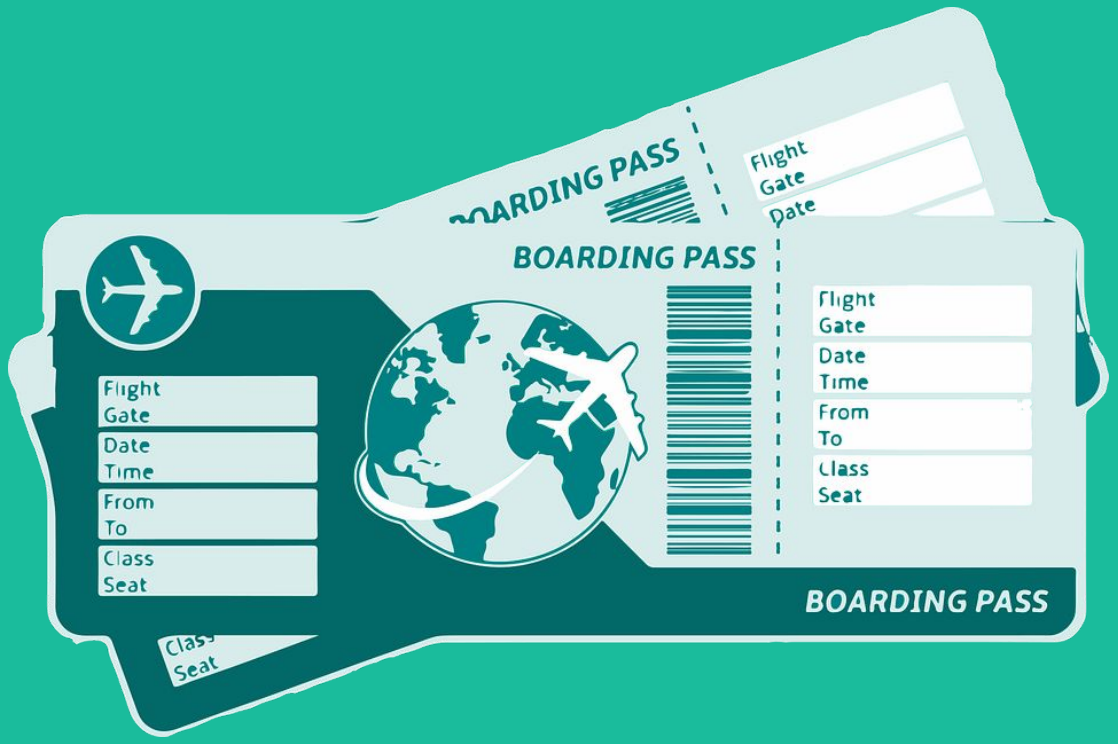
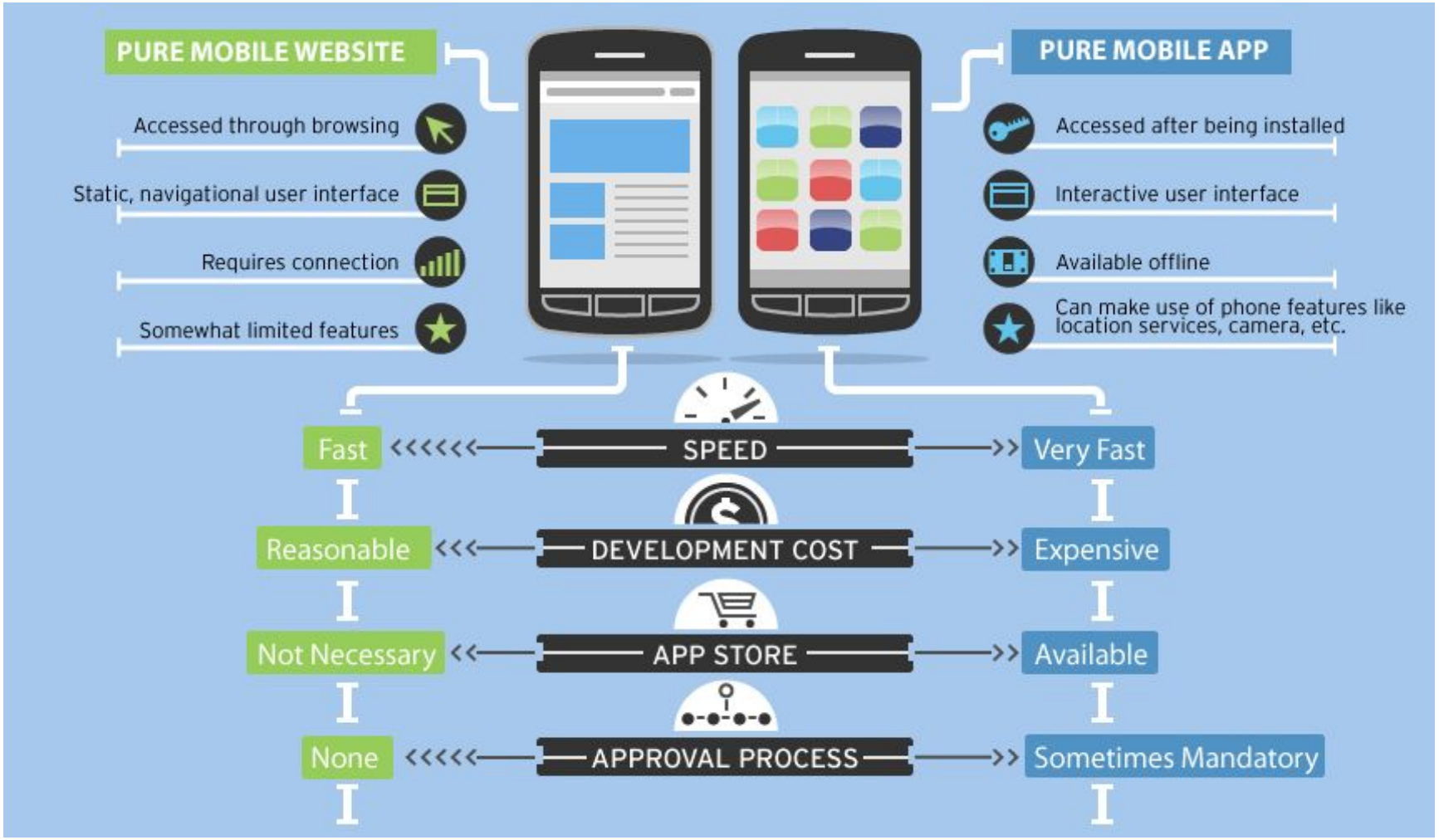


