

# Interaktionsprogrammering

## TDDC73

Anders Fröberg

[anders.froberg@liu.se](mailto:anders.froberg@liu.se)

# Agenda

- Declarative UI (repeat)
- Navigation
- Network

# Declarative UI

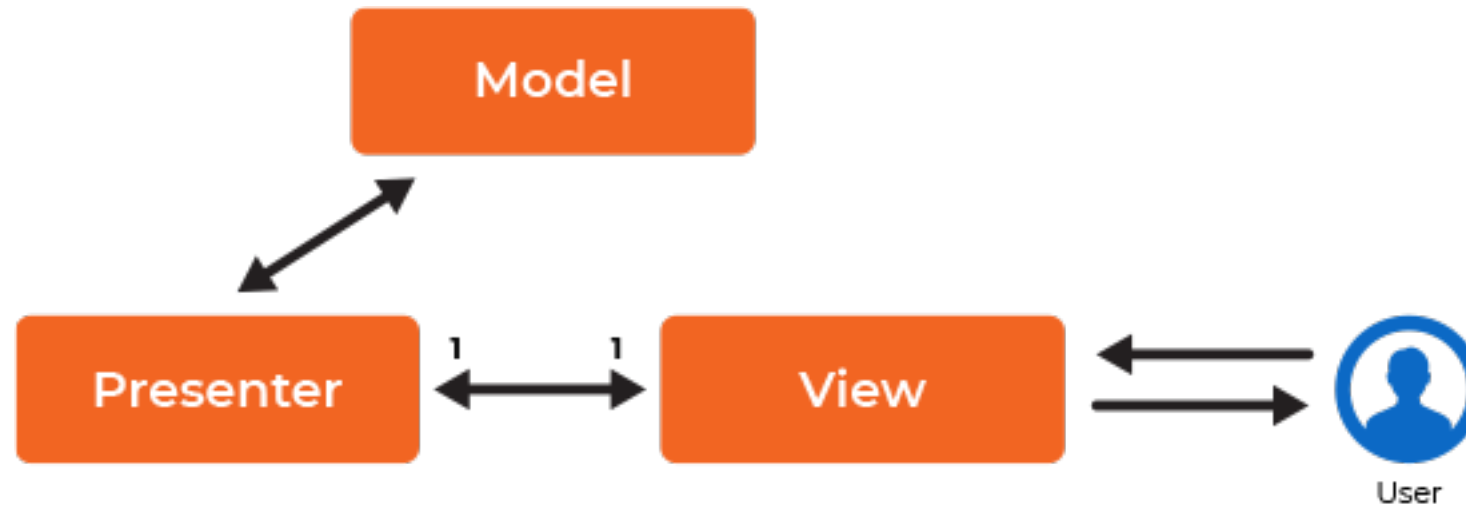
The Modern Shift in Mobile Development

Flutter React Native Jetpack Compose

# 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

# Model-View-Presenter MVP



**MVP Pattern**

# Demo

- Android Adapter example


# The Paradigm Shift

---

## Imperative

---


The Old Way (Android XML, UIKit)

- ✓ Manually construct the UI.
- ✓ Mutate it manually when data changes.
- ✓ *"Find button ID, set color red."*
- ✓  Prone to synchronization bugs.

## Declarative

---

The Modern Way (Flutter, RN, Compose)

- ✓ Describe the current state.
- ✓ Framework rebuilds the UI.
- ✓ *"State is loading, show spinner."*
- ✓  Single source of truth.

# The Core Concept

**UI = fstate**

Your User Interface is a direct result (function) of your current application  
State.

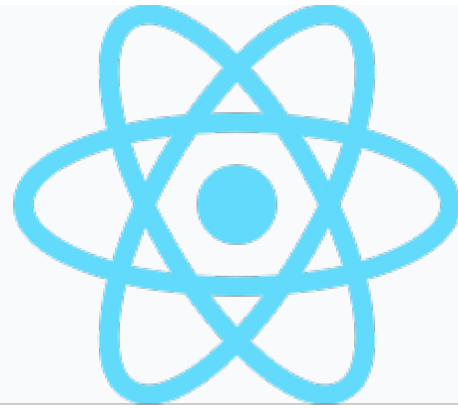
# The Contenders

---



## Flutter

Google's UI toolkit for building natively compiled applications from a single codebase using **Dart**.



## React Native

Meta's framework for building native apps using **JavaScript** and the React paradigm.



## Jetpack Compose

Android's modern toolkit for building native UI using **Kotlin** composable functions.

# 1. Flutter (Dart)

---

## "Everything is a Widget"

- ✓ **Language:** Dart
- ✓ **Structure:** A tree of widgets. Code reflects the visual hierarchy.
- ✓ **Trigger:** `setState()` marks the widget as dirty to rebuild.
- ✓ **Rendering:** Uses the Skia/Impeller engine to draw pixels directly to the screen.



# Flutter Code: The Counter

```
class _CounterScreenState extends State {  
  int count = 0; // 1. The State  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      body: Column(  
        children: [  
          Text('Count: $count'), // 2. The UI  
          ElevatedButton(  
            onPressed: () {  
              setState(() { // 3. The Trigger  
                count++;  
              });  
            },  
            child: Text('Increment'),  
          ),  
        ],  
      ),  
    );  
  }  
}
```

