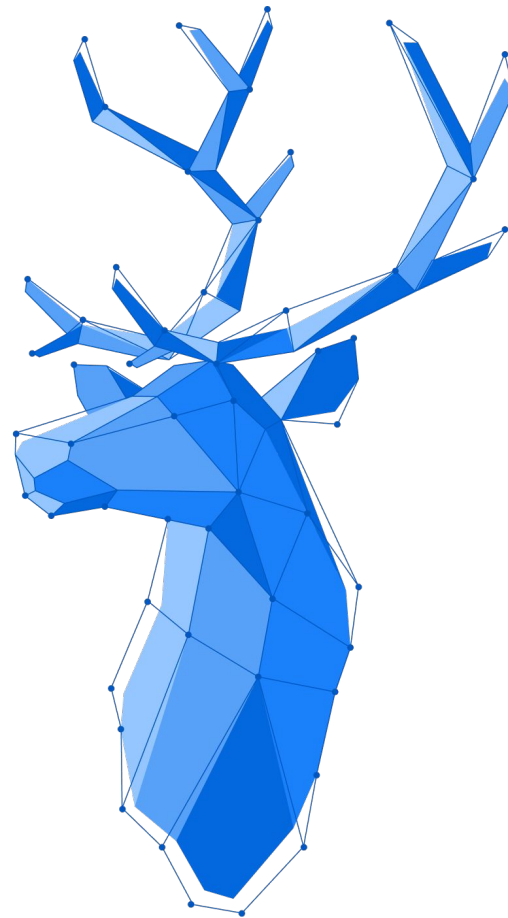


Theming and Styling Applications, Part 4

Grid dynamic styling

Recap, parts 1-3

- Theme variants
- Css variables
- Applying styles to Shadow DOM
- Lumo Utility Classes
- Custom Theme
- Using resources



Grid dynamic styling

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

Grid dynamic styling

Part name generator

A Grid contains dynamically populated data and can contain any number of rows. Without any help, targeting a specific row or cell with CSS rules is difficult or impossible; that's why Grid offers part name generators that allow the creation of part attributes based on the row data.

```
grid.setPartNameGenerator(product -> product.isAvailable() ? "" : "unavailable" );
```

Grid dynamic styling

Styling a whole row

To style the whole **row**, call `setPartNameGenerator` on the **Grid**, which applies the part name on all the cells in the row.

```
grid.setPartNameGenerator(this::generateWarnPartName);
```

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

Grid dynamic styling

Styling a cell

To style a **cell**, call `setPartNameGenerator` on a **Column**.

```
grid.addColumn(MonthlyExpense::getSpending)
    .setHeader("Expenses")
    .setPartNameGenerator(this::generateWarnPartName);
```

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

Grid dynamic styling

Style grid parts in the theme

Then add a style for the part name in the theme stylesheet:

```
frontend/themes/my-theme/styles.css
```

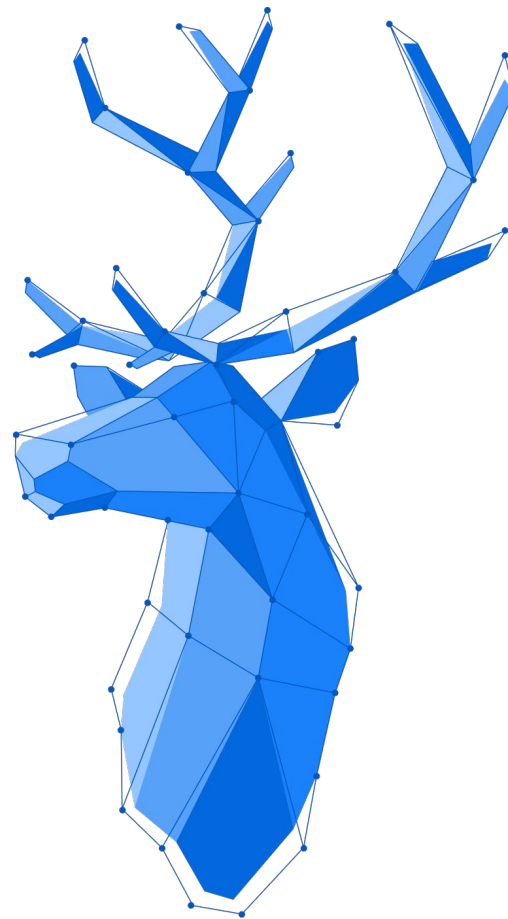
```
vaadin-grid::part(warn) {  
  color: red;  
}
```


