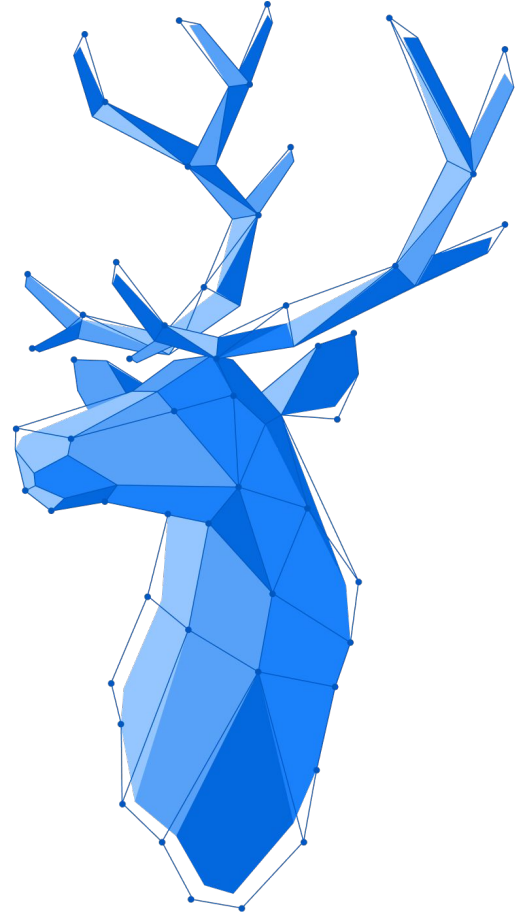


vaadin}>

# Layouting – Positioning components

Vaadin Flow



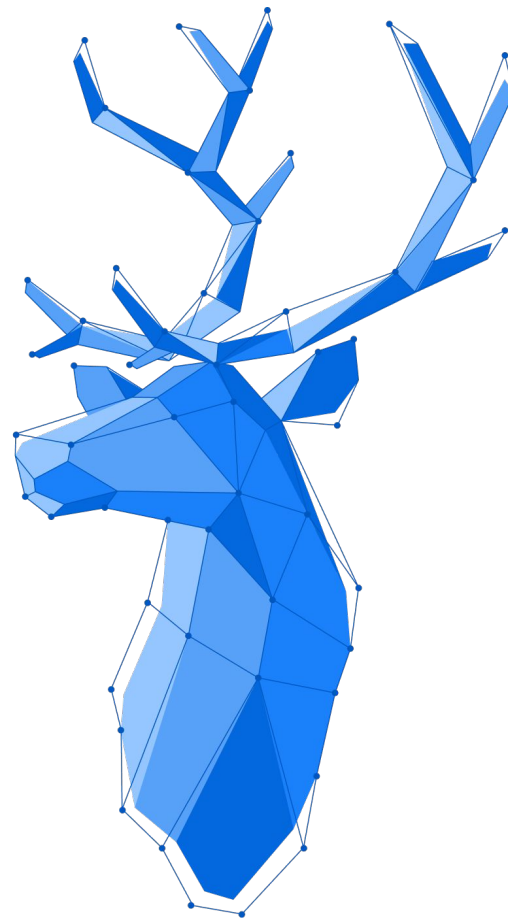
Vaadin training set

# Vaadin Foundation

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

# Agenda

- Part 1:
  - Horizontal & Vertical Layouts
- Part 2:
  - Flexbox
  - Exercise 1
- Part 3:
  - FormLayout
  - Exercise 2
- Part 4:
  - Vaadin Board and App Layout
  - Exercise 3



# Layouting, Part 1

Horizontal & Vertical Layouts

# HorizontalLayout & VerticalLayout

# Adding components

```
HorizontalLayout layout = new HorizontalLayout();  
// or VerticalLayout layout = new VerticalLayout();  
layout.add(new DatePicker("DatePicker"));  
layout.add(new TextField("TextField"));  
layout.add(new ComboBox("ComboBox"));
```

## VerticalLayout

DatePicker

TextField

ComboBox

## HorizontalLayout

DatePicker

TextField

ComboBox

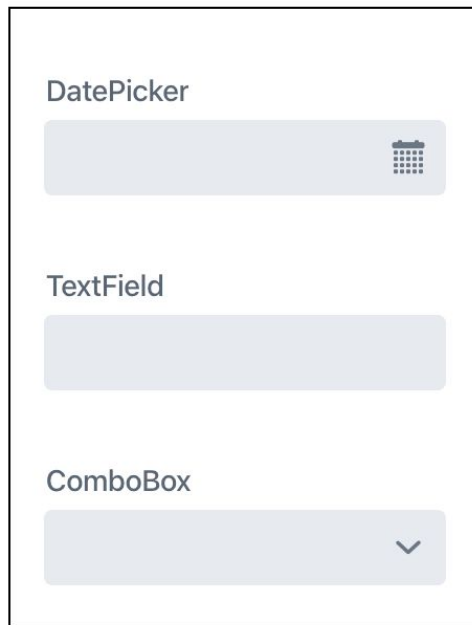
# Padding

Padding means space around the **inner** side of the border of the layout.

Padding can be turned on and off with `setPadding()`

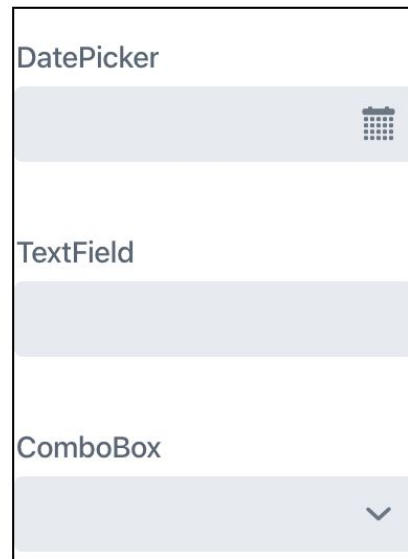
```
layout.setPadding(false);  
layout.setPadding(true);
```

With Padding



The image shows three UI components stacked vertically within a container. Each component is a light gray rectangle with a thin black border. The components are labeled 'DatePicker', 'TextField', and 'ComboBox' above them. The 'DatePicker' has a calendar icon on the right. The 'TextField' is empty. The 'ComboBox' has a downward arrow icon on the right. There is a significant amount of white space (padding) between the components and between the components and the container borders.

Without Padding



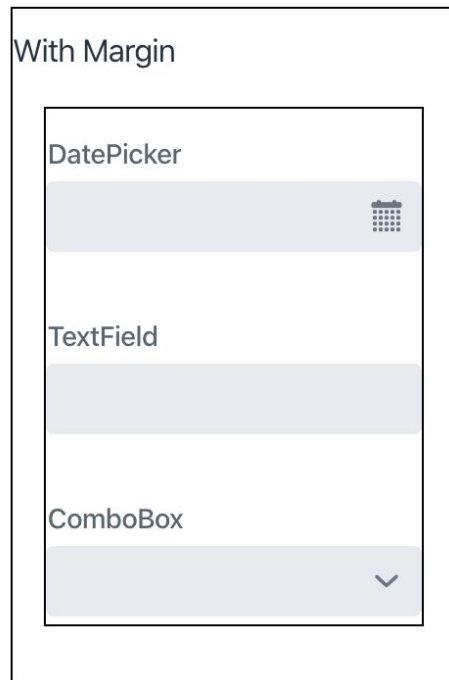
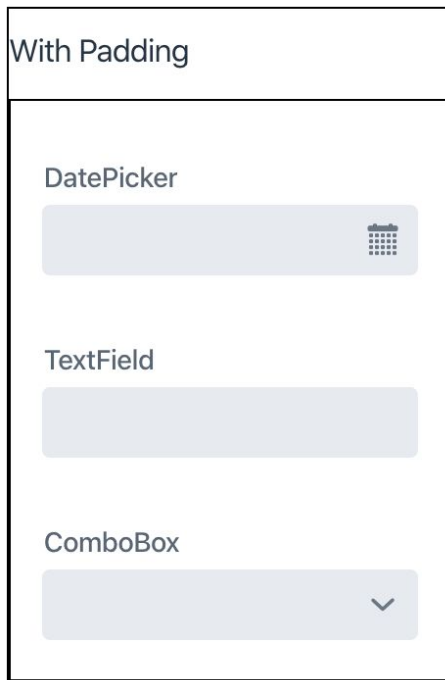
The image shows the same three UI components stacked vertically within a container, but without padding. The components are labeled 'DatePicker', 'TextField', and 'ComboBox' above them. The 'DatePicker' has a calendar icon on the right. The 'TextField' is empty. The 'ComboBox' has a downward arrow icon on the right. The components are tightly packed together with no white space between them or between them and the container borders.

# Margin

Margin means space around the **outer** side of the border of the layout.

Margin can be turned on and off with `setMargin()`

```
layout.setMargin(false);  
layout.setMargin(true);
```





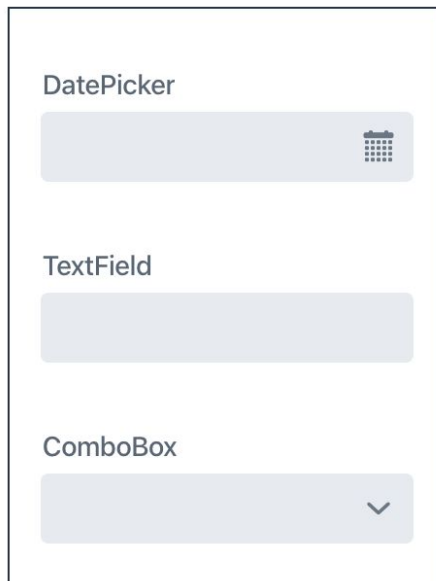
# Spacing

Spacing means the space between the components in the layout.

Spacing can be turned on and off with `setSpacing()`

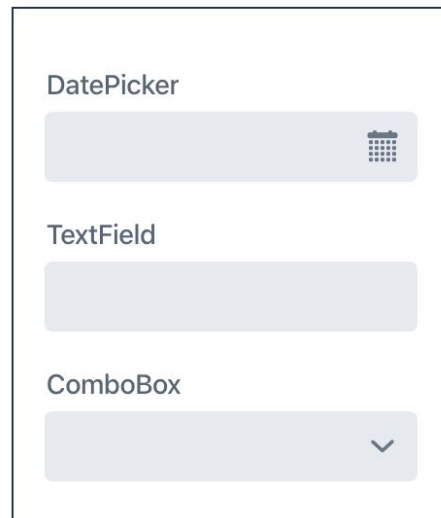
```
layout.setSpacing(false);  
layout.setSpacing(true);
```

VerticalLayout with spacing



A vertical container with three components: DatePicker, TextField, and ComboBox. The components are stacked vertically with visible gaps between them. The DatePicker has a calendar icon, the TextField is a simple input box, and the ComboBox has a dropdown arrow.

VerticalLayout without spacing



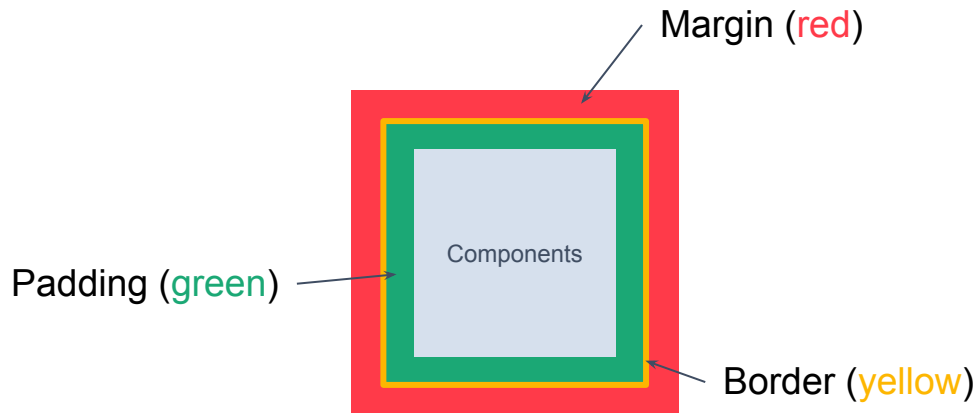
A vertical container with three components: DatePicker, TextField, and ComboBox. The components are stacked vertically with no gaps between them, appearing tightly packed.

# Margin vs Padding

Padding is added to the inside of the layout while margin is added to the outside.

If you give the layout a background color:

- the padding will be colored
- the margin will not



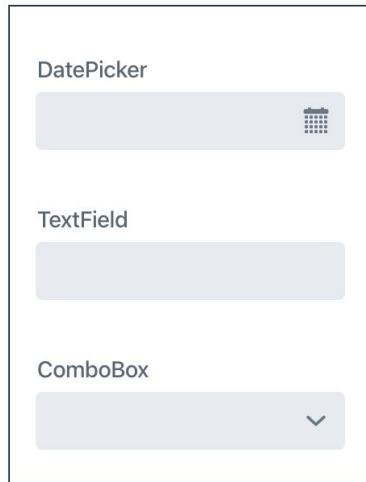
# Default values

By default, `VerticalLayout` has both padding and spacing.

`HorizontalLayout` has spacing but no padding.

Neither `VerticalLayout` nor `HorizontalLayout` has margin by default.

VerticalLayout

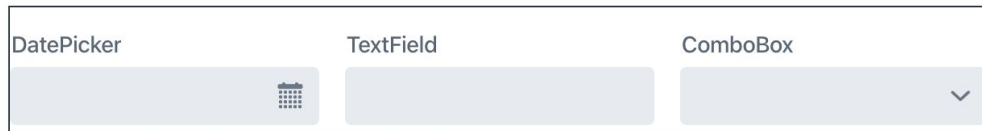
A vertical container showing three UI components stacked on top of each other with visible spacing and padding. The components are a DatePicker, a TextField, and a ComboBox.

DatePicker

TextField

ComboBox

HorizontalLayout

A horizontal container showing three UI components side-by-side with visible spacing but no padding. The components are a DatePicker, a TextField, and a ComboBox.