## Key Takeaways

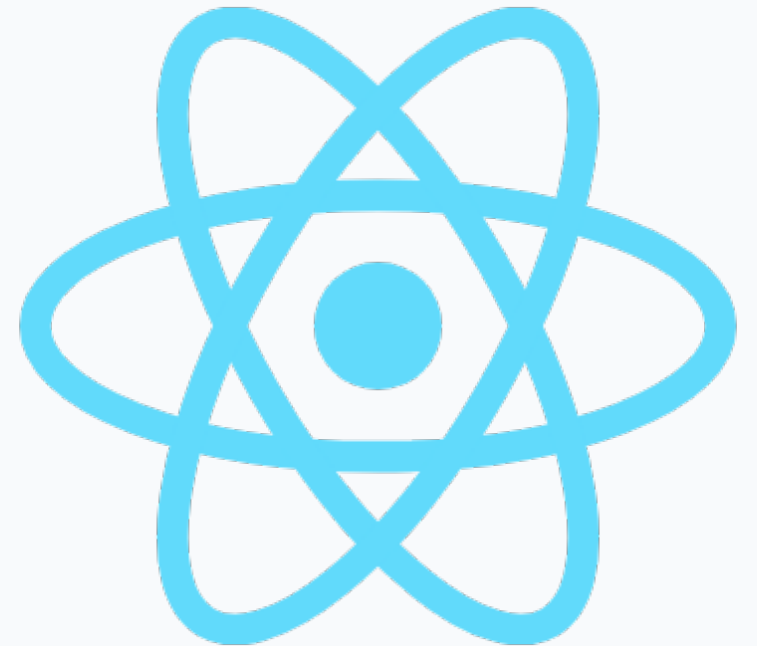
- ✓ **Nesting:** The code structure (Column > Text, Button) matches the visual layout.
- ✓ **setState:** This specific function tells Flutter to run build() again.
- ✓ **Explicit:** Dart requires explicit widget instantiation (e.g., Text()).

## 2. React Native (JS/TS)

---

### Components & Hooks

- ✓ **Language:** JavaScript / TypeScript (JSX).
- ✓ **Structure:** Uses the Virtual DOM concept.
- ✓ **Trigger:** `useState()` hooks handle data changes.
- ✓ **Rendering:** Bridges JS code to actual native UI components (Views, Buttons).



# React Native Code: The Counter

```
const CounterScreen = () => {  
  // 1. The State (Hook)  
  const [count, setCount] = useState(0);  
  
  return (  
    // 2. The UI description (JSX)  
  
    <Count count={count} />  
  
    <Increment  
      onPress={() => setCount(count + 1)} // 3. Trigger  
    />  
  
  );  
};
```

## Key Takeaways

- ✓ **Hooks:** `useState` is separated from the UI logic.
- ✓ **JSX:** Looks like HTML but is actually JavaScript sugar.
- ✓ **Bridges:** The `<View>` tag becomes a `ViewGroup` on Android or `UIView` on iOS.

# 3. Jetpack Compose (Kotlin)

---

## Composable Functions

- ✓ **Language:** Kotlin
- ✓ **Structure:** Functions annotated with `@Composable` emit UI.
- ✓ **Trigger:** `mutableStateOf()` with `remember`.
- ✓ **Rendering:** Intelligent recomposition (Gap Buffer) skips unchanged elements.



# Compose Code: The Counter

```
@Composable
fun CounterScreen() {
    // 1. The State (Remembered)
    var count by remember { mutableStateOf(0) }

    // 2. The UI description
    Column {
        Text(text = "Count: $count")

        Button(onClick = { count++ }) { // 3. Trigger
            Text("Increment")
        }
    }
}
```

## Key Takeaways

- ✓ **Functions:** The UI is just a function, not a class.
- ✓ **Magic State:** Just changing `count++` triggers the update. No `setState` boilerplate needed.
- ✓ **Kotlin DSL:** Clean, concise syntax without XML.

# Framework Comparison

---

Feature	Flutter	React Native	Jetpack Compose
Building Block	Widget (Class)	Component (Function)	Composable (Function)
Markup	Dart (Nested)	JSX (XML-in-JS)	Kotlin DSL
State Trigger	setState()	setCount() (Hook)	count.value = x
Rendering	Skia Engine (Canvas)	Native Views (Bridge)	Native Canvas

# The Mental Model

---



## Define State

Identify the data that changes (e.g.,  
isLoading, userCount).



## Describe UI

Write the function that maps that  
state to a visual layout.



