

TDDC73 - LAYOUT, INTERACTION, AND CODE- ORGANIZATION

Agenda

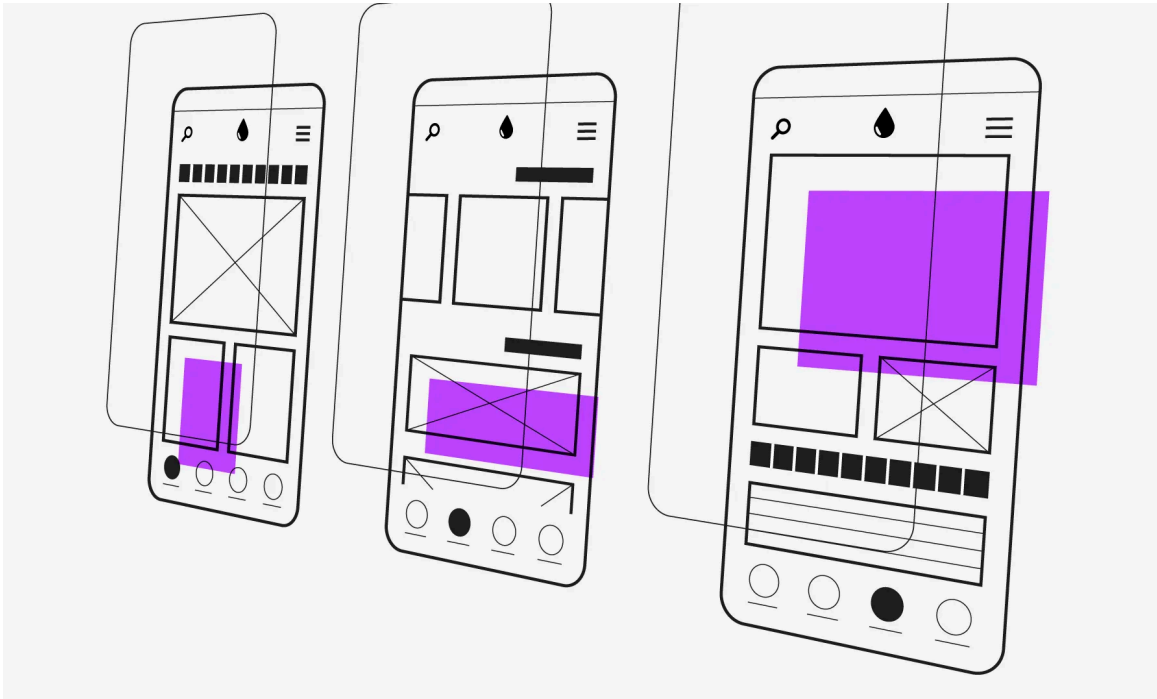
- Questions
- Course page (www.ida.liu.se/~TDDC73)
- Grouping Code (Framework/API/SDK)
- Layout
 - Android - standard and Android - compose
 - React-Native Flexbox
 - Flutter
- Manage interaction
 - Callbacks/Listeners
- State and Structuring your code Model-View-Whatever
 - MVC ,MVP ,MVVM
- Demo

Framework

- Functions/Classes/Objects
 - Libraries
 - Framework
- SDK Software Development Kit
 - We add some tools

Layouts

- Putting stuff in the right place
- Keeping things in their proper place
- Make shit look and hopefully work nice



Standard Android layouts

- ViewGroup - a View that supports adding child views
 - LinearLayout
 - Left-to-Right or Top-to-Bottom
 - FrameLayout
 - Designed for a single View child
 - RelativeLayout
 - Views placed in relation to each other or the parent
 - GridLayout
 - Rows and Columns
 - ConstraintLayout
 - RelativeLayout on steroids
 - MotionLayout
 - ConstraintLayout on steroids

Layout in Compose



Column



Row



Box

Column

```
@Composable
fun ArtistCardColumn() {
    Column {
        Text("Alfred Sisley")
        Text("3 minutes ago")
    }
}
```

Alfred Sisley

3 minutes ago

Row

```
@Composable
fun ArtistCardRow(artist: Artist) {
    Row(verticalAlignment = Alignment.CenterVertically) {
        Image(bitmap = artist.image, contentDescription = "Artist image")
        Column {
            Text(artist.name)
            Text(artist.lastSeenOnline)
        }
    }
}
```



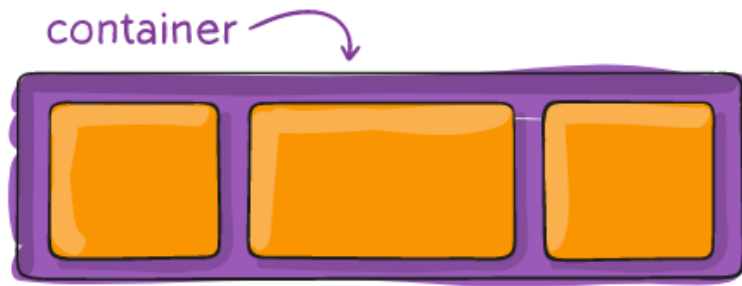
Alfred Sisley

3 minutes ago

Box

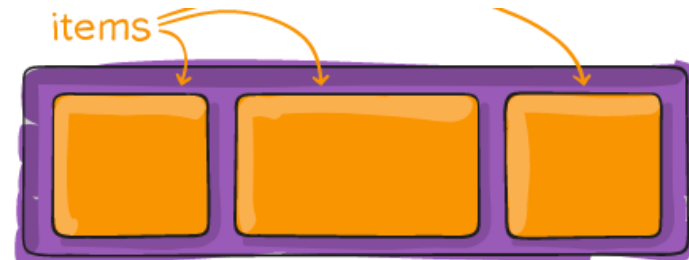
```
@Composable
fun ArtistAvatar(artist: Artist) {
    Box {
        Image(bitmap = artist.image, contentDescription = "Artist image")
        Icon(Icons.Filled.Check, contentDescription = "Check mark")
    }
}
```



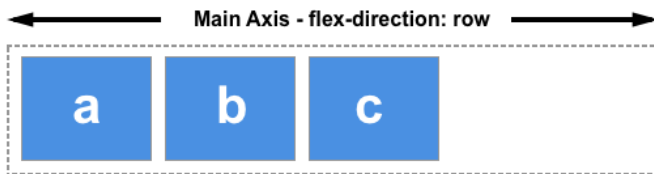


Flexbox layout

- Multiple properties are used together in order to use Flexbox layout
- Some properties are set on the container (parent, flex container)
- Some properties are set on the elements in the container (children, flex items)
- https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout
- <https://css-tricks.com/snippets/css/a-guide-to-flexbox/>



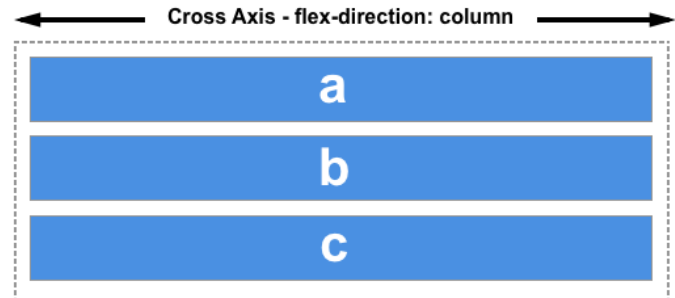
Main Axis and Cross Axis



Choose `column` or `column-reverse` and your main axis will run top to bottom of the page in the block direction.