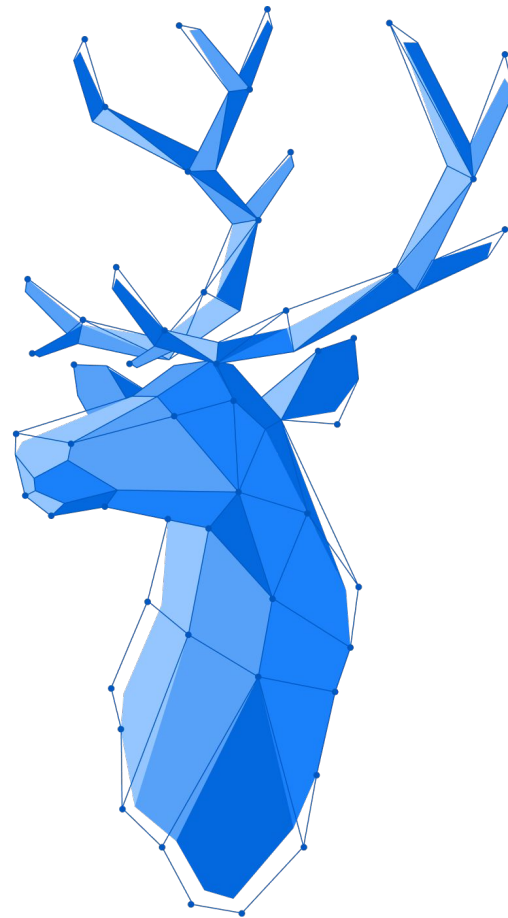
DatePicker

TextField

ComboBox

# Summary, Part 1

- `HorizontalLayout` & `VerticalLayout`
- Adding components
- Padding
- Margin
- Spacing

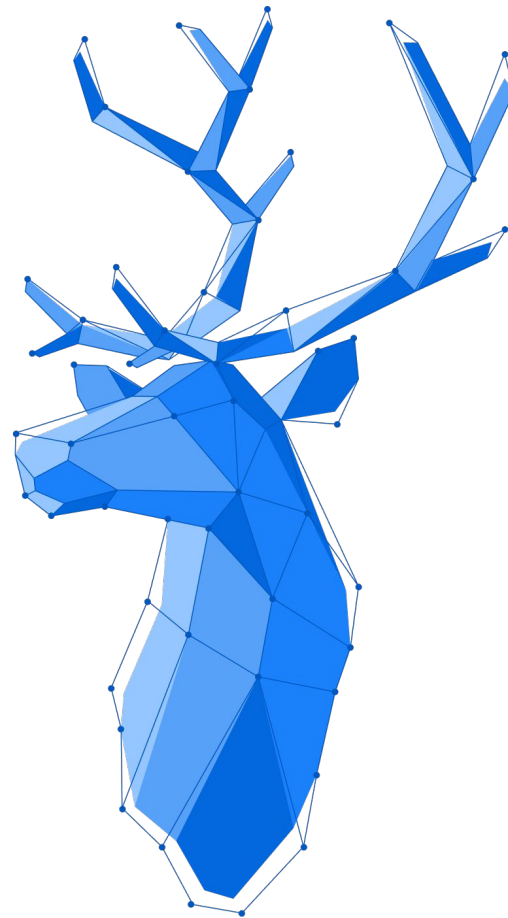


# Layouting, Part 2

Flexbox and Alignment

# Recap, Part 1

- `HorizontalLayout` & `VerticalLayout`
- Adding components
- Padding
- Margin
- Spacing



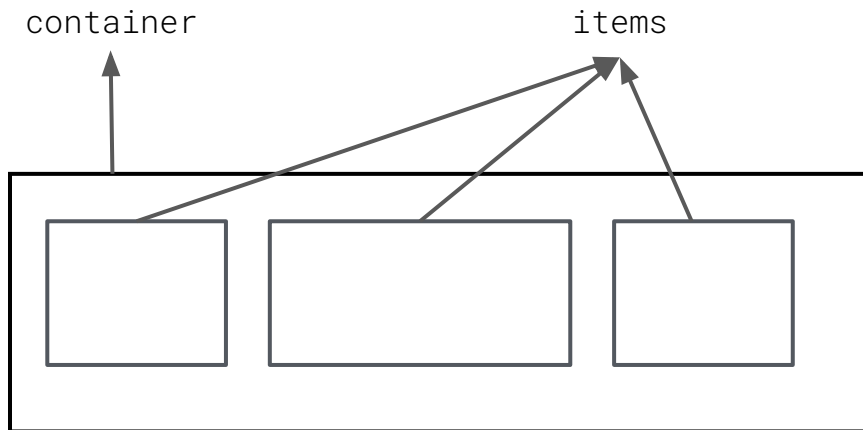
# Flexbox and FlexLayout

# Flexbox

Flexbox is a CSS layouting feature.

It has two main concepts: container and items.

Flexbox is a very powerful and highly configurable layouting model, offering you lots of freedom in what the layouting should look like.

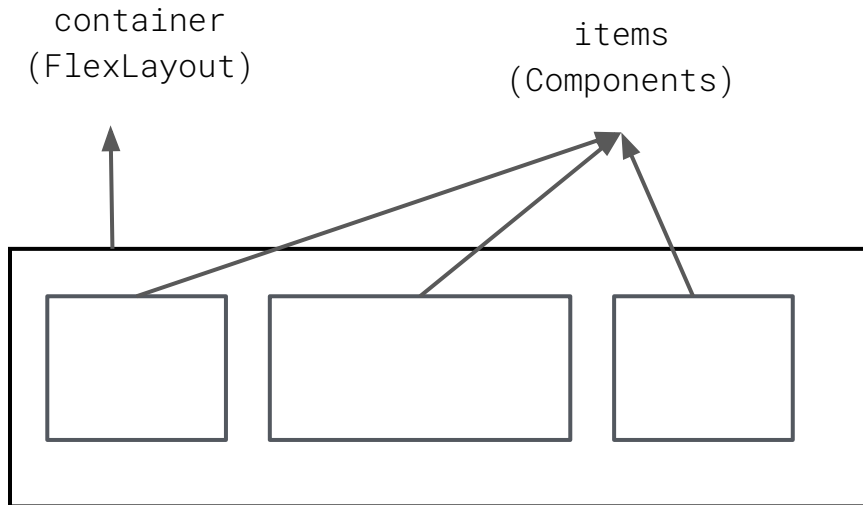




# FlexLayout

FlexLayout is a Flow layout component that offers Flexbox layouting through a Java API.

FlexLayout offers you an easy way to take advantage the different layout configuration options from Flexbox, but requires some understanding of how the underlying mechanisms work.

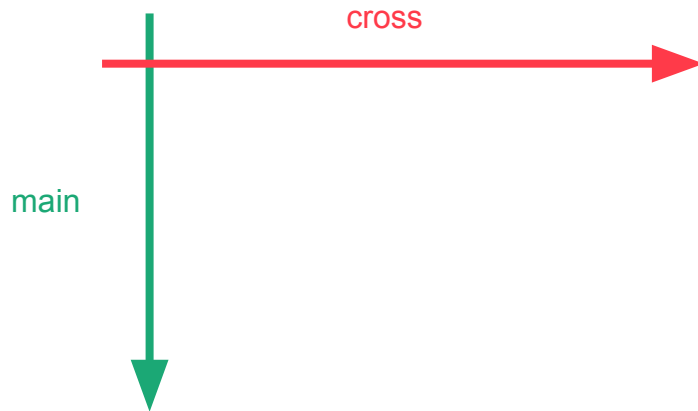
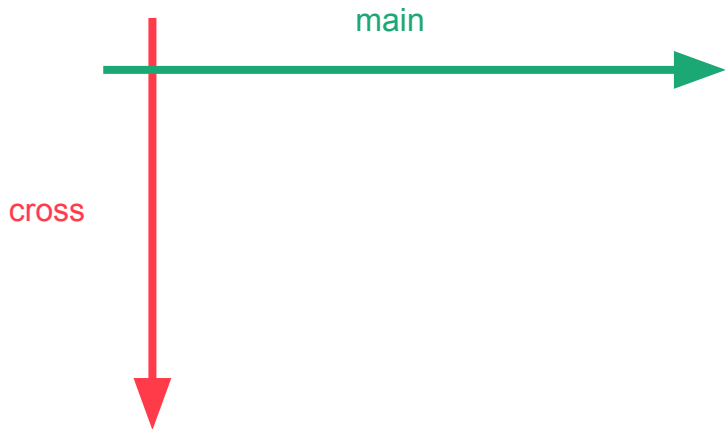


# Alignment in Flexbox

# Main and Cross Axes

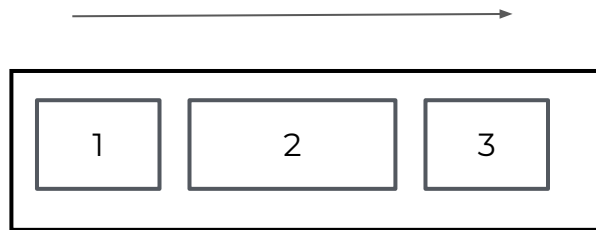
The **main axis** is the axis along which items are laid out inside a flexbox.

The **cross axis** runs perpendicular to the main axis.

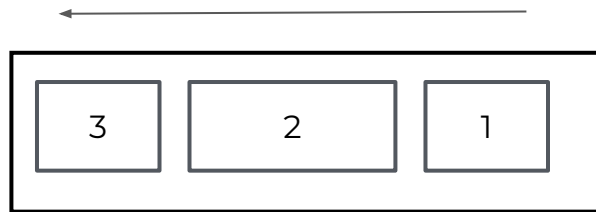


# flex-direction

flex-direction is a CSS property that applies to the container.



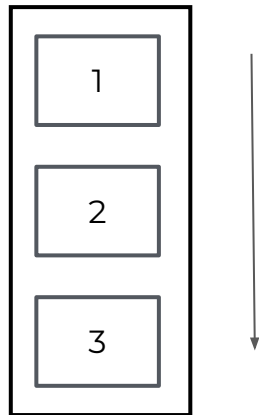
`flex-direction: row`



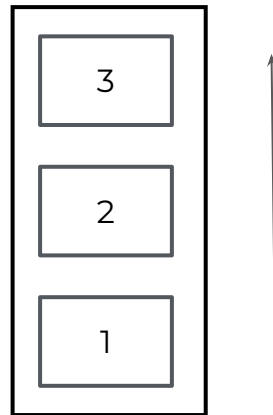
`flex-direction: row-reverse`

# flex-direction

flex-direction is a CSS property that applies to the container.



`flex-direction:column`



`flex-direction:column-reverse`

# Java API for flex-direction

There is a **FlexDirection** enum and a Java API for setting the flex-direction of a FlexLayout

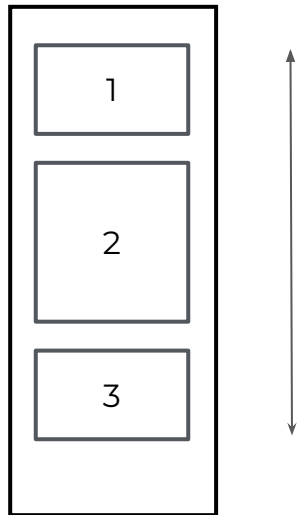
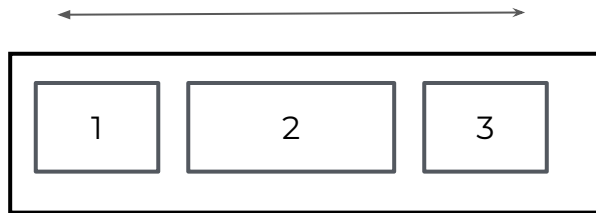
```
flexLayout.setFlexDirection(FlexDirection.ROW);  
flexLayout.setFlexDirection(FlexDirection.ROW_REVERSE);  
flexLayout.setFlexDirection(FlexDirection.COLUMN);  
flexLayout.setFlexDirection(FlexDirection.COLUMN_REVERSE);
```

# Alignment

**justify-content** determines how the items are positioned along the **main** axis.

For a horizontal layout, it's the horizontal axis.

For a vertical layout, it's the vertical axis.



# Alignment on the main axis

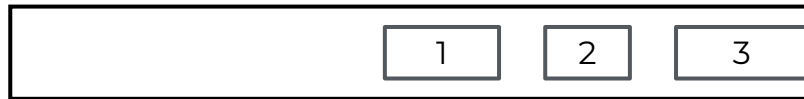
**flex-start:** Default value. Items are positioned at the beginning of the main axis.

**flex-end:** Items are positioned at the end of the main axis.

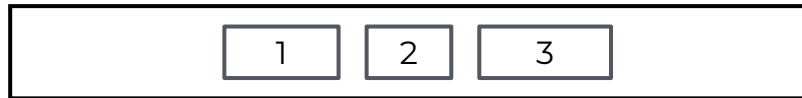
**center:** Items are positioned at the center of the main axis.



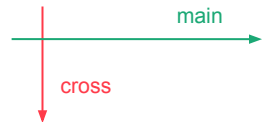
`justify-content: flex-start`



`justify-content: flex-end`



`justify-content: center`



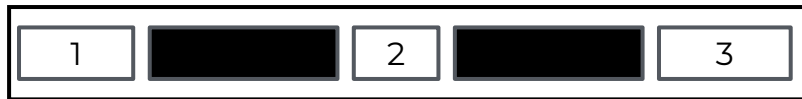


# Alignment on the main axis

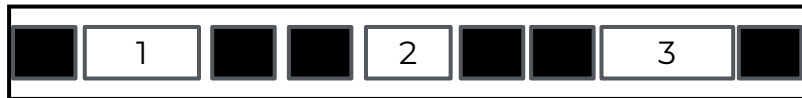
**space-between:** items are evenly distributed on the main axis, with equal space *between* them.

**space-around:** items are evenly distributed on the main axis with equal space *around* them.

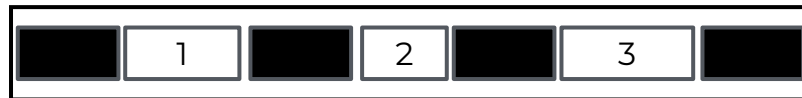
**space-evenly:** items are evenly distributed on the main axis, with equal space between both items and the edges of the container.



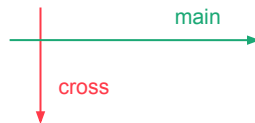
justify-content: space-between



justify-content: space-around



justify-content: space-evenly



# Alignment on the main axis

There is a **JustifyContentMode** enum and a Java API for doing the alignment on the primary axis

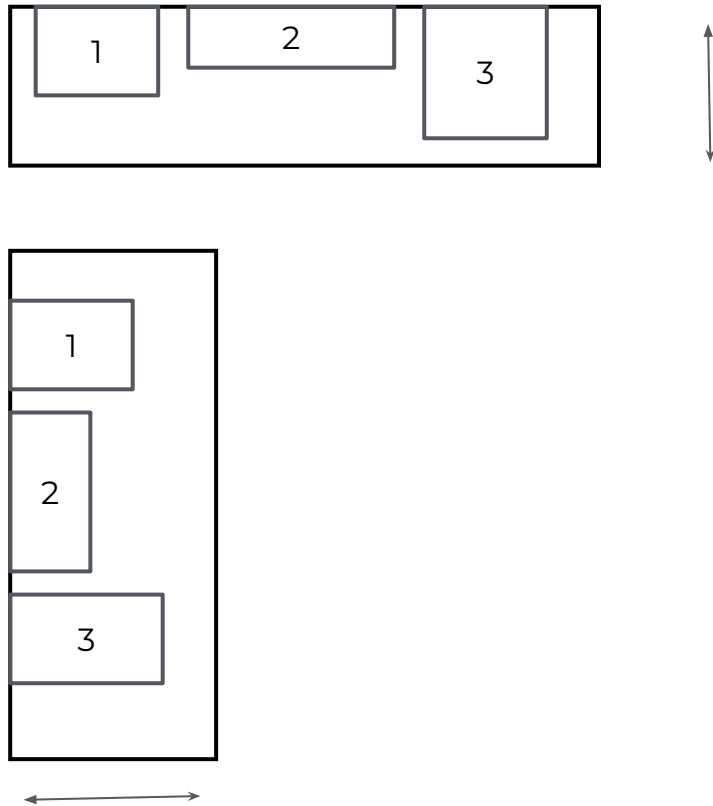
```
layout.setJustifyContentMode(FlexComponent.JustifyContentMode.AROUND);  
layout.setJustifyContentMode(FlexComponent.JustifyContentMode.BETWEEN);  
layout.setJustifyContentMode(FlexComponent.JustifyContentMode.CENTER);  
layout.setJustifyContentMode(FlexComponent.JustifyContentMode.END);  
layout.setJustifyContentMode(FlexComponent.JustifyContentMode.EVENLY);  
layout.setJustifyContentMode(FlexComponent.JustifyContentMode.START);
```

# Alignment

**align-items** determines how the items are positioned on the **cross** axis.

For a horizontal layout, it's the vertical axis.

For a vertical layout, it's the horizontal axis.

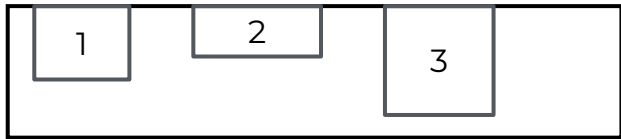


# Alignment on the cross axis

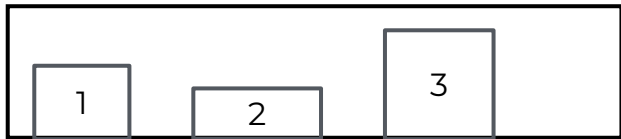
**flex-start:** Items are positioned at the start of the cross axis.