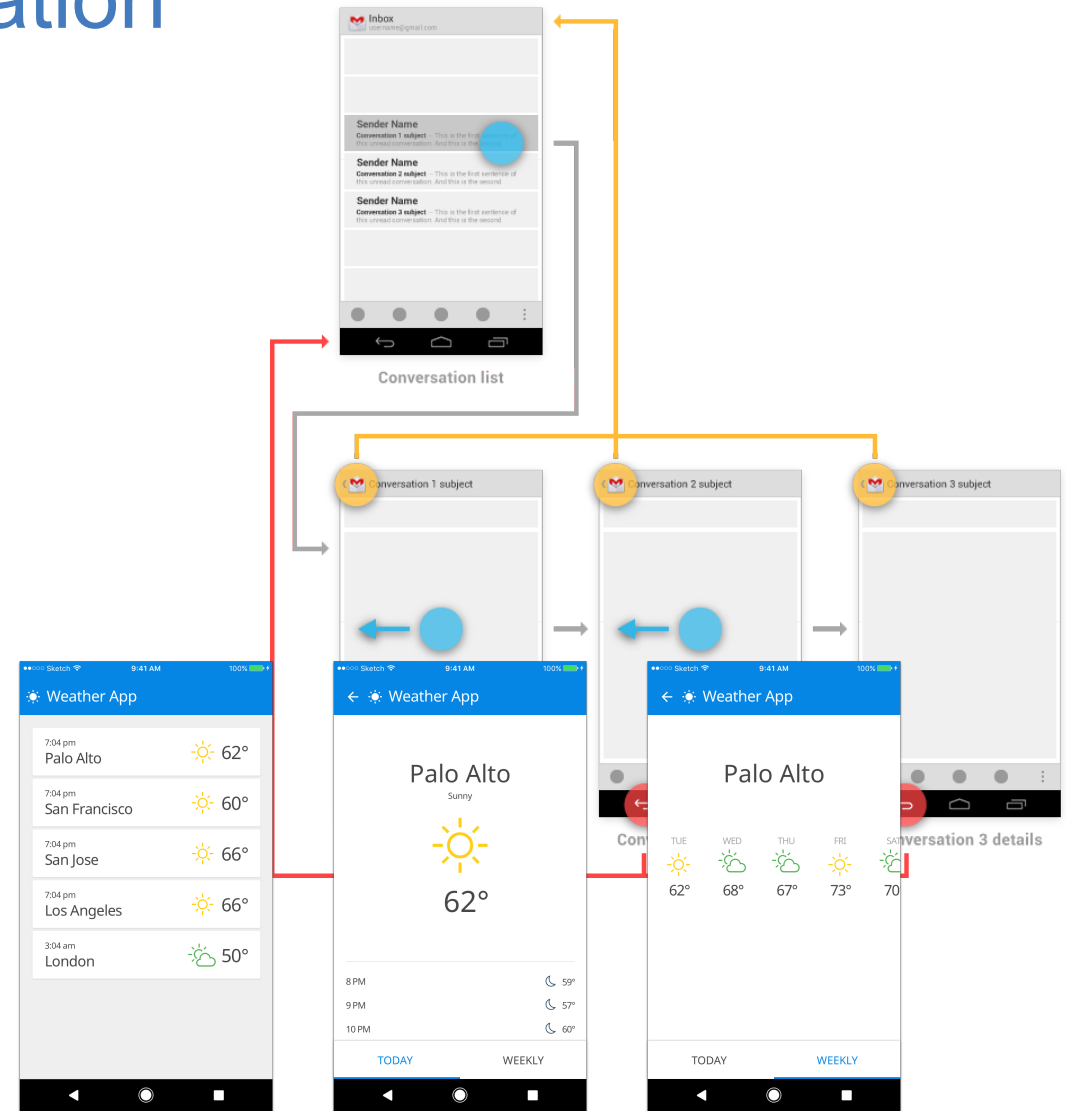
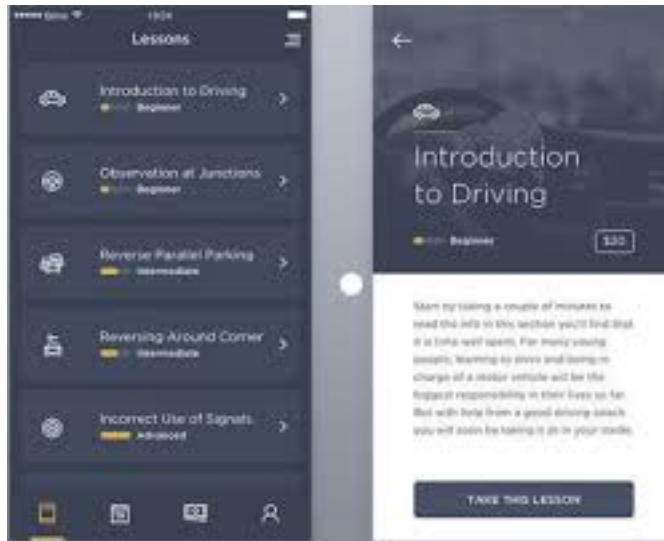
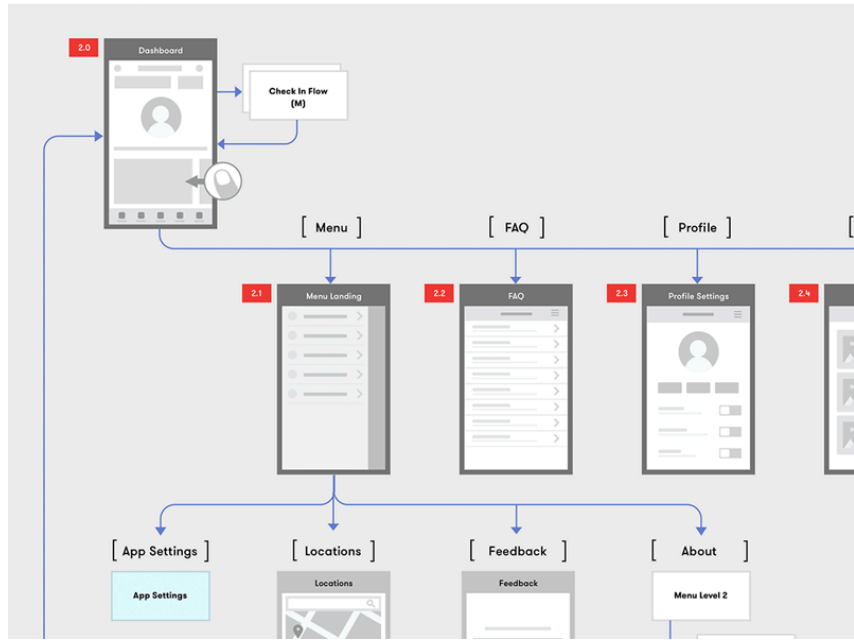
## Let Go

Trust the framework to update the  
view when state changes.

# Navigation in Modern UI Frameworks

A deep dive into Flutter, React Native, and Jetpack Compose.

# Navigation



---

# Part 1: The Mental Model Shift

Moving from Imperative to Declarative

Routing

---

# The Architecture Shift

## Imperative (Old)

Focuses on the **action** of moving.

```
// Push Screen B  
navigator.push(ScreenB());
```

*Problem:* If the app crashes, history is lost. Deep linking is fragile.

## Declarative (New)

Focuses on the **state** of the app.

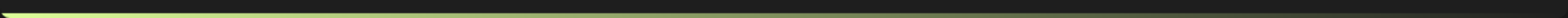
```
// State determines Route  
if (state.hasDetails) {  
  return DetailsScreen();  
}
```

*Benefit:* The URL is a reflection of state. History can be rebuilt from any point.



# Flutter

The Router API and `go_router`



# Flutter

- Routes
  - Pages/Screens
- Navigation
  - To move between Routes
  - Push and Pop

# Routes

```
class FirstRoute extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text('First Route'),  
      ),  
      body: Center(  
        child: RaisedButton(  
          child: Text('Open route'),  
        ),  
      ),  
    );  
  }  
}
```

```
class SecondRoute extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text("Second Route"),  
      ),  
      body: Center(  
        child: RaisedButton(  
          child: Text('Go back!'),  
        ),  
      ),  
    );  
  }  
}
```

# Navigation

```
child: RaisedButton(  
  child: Text('Open route'),  
  onPressed: () {  
    Navigator.push(  
      context,  
      MaterialPageRoute(builder: (context) =>  
        SecondRoute()),  
    );  
  },  
),
```

```
routes: {  
  '/': (context) => FirstScreen(),  
  '/info': (context) => SecondRoute(),  
},
```

```
Navigator.pushNamed(context, '/info');
```

```
child: RaisedButton(  
  onPressed: () {  
    Navigator.pop(context);  
  },  
  child: Text('Go back!'),  
),
```

# Centralized Routing

Flutter's `go_router` allows you to define a "map" of your entire app structure upfront.