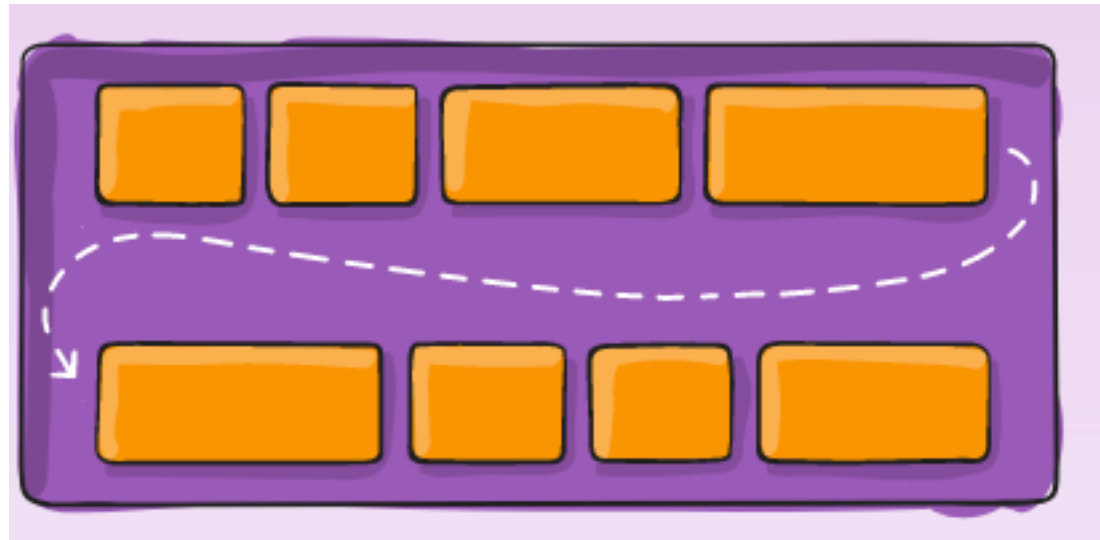
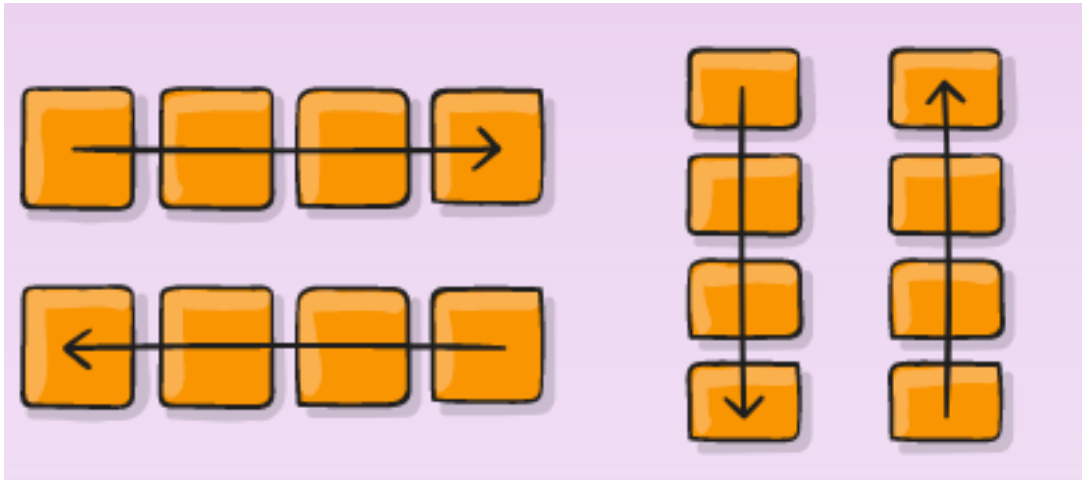
If your main axis is `column` or `column-reverse` then the cross axis runs along the rows.



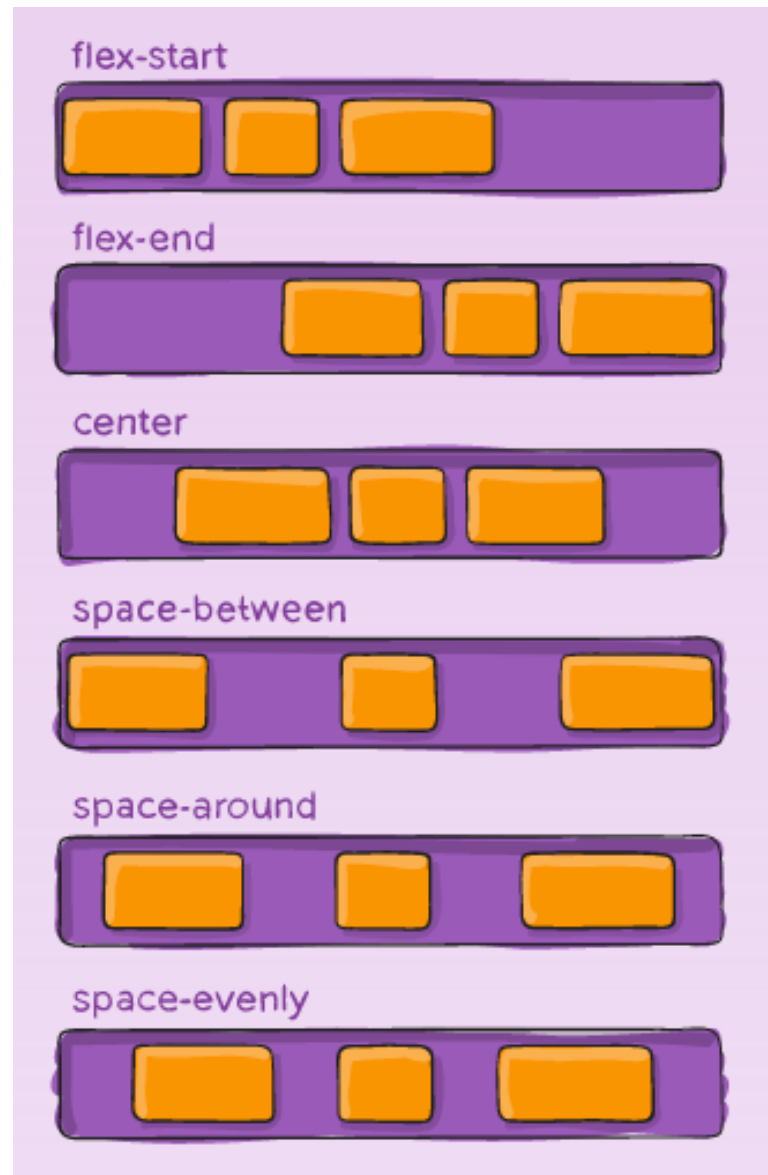
Some parent container properties

- flex-direction: row | row-reverse | column | column-reverse
- flex-wrap: nowrap | wrap | wrap-reverse
- justify-content: flex-start | flex-end | center | space-between
- align-items: stretch | flex-start | flex-end | center | baseline
- flex-flow: <flex-direction> <flex-wrap>