Exercise 3

Style a Grid cell/column based on its content

Summary, Part 4

- Part name generator
- Styling whole row
- Styling a cell

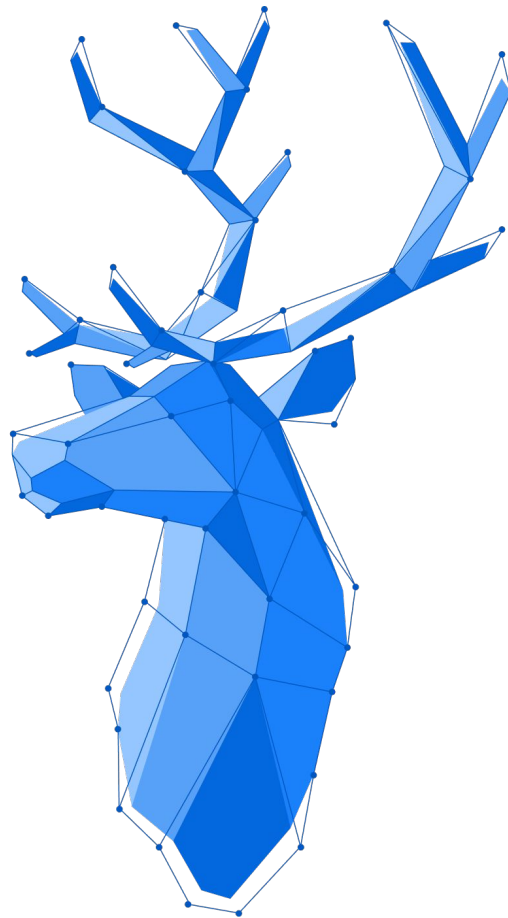


Theming and Styling Applications, Part 5

Case study: Custom font

Recap, parts 1-4

- Theme variants
- Css variables
- Applying styles to Shadow DOM
- Applying styles to overlays
- Styling other UI elements
- Lumo Utility Classes
- Grid styling



Case study: Custom font

Replace default font faces in the Lumo theme

Case study: Custom font

Custom font faces

Using a custom font is a common requirement, especially for enterprise applications.

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

The following slides will show you how to use a custom font in a default Lumo theme.

Case study: Custom font

Get a font

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](#)

Case study: Custom font

Find or Generate Web Font Kit

Next, you need a **Web Font Kit** for the font. Many font providers like [Google Fonts](#) provide the Web Font Kits out of the box. Otherwise, you can generate one.

You can upload your .ttf or .otf file to the [Webfont Generator](#) to generate the Kit.

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

Case study: Custom font

Place the fonts in your theme

Put the font files into the right place. Create a folder called “fonts” under your theme folder and put the .woff, .woff2, and stylesheet.css files under the new folder.

frontend

└─ themes

└─ my-theme

└─ fonts/

└─ └─ chilanka-regular-webfont.woff

└─ └─ chilanka-regular-webfont.woff2

└─ └─ stylesheet.css

└─ theme.json

└─ styles.css

Case study: Custom font

Import font

Import the stylesheet.css by adding the import declaration to the master style sheet.

```
frontend/themes/my-theme/styles.css
```

```
@import 'fonts/stylesheet.css';
```

Case study: Custom font

Apply font

Use the font with `--lumo-font-family` variable in your master style sheet.

frontend/themes/my-theme/styles.css

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

Case study: Custom font

Fallback font

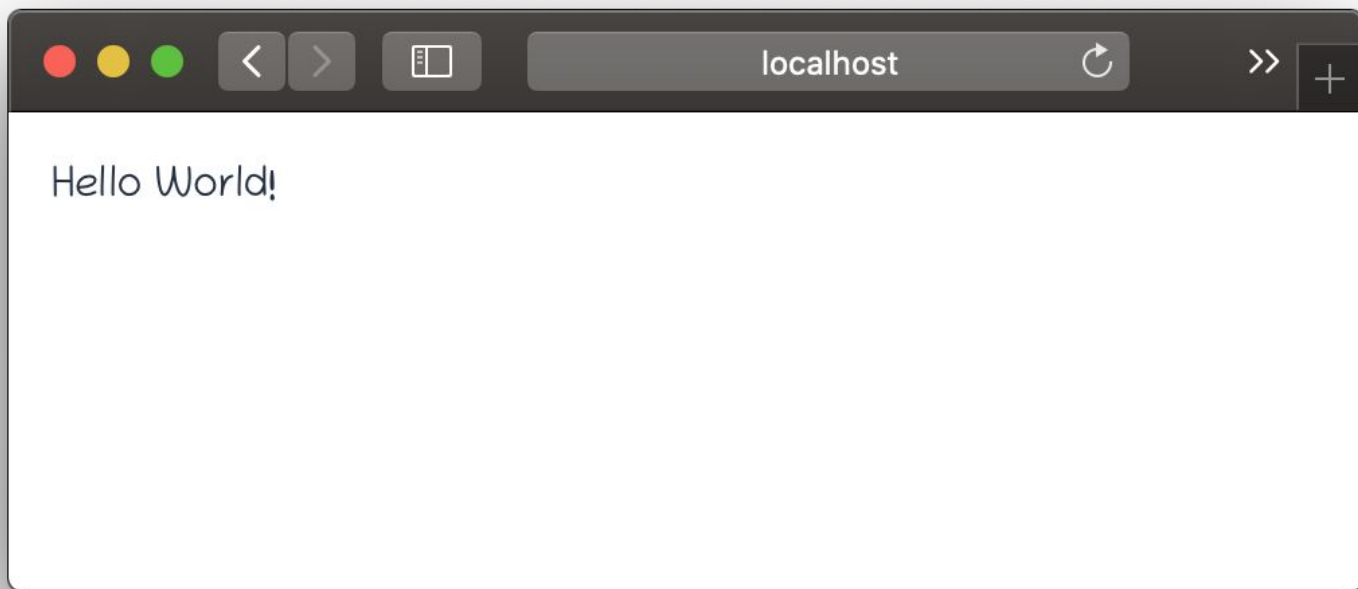
It's also a good practice to define at least one fallback font for cases where the web font fails to load.

```
frontend/themes/my-theme/styles.css
```