- Eliminates "spaghetti code" navigation calls.
- Deep linking is a first-class citizen.
- Redirects are handled declaratively at the top level.



---

# React Native

Component-based vs. File-System

Routing

---

# Two Major Paradigms

## 1. React Navigation (Standard)

The traditional, imperative, component-based approach. You define a `Stack.Navigator` and wrap screens.

## 2. Expo Router ('Modern')

Inspired by Next.js. File-system based routing. Files in the `app/` directory automatically become routes.

→ `app/index.js` → `/`

→ `app/details/[id].js` → `/details/123`



# React-Native

- React-Native
  - Multiple ways, both standard and external libraries
- Screens
  - Stack-Navigator
  - Routes describe Screens

# Screen (Just a component)

```
function AndersScreen({ navigation }) {  
  return (  
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>  
      <Text>Home Screen</Text>  
      <Button  
        title="Go to TDDC73"  
        onPress={() => navigation.navigate('TDDC73')}  
      />  
    </View>  
  );  
}
```

```
Function App(){  
  Yada yada ...  
  
  return (<NavigationContainer>  
    <Stack.Navigator initialRouteName="Home">  
      <Stack.Screen name="Anders" component={AndersScreen} />  
      <Stack.Screen name="TDDC73" component={TDDC73Screen} />  
    </Stack.Navigator>  
    </NavigationContainer>);  
}
```

# Push,navigate,Back

```
function TDDC73Screen({ navigation }) {
  return (
    .....
    <Button
      title="Go to Anders... again"
      onPress={() => navigation.navigate('Anders')}
    />
    <Button
      title="Go back"
      onPress={() => navigation.goBack()}
    />
    <Button
      title="Go to Anders"
      onPress={() => navigation.push('Anders')}
    />
    </View>
  );
}
```



# Jetpack Compose

Type-Safe Navigation in Kotlin

# No More Strings

Historically, Android navigation relied on fragile string paths like "profile/{id}". The new standard uses **Type-Safe Navigation**.

- Define routes as @Serializable objects.
- Pass arguments as typed properties, not string segments.
- Compile-time safety guarantees.



# Serializable Routes

Instead of manually parsing strings, we define data classes.

The library handles serialization automatically.

This prevents crashes caused by typos in route names or missing arguments.

```
@Serializable
data class Profile(val id: String)

NavHost(navController, startDestination = Home) {
    composable { backStackEntry →
        // Arguments are type-safe!
        val profile: Profile =
backStackEntry.toRoute()
            ProfileScreen(id = profile.id)
    }
}
```

# Key Challenges



## Complex Objects

**Anti-Pattern:** Passing full User objects in the URL. Android can kill processes, losing large data payloads. Always pass IDs and re-fetch.



## Nested Navigation

Managing independent stacks for Bottom Tabs (e.g., Home vs. Settings) requires explicit state saving/restoring in Compose.



## Deep Linking

Syncing web URLs with app state is tricky. Requires precise manifest configuration (Android) or Info.plist (iOS) setup.

# Summary Comparison

Feature	Flutter (go_router)	React Native (Expo)	Jetpack Compose
<b>Definition</b>	Centralized List	File-System Based	Type-Safe DSL
<b>Arguments</b>	Path Params (String)	Search Params (String)	Serializable Objects
<b>Deep Linking</b>	Native Router Support	Automagic (Expo)	Intent Filters
<b>Boilerplate</b>	Medium	Very Low	Medium

# Advanced Networking

Architecture & Asynchronous Patterns in Modern UI

Frameworks

Flutter · React Native · Kotlin

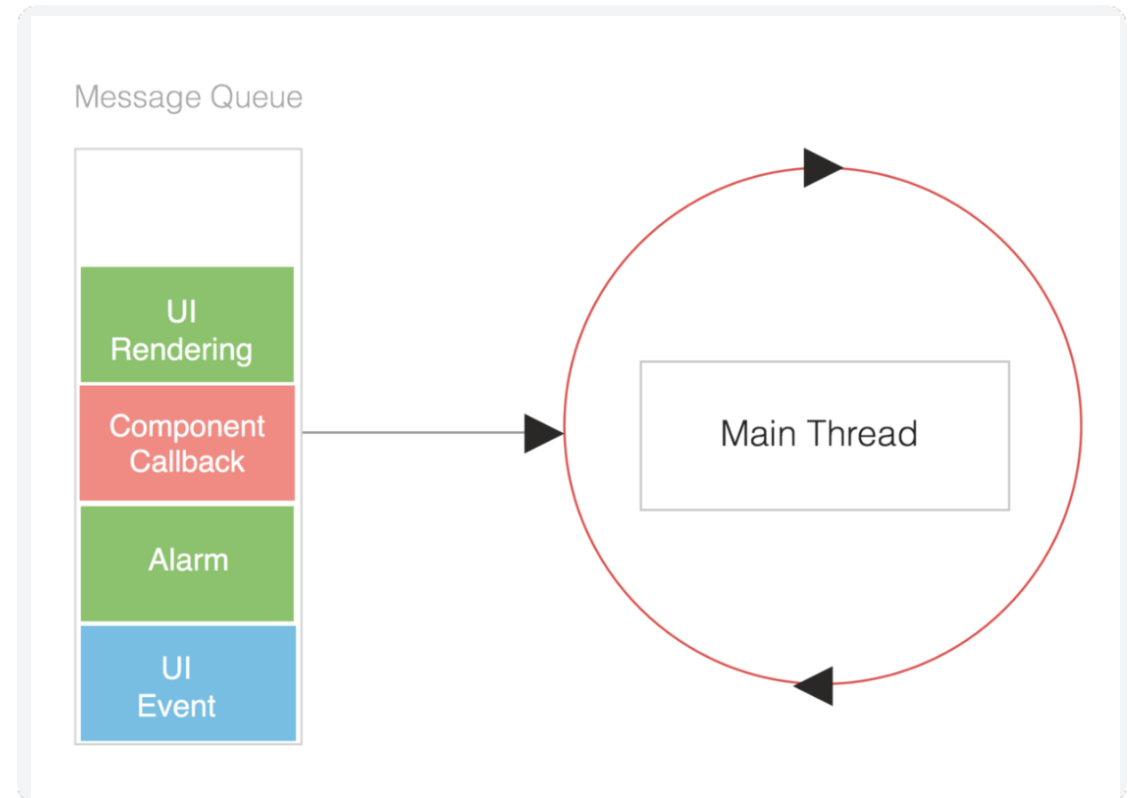
# The Foundation: Asynchronous Programming

## The Golden Rule

Never block the Main Thread (UI Thread). It runs at 60fps (16ms per frame).

