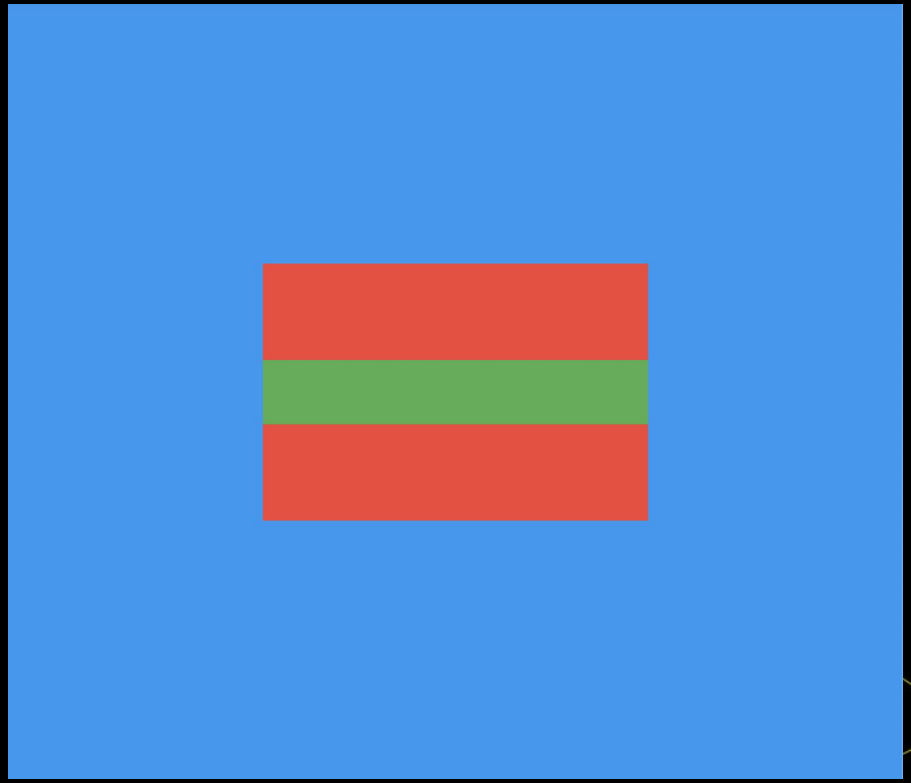
LeanCode

Programming Mobile Applications in Flutter

Layouting 2

Lecture 2 recap

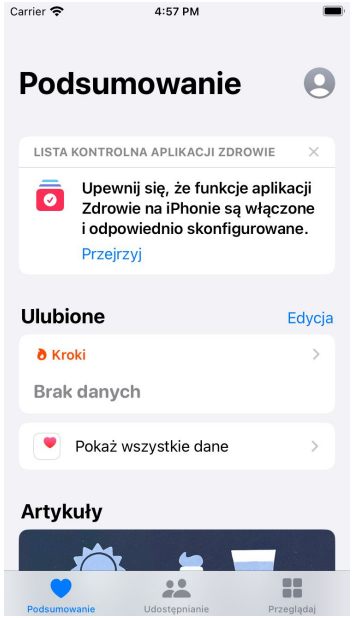
LeanCode

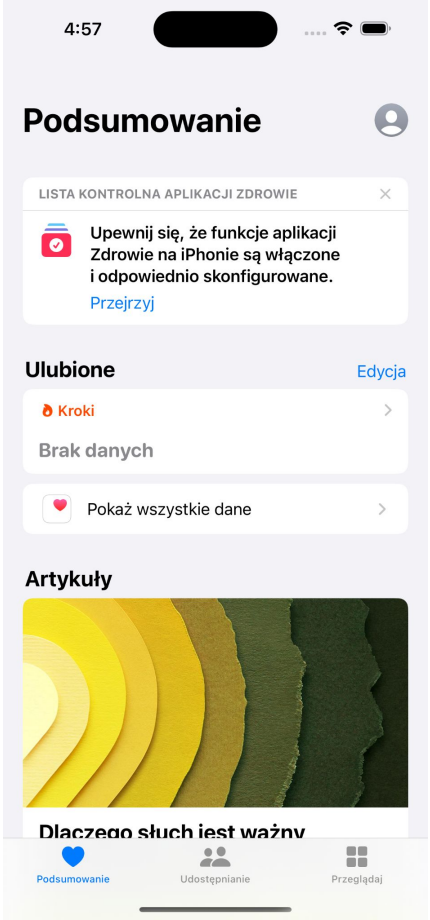


LeanCode

```
child: Container(
  constraints: BoxConstraints.tight(const Size(300, 200)),
  color: Colors.red,
  child: Align(
    alignment: const Alignment(1,0),
    child: Container(
      width: 350,
      height: 50,
      color: Colors.green,
    ),
  ),
),
```

Flexible layout

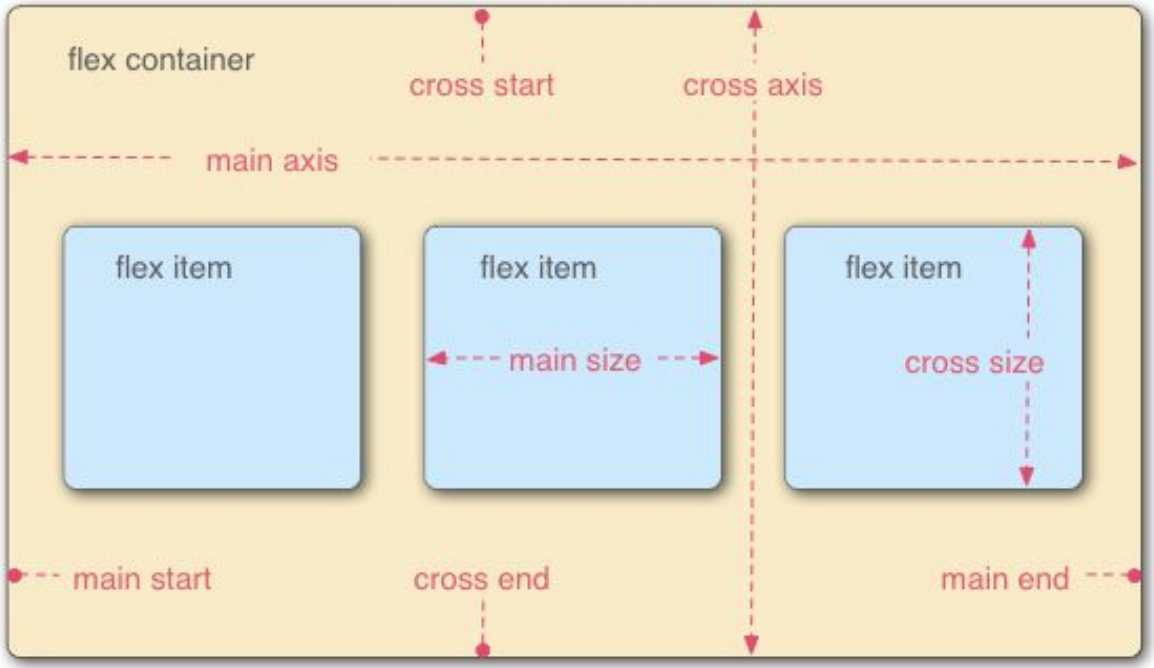




Flex

Flex

- A *Widget*
- *Container* has the ability to alter its items' width/height to best fill the available space
- Two axes
  - Main
  - Cross
- A weight



Source: https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Flexbox

Row

- A *Widget* that displays its children in a horizontal array
- Extends *Flex*
- Doesn’t scroll
    - It is considered an error to have more children in a *Row* than will fit in the available room

Column

A *Widget* that displays its children in a vertical array

Does every widget inside *Flex* must have a flex factor?