**flex-end:** Items are positioned at the end of the cross axis.

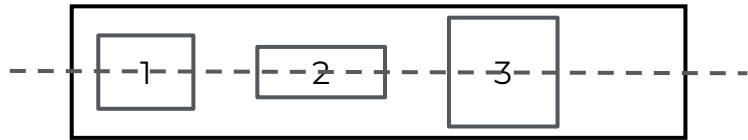
**center:** Items are positioned at the center of the cross axis.



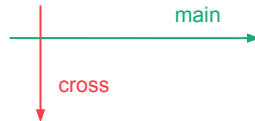
`align-items: flex-start`



`align-items: flex-end`



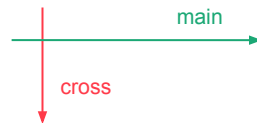
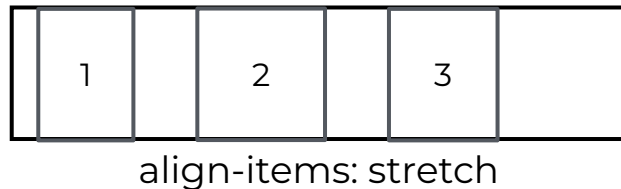
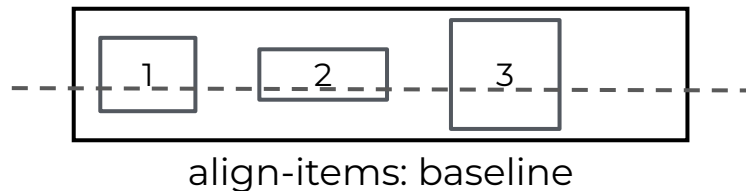
`align-items: flex-center`



# Alignment on the cross axis

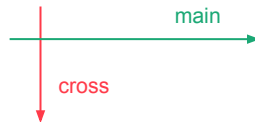
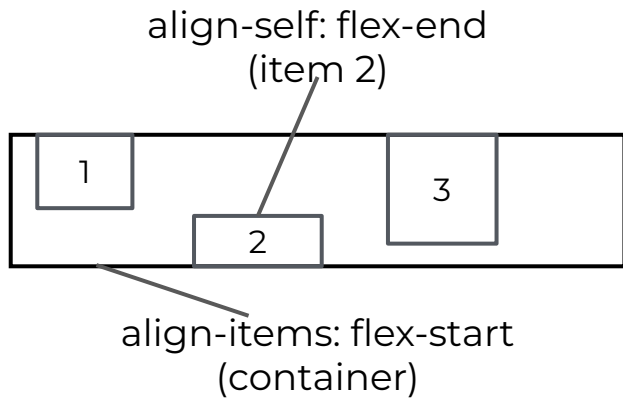
**flex-baseline:** Items are aligned along the container's baseline.

**flex-stretch:** Default value. Items are stretched to fill the container along the cross axis.



# Alignment on the cross axis

Also possible to align an individual item differently with **align-self**



# Alignment on the cross axis

There is an **Alignment** enum and a Java API for doing the alignment on the cross axis

```
//For the container, to align all its items  
layout.setAlignItems(FlexComponent.Alignment.BASELINE);  
layout.setAlignItems(FlexComponent.Alignment.CENTER);  
layout.setAlignItems(FlexComponent.Alignment.END);  
layout.setAlignItems(FlexComponent.Alignment.START);  
layout.setAlignItems(FlexComponent.Alignment.STRETCH);
```

# Alignment on the cross axis

There are also helper methods for `HorizontalLayout` and `VerticalLayout` to do the alignment on the cross axis.

```
// To align all the items on the vertical axis for a horizontal layout
```

```
horizontalLayout.setDefaultVerticalComponentsAlignment(FlexComponent.Alignment.BASELINE);
```

```
// To align all the items on the horizontal axis for a vertical layout
```

```
verticalLayout.setDefaultHorizontalComponentsAlignment(FlexComponent.Alignment.BASELINE);
```



# Alignment on the cross axis

There is a **Alignment** enum and a Java API for doing the alignment on the cross axis for individual items

```
// For individual item(s)  
layout.setAlignSelf(FlexComponent.Alignment.BASELINE, item);  
layout.setAlignSelf(FlexComponent.Alignment.CENTER, item);  
layout.setAlignSelf(FlexComponent.Alignment.END, item);  
layout.setAlignSelf(FlexComponent.Alignment.START, item);  
layout.setAlignSelf(FlexComponent.Alignment.STRETCH, item);
```

# Alignment on the cross axis

There are also helper methods for `HorizontalLayout` and `VerticalLayout` to do the alignment for individual items on the cross axis.

*//To align an individual item on the vertical axis for a horizontal layout*

```
horizontalLayout.setVerticalComponentsAlignment(FlexComponent.Alignment.END, component);
```

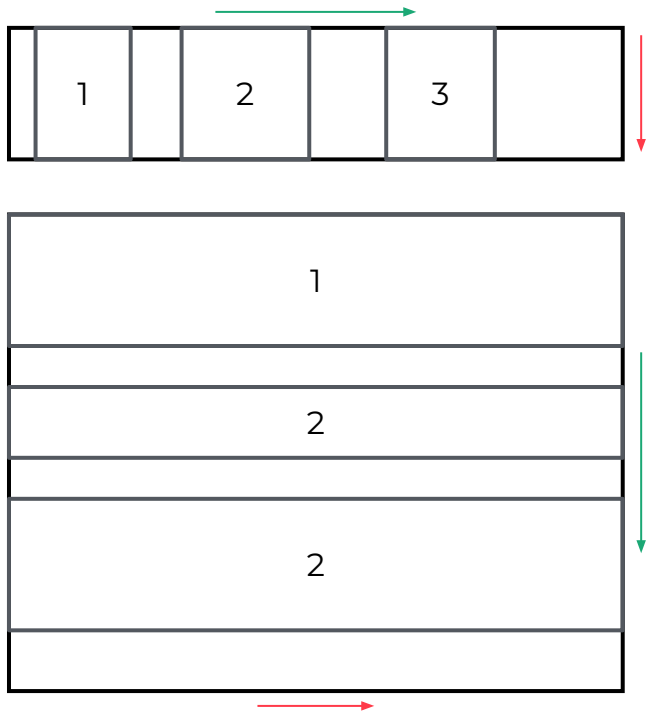
*//To align an individual item on the horizontal axis for a vertical layout*

```
verticalLayout.setHorizontalComponentsAlignment(FlexComponent.Alignment.END, component);
```

# Use case - full width/height

To make all the child items have full height in a horizontal layout or full width in a vertical layout, use

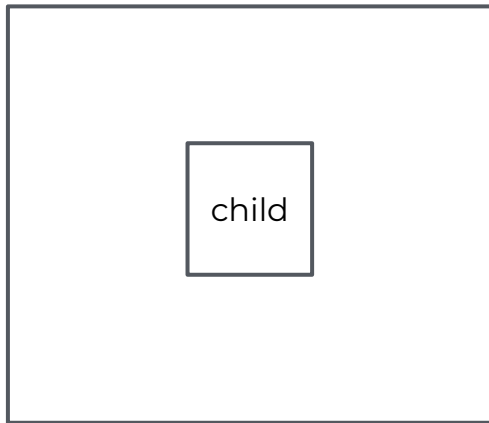
```
layout.setAlignItems(  
    FlexComponent.Alignment.STRETCH);
```



# Use case – centering

To center a child item, you can combine **Alignment** and **JustifyContentMode**

```
layout.setAlignItems(  
    FlexComponent.Alignment.CENTER);  
layout.setJustifyContentMode(  
    FlexComponent.JustifyContentMode.CENTER);
```



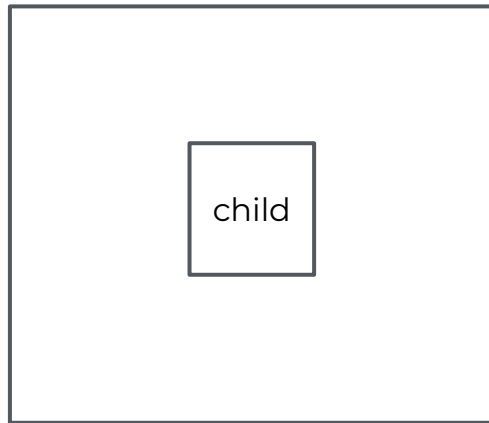
# Use case - centering

You can also use a CSS trick

```
child.getElement().getStyle()  
    .set("margin", "auto");
```

Or use LumoUtility to add a corresponding  
CSS class

```
child.addClassName(LumoUtility.Margin.AUTO);
```

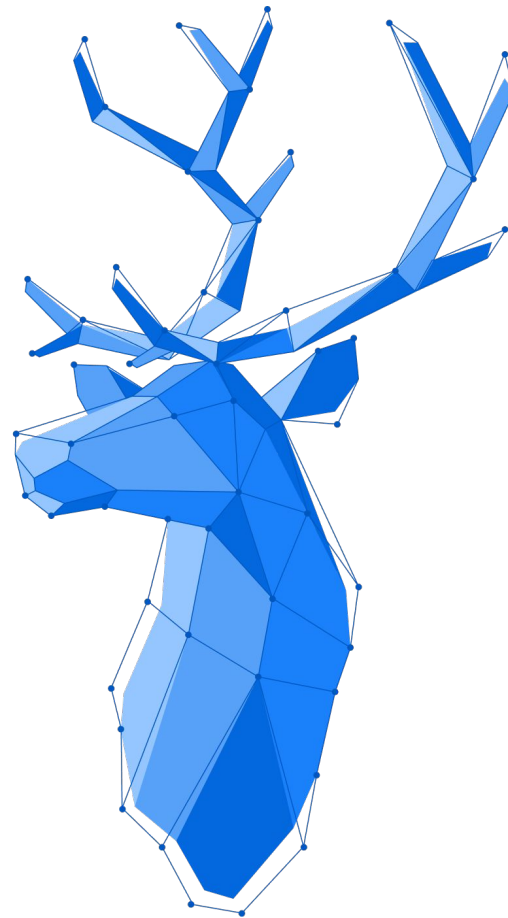


# Flexbox property summary

Property	Possible values	Description
<b>flex-direction</b>	row / column / column-reverse / row-reverse	Defines the main axis and the direction
<b>justify-content</b>	flex-start / flex-end / center / space-between / space-around / space-evenly	Distribution of space between and around items along the main axis
<b>align-items</b>	flex-start / flex-end / center / stretch / baseline	Alignment of items on the cross axis
<b>align-content</b>	flex-start / flex-end / center / stretch / space-between / space-around	Distribution of space between and around items along the cross axis
<b>align-self</b>	auto / flex-start / flex-end / center / baseline / stretch	Overrides an individual item's <b>align-items</b> value

# Summary, Part 2

- Flexbox and FlexLayout
- Alignment on the primary axis
- Alignment on the secondary axis
- Full height/width
- Centering



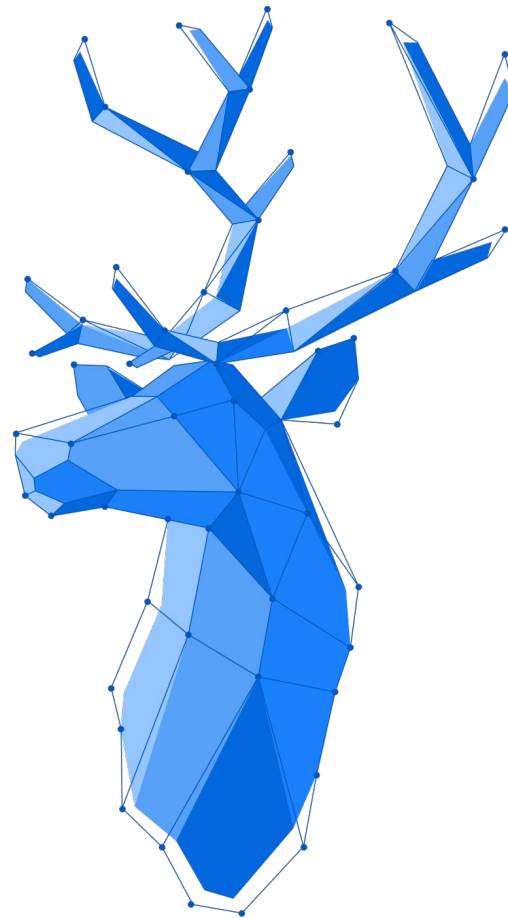
# Layouting, Part 3

Sizing in Flexbox



# Recap, parts 1 & 2

- Basic layouts (Horizontal, Vertical)
- Flexbox and FlexLayout
- Aligning items



# Sizing in Flexbox

# Sizing

There are convenient APIs for setting the size of a component

```
// set width/height of a component, e.g. setWidth/Height("200px"), setWidth/Height("100%")  
component.setWidth("200px");  
component.setHeight(100f, Unit.PERCENTAGE);
```

```
// A shorthand for setWidth/Height("100%")  
component.setWidthFull();  
component.setHeightFull();
```

# Sizing

There are convenient APIs for setting the size of a component

```
// Set the min/max height/width of a component, could be useful for responsive layouting  
component.setMinWidth("500px");  
component.setMinHeight(500, Unit.PIXELS);  
component.setMaxWidth(20, Unit.PERCENTAGE);  
component.setMaxHeight("1000em");
```

# Sizing

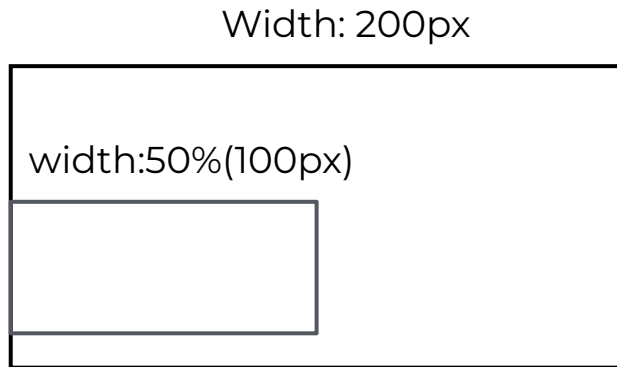
There are convenient APIs for setting the size of a component

```
// A shorthand for setWidth("100%") and setHeight("100%")  
component.setSizeFull();
```

```
// Remove the height and width of the component  
component.setSizeUndefined();
```

# Relative size

Relative size is relative to the parent component. E.g., 50% means 50% of the parent component's height/width.



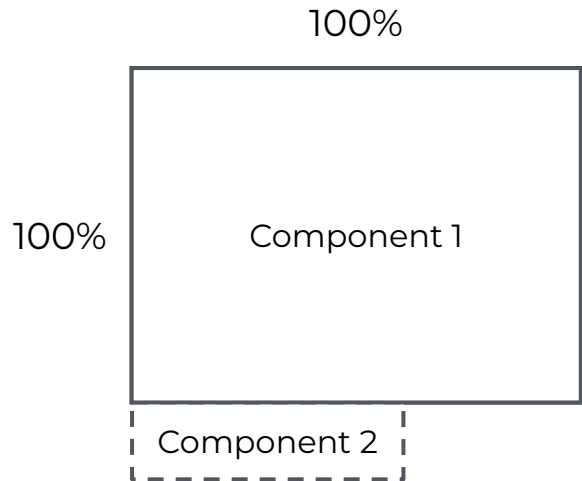
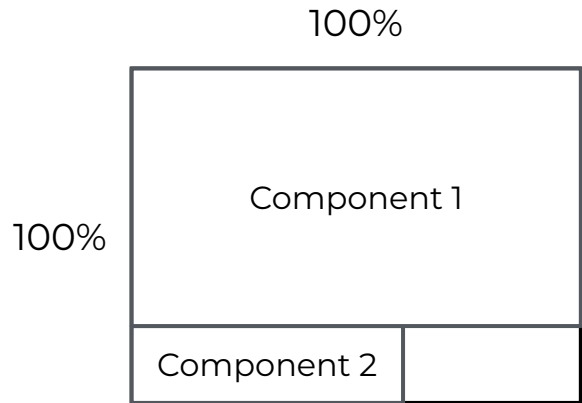
# Sizing

Given the code below, which layouting on the right side is correct?

```
VerticalLayout layout = ...  
layout.setSizeFull();
```

```
Component component1 = ...  
layout.add(component1);  
Component component2 = ...  
layout.add(component2);
```

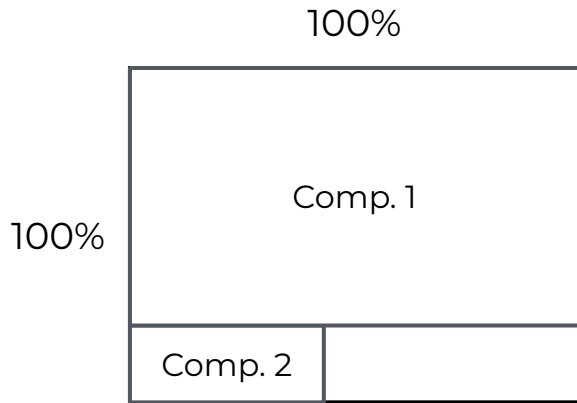
```
button1.setSizeFull();
```



# Sizing

Size is also affected by **flex-shrink** and **flex-grow**.