Network requests are slow (50ms - 5000ms). Running them on the main thread causes the app to freeze (ANR).

We must offload these tasks to background threads or event loops using **Futures**, **Promises**, or **Coroutines**.



# Why Async? Visualizing "The Block"

---

Imagine your app is a single railway track. If a heavy train stops on the track, nothing else moves.

**Blocking...**

Network Request (Waiting 2s). UI  
Freezes.

**4. Update UI**

Data returns and updates screen.



**1. User Tap**

Instant feedback required.



**Non-Blocking**

Request moves to background. UI  
continues.

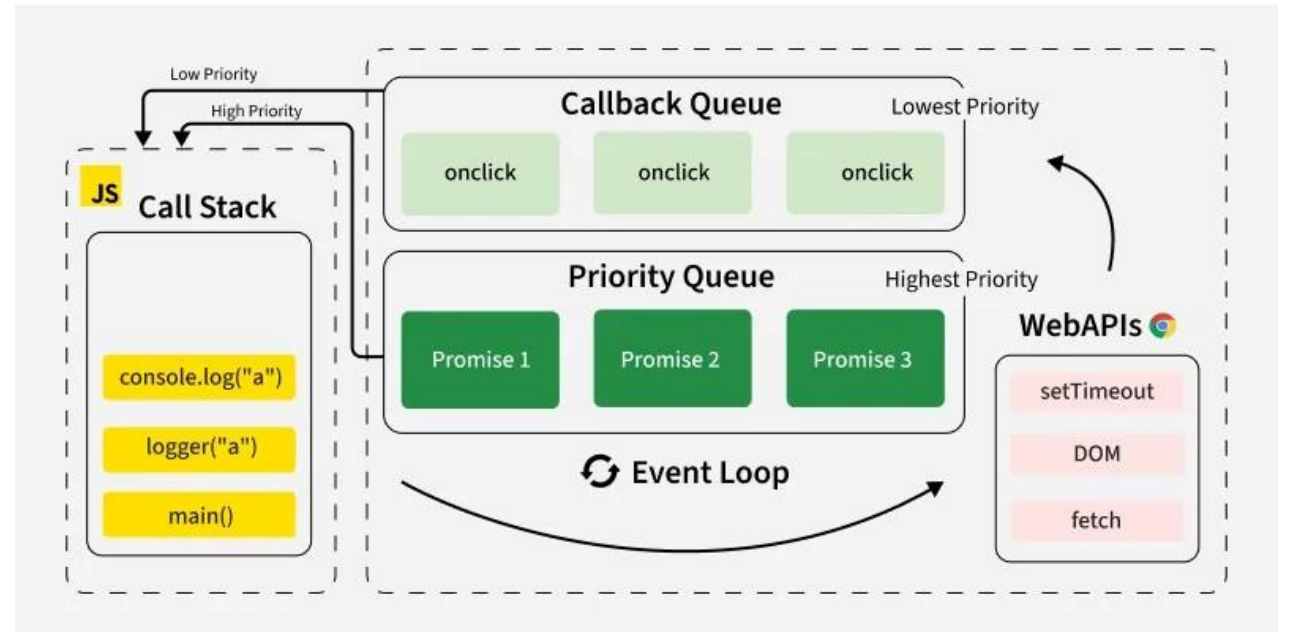


# Model 1: The Event Loop (Flutter/JS)

## Single Threaded Efficiency

Dart and JS cannot technically do two things at once. They rely on an **Event Loop**.

- Code runs until it hits an await.
- The heavy task is offloaded to the system.
- The Event Loop continues processing user inputs and drawing frames.
- When data returns, the task is queued to run next.



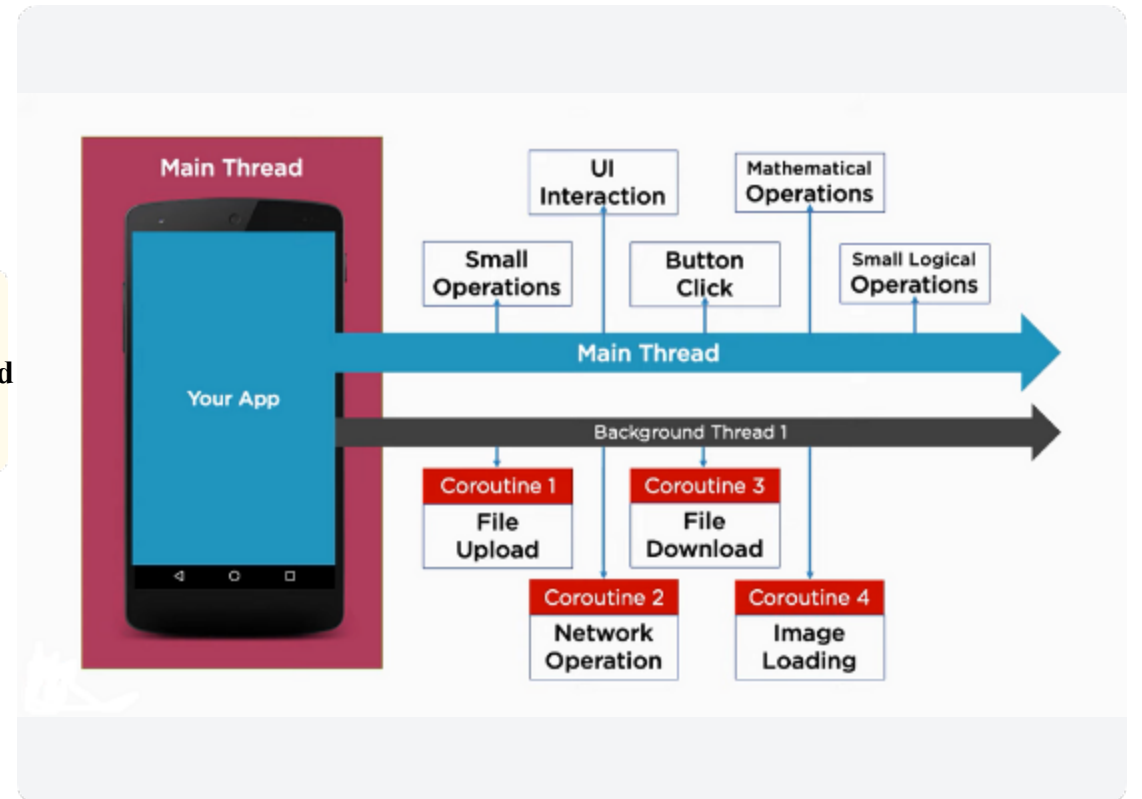
# Model 2: Coroutines (Kotlin)