Chapter 11. Mobile Applications

Bilkent University | CS443 | 2020, Spring | Dr. Orçun Dayıbaş

Introduction

- **Mobile App vs. Mobile website**



https://www.businessinsider.com/a-mobile-website-might-be-what-your-business-needs-2013-3

Dart & Flutter

- **Why Dart & Flutter?**
  - Optimized for UI
    - Asynchronous by design
    - Shared-state concurrency is error-prone and can lead to complicated code. Instead of threads, all Dart code runs inside of isolates. Each isolate has its own memory heap, ensuring that no isolate's state is accessible from any other isolate
  - Productive Development
    - Flutter has a hot reload feature that helps you quickly and easily experiment, build UIs, add features, and fix bugs
    - Flutter provides static analysis which allows you to find problems before executing a single line of code
  - Fast on all Platforms
    - Dart has an AOT (Ahead of Time) compiler, which compiles to fast, predictable, native code that allows almost all of Flutter to be written in Dart. This not only makes Flutter fast but ensures that virtually everything (including all the widgets) can be customized

https://www.educative.io/courses/learn-dart-first-step-to-flutter

Dart

- **Hello world**
  - Dart uses imperative programming style
  - main() is the main entry point

```
1  main() {
2    // Printing the text 'Hello World'
3    print("Hello World");
4  }
```

- **Libraries**
  - Ex: dart:io provides I/O operations

```
1  import 'dart:io';
2
3  main() {
4      print("Hello " + stdin.readLineSync());
5  }
```

https://www.educative.io/courses/learn-dart-first-step-to-flutter

Dart

- **Object-oriented & Statically typed**
  - Very similar to Java
  - Language specification: https://dart.dev/guides/language/spec
  - Built-in data types: Numbers, Strings, Booleans, Lists, Sets, Maps, Runes, Symbols
  - Value & reference types
    - a reference type holds the memory address location of the value, while value types hold the value themselves
    - In most languages (like Java), primitive data types are value types, but in Dart, all data types are objects (all variables specifically store references and are referring to objects)

```
1  main() {
2      int notInitialized;
3      print(notInitialized);
4  }
```

https://dartpad.dartlang.org/

Dart

- **Numbers**
  - num, integer and double

```dart
void main() {
  num a = 12.2;
  int b = 2;
  double c = 2.33;
  int hex = 0x004F;
  print(a+b+c+hex);
}
```

- **Strings**

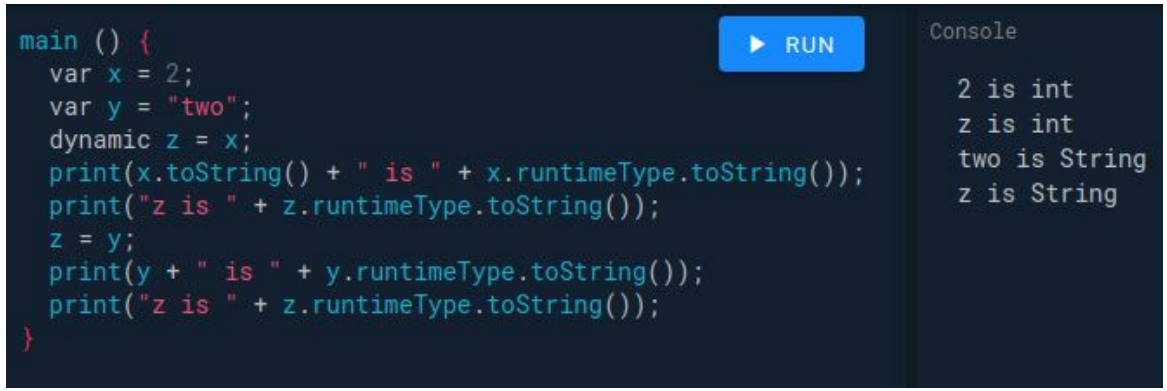
```dart
// Single Quotes
print('Using single quotes');

// Double Quotes
print("Using double quotes");

// Single quotes with escape character \
print('It\'s possible with an escape character');

// Double quotes
print("It's better without an escape character");
```

```dart
main() {
  String country = "Japan";

  print("I want to visit $country");
}
```

```dart
main() {
  var multilineString = """This is a
multiline string
consisting of
multiple lines""";

  print(multilineString);
}
```

Dart

- **Type inference**
  - Although types are mandatory in Dart, type annotations are optional because of type inference.
    - var → type is fixed by the first assignment
    - dynamic → a variable can hold objects of many types




```
main () {
  var x = 2;
  var y = "two";
  dynamic z = x;
  print(x.toString() + " is " + x.runtimeType.toString());
  print("z is " + z.runtimeType.toString());
  z = y;
  print(y + " is " + y.runtimeType.toString());
  print("z is " + z.runtimeType.toString());
}
```


```
main() {
  double type1 = 2.0;
  num type2 = 15;
  String type3 = "CS443";
  bool type4 = true;

  print(type1 is int);
  print((type2 as int)<<1);
  print((type1 as int)<<2);
  print(type3 is String);
  print(type4 is double);
}
```