Note that it affects not only relative size but also absolute size, e.g. 100px.





# flex-shrink

It defines how items should shrink when there isn't enough space in the container.

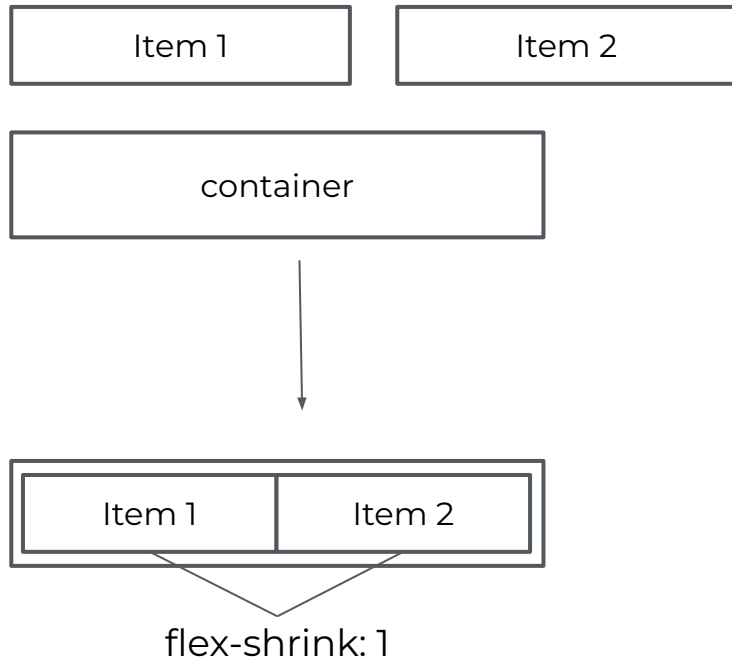


# flex-shrink

Setting flex-shrink to 1 means an item will shrink to fit into the container.

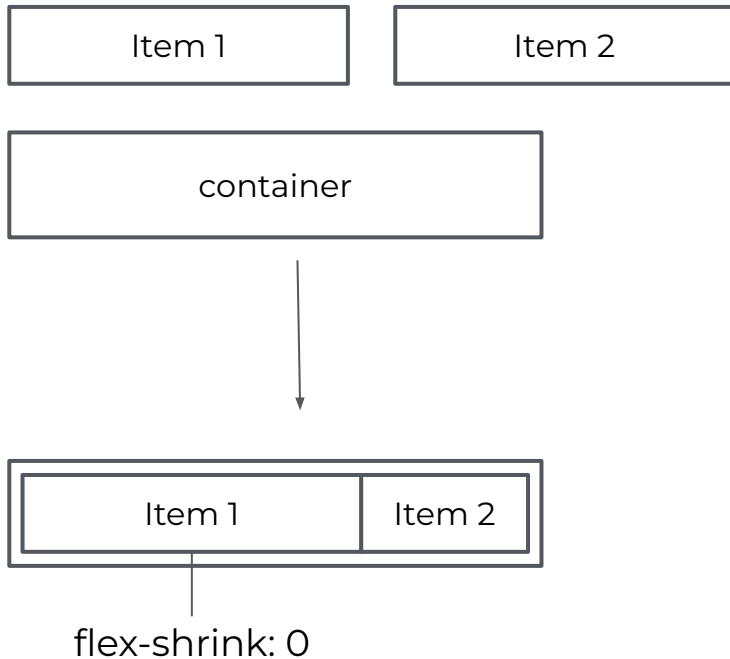
If multiple items have flex-shrink set to 1, they will all shrink equally.

This is the default value, but some Vaadin components overrides it.



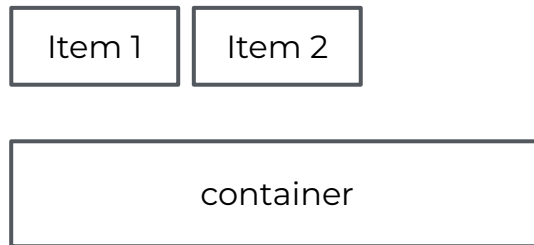
# flex-shrink

Setting flex-shrink to 0 will prevent an item from shrinking.



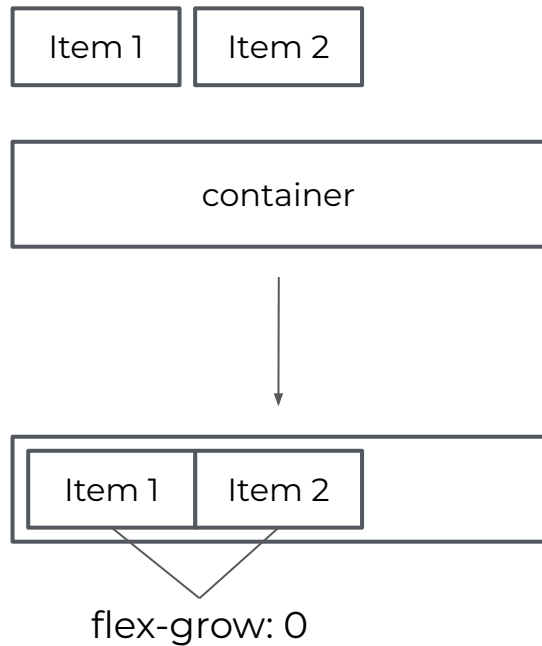
# flex-grow

Defines how to distribute **free space**, when the container is bigger than the size of the items.



# flex-grow

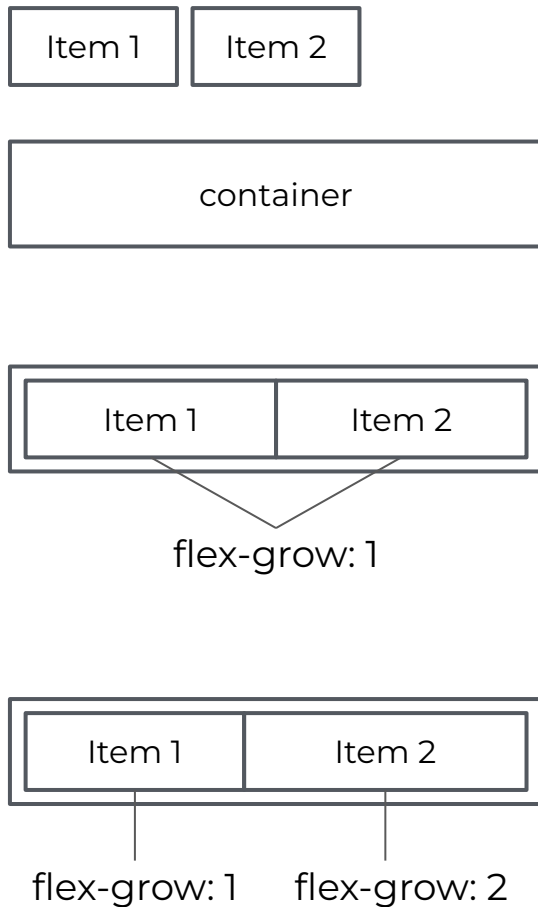
The default value is 0, which means an item won't take up any free space.



# flex-grow

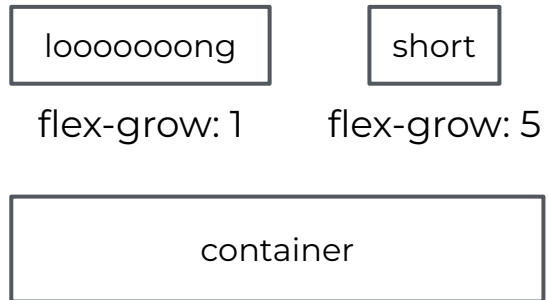
Define flex-grow: 1 for each item will make them grow equally.

Define flex-grow with a different value for each item will make them grow proportionally



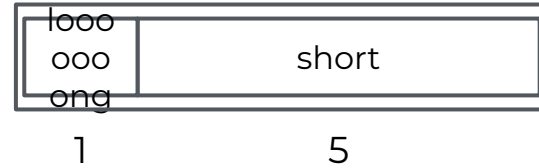
# Quiz

How will the items fit into the container?



# Quiz

**Free space** is distributed according to 1:5 ratio. The width of the items themselves have nothing to do with it.





# Java API for flex-grow

You can set the flex-grow of an item from the parent layout via the `setFlexGrow()` Java API

```
layout.setFlexGrow(3, item);
```

# Java API for flex-grow

There is also a shorthand method `expand()` for setting the flex-grow to 1.

```
layout.expand(item);
```

⇔

```
layout.setFlexGrow(1, item);
```

# Java API for flex-shrink

There is Java API for setting the flex-shrink for child components **on the layout**

```
layout.setFlexShrink(0, component);
```

This is the same as

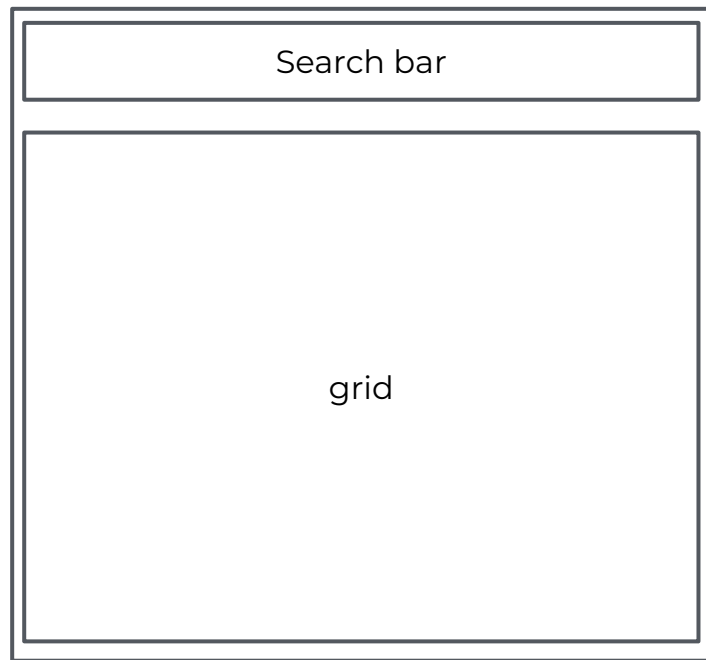
```
item.getElement().getStyle().set("flex-shrink", "0");
```

or

```
item.addClassName(LumoUtility.Flex.SHRINK_NONE);
```

# Use case #1

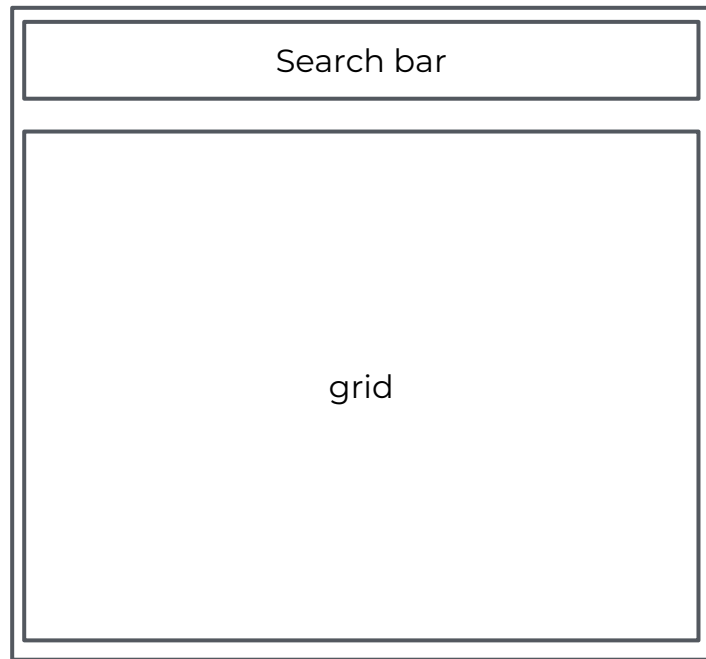
How to expand a component?



# Use case #1

How to expand a component?

```
layout.setAlignItems(Alignment.STRETCH);  
layout.expand(grid);  
layout.setSizeFull();
```



## Use case #2

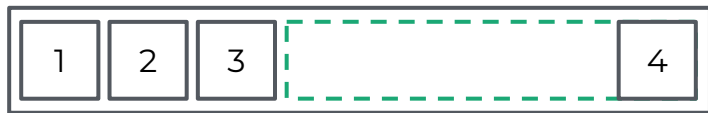
How to implement layout like this?



## Use case #2

One way is to wrap item 4 into a FlexLayout.

```
FlexLayout wrapper = new FlexLayout(item4);  
layout.expand(wrapper);  
wrapper.setJustifyContentMode(  
    FlexComponent.JustifyContentMode.END)  
;
```



## Use case #2

You could also use CSS:

```
child4.getStyle().set(  
    "margin-left", "auto");
```

Or use LumoUtility CSS classes:

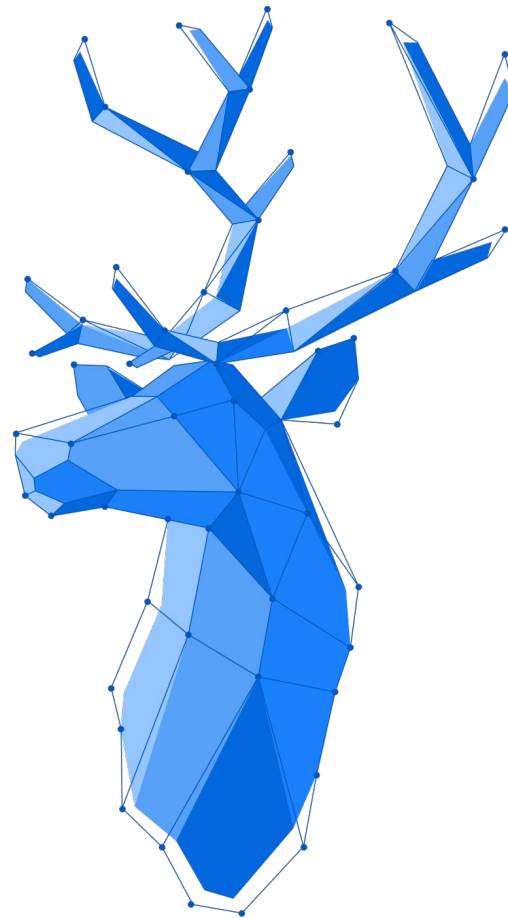
```
child4.addClassName(LumoUtility.Margin.Left.AUTO);
```