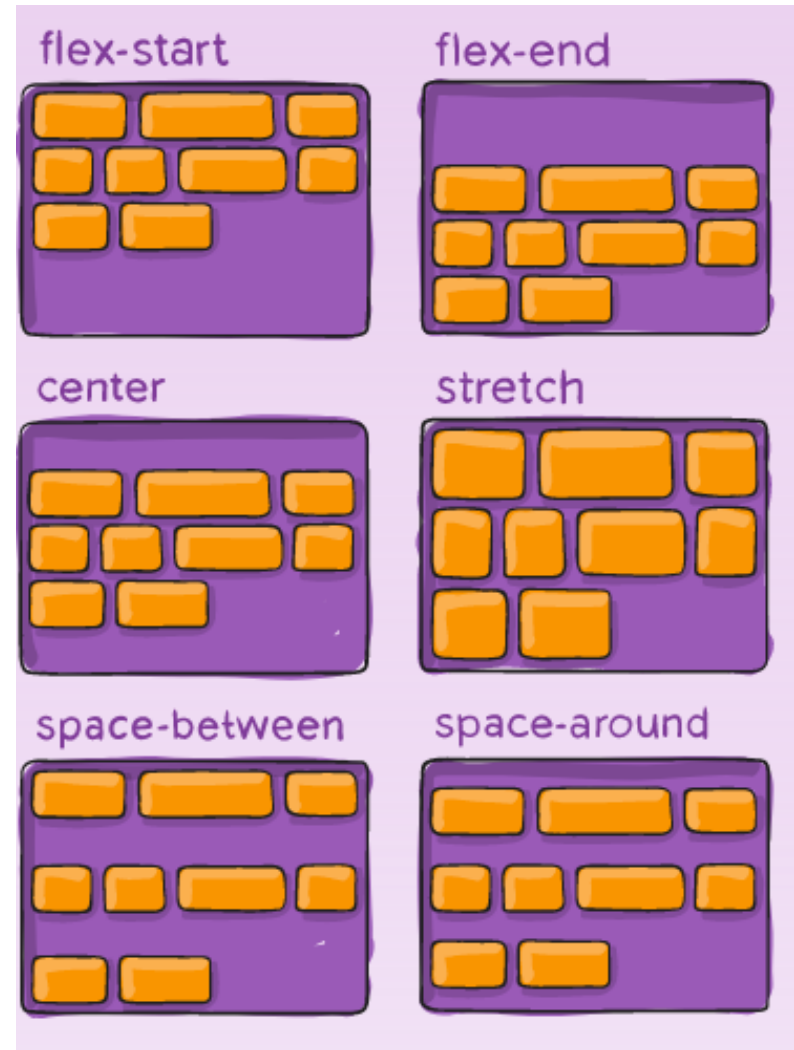
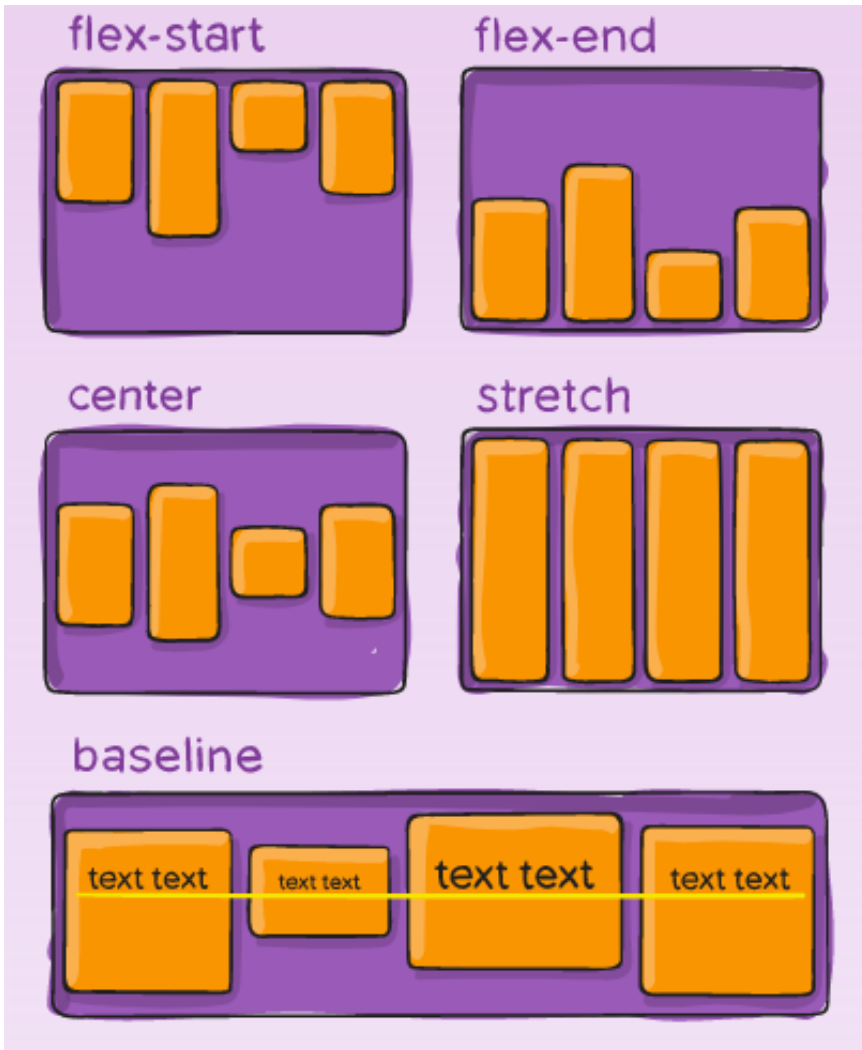
flex-direction and flex-wrap