No

Row

Row

Row’s layout algorithm

1. Layout each child a null or zero flex factor with unbounded horizontal constraints and the incoming vertical constraints
2. Divide the remaining horizontal space among the children with non-zero flex factors according to their flex factor
3. Layout each of the remaining children with the same vertical constraints as in step 1, but instead of using unbounded horizontal constraints, use horizontal constraints based on the amount of space allocated in step 2.
4. The height of the *Row* is the max height of the children
5. The width of the *Row* is determined by the *mainAxisSize* property
6. Determine the position for each child according to the *mainAxisAlignment* and the *crossAxisAlignment*

Demo!

Flexible

- A widget that controls how a child of a *Flex* flexes
- Does not require the child to fill the available space

Expanded

- Extends *Flexible*
- Forces the child to expand to fill the available space

Spacer

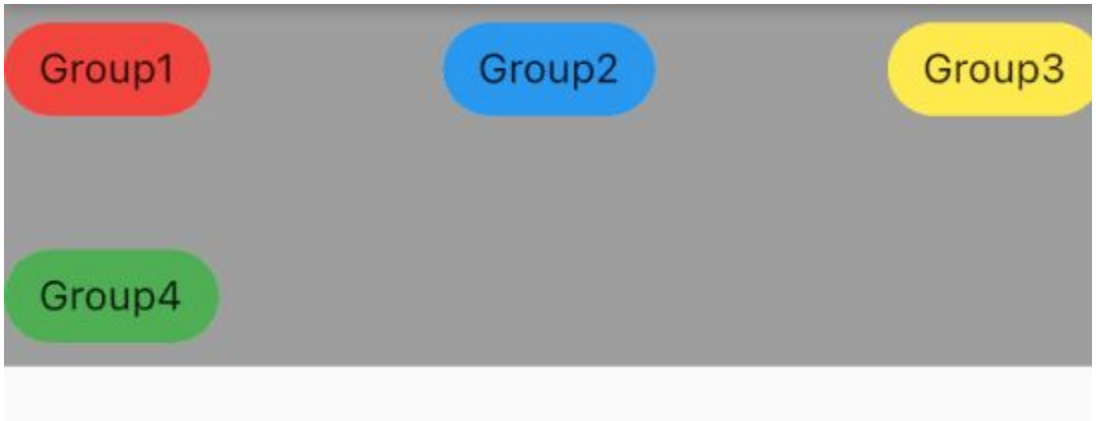
- Creates an adjustable, empty spacer that can be used to tune the spacing between widgets in a *Flex* container
- Widget will take up any available space

Demo!

Wrap

- A *Widget* that displays its children in multiple horizontal or vertical runs
- ~~Flex!~~
- *spacing* - a gap between widgets in main axis
- *runSpacing* - a gap in cross axis

Wrap



```
Container(
  color: Colors.grey,
  child: Wrap(
    spacing: 80,
    runSpacing: 30,
    children: const [
      Chip(
        label: Text("Group1"),
        backgroundColor: Colors.red,
      ),  // Chip
      Chip(
        label: Text("Group2"),
        backgroundColor: Colors.blue,
      ),  // Chip
      Chip(
        label: Text("Group3"),
        backgroundColor: Colors.yellow,
      ),  // Chip
      Chip(
        label: Text("Group4"),
        backgroundColor: Colors.green,
      ),  // Chip
    ],
  ),  // Wrap
),  // Container
```

SingleChildScrollView

- A box in which a single widget can be scrolled
- When to use?
  - There is the concern that in some cases, there might not be enough room to see the entire contents
  - Some devices have unusually small screens
  - Application can be used in landscape mode
  - Application is being shown in a small window in split-screen mode