- **Operators (Arithmetic, Equality, Relational)**
  - Nothing special (very similar to Java)

Dart

- **Other operators**

Operator	Name	Meaning
`()`	Function application	Represents a function call
`[]`	List access	Refers to the value at the specified index in the list
`.`	Member access	Refers to a property of an expression; example: `foo.bar` selects property `bar` from expression `foo`
`?.`	Conditional member access	Like `.`, but the leftmost operand can be null; example: `foo?.bar` selects property `bar` from expression `foo` unless `foo` is null (in which case the value of `foo?.bar` is null)

- **Cascade notation**
  - to make a sequence of operations on the same object

```
querySelector('#confirm') // Get an object.
  ..text = 'Confirm' // Use its members.
  ..classes.add('important')
  ..onClick.listen((e) => window.alert('Confirmed!'));
```

```
var button = querySelector('#confirm');
button.text = 'Confirm';
button.classes.add('important');
button.onClick.listen((e) => window.alert('Confirmed!'));
```

Dart

- **Control flow statements**
  - if, else, for, while, do-while, etc.
  - Nothing special (very similar to Java)
- **Collections**
  - List, Set, Map, etc.
  - Nothing special (very similar to Java)
- **Class definition**
  - Neither main() nor Bicycle is declared as public, because all identifiers are public by default.
  - Dart doesn't have keywords for public, private, or protected.
  - “new” is optional (to create an instance)

```
class Bicycle {
  int cadence;
  int _speed = 0;
  int get speed => _speed;
  int gear;

  Bicycle(this.cadence, this.gear);

  void applyBrake(int decrement) {
    _speed -= decrement;
  }

  void speedUp(int increment) {
    _speed += increment;
  }

  @override
  String toString() => 'Bicycle: $_speed mph';
}

void main() {
  var bike = Bicycle(2, 1);
  print(bike);
}
```

Dart

- **Class definition (link)**
  - Uninitialized variables (even numbers) have the value null
  - The Dart compiler enforces privacy for any identifier prefixed with an underscore.
  - By default, Dart provides implicit getters and setters for all public instance variables.
    - You don't need to define your own getters/setters unless you want to enforce read-only or write-only variables, compute or verify a value, or update a value elsewhere.
    - instance variables can be accessed using bike.gear or bike.cadence.
  - Constructor without a body is a valid definition

```
Bicycle(this.cadence, this.speed, this.gear);
```

```
Bicycle(int cadence, int speed, int gear) {
  this.cadence = cadence;
  this.speed = speed;
  this.gear = gear;
}
```

  - Java vs. Dart
    - Dart version is more compact at 23 lines instead of 40

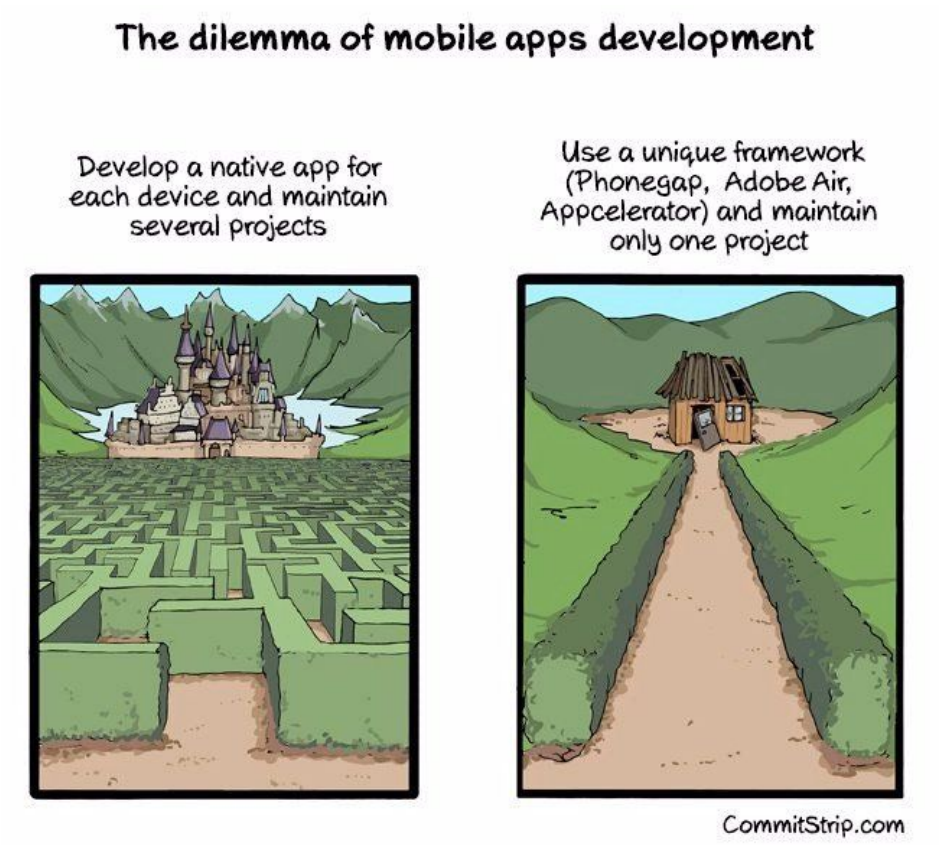
Dart

- **Function definition**
  - You can define top-level functions
  - “=>” can be used as a syntactic sugar (example)
- **Exception Handling**
  - syntax

```
try {
  // code that might throw an exception
}
on Exception1 {
  // code for handling exception
}
catch Exception2 {
  // code for handling exception
}finally {
  // code that should always execute; irrespective of the exception
}
```