justify-content



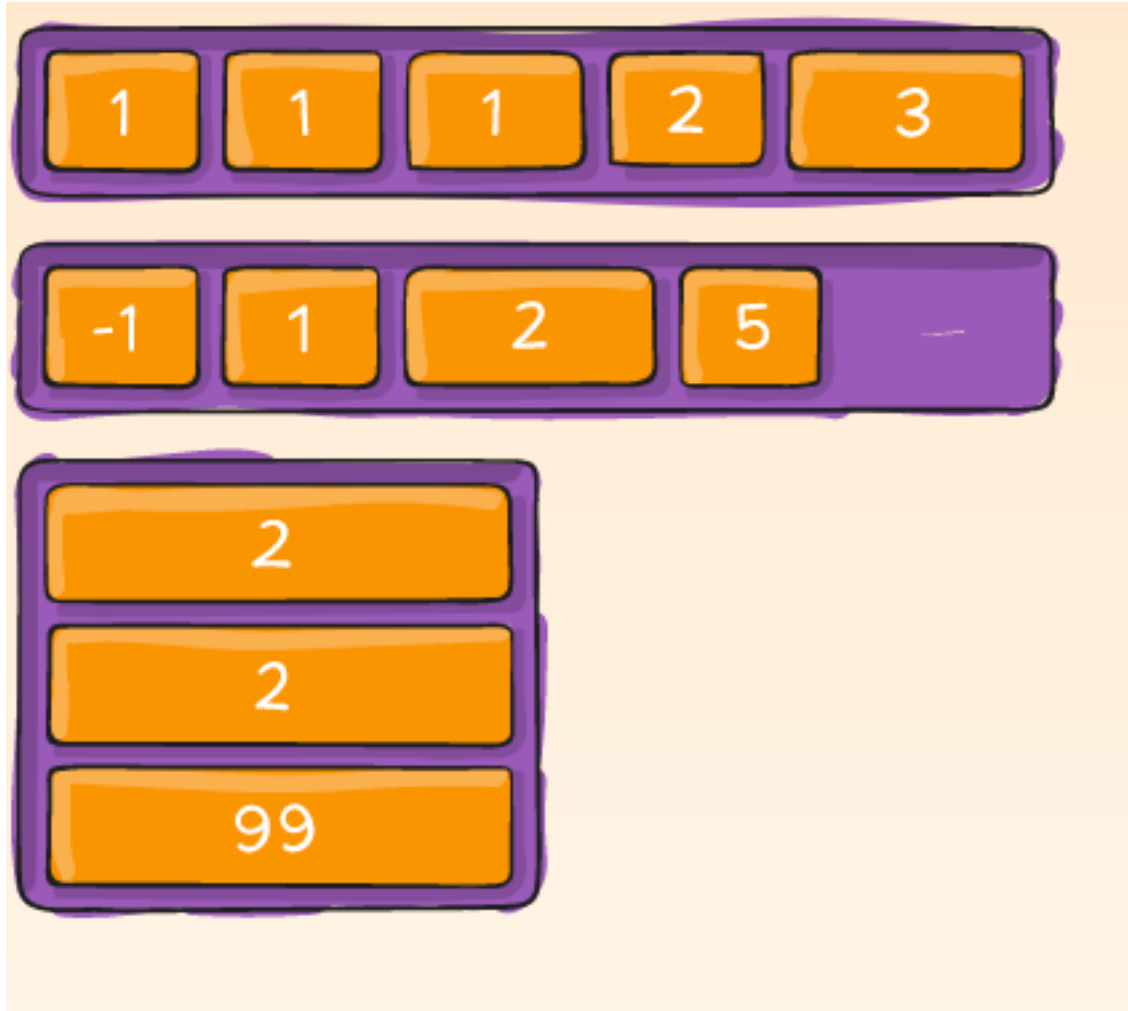
align-items and align-content



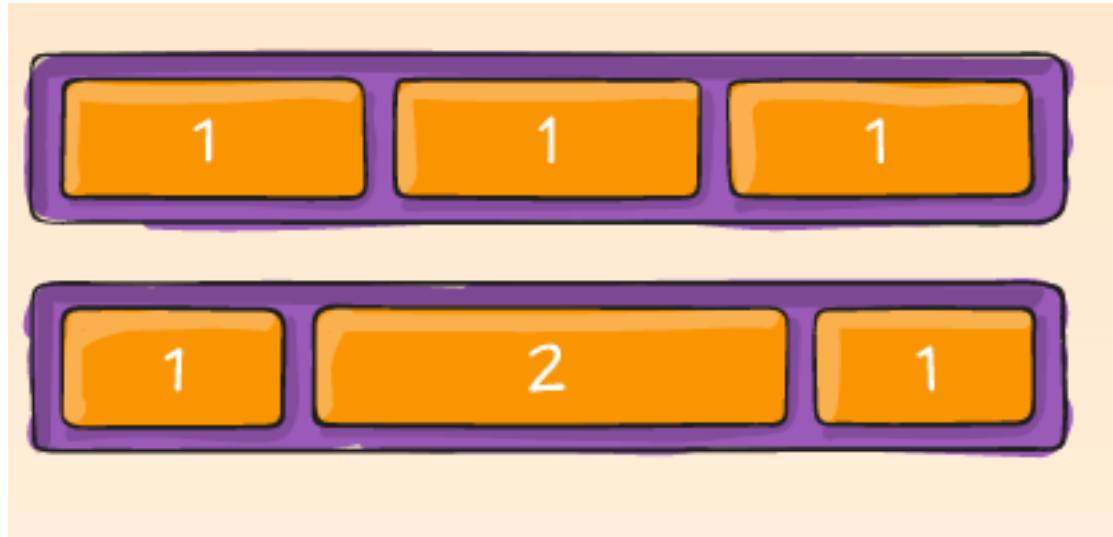
Properties set on children

- `order`: <integer>
- `flex-grow`: <positive number>
- `flex-shrink`: <positive number>
- `flex-basis`: <width/height>
- `align-self`: `auto` | `flex-start` | `flex-end` | `center` | `baseline` | `stretch`;
- `flex` (see reference documentation)