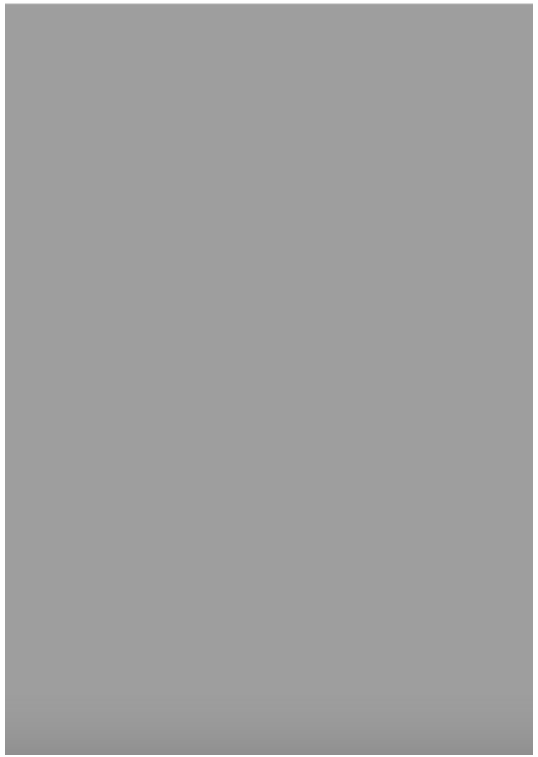
List view

- A scrollable list of widgets arranged linearly
- It displays its children one after another in the scroll direction. In the cross axis, the children are required to fill the *ListView*
- How to create?
  - The default constructor takes an explicit [*List<Widget>]* of children
  - The *ListView.builder* constructor takes an *IndexedWidgetBuilder*, which builds the children on demand
  - The *ListView.separated* constructor takes two *IndexedWidgetBuilder's*: *itemBuilder* builds child items on demand, and *separatorBuilder* similarly builds separator children which appear in between the child items
  - The *ListView.custom* constructor takes a *SliverChildDelegate*, which provides the ability to customize additional aspects of the child model

Reply 1

Reply 2

Reply 3

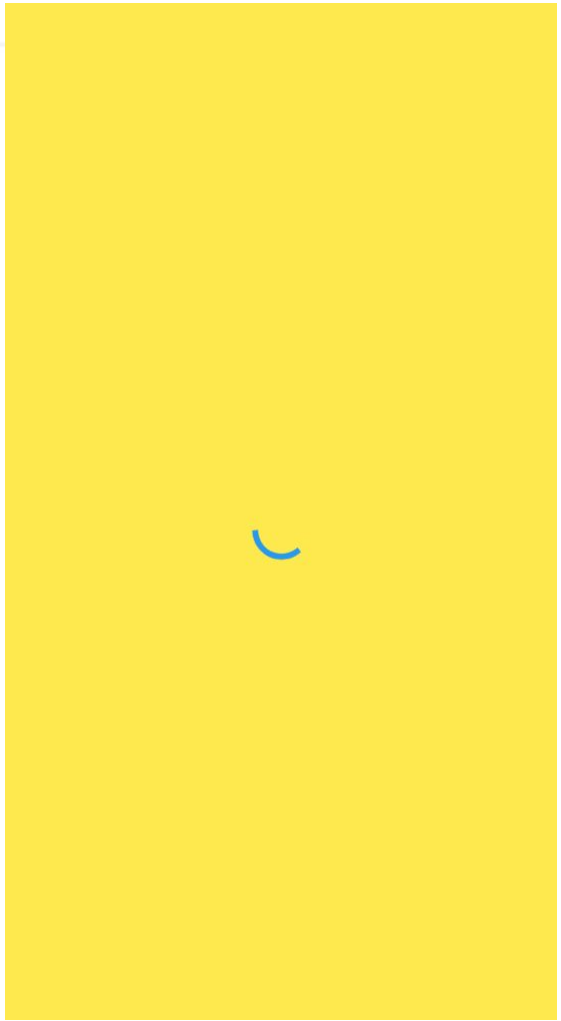


```
Container(
  color: Colors.grey,
  child: ListView(
    children: [
      Container(
        padding: const EdgeInsets.all(16),
        color: Colors.white,
        child: Row(
          children: const [Text("Reply 1"), Spacer()],
        ),  // Row
      ),  // Container
      Container(
        padding: const EdgeInsets.all(16),
        color: Colors.blue,
        child: Row(
          children: const [Spacer(), Text("Reply 2")],
        ),  // Row
      ),  // Container
      Container(
        padding: const EdgeInsets.all(16),
        color: Colors.white,
        child: Row(
          children: const [Text("Reply 3"), Spacer()],
        ),  // Row
      ),  // Container
    ],
  ),  // ListView
),  // Container
```

Demo!