Flutter

- **Motivation**
  - Open-source UI software development kit created by Google
  - Motto: “Build beautiful native apps in record time”



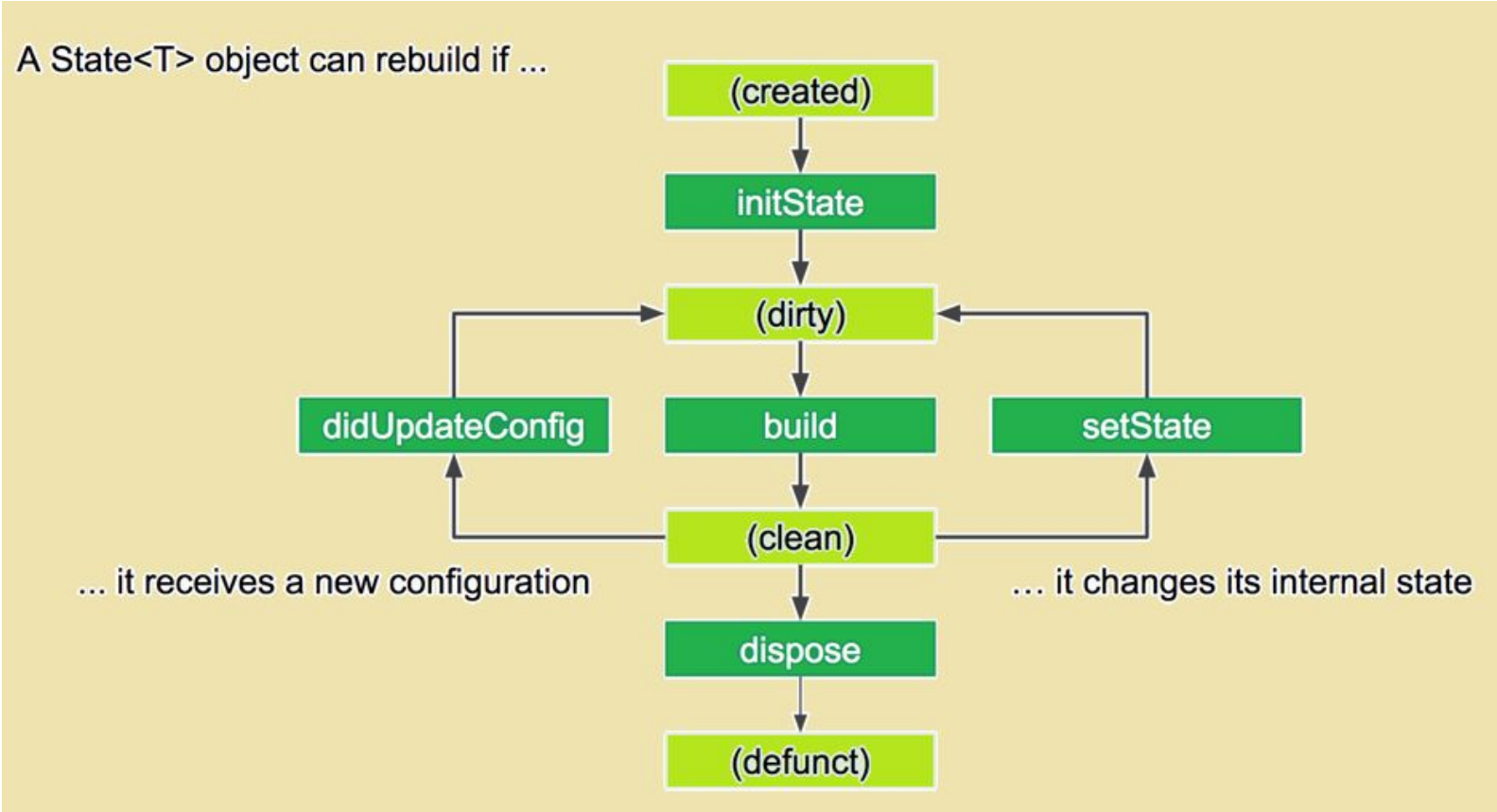
Flutter

- **Components**
  - **Flutter Engine :** It is primarily based on C++ language and is used to render low level graphics using Google's open source graphic library Skia.
  - **Foundation Library:** All the basic building blocks for writing a Flutter application are packaged under this and written in Dart.
  - **Widgets:** A widget in Flutter represents an immutable description of part of the user interface; all graphics, including text, shapes, and animations are created using widgets.
  - **Design Specific Widgets:** The Flutter framework contains two sets of widgets which conform to specific design languages namely Material Design for Android Side and Cupertino Style for iOS side.

https://medium.com/gradeup/mobile-development-approaches-and-flutter-architecture-flutter-part-i-a7e08838c97a