## Lightweight Threads

Kotlin uses **Coroutines**. Unlike OS threads which are expensive, you can run 100,000+ coroutines on a single thread.

### Suspend vs Block

When a function **suspends**, it pauses execution but **frees up the underlying thread** to do other work.



# Flutter: Async Syntax

---

## Futures & Await

A Future is a handle to a value that doesn't exist yet. await unboxes it.

- async: Marks function as returning a Future.
- await: Pauses execution of the current function.

```
Future<String> getData() async {  
  // Simulates network delay  
  await Future.delayed(Duration(seconds: 2));  
  return "Loaded";  
}
```

```
void main() async {  
  print("Start");  
  String result = await getData();  
  print(result);  
}  
  
void main() async {  
  String data="No data"  
  print("Start");  
  getData().then((value)=>data=value);  
  print(result);  
}
```

# React Native: Async Syntax

---

## Promises

Standard JS Promises representing eventual completion.

- `async/await` is syntactic sugar over Promises.
- Keeps code looking synchronous and readable.

```
const getData = () => {  
  return new Promise((resolve) => {  
    setTimeout(() => {  
      resolve("Loaded");  
    }, 2000);  
  });  
};
```

```
const main = async () => {  
  console.log("Start");  
  const result = await getData();  
  console.log(result);  
};
```

# Kotlin: Async Syntax

---

## Suspend Functions

The suspend keyword is magic.

- Can only be called from a coroutine or another suspend function.
- Used with viewModelScope in Android.

```
suspend fun getData(): String {  
    delay(2000L) // Non-blocking pause  
    return "Loaded"  
}
```

```
suspend fun main() {  
    println("Start")  
    val result = getData()  
    println(result)  
}
```

```
fun main()=runBlocking {  
    println("Start")  
    val result = getData()  
    println(result)  
}
```

# Protocol Comparison

---

## REST

- **Resource Based:** /users, /posts
- **Fixed Data:** Server decides what you get.
- **Caching:** Easy (standard HTTP).

## GraphQL

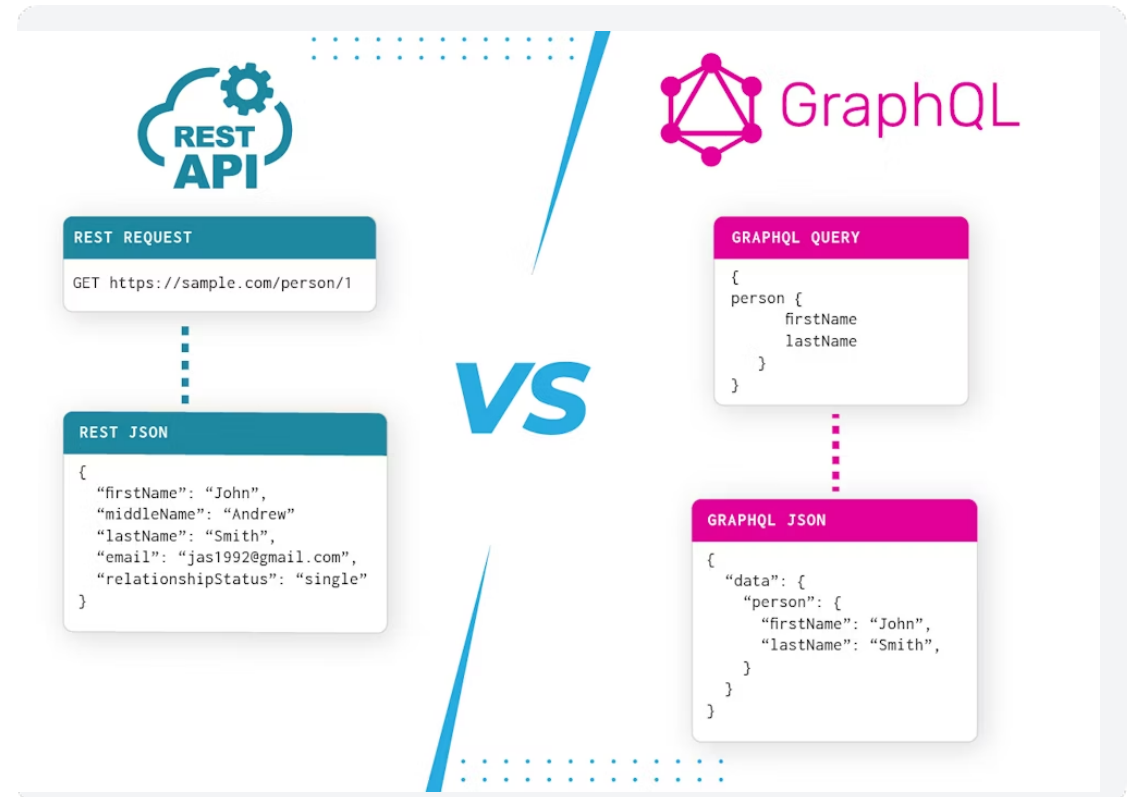
- **Query Based:** Single endpoint /graphql.
- **Client Driven:** You ask for specific fields.
- **Caching:** Harder (requires specialized clients).