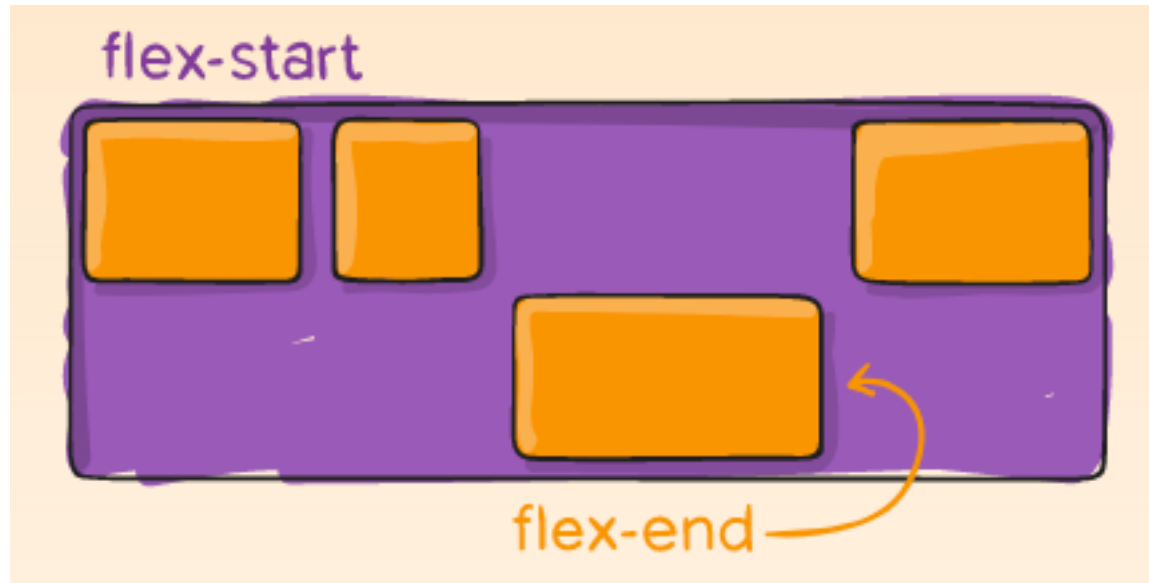
order



flex-grow and flex-shrink

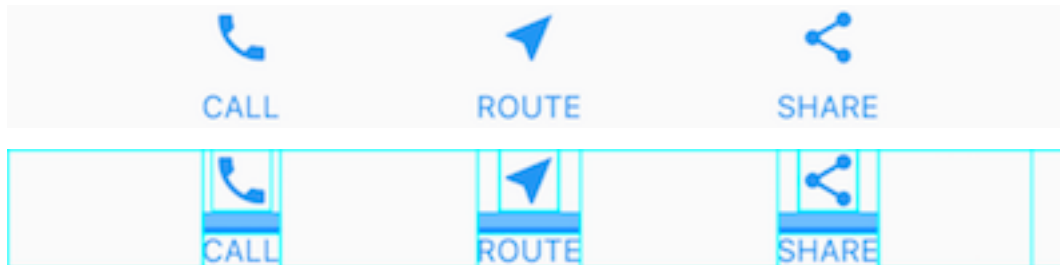


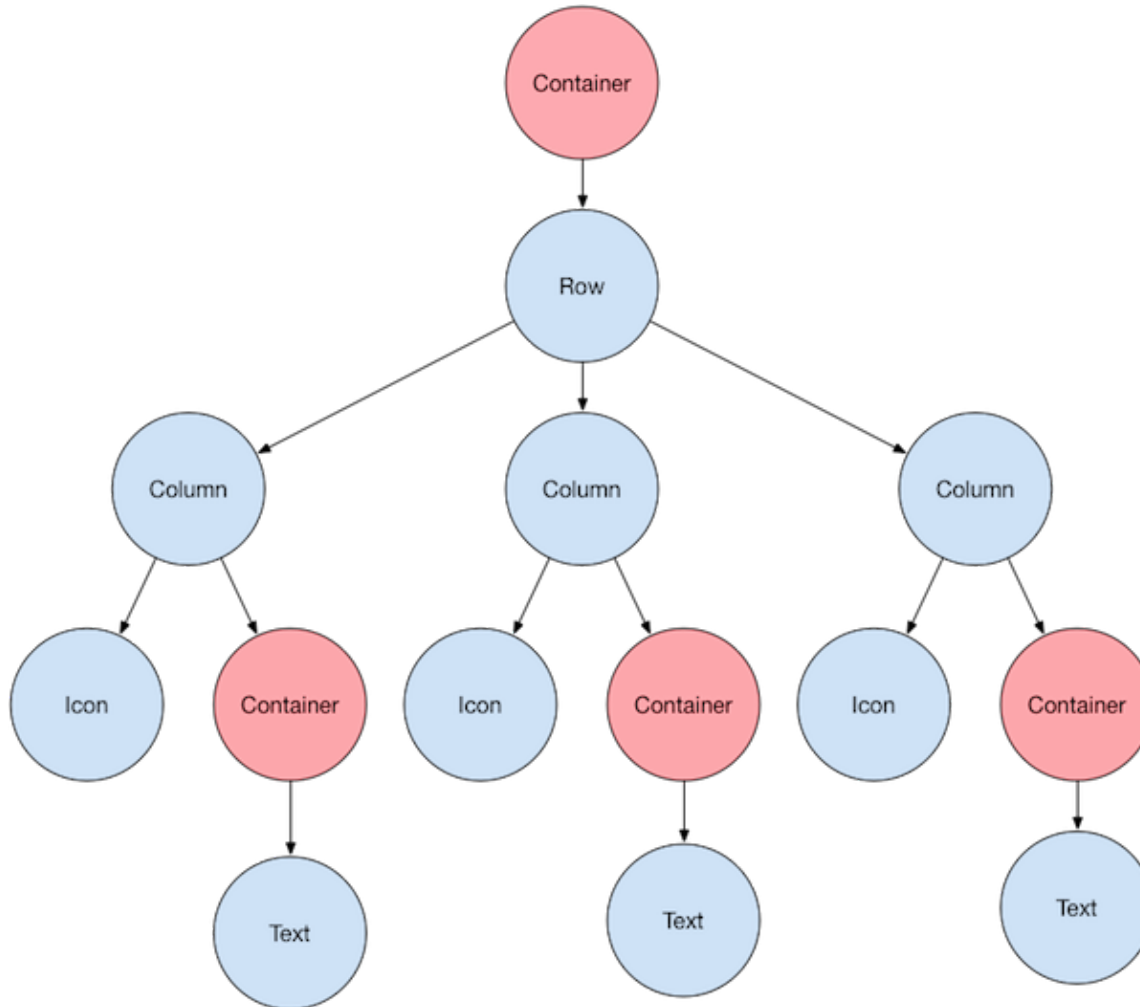
align-self



Layout in Flutter

- Based on Widgets in rows and columns
- Everything is a widget (almost)





Layout widgets

- Containers are layout widgets

- <https://flutter.dev/docs/development/ui/widgets/layout>

Maganging Interaction

When the users does something in our application

Widget Messages

- Messages from widgets to your code
 - Information about a change in state
 - Messages of user generated actions
 - Mouse movement, keyboard action
 - Touch , Sensors
- Widgets has a set of Messages (zero or more) you choose which you care about

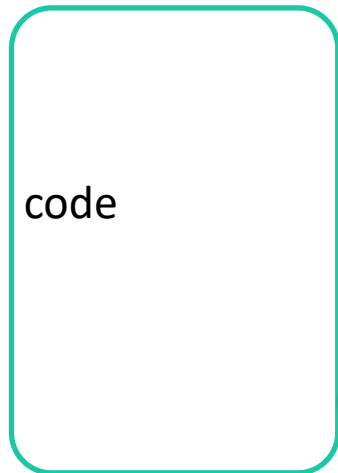
Widget Messages

- Messages passed through callback functions
- You registers yourself with component using listeners
 - android : `setOnClickListener`
- Receive notification
 - `onClick(View v)`

Listeners



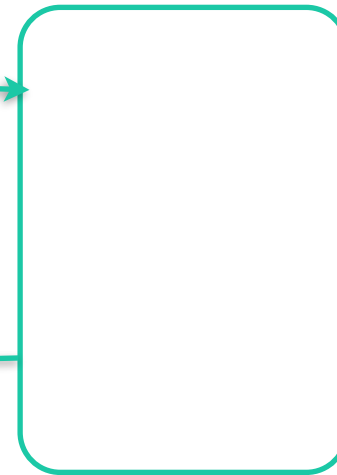
Your code



add_A_Listener(this)



Button/Widget



callback_function(Event)



Button-click kotlin

```
button.setOnClickListener{  
    counter++  
    textView.text = "Click counter : $counter"  
}
```

```
button.setOnClickListener(object: View.OnClickListener {  
    override fun onClick(v: View?) {  
        counter++  
        textView.text = "Click counter : $counter"  
        v?.setBackgroundColor(Color.MAGENTA)  
    }  
})
```

Button-click flutter

```
FlatButton(  
  child: Text('I am FlatButton'),  
  onPressed: () {  
    print('You tapped on FlatButton');  
  },  
),
```