Flutter

- **Widgets & states**
  - Only stateful widget can hold a state



https://www.developerlibs.com/2019/12/flutter-lifecycle-widgets.html

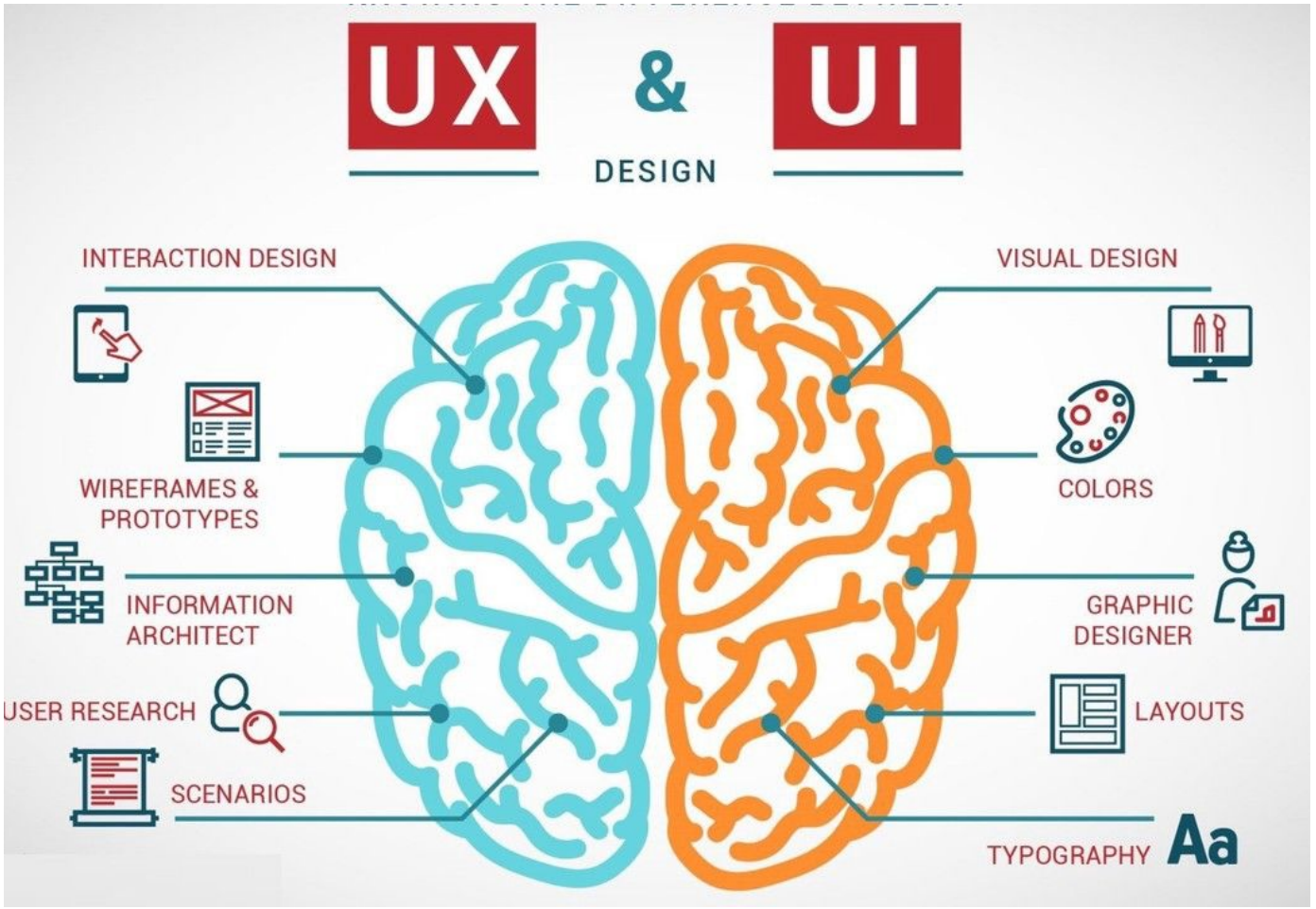
Flutter

- **Widgets & states**
  - Almost everything is a widget in Flutter
  - Although it's convenient, it's not recommended to put an API call in a build() method (use initState or Constructor).
    - Flutter calls the build() method every time it needs to change anything in the view, and this happens surprisingly often. Leaving the fetch call in your build() method floods the API with unnecessary calls and slows down your app.
- **Async & await**
  - Future is a core Dart class for working with async operations. A Future object represents a potential value or error that will be available at some time in the future
  - The http.Response class contains the data received from a successful http call
  - "await" is used to wait for the result of async call
- **Let's dive in...**

Hands-on Session (Flutter)

User Experience (UX)

- **UI vs. UX**



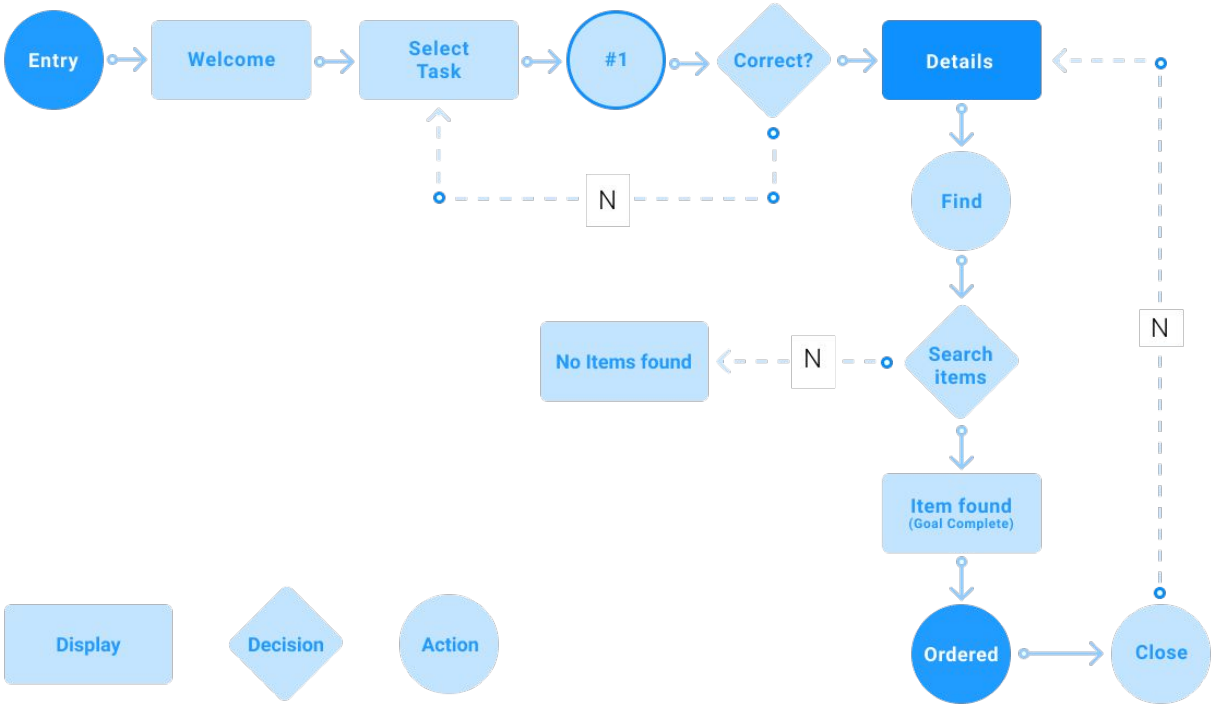
User Experience (UX)