Stack

- A widget that positions its children relative to the edges of its box
- Useful if you want to overlap several children in a simple way, for example having some text and an image, overlaid with a gradient and a button attached to the bottom
- Each child of a *Stack* widget is either *positioned* or *non-positioned*
    - Positioned children are those wrapped in a *Positioned* widget that has at least one non-null property



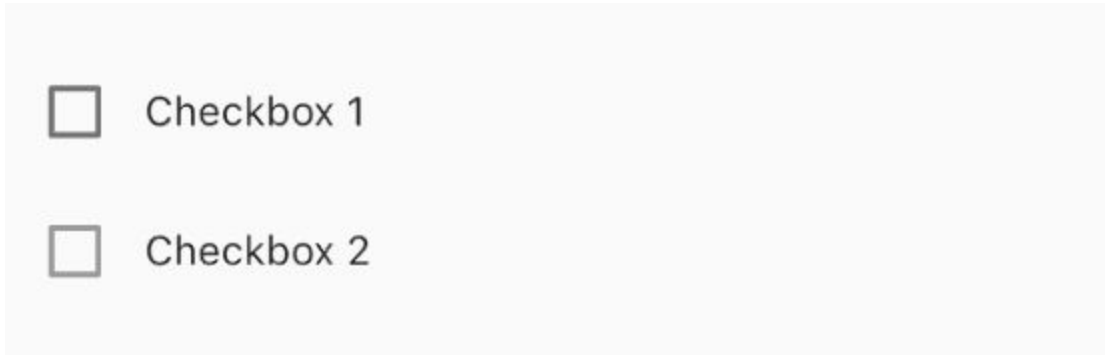
```
Stack(
  children: [
    Container(
      width: double.maxFinite,
      height: double.maxFinite,
      color: Colors.yellow,
    ),  // Container
    const Center(child: CircularProgressIndicator())
  ],
),  // Stack
```

State

Any data that’s needed to create your UI at a certain point in time

Stateful Widget

- A widget that has mutable state
- State is information that
    - Can be read synchronously when the widget is built
    - Might change during the lifetime of the widget
- It is the responsibility of the widget implementer to ensure that the *State* is promptly notified when such state changes, using *State::setState*



```
Column(
  children: [
    Row(
      children: const [
        Checkbox(onChanged: null, value: false),
        Text("Checkbox 1")
      ],
    ),  // Row
    Row(
      children: const [
        Checkbox(onChanged: null, value: false),
        Text("Checkbox 2")
      ],
    ),  // Row
  ],
),  // Column
```

Demo!

Navigation

Imperative navigation

- *Navigator* — a widget that manages a stack of *Route* objects
- *Route* — an object managed by a *Navigator* that represents a screen, typically implemented by classes like *MaterialPageRoute*
- *Routes* are pushed and popped onto the *Navigator's* stack with either named routes or anonymous routes
- *MaterialApp* and *CupertinoApp* already use a *Navigator* under the hood
- You can access the navigator using *Navigator.of()* or display a new screen using *Navigator.push()*, and return to the previous screen with *Navigator.pop()*

Using anonymous routes

```
MaterialButton(
  onPressed: () {
    Navigator.push(
      context,
      MaterialPageRoute(
        builder: (context) {
          return Sample2Page();
        },
      ),  // MaterialPageRoute
    );
  },
  child: const Text("Click"),
)  // MaterialButton
```

Using named routes

```
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      routes: {
        '/': (context) => const Sample1Page(),
        '/sample2': (context) => const Sample2Page(),
      },
    );  // MaterialApp
  }
}
```

Using named routes

```dart
class Sample1Page extends StatelessWidget {
  const Sample1Page({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(),
      body: Column(
        children: [
          MaterialButton(
            onPressed: () {
              Navigator.pushNamed(context, '/sample2');
            },
            child: const Text("Click"),
          ) // MaterialButton
        ],
      ), // Column
    ); // Scaffold
  }
}
```