# Summary, Part 3

- Sizing
- Flex-shrink
- Flex-grow

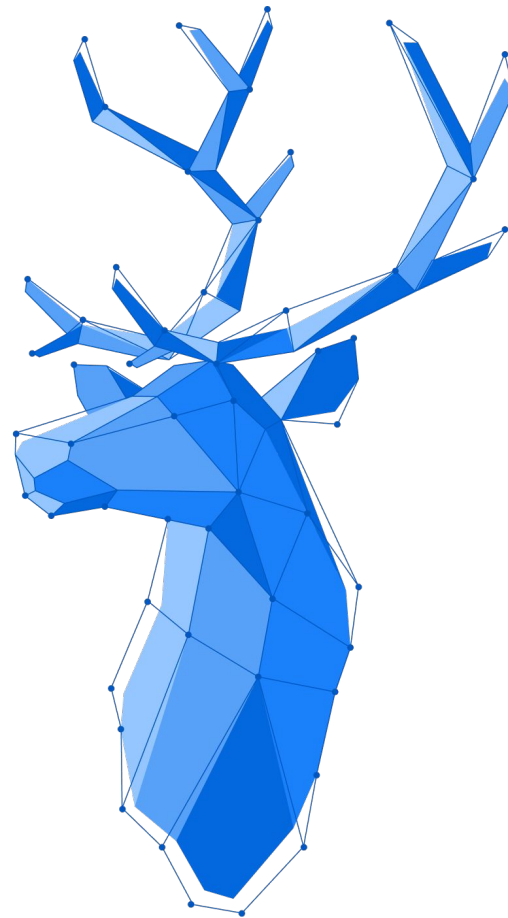


# Layouting, Part 4

Wrapping items in Flexbox

# Recap, parts 1 – 3

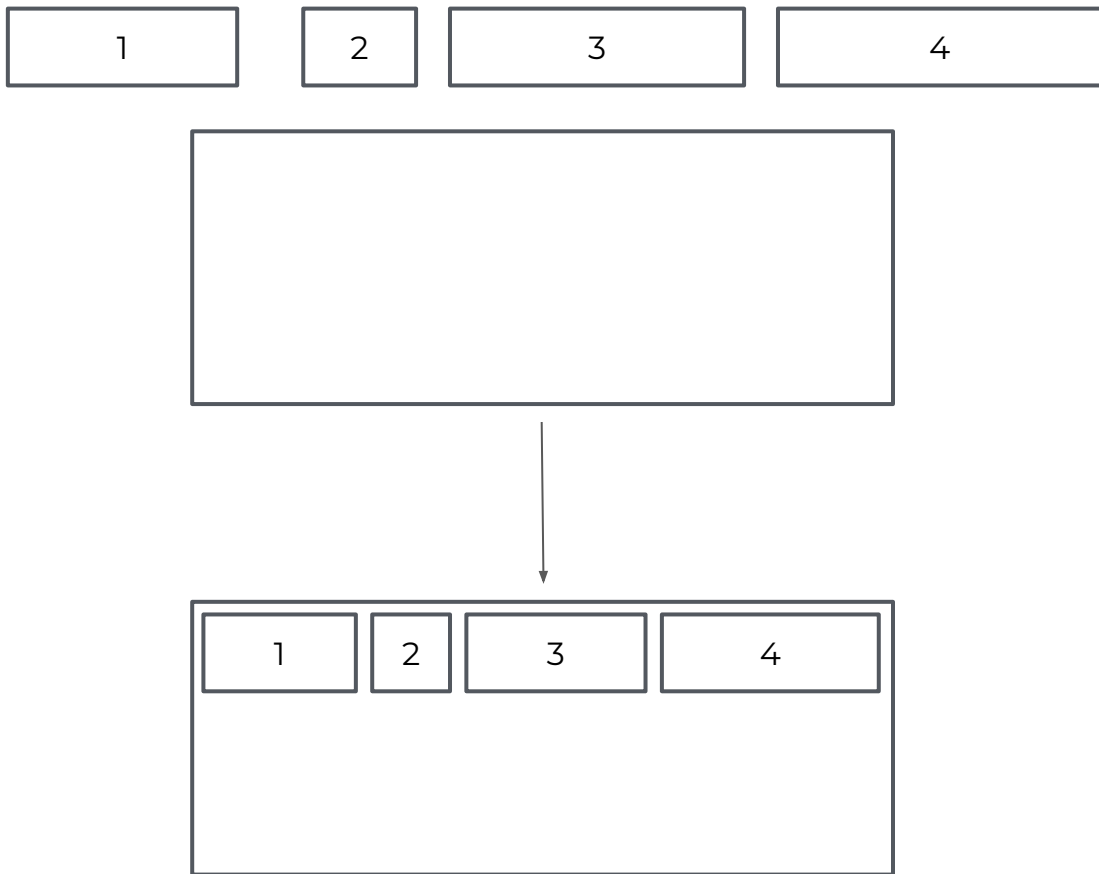
- Basic layouts (Horizontal, Vertical)
- Flexbox and FlexLayout
- Aligning items
- Sizing
- Flex-shrink & Flex-grow



# Wrapping items in Flexbox

# Wrap items

By default, items will shrink themselves to fit into one line, even though there is enough space to start a new line.

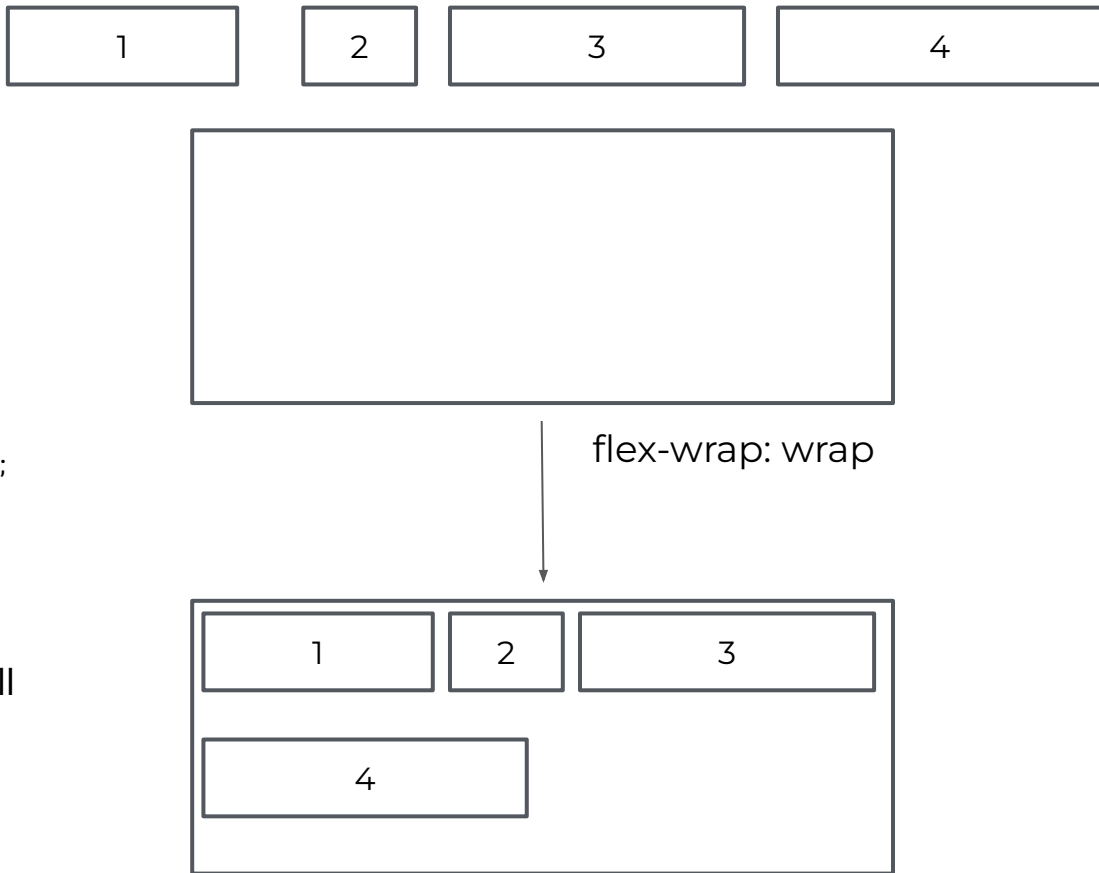


# Wrap items

You can configure the layout by setting **flex-wrap** to **wrap** to make the items wrap into new lines.

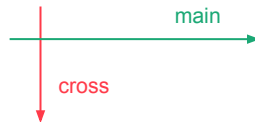
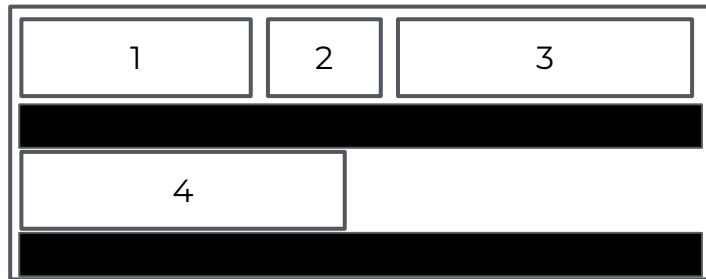
```
layout.setFlexWrap(FlexLayout.FlexWrap.WRAP);
```

Note: while you can use wrapping with `HorizontalLayout` and `VerticalLayout` through CSS, they will probably look bad if you have spacing enabled - better use `FlexLayout` if you need wrapping.



# Free space between lines

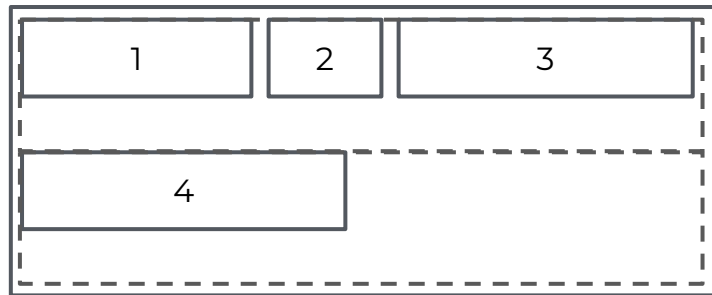
When there are multiple lines in the container, you can decide how to distribute the free space between the lines with the **align-content** property.



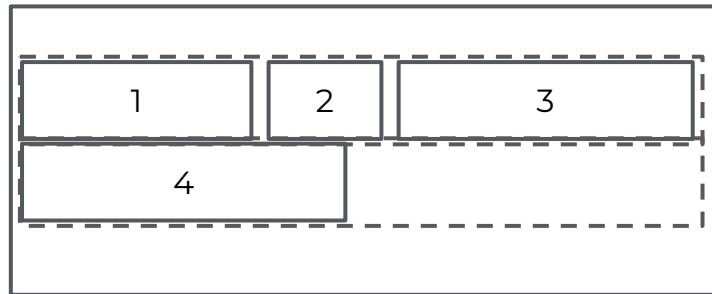
# align-content

**stretch:** The **default** value. Each row (dashed lines in the picture) will stretch equally to take free space. There will be space inside the row if the items are smaller than the row.

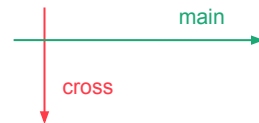
**center:** rows (dashed lines in the picture) are packed at the center of the container.



**align-content: stretch**



**align-content: center**

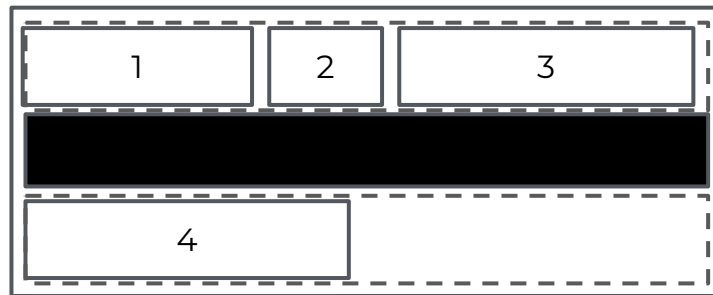




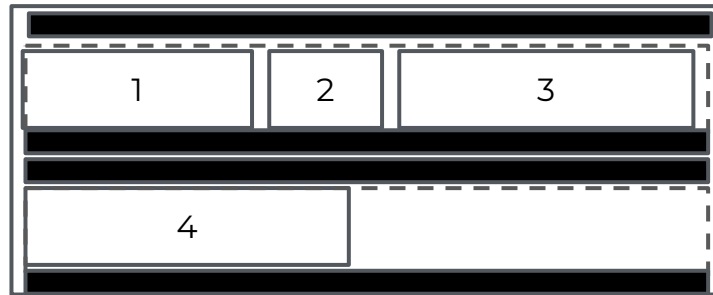
# align-content

**space-between:** rows are evenly distributed; the first row is at the start of the container while the last row is at the end.

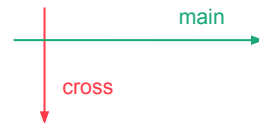
**space-around:** rows are evenly distributed with equal space around them.



**align-content: space-between**



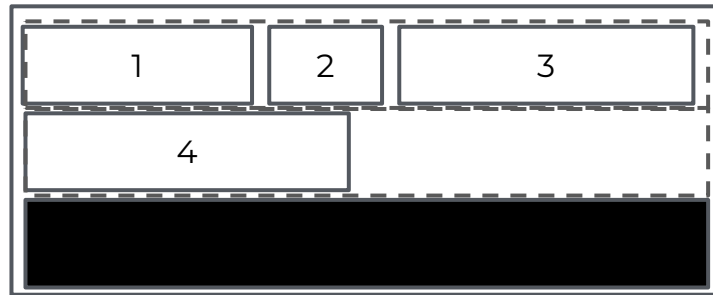
**align-content: space-around**



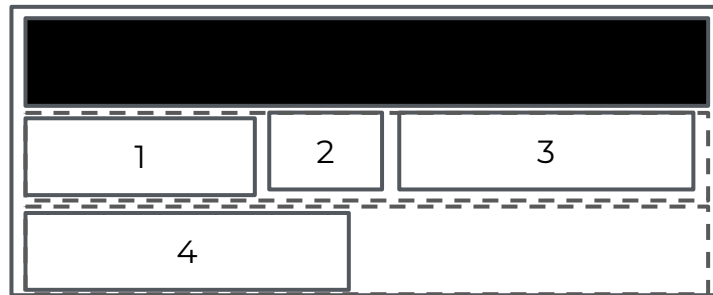
# align-content

**flex-start:** rows are packed at the start of the container.

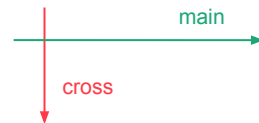
**flex-end:** rows are packed at the end of the container.



**align-content: flex-start**



**align-content: flex-end**



# Java API

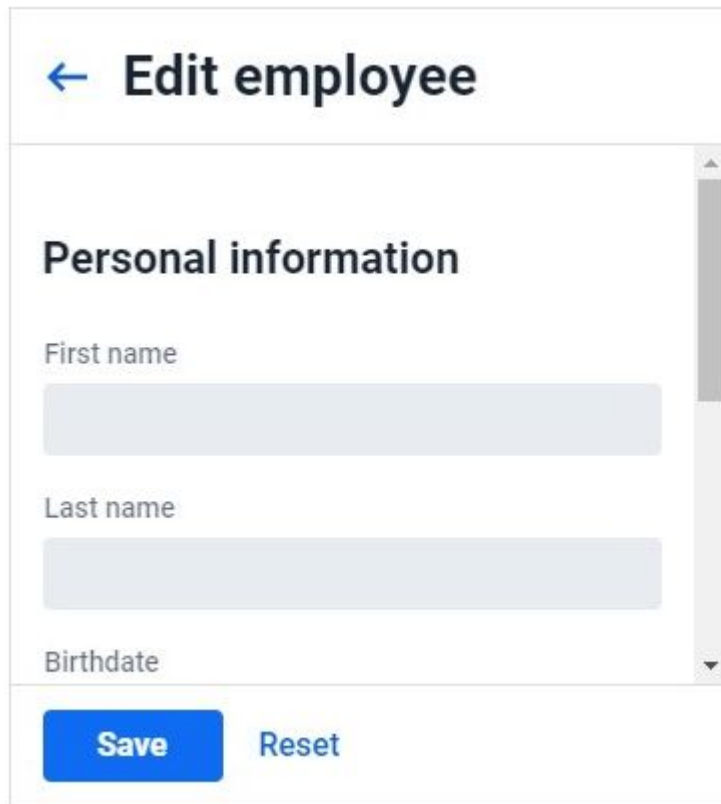
There is a Java API for FlexLayout to set the align-content

```
flexLayout.setAlignContent(FlexLayout.ContentAlignment.START);  
flexLayout.setAlignContent(FlexLayout.ContentAlignment.END);  
flexLayout.setAlignContent(FlexLayout.ContentAlignment.CENTER);  
flexLayout.setAlignContent(FlexLayout.ContentAlignment.SPACE_BETWEEN);  
flexLayout.setAlignContent(FlexLayout.ContentAlignment.SPACE_AROUND);  
flexLayout.setAlignContent(FlexLayout.ContentAlignment.STRETCH);
```

# Scroller

Scroller is a component container for creating scrollable areas in the UI.