- **7 factors**
  - Useful & Valuable
    - It has to have a purpose (clear, concise one)
    - It has to deliver a value
  - Usable
    - “The iPod wasn’t the first MP3 player but it was the first truly usable one”
  - Findable
    - The content within them must be easy to find
  - Credible
    - “The ability of the user to trust in the product that you’ve provided”
  - Desirable
    - Desirability is conveyed in design through branding, image, identity, aesthetics and emotional design
  - Accessable
    - Accessibility is about providing an experience which can be accessed by users of a full range of abilities

https://www.interaction-design.org/literature/article/the-7-factors-that-influence-user-experience

User Flow Design

- **User flows / UX flows**
  - Diagrams that display the complete path a user takes when using a product
  - The user flow lays out the user's movement through the product, mapping out each and every step the user takes

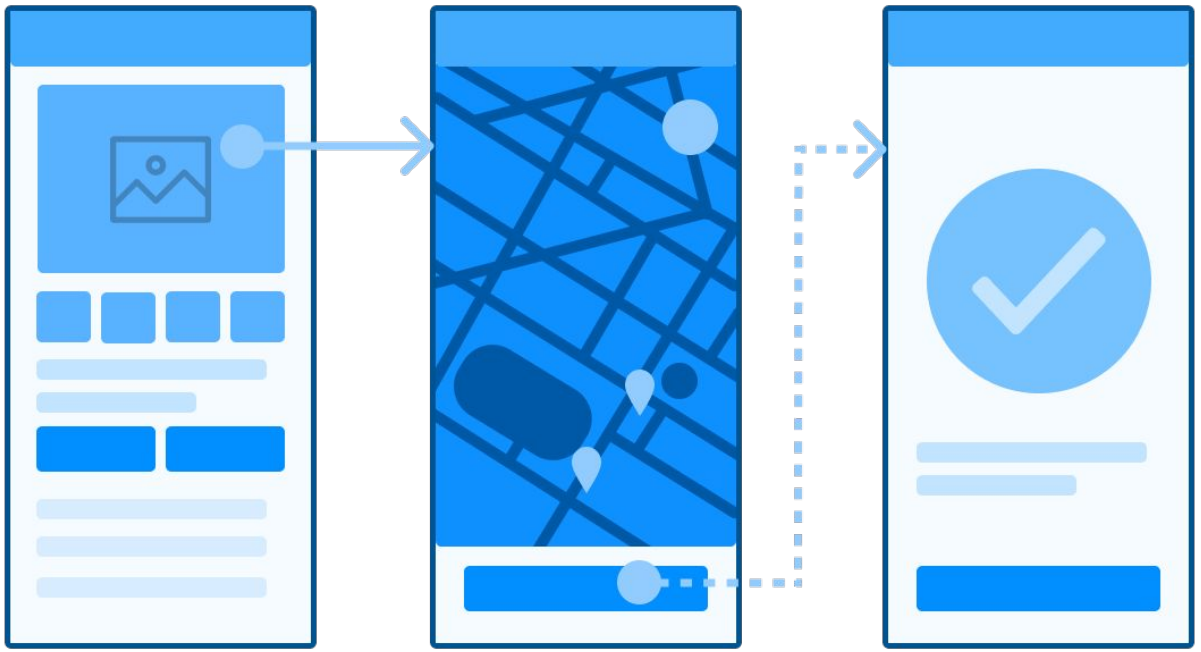


User Flow Design

- **Types of user flow charts**
  - Task flows
    - Task flows focus on how users travel through the platform while performing a specific task.
    - They generally show only one path and don’t include multiple branches or pathways like a traditional user flow might.
    - These are best used when the task being analyzed is accomplished similarly by all users. When using task flows, it is assumed that all users will share a common starting point and have no variability in the way the task is carried out.