Flutter has multiple Button widgets

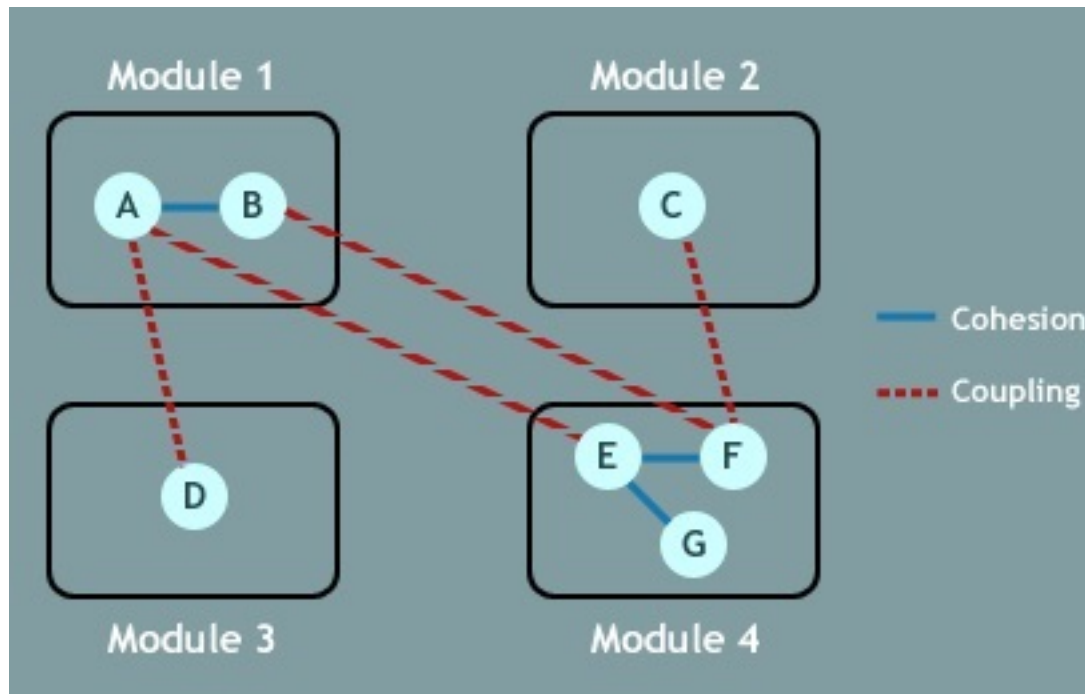
Button-click React Native

```
<Button
  onPress={() => {
    alert('You tapped the button!');
  }}
  title="Press Me"
/>
```

Organisation of code

- Biggest challenge of UI development (Ok one of)
 - Maintaining the correct and same state between the UI and the system/model/backend
- Separation of concerns
- Architecture Presentation Patterns
 - MVC
 - MVP
 - MVVM

Cohesion and Coupling



From Chaos to Control

Mastering UI State & The Architectures That Tame It

What is UI State?

What is State?

All the data that describes your application's condition at a given point in time.

What is UI State?





All the data your user interface needs to draw itself correctly.

Music Player Examples

UI State is the answer to all of these questions:

- Is a song playing? (`isPlaying``)
- What is the volume? (`volume``)
- What songs are in the list? (`playlist``)
- Is there an error? (`errorMessage``)
- Are we waiting for data? (`isLoading``)

The Problem: When State Becomes Chaos

-  **Shared State:** Multiple components need the same data (e.g., user login). Where does this state "live"?
-  **Synchronization:** If a user logs out from one component, how do all other components know to update immediately?
-  **Prop Drilling:** Passing data down through 5-6 intermediate components that don't need the data themselves.
-  **Unpredictable Mutations:** Any component can change any state at any time, leading to a nightmare of bugs.

Classic Architectures



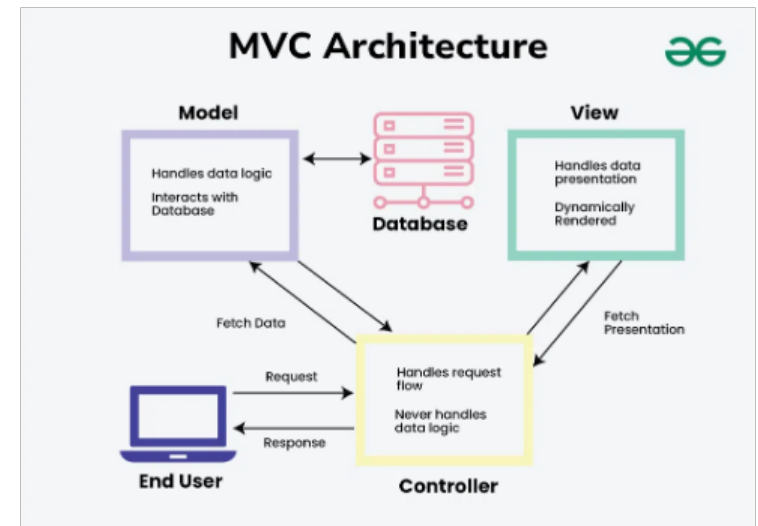
Our first attempts at separating concerns and
taming the chaos.

Classic Pattern 1: Model-View-Controller (MVC)

The Original "Separation of Concerns"

This pattern was the first major attempt to untangle logic from the UI.

- **Model:** The data and business logic. The single source of "truth."
- **View:** The UI. What the user sees.
- **Controller:** Takes user input (e.g., button clicks) and updates the Model.
- **The Flow:** The Controller updates the Model. The Model then notifies the View to update itself.



The Evolutions: MVP & MVVM

Model-View-Presenter (MVP)

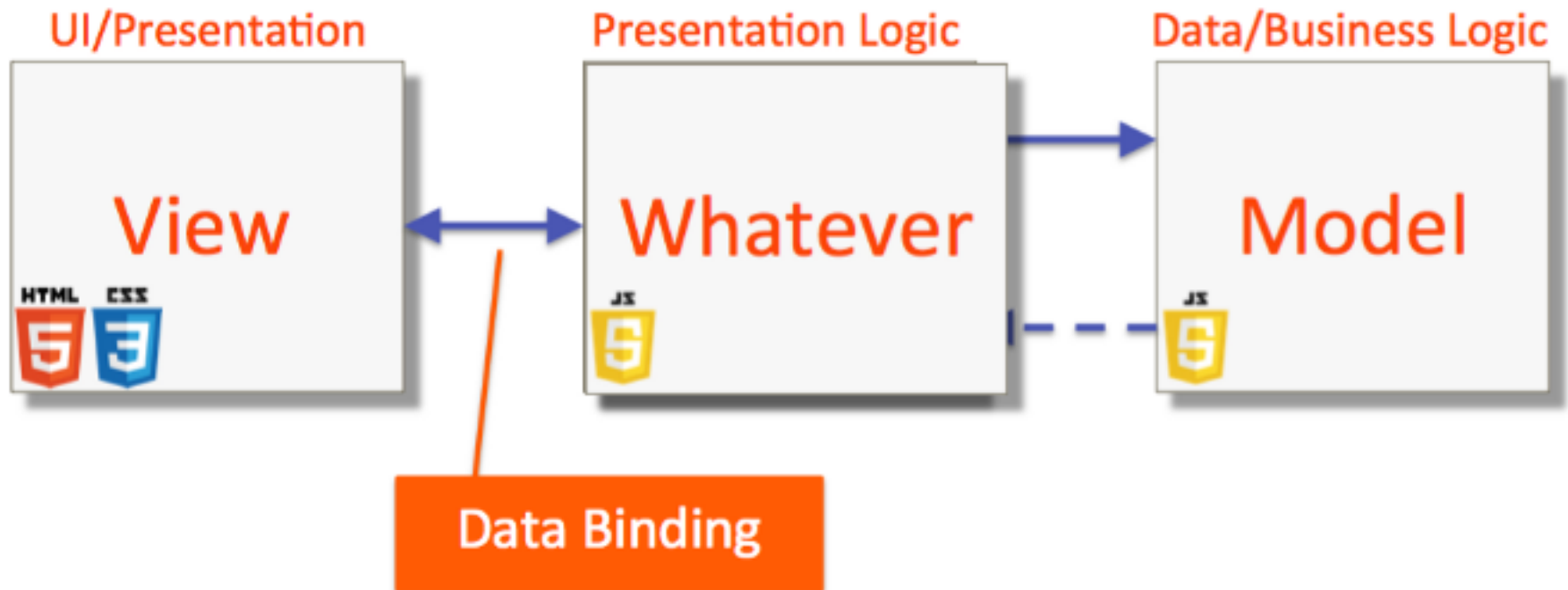
An evolution to make the View "dumber." The Presenter acts as a middle-man, fetching data from the Model and pushing it to the View. The View just passes user events to the Presenter. This improves testability.