Scroller has four different scroll directions: **vertical**, **horizontal**, **both**, and **none**. Scroller's default scroll direction is **both**.

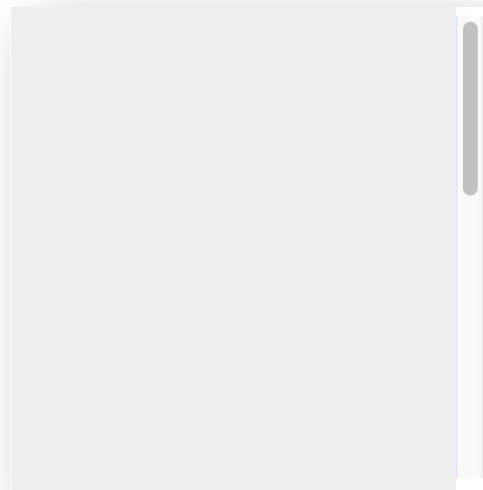


The image shows a web form titled "Edit employee" with a blue back arrow icon. The form contains a section titled "Personal information" with three input fields: "First name", "Last name", and "Birthdate". A vertical scrollbar is visible on the right side of the form, indicating that the content is scrollable. At the bottom of the form, there are two buttons: a blue "Save" button and a "Reset" button.

# Scrolling

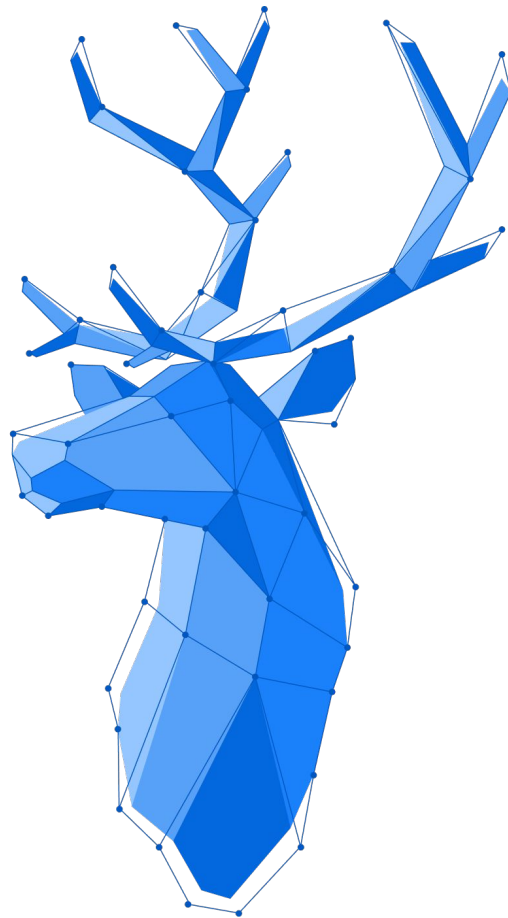
A scrolling container can also be implemented with Element API and/or CSS:

```
//enable vertical scroll bar  
layout.getStyle().set("overflow-y", "auto");  
//enable horizontal scroll bar  
layout.getStyle().set("overflow-x", "auto");  
//enable both horizontal and vertical scroll bars  
layout.getStyle().set("overflow", "auto");
```



# Summary, Part 4

- Wrapping items
- Aligning wrapped items
- Scrolling

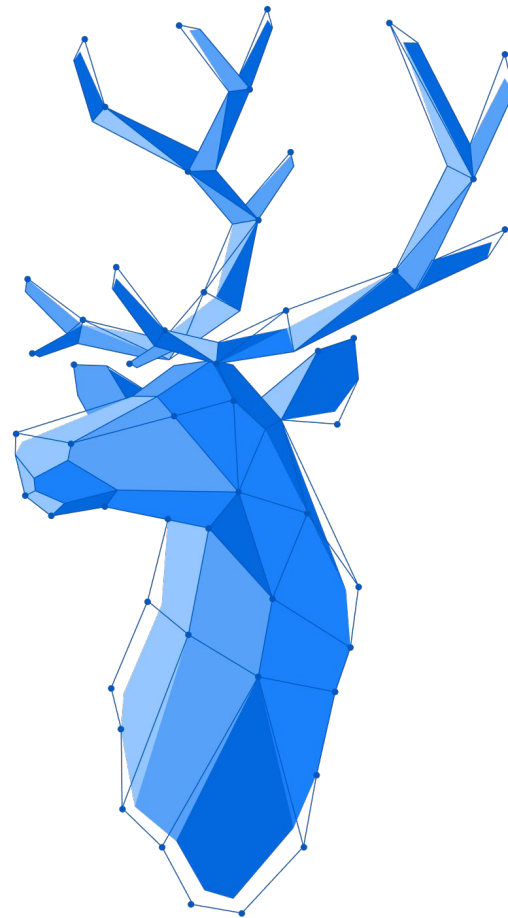


# Layouting, Part 5

Exercise

# Recap, parts 1 – 4

- Basic layouts (Horizontal, Vertical)
- Flexbox and FlexLayout
- Aligning and sizing
- Flex-shrink & Flex-grow
- Wrapping items





# Exercise 1

Compose an application layout

# Layouting, Part 3

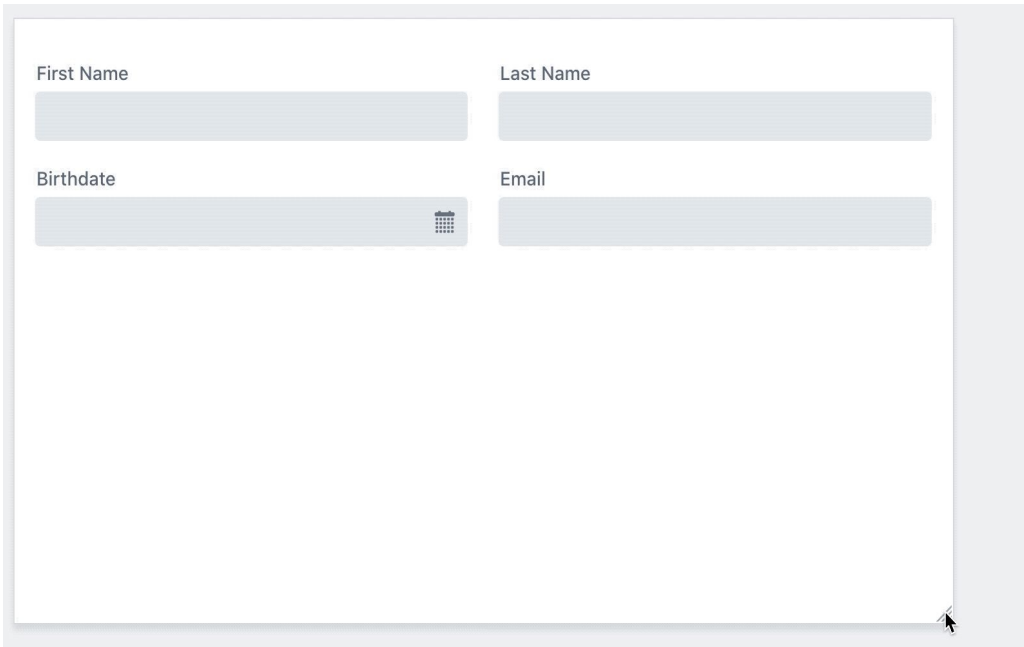
FormLayout

# FormLayout

A responsive layout designed for displaying forms

# Responsive

One significant benefit of using `FormLayout` is that it's responsive out of the box.



A screenshot of a responsive form layout. The form is contained within a light gray border. It features four input fields arranged in a 2x2 grid. The top row contains 'First Name' and 'Last Name'. The bottom row contains 'Birthdate' and 'Email'. Each field is represented by a light gray rectangular input box. The 'Birthdate' field includes a small calendar icon on its right side. A mouse cursor is visible at the bottom right corner of the form container.

First Name	Last Name
Birthdate	Email

# Responsive

The responsiveness works by showing a different number of columns depending on the width of the Formlayout.

First Name <input type="text"/>	Last Name <input type="text"/>
Birthdate <input type="text"/>	Email <input type="text"/>

Column 1

Column 2

# Responsive

By default, it shows 2 columns when the width is more than 40em, only 1 column otherwise.

First Name	Last Name
<input type="text"/>	<input type="text"/>
Birthdate	Email
<input type="text"/>	<input type="text"/>

Column 1

Column 2

# Responsive Step

The number of columns can also be customised via setting the responsive steps.

First Name	Last Name
<input type="text"/>	<input type="text"/>
Birthdate	Email
<input type="text"/>	<input type="text"/>

Column 1

Column 2

# Responsive Step

When the width is more than the [first parameter], then there should be [second parameter] columns.

```
/**  
 * Show 2 columns when the width is >= 50em.  
 * Show 1 column when the width is [0-50) em  
 */  
formLayout.setResponsiveSteps(  
    new ResponsiveStep("0", 1),  
    new ResponsiveStep("50em", 2));
```

First Name	Last Name
<input type="text"/>	<input type="text"/>
Birthdate	Email
<input type="text"/>	<input type="text"/>

Column 1

Column 2



# Add components

The first way of adding components to a `FormLayout` is to use the **`add()`** method, as in any other type of layouts.

In this way, a **label** is set for the added component and displays on **top** of the component.

```
FormLayout formLayout = new FormLayout();
```

```
TextField firstName = new  
TextField("First Name");  
formLayout.add(firstName);
```

First Name	Last Name
<input type="text"/>	<input type="text"/>
Birthdate	Email
<input type="text"/>	<input type="text"/>

Column 1

Column 2

# Add components

The second way of adding components to a `FormLayout` is to use the **`addFormItem()`** method

In this way, the added component is wrapped into a **form item**, and the label should be set for the form item.

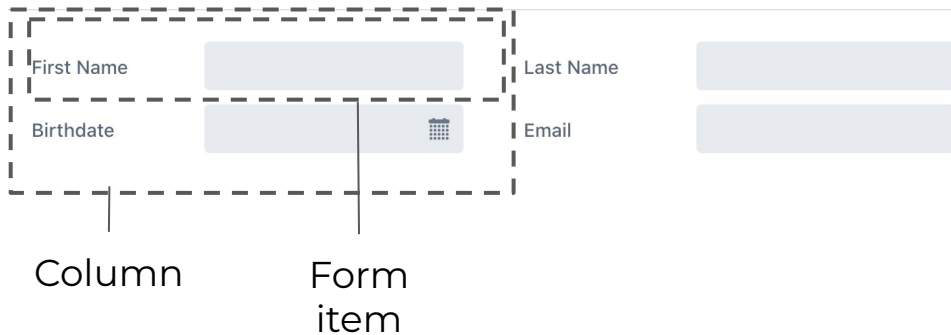
A **label** displays to the **left** side of a component.

```
FormLayout formLayout = new FormLayout();
```

```
TextField firstName = new TextField();
```

```
FormItem formItem =
```

```
    formLayout.addFormItem(firstName, "First Name");
```

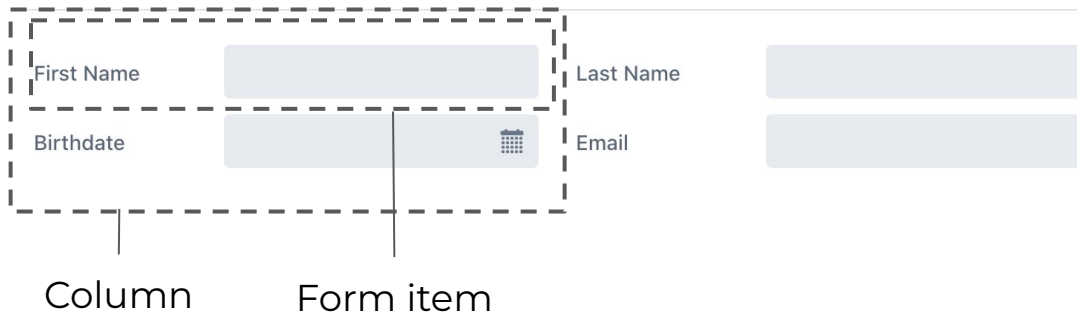


# Add components

When using `addFormItem()`, it's normally good to set full width for the component, so the component will take the full width of the available space in the form item.

```
FormLayout formLayout = new FormLayout();
```

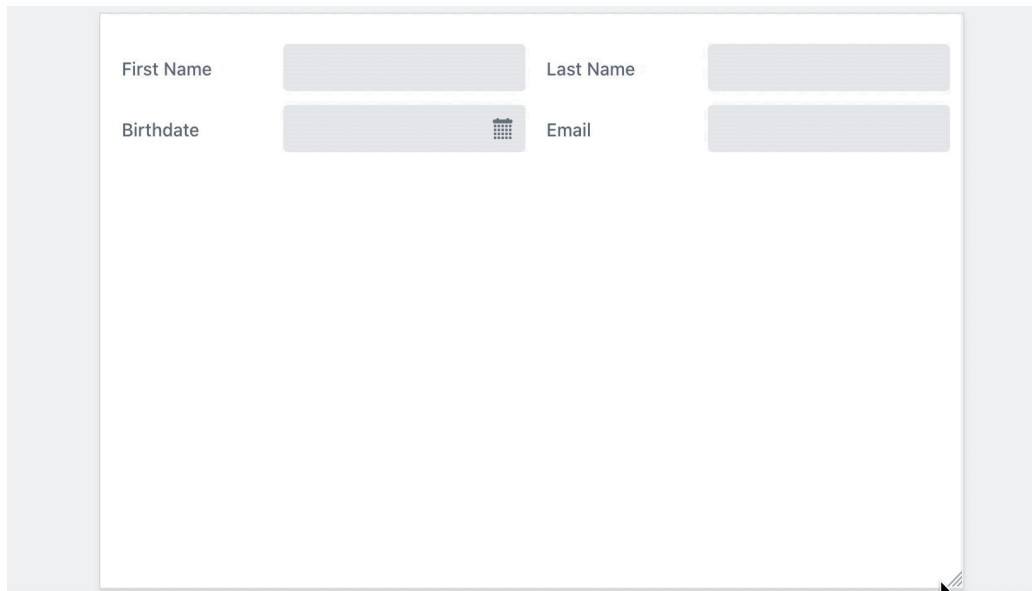
```
TextField firstName = new TextField();  
formLayout.addFormItem(firstName, "First  
Name");  
firstName.setWidthFull();  
...
```



# Responsive label position

The most significant benefit of using `addFormItem()` is that the position of the labels can be changed responsively.

The default is that when the width is less than 20em, the labels will come to the top of the components.