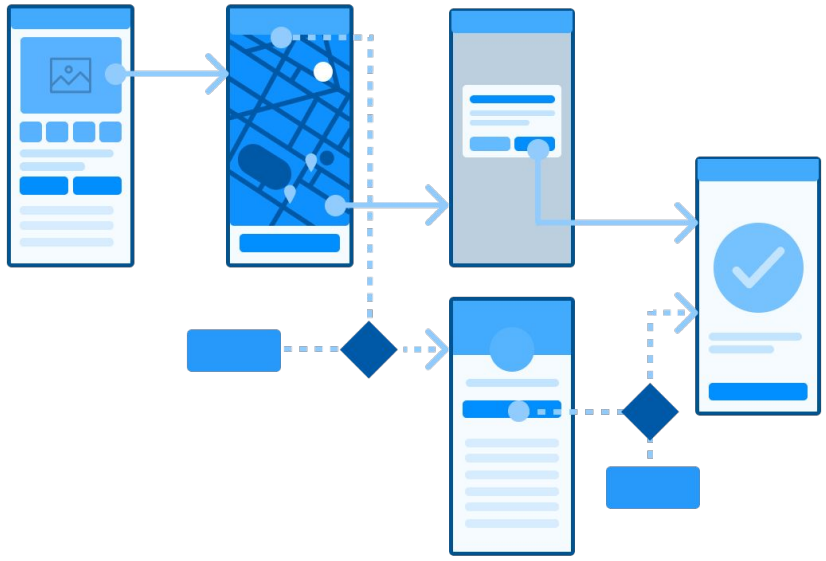
User Flow Design

- **Types of user flow charts**
  - Wire flows
    - Wireflows are a combination of wireframes and flowcharts.
    - They utilize the layout of individual screens as elements within the diagram.
    - Wireflows add page context to UX flows.



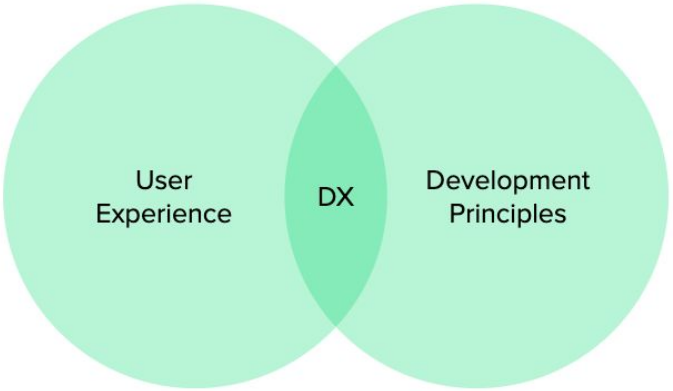
User Flow Design

- **Types of user flow charts**
  - User flows
    - User flows focus on the way your target audience will interact with the product.
    - They emphasize that all users might not perform tasks the same and may travel in different paths.
    - They are typically attached to a specific persona and entry point.
      - When using this type of flowchart, you may have many different scenarios that start at different places.
      - However, the main task or accomplishment is usually always the same.



Developer Experience (DX)

- **UX vs. DX**
  - Developer Experience is the equivalent of User Experience when the primary user of the product is a developer
    - DX cares about the developer experience of using a product, its libs, SDKs, documentation, frameworks, open-source solutions, general tools, APIs, etc.
    - Developers deserve solutions as well designed as non-technical people
  - DX and UX share some principles, but with differences in good practice
    - this is because developers have different needs in their daily context compared to an average user.
    - DX is important for the same reasons that UX is important