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

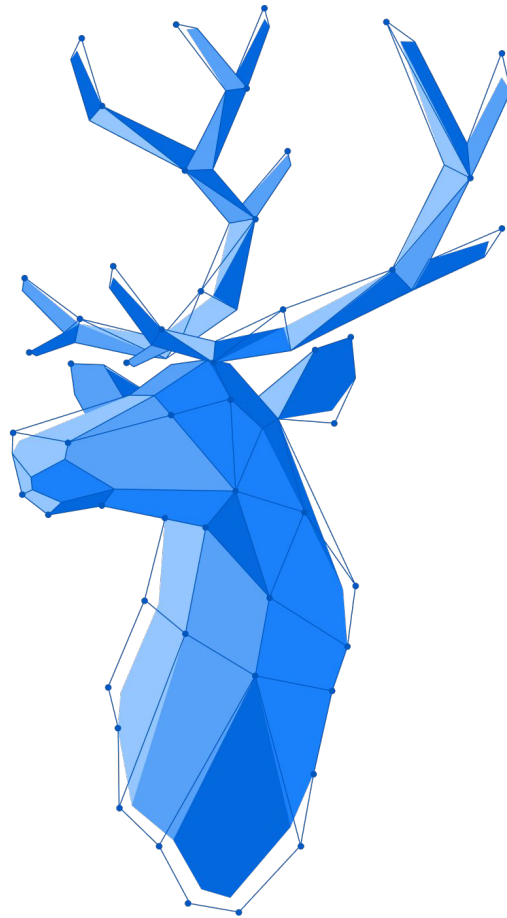
Case study: Custom font

The result



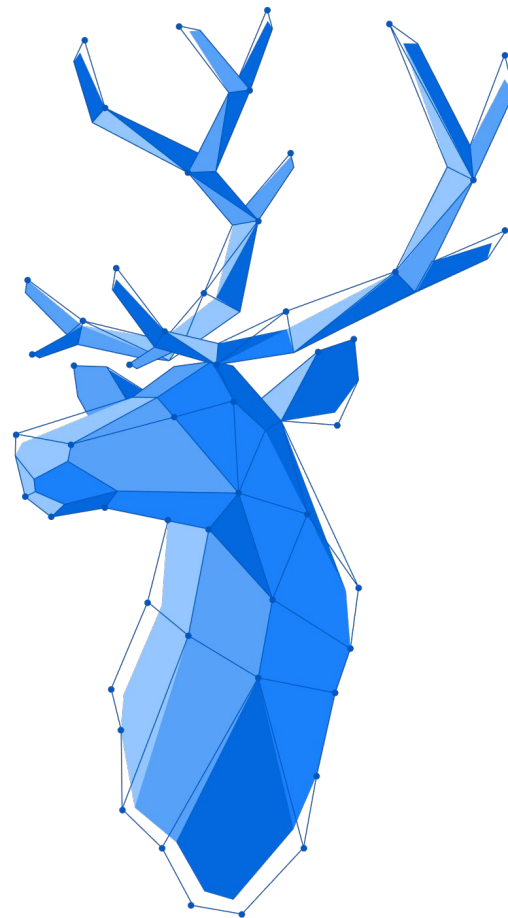
Summary, Part 5

- Generate Web Font Kit
- Import font
- Apply font



Summary

- Styling Vaadin Components
- Lumo
- More advanced styling options
- Styling Other UI Elements
- Application Theme
- Grid Dynamic Styling
- Custom fonts



Thank you!