Model-View-ViewModel (MVVM)

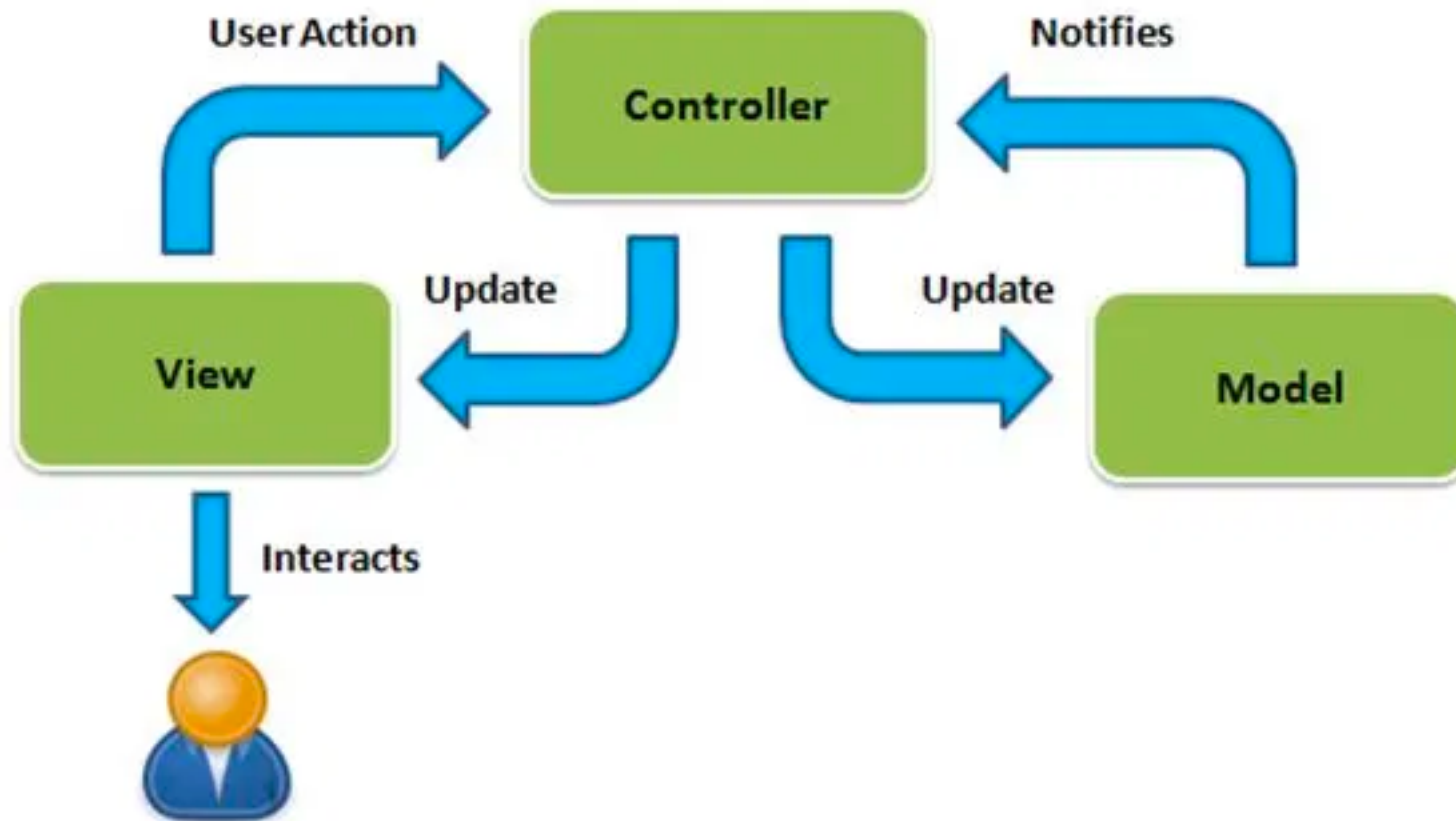
Very popular in modern Android. The ViewModel exposes state (data) and the View "binds" to it, updating automatically. The View is completely separate from the business logic, making it highly testable and robust.