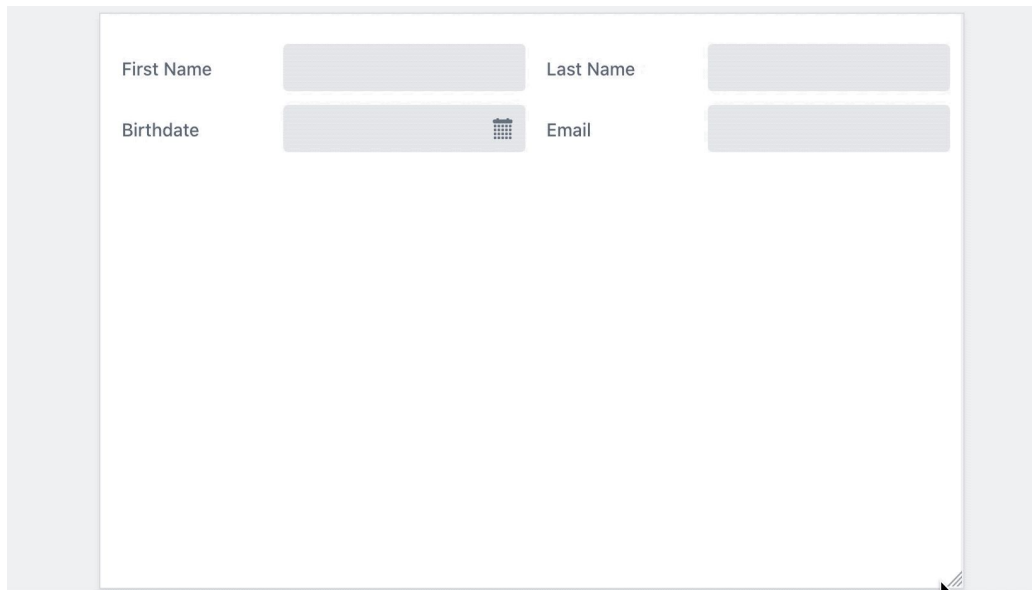
The image displays a form with four input fields arranged in a 2x2 grid. The labels are positioned above the input fields. The labels are 'First Name', 'Last Name', 'Birthdate', and 'Email'. The input fields are light gray rectangles. The 'Birthdate' input field has a small calendar icon to its right. The form is set against a light gray background.

First Name	<input type="text"/>	Last Name	<input type="text"/>
Birthdate	<input type="text"/>	Email	<input type="text"/>

# Responsive label position

The position of the labels can also be customised via setting the responsive steps.

```
/**  
 * Show 2 columns when the width is >= 50em.  
 * Show 1 column with label on the left side  
 * when the width is [20-50) em.  
 * Show 1 column with label on the top  
 * when the width is [0-20) em.  
 */  
f1.setResponsiveSteps(  
    new ResponsiveStep("0", 1,  
        LabelsPosition.TOP),  
    new ResponsiveStep("20em", 1),  
    new ResponsiveStep("50em", 2));
```



# Add complex components

Components can be grouped into a layout and then add the layout to the `FormLayout`.

First Name	<input type="text"/>	Last Name	<input type="text"/>
Birthdate	<input type="text"/>	Email	<input type="text"/>
Phone	<input type="text"/>	<input type="checkbox"/>	Do not call

```
FormLayout formLayout = new FormLayout();

FlexLayout phoneLayout = new FlexLayout();
phoneLayout.setWidthFull();
TextField phone = new TextField();
Checkbox noCall = new Checkbox("Do not call");
phoneLayout.add(phone, noCall);
phoneLayout.expand(phone);
phoneLayout.setAlignItems(Alignment.CENTER);

formLayout.addFormItem(phoneLayout, "Phone");
...
```

# Add complex components

Could also let a component span multiple columns by using **`formLayout.setColspan()`**

First Name	<input type="text"/>	Last Name	<input type="text"/>
Birthdate	<input type="text"/>	Email	<input type="text"/>
Phone	<input type="text"/>		
			<input type="checkbox"/> Do not call

When using `add()`, set column span on the **component**.

When using `addFormItem()`, set column span on the **form item**.

```
formLayout.setColspan(formItem, 2);  
formLayout.setColspan(component, 2);
```

If your column span doesn't seem to be working, it's probably because you set it on the component instead of the form item.

# Force a new row

Sometimes, instead of showing a component on the second column, you might want to force it to a new row.

You can achieve this by adding a `<br>`.

```
PasswordField password = new
PasswordField();
formLayout.addFormItem(password,
"Password");

formLayout.getElement().appendChild(
    ElementFactory.createBr());

PasswordField repeatPwd = new
PasswordField();
formLayout.addFormItem(
    repeatPwd, "RepeatPassword");
```

First Name	<input type="text"/>	Last Name	<input type="text"/>
Birthdate	<input type="text"/>	Email	<input type="text"/>
Password	<input type="password"/>		
RepeatPassword	<input type="password"/>		

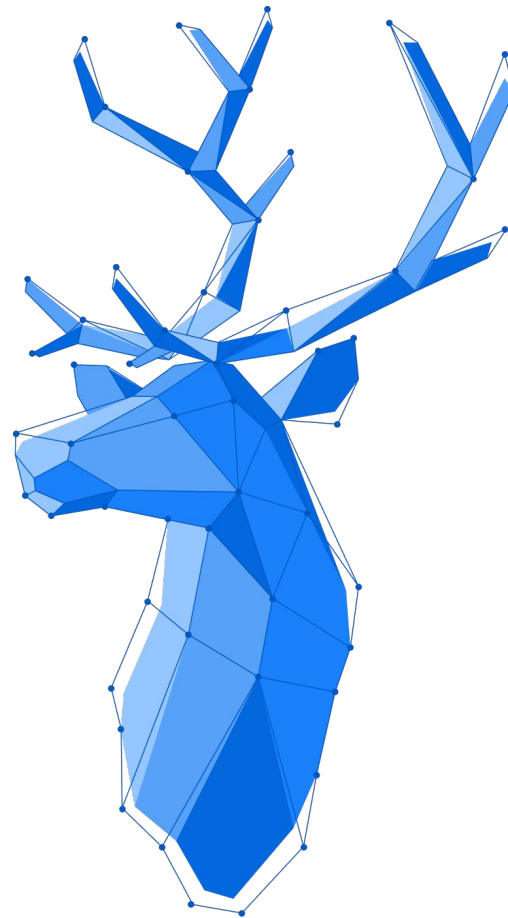


# Exercise 2

Build a form with `FormLayout`

# Summary, Part 5

- FormLayout
- Responsive Steps
- Label position
- Complex components
- Exercise

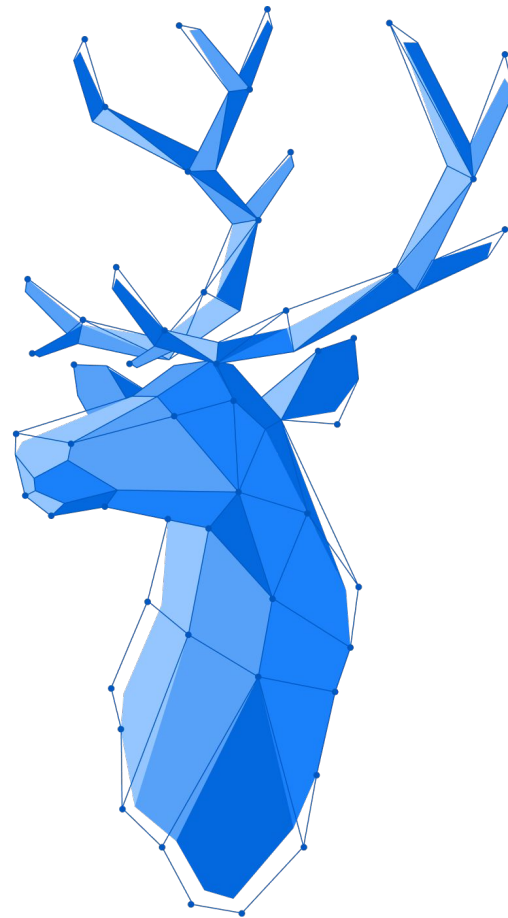


# Layouting, Part 4

Board and AppLayout

# Recap, parts 1 – 5

- Basic layouts (Horizontal, Vertical)
- Flexbox and FlexLayout
- FormLayout



# Vaadin Board

Automatic responsive layout

# Vaadin Board

Vaadin Board is a **commercial** component.

Start a **free trial** by clicking on the prompt in the browser when seeing one

Click to validate your Vaadin Subscription

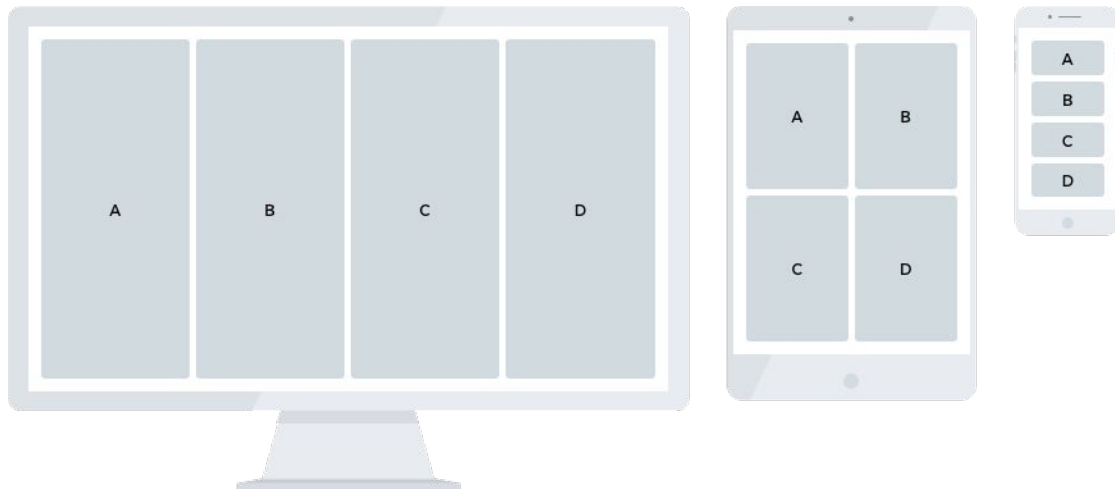


# Vaadin Board

Vaadin Board is based on rows and columns.

You add components to the rows, Vaadin will make it responsive for you.

```
Board board = new Board();  
Component a = createComponent("A");  
Component b = createComponent("B");  
Component c = createComponent("C");  
Component d = createComponent("D");  
  
board.addRow(a, b, c, d);
```

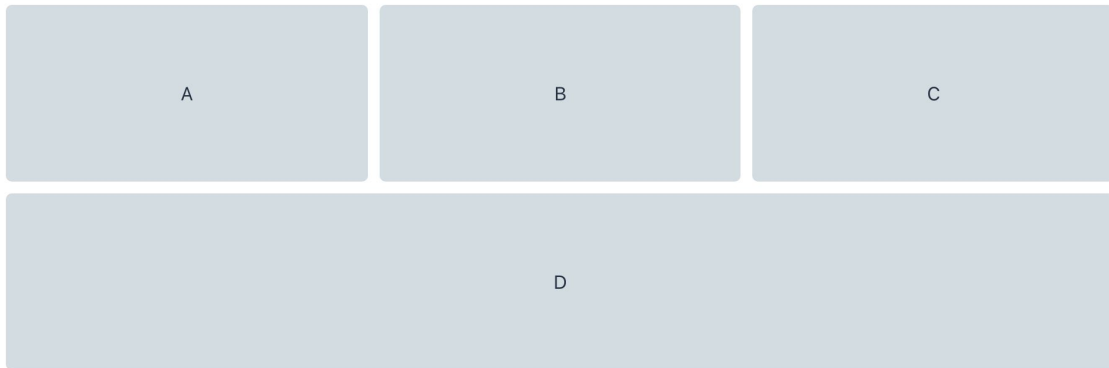


# Vaadin Board

You can add multiple rows.

```
Board board = new Board();  
Component a = createComponent("A");  
Component b = createComponent("B");  
Component c = createComponent("C");  
Component d = createComponent("D");
```

```
board.addRow(a, b, c);  
board.addRow(d);
```





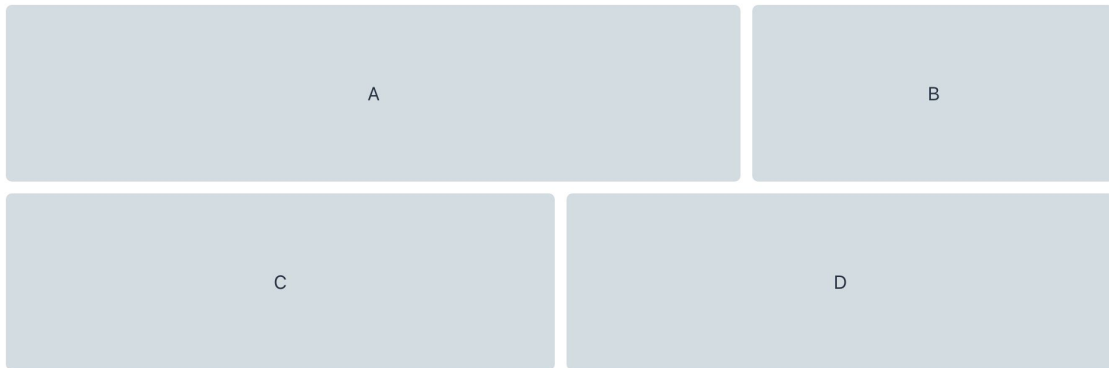
# Vaadin Board

You can set a column span.

```
Board board = new Board();  
Component a = createComponent("A");  
Component b = createComponent("B");  
Component c = createComponent("C");  
Component d = createComponent("D");
```

```
Row row1 = board.addRow(a, b);  
row1.setComponentSpan(a, 2);
```

```
board.addRow(c, d);
```



# Vaadin Board

Limitation: A row can only have up to **4** columns.

```
Board board = new Board();
Component a = createComponent("A");
Component b = createComponent("B");
Component c = createComponent("C");
Component d = createComponent("D");
Component e = createComponent("E");
board.addRow(a, b, c, d, e);
```

```
Caused by: java.lang.IllegalArgumentException: A row can only contain 4 columns
    at com.vaadin.flow.component.board.Row.throwIfTooManyColumns(Row.java:175)
    at com.vaadin.flow.component.board.Row.add(Row.java:99)
    at com.vaadin.flow.component.board.Row.<init>(Row.java:84)
    at com.vaadin.flow.component.board.Board.addRow(Board.java:78)
    at com.vaadin.starter.skeleton.MainView.<init>(MainView.java:109)
    ... 53 more
```

# Exercise 3

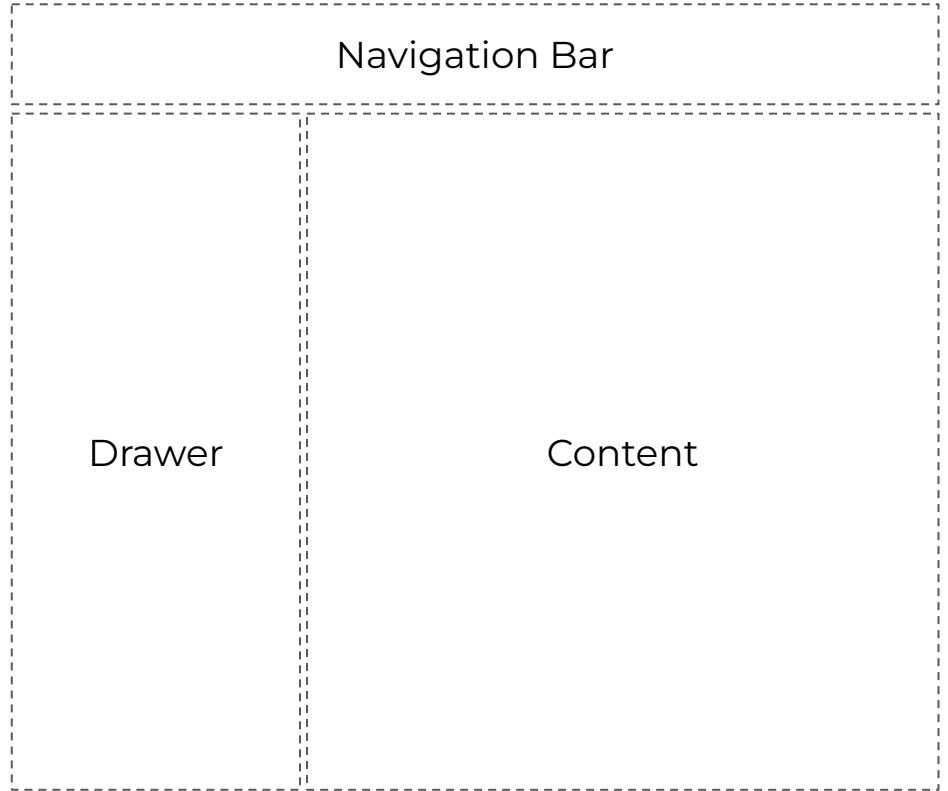
Use Board

# App Layout

A quick and easy way to get a common application layout

# App Layout

An App Layout contains 3 parts: the navigation bar, the drawer and the content.