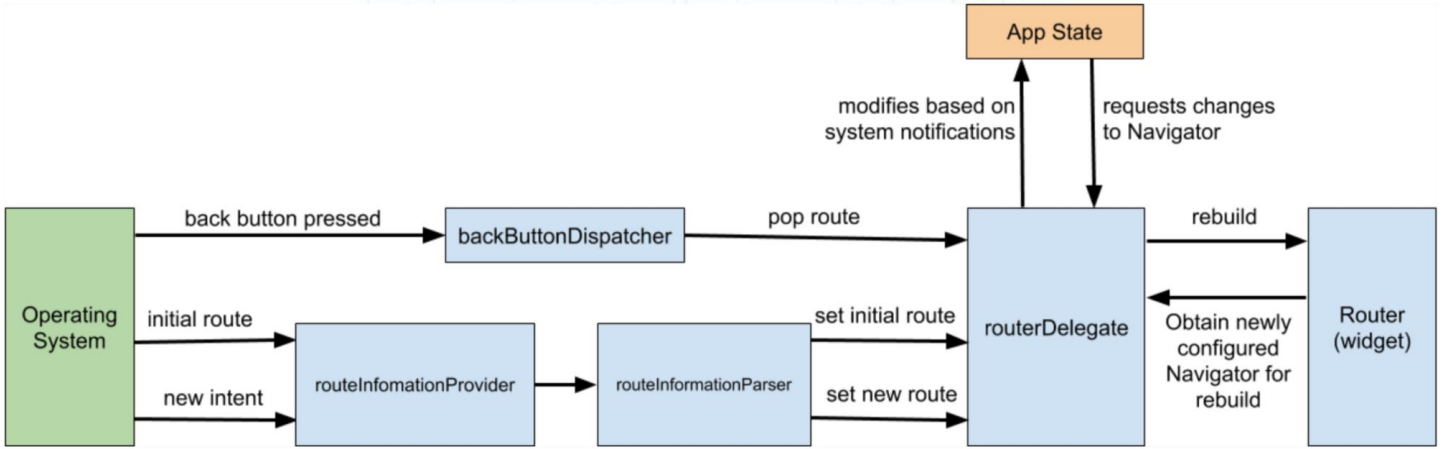
Imperative navigation arguments

- You can pass argument in page's constructor
- *Navigator::push - Future<T?> push*
- You can pass result as *Navigator::pop* parameter

Declarative navigation

- *Page* — an immutable object used to set the navigator's history stack.
- *Router* — configures the list of pages to be displayed by the *Navigator*.
- *RouteInformationParser* - takes the *RouteInformation* from *RouteInformationProvider* and parses it into a user-defined data type.
- *RouterDelegate* — defines app-specific behavior of how the *Router* learns about changes in app state and how it responds to them. Its job is to listen to the *RouteInformationParser* and the app state and build the *Navigator* with the current list of *Pages*.
- *BackButtonDispatcher* — reports back button presses to the *Router*

Declarative navigation



Source: https://medium.com/flutter/learning-flutters-new-navigation-and-routing-system-7c9068155ade

Using declarative navigation

```
Navigator(
  pages: [
    const MaterialPage(child: Sample1Page()),
    if (_showSecondPage) const MaterialPage(child: Sample2Page()),
  ],
  onPopPage: (route, result) {
    if (!route.didPop(result)) {
      return false;
    }
    setState(() {
      _showSecondPage = false;
    });
    return true;
  },
), // Navigator
```

Declarative navigation

- *RouteInformationParser* - converts information into user-defined data type
    - For example: parse URI and convert to object that represents path
    - Use case: deep links
- *RouterDelegate* — builds *Navigator,* converts user-defined data type into page

Declarative navigation

```
return MaterialApp.router(
  title: "Lecture 3",
  routeInformationParser: _routeInformationParser,
  routerDelegate: _routerDelegate,
);
```

Great article about declarative navigation:

https://medium.com/flutter/learning-flutters-new-navigation-and-routing-system-7c9068155ade

Questions?