ORGANIZING YOUR CODE

MVC/MVP/MVVM



Model-View-Controller - MVC

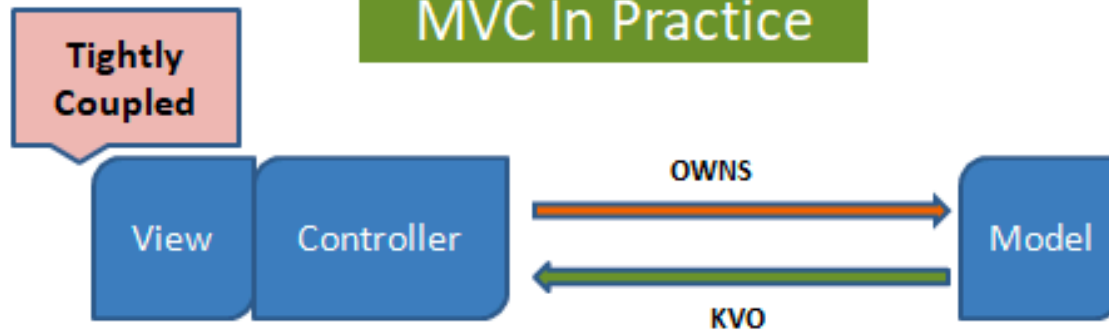


MVC In Theory

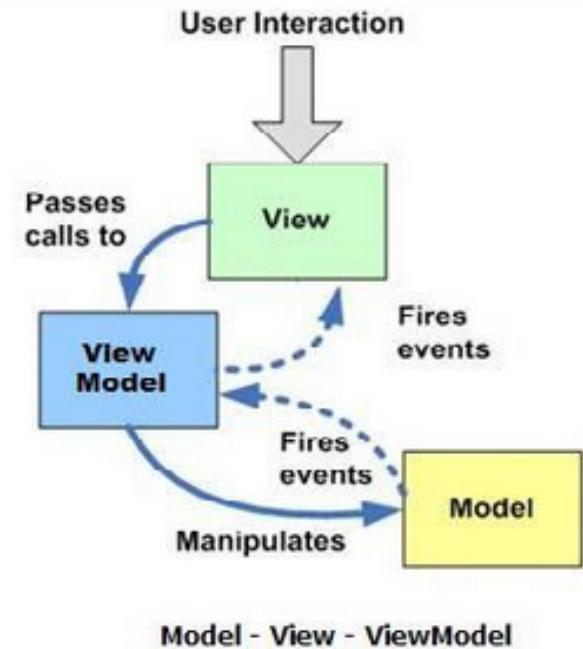
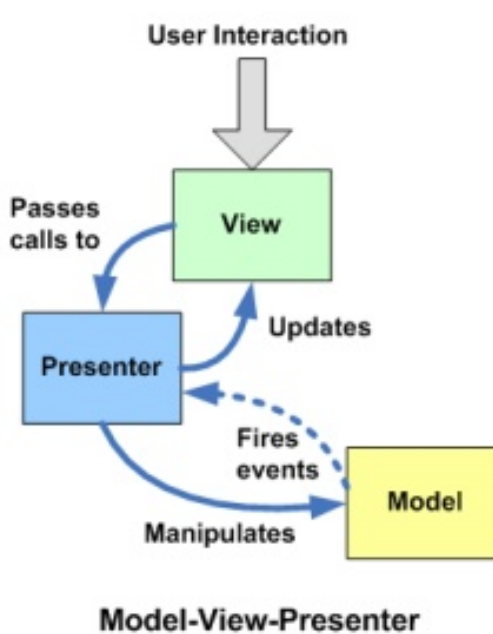
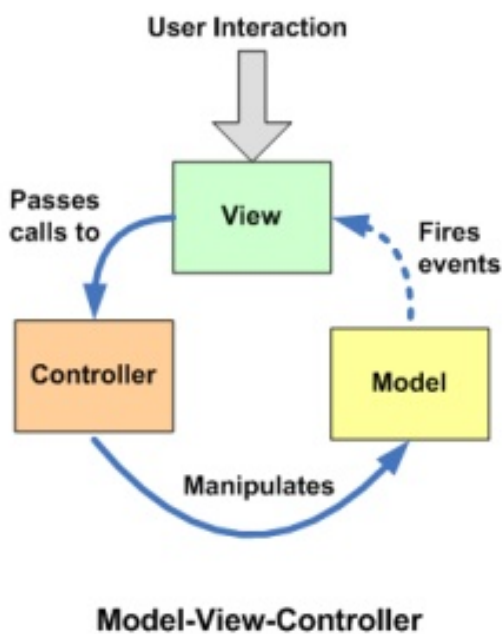


KVO = Key-Value Observing

MVC In Practice



Overview

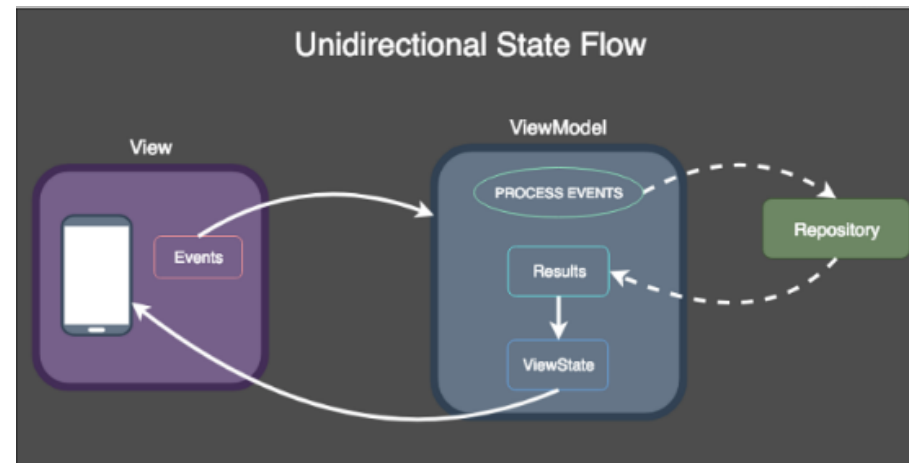


The Revolution: Unidirectional Data Flow

The Flux/Redux Idea

Data **always** flows in a single, predictable direction. This is the foundation of React Native, Flutter, and Compose.

- **1. Store:** A single source of truth holds all state.
- **2. View:** Renders the UI based on the state.
- **3. Action:** User interaction dispatches an action (describes **what happened**).
- **4. Reducer:** A function returns a **new** state based on the action.



The Modern Insight



Not all state is created equal. We must specialize our approach and stop treating every piece of data the same way.

Type 1: Local UI State



React Native

The `useState` hook. It's fast, simple, and co-located with your component.

(e.g., `const [isEnabled, setIsEnabled] = useState(false);`)



Flutter

A `StatefulWidget` and the `setState()` method. The state lives and dies with the widget.

(e.g., `setState(() { _isEnabled = false; });`)

Jetpack Compose

The `remember` and `mutableStateOf` functions. The state is "remembered" across recompositions.

(e.g., `var isEnabled by remember { mutableStateOf(false) }`)

Type 2: Shared Client State



React Native

For theme, auth, or shopping cart.

Tools: Context API, Zustand, Jotai, or Redux Toolkit.



Flutter

A model `ChangeNotifier` updates all listeners.

Tools: Provider or Riverpod.

Jetpack Compose

An Android `ViewModel` exposes a `StateFlow`.

Tools: ViewModel + StateFlow.

Type 3: Server State (Cache)



React Native

Manages `isLoading`, `error`, `data` and caching for you.

Tools: React Query (TanStack Query) or SWR.



Flutter

Use `FutureBuilder` or `StreamBuilder` to react to state.

Tools: Repository Pattern + Provider/Riverpod.

Jetpack Compose

Expose a sealed `UiState` class (Loading, Success, Error).

Tools: Repository Pattern + ViewModel/StateFlow.

How to Choose: React Native



Local State: Always start with `useState``. Keep it simple.



Server State: Use **React Query (TanStack Query)** for *all* API data. This is the modern standard.



Shared State: Use **Zustand** or **Context API** for simple global state (theme, auth status).



Complex State: Only use **Redux Toolkit** if you have extremely complex *client-side* logic.

How to Choose: Flutter

Flutter

1. Start with `StatefulWidget` for purely local state.
2. Use **Provider** or **Riverpod** as soon as state needs to be shared or lifted from a widget.
3. Combine this with a **Repository Pattern** for all data fetching and business logic.

How to Choose: Jetpack Compose

Jetpack Compose

1. Start with `remember` and **State Hoisting** (passing state down and events up).
2. Use an **Android ViewModel** + **StateFlow** as soon as state gets complex, needs to be shared, or must survive screen rotation.
3. Combine this with a **Repository Pattern** inside the ViewModel.

Summary



Key Takeaway

Specialize Your State

Modern UI is declarative: **UI = f(state)**.

The key is to stop looking for **one** state manager. Use the right tool for the right job:

- **Local State** (in the component)
- **Server Cache** (e.g., React Query, Repository)
- **Shared Client State** (e.g., Zustand, Provider, ViewModel)

Questions?

Demo