# The Problem: Over-fetching

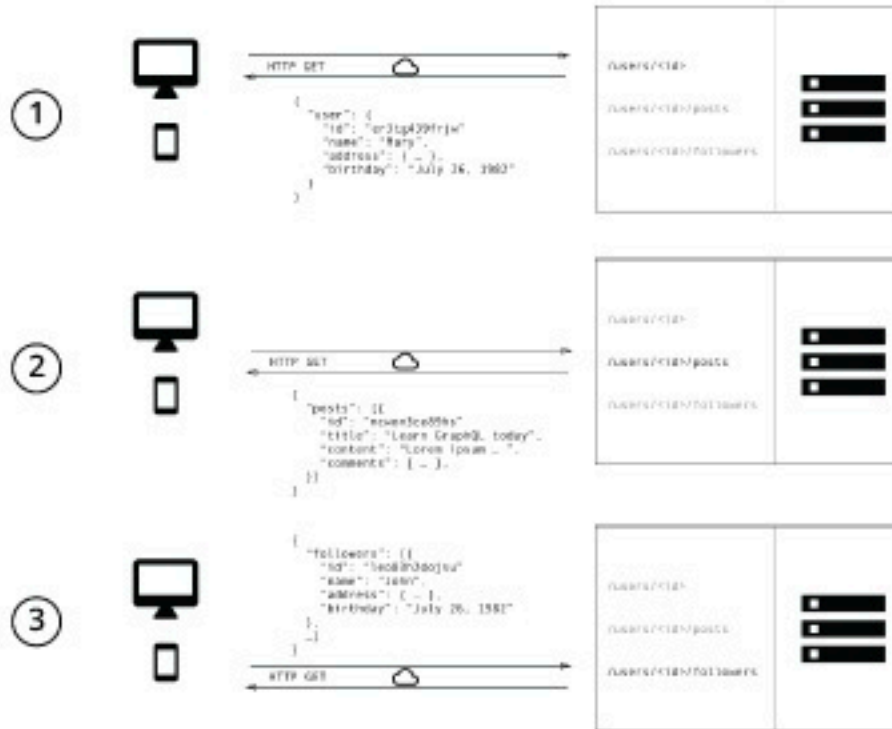
In REST, you often download more than you need.

**Scenario:** You just need a user's name.

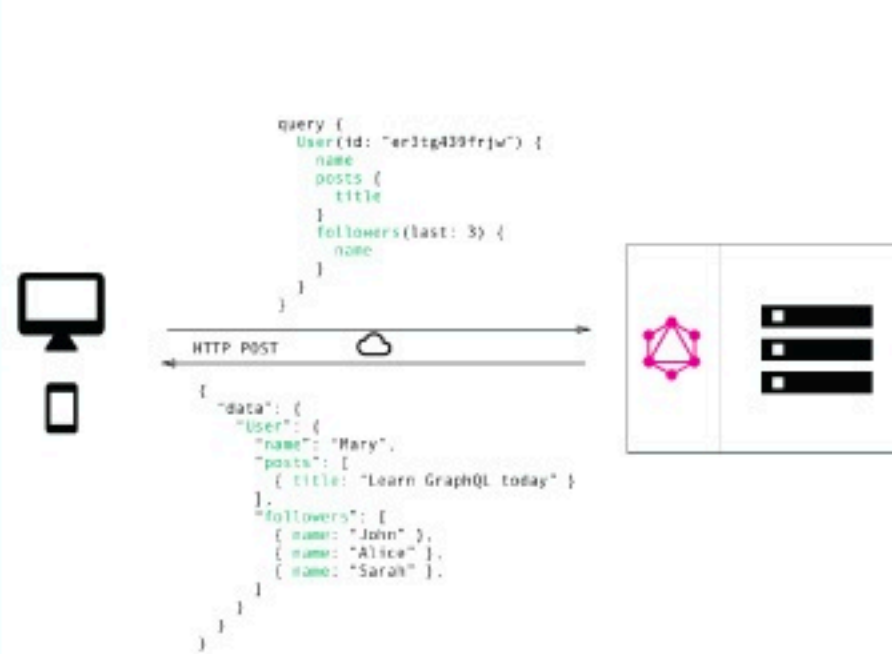
**Result:** You get their name, address, bio, history, and preferences.



# REST API



# GraphQL



# REST vs. GraphQL

## REST

```
GET /api/pizza/cheese
GET /api/pizza/supreme
GET /api/pizza/margherita
```

- Filter down the data
- Perform waterfall requests for related data
- Aggregate the data yourself

## GraphQL

```
query {
  pizza {
    cheese {
      mozzarella
    }
    toppings {
      basil
      mushrooms
    }
  }
}
```

- Receive exactly what you ask for
- No aggregating or filtering data

# GraphQL

- A query language runtime for server-side
- Front-end driven
  - You decide what you want
  - Subway vs Pressbyrån
- Single endpoint
- Typed

# Schema Fields and Types

```
type Query {  
  me: User  
}  
  
type User {  
  id: ID  
  name: String  
}
```

```
type Query {  
  bookById(id: ID): Book  
}
```

```
type Book {  
  id: ID  
  name: String  
  pageCount: Int  
  author: Author  
}
```

```
type Author {  
  id: ID  
  firstName: String  
  lastName: String  
}
```

# Queries

```
{
  bookById(id: "book-1"){
    id
    name
    pageCount
    author {
      firstName
      lastName
    }
  }
}
```



```
{
  "bookById": {
    "id": "book-1",
    "name": "Harry Potter and the Philosopher's Stone",
    "pageCount": 223,
    "author": {
      "firstName": "Joanne",
      "lastName": "Rowling"
    }
  }
}
```

# Flutter: REST (Dio)

---

## Dio Package

A powerful HTTP client for Dart.

- Automatic JSON serialization for payloads.
- Robust error handling with `DioException`.

```
final dio = Dio();

Future<void> createPost() async {
  try {
    final res = await dio.post(
      'https://api.../posts',
      data: { 'title': 'Hello' },
    );
    print(res.data['id']);
  } on DioException catch (e) {
    print(e.message);
  }
}
```

# Flutter: GraphQL

---

## Mutation Widget

The `graphql\_flutter` package uses widgets to handle state.

- Declarative UI updates.
- Variables passed via `runMutation`.

```
Mutation(  
  options: MutationOptions(  
    document: gql(r"""  
      mutation Add($title: String!) {  
        create(title: $title) { id }  
      }  
    """),  
  ),  
  builder: (runMutation, result) {  
    return Btn(  
      onPressed: () => runMutation({  
        'title': 'New Post'  
      }),  
      child: Text('Submit'),  
    );  
  },  
);
```

# React Native: REST (Axios)

---

## Axios

Standard choice for JS/TS.