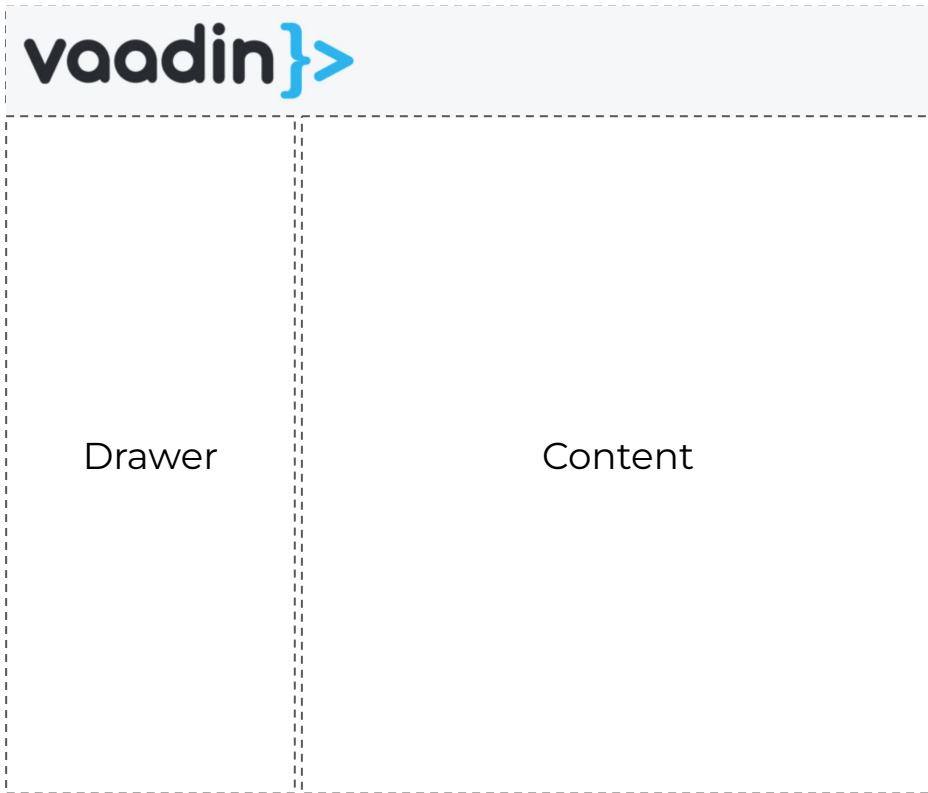


# App Layout

Add components to the navigation bar

```
AppLayout appLayout = new AppLayout();
```

```
Image img = new Image("logo-url", "Logo");  
appLayout.addToNavbar(img);
```

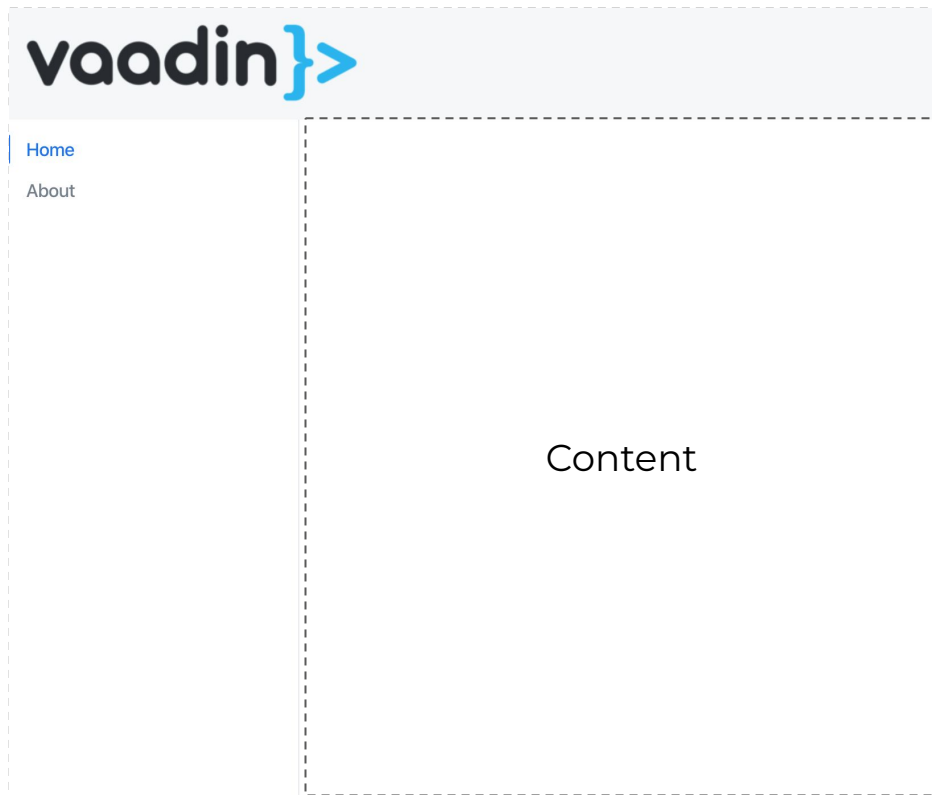


# App Layout

Add components to the drawer

```
AppLayout appLayout = new AppLayout();
```

```
Tabs tabs = new Tabs(  
    new Tab("Home"), new Tab("About"));  
tabs.setOrientation(Tabs.Orientation.VERTICAL);  
appLayout.addToDrawer(tabs);
```

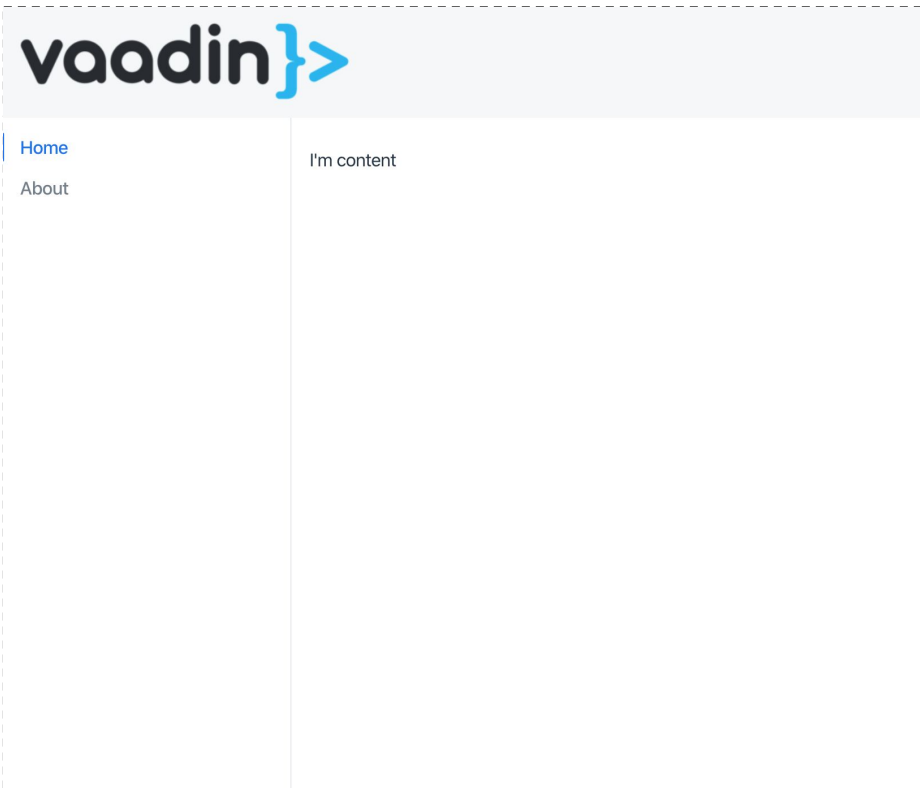


# App Layout

Set content

```
AppLayout appLayout = new AppLayout();
```

```
Component content = new Paragraph("I'm  
content");  
appLayout.setContent(content);
```

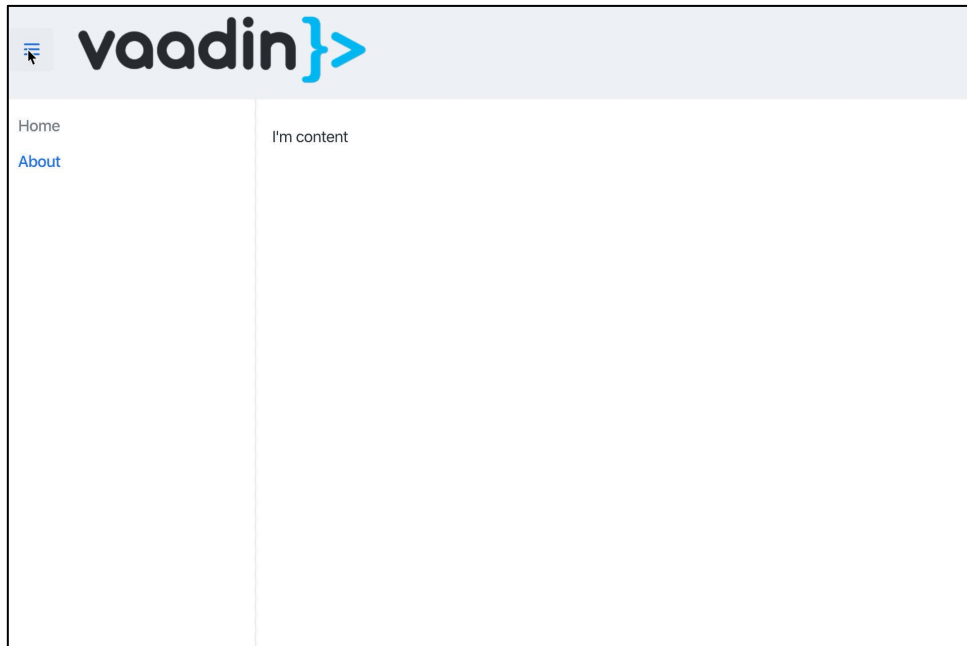




# App Layout

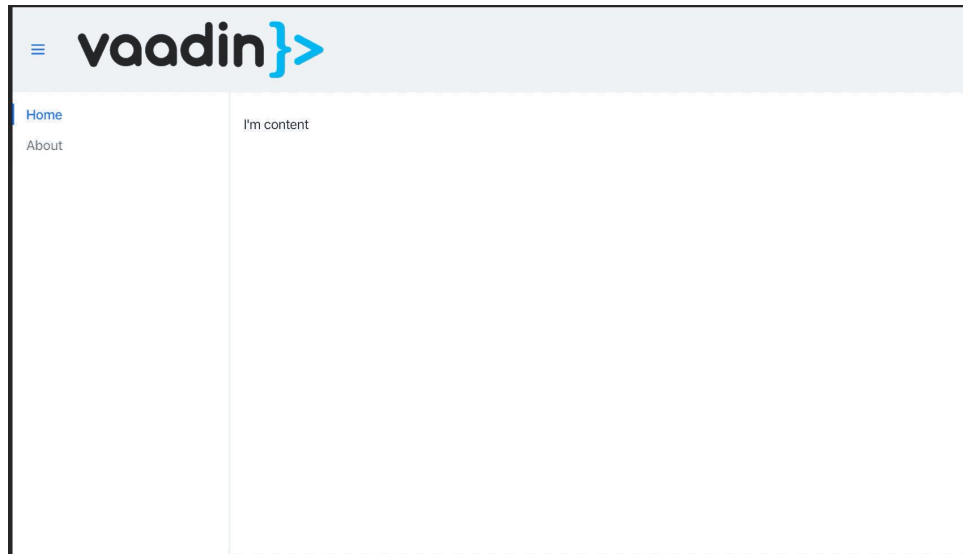
Add a drawer toggle button to control the visibility of the drawer.

```
AppLayout appLayout = new AppLayout();  
appLayout.addToNavbar(new DrawerToggle(),  
    new Image("logo-url", "Logo"));
```



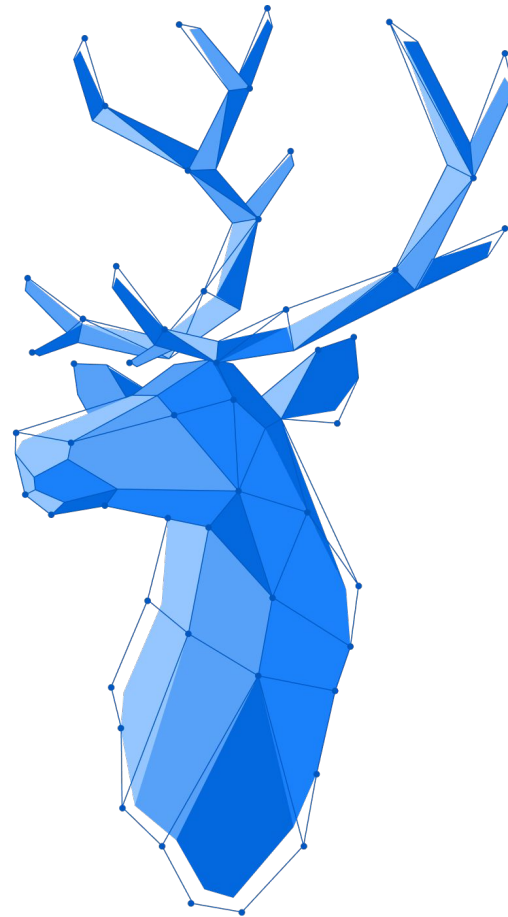
# App Layout

It's responsive out-of-the-box



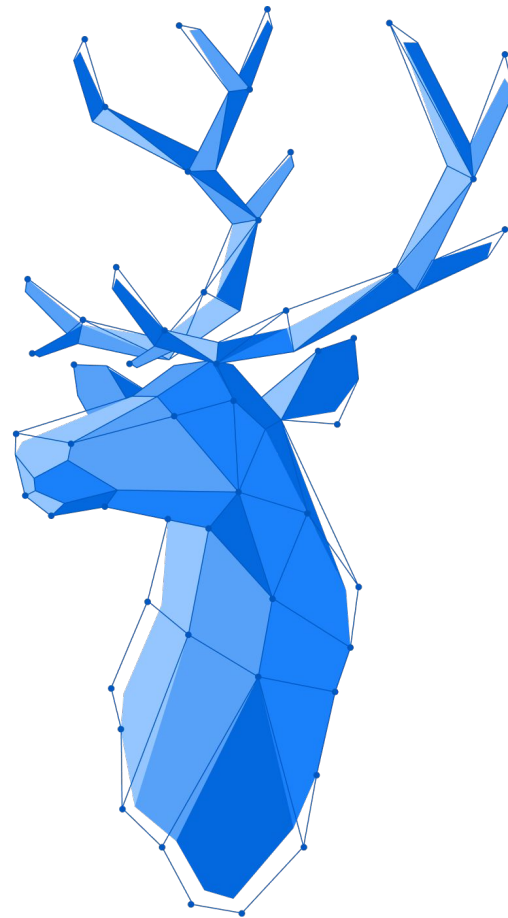
# Summary, Part 5

- Vaadin Board
- AppLayout



# Summary for Layouting

- `HorizontalLayout` & `VerticalLayout`
- `Flexbox` & `FlexLayout`
- `FormLayout`
- Vaadin Board
- App Layout



**Thank you!**