- Automatic JSON parsing/stringifying.
- Clean `async/await` syntax.

```
import axios from 'axios';

const sendData = async () => {
  try {
    const res = await axios.post(
      'https://api.../posts',
      { title: 'Hello' }
    );
    console.log(res.data.id);
  } catch (err) {
    console.error(err);
  }
};
```

# React Native: GraphQL

---

## Apollo Hooks

Apollo Client integrates via Hooks.

- `useMutation` returns a trigger function.
- Handles loading/error state automatically.

```
const ADD = gql`
  mutation Add($title: String!) {
    create(title: $title) { id }
  }
`;

function MyComponent() {
  const [addPost] = useMutation(ADD);

  return (
    <Button
      title="Save"
      onPress={() => addPost({
        variables: { title: "Hi" }
      })}
    />
  );
}
```

# Kotlin: REST (Retrofit)

---

## Type Safety

Retrofit turns your API into an Interface.

- Strict data classes for Request/Response.
- Integrates seamlessly with Coroutines.

```
interface Api {  
    @POST("posts")  
    suspend fun add(@Body p: Post): Resp  
}
```

```
viewModelScope.launch {  
    try {  
        val res = api.add(Post("Hi"))  
        println(res.id)  
    } catch (e: Exception) {  
        println(e)  
    }  
}
```

# Kotlin: GraphQL (Apollo)

---

## Code Generation

Apollo Kotlin generates models from your `.graphql` files.

- No manual parsing.
- Compile-time safety for queries.

```
viewModelScope.launch {
    val res = client.mutation(
        CreatePostMutation(title = "Hi")
    ).execute()

    if (res.hasErrors()) {
        println(res.errors)
    } else {
        println(res.data?.create?.id)
    }
}
```

# The Power of Interceptors

---

## Request Interceptor

Runs **before** the request leaves.

- Add Auth Tokens (Bearer).
- Add Headers (Language, OS).
- Logging.

## Response Interceptor

Runs **after** the response arrives.

- Global Error Handling.
- Token Refresh on 401.
- Data formatting.

# Flutter: Dio Interceptors

---

## Centralized Logic

Add this once to your Dio instance, and every API call is authenticated automatically.

```
dio.interceptors.add(
  InterceptorsWrapper(
    onRequest: (opts, handler) {
      opts.headers['Auth'] = 'Bearer $token';
      return handler.next(opts);
    },
    onError: (err, handler) {
      if (err.response?.statusCode == 401) {
        // Refresh Token Logic
      }
      return handler.next(err);
    },
  ),
);
```

# React Native: Axios Interceptors

---

## Global Configuration

Axios allows chaining interceptors for granular control.

```
axios.interceptors.request.use((cfg) => {  
  cfg.headers.Auth = `Bearer ${token}`;  
  return cfg;  
});
```

```
axios.interceptors.response.use(  
  (res) => res,  
  (err) => {  
    if (err.response.status === 401) {  
      // Logout or Refresh  
    }  
    return Promise.reject(err);  
  }  
);
```

# Kotlin: OkHttp Interceptors

---

## The Chain

OkHttp uses the chain of responsibility pattern.

```
class AuthInterceptor : Interceptor {  
    override fun intercept(chain: Chain): Resp {  
        val req = chain.request()  
        .newBuilder()  
        .header("Auth", "Bearer $token")  
        .build()  
  
        return chain.proceed(req)  
    }  
}
```

# Image Sources

---



[https://images.ctfassets.net/piwi0eufbb2g/6rBPAuvllhrPRcmvUDqzUT/7dd8f711df3cd6993ab2402aa83a42ee/Thread\\_in\\_Android\\_-\\_ss2.png](https://images.ctfassets.net/piwi0eufbb2g/6rBPAuvllhrPRcmvUDqzUT/7dd8f711df3cd6993ab2402aa83a42ee/Thread_in_Android_-_ss2.png)  
Source: [www.topcoder.com](http://www.topcoder.com)

---



<https://media.geeksforgeeks.org/wp-content/uploads/20250208123836185275/Event-Loop-in-JavaScript.jpg>  
Source: [geeksforgeeks.org](http://geeksforgeeks.org)

---



[https://miro.medium.com/0\\*Thub8gCOOP1NyjwT.png](https://miro.medium.com/0*Thub8gCOOP1NyjwT.png)  
Source: [medium.com](http://medium.com)

---



<https://imgix.cosmicjs.com/7d9a6f90-7080-11eb-87a2-9be5e90cdf74-GraphQLvsRest.png?w=1000&auto=format&dpr=1>  
Source: [www.cosmicjs.com](http://www.cosmicjs.com)

# Image Sources

---



[https://storage.googleapis.com/cms-storage-bucket/lookup\\_flutter\\_vertical.a9d6ce81aee44ae017ee.png](https://storage.googleapis.com/cms-storage-bucket/lookup_flutter_vertical.a9d6ce81aee44ae017ee.png)

Source: [flutter.dev](https://flutter.dev)

---



<https://upload.wikimedia.org/wikipedia/commons/a/a7/React-icon.svg>

Source: [en.wikipedia.org](https://en.wikipedia.org)

---



<https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEjC97Z8BResg5dIPqczsRCFhP6zewWX0X0e7fVPG->

[G7PuUZwwZVsi9OPoqJYkgqT2h0FI95SsmWzVEgpt8b8HAqFIlxZ98TFtY4IE0b8UrtVJ2HrJebRwl6C9DslsQDI9KnBlrdHS6LtkY/s1600/jetpack+compose+icon\\_RGB.png](https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEjC97Z8BResg5dIPqczsRCFhP6zewWX0X0e7fVPG-G7PuUZwwZVsi9OPoqJYkgqT2h0FI95SsmWzVEgpt8b8HAqFIlxZ98TFtY4IE0b8UrtVJ2HrJebRwl6C9DslsQDI9KnBlrdHS6LtkY/s1600/jetpack+compose+icon_RGB.png)

Source: [android-developers.googleblog.com](https://android-developers.googleblog.com)