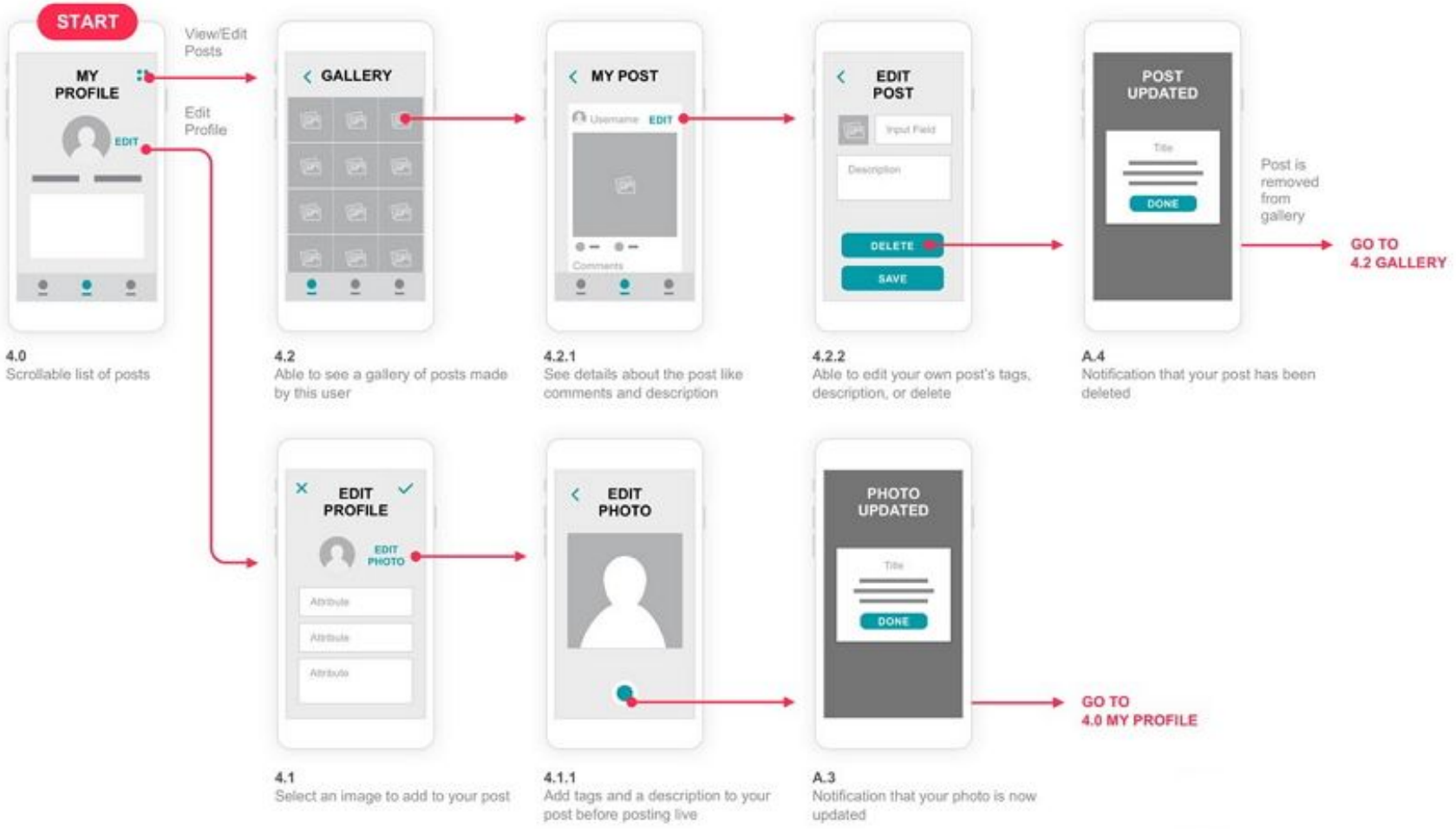


Developer Experience (DX)

- **Pillars of DX**
  - Function
    - The foundation of the developer experience, a dev tool is as good as the role it offers to perform an activity. If it doesn't work, it's no use, there's no DX.
  - Stability
    - In addition to working, your product has to have high performance and reliability, of course, the software is subject to bugs, so it is important to quickly fix product errors so as not to cause major harm to users.
    - Instability in the relationship of trust, the perception of value drops dramatically.
  - Ease of Use
    - Ease of use in DX is beyond what it seems, it's not just about navigating the tool, but also accessing what you need at all stages of the journey quickly and efficiently (Rich documentation, use cases, communities, knowledge bases, keyboard shortcuts, snippets, intuitive filters, etc.)
  - Clarity
    - Clarity is about giving the developer full visibility of the possible consequences involved in an action and the history of these actions.

Case Study

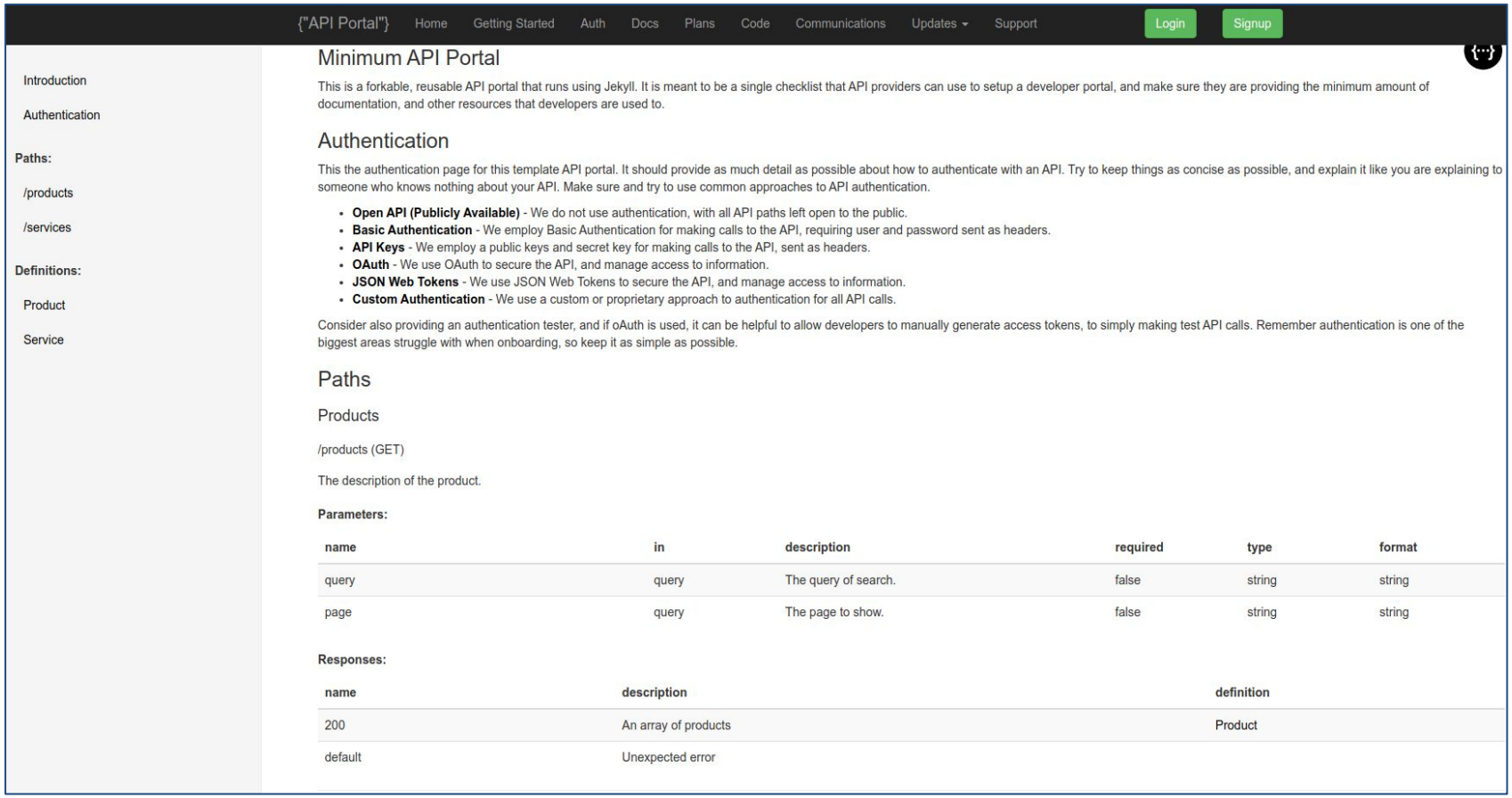
- **User flow design (Level 0)**



https://www.mockplus.com/blog/post/user-flow-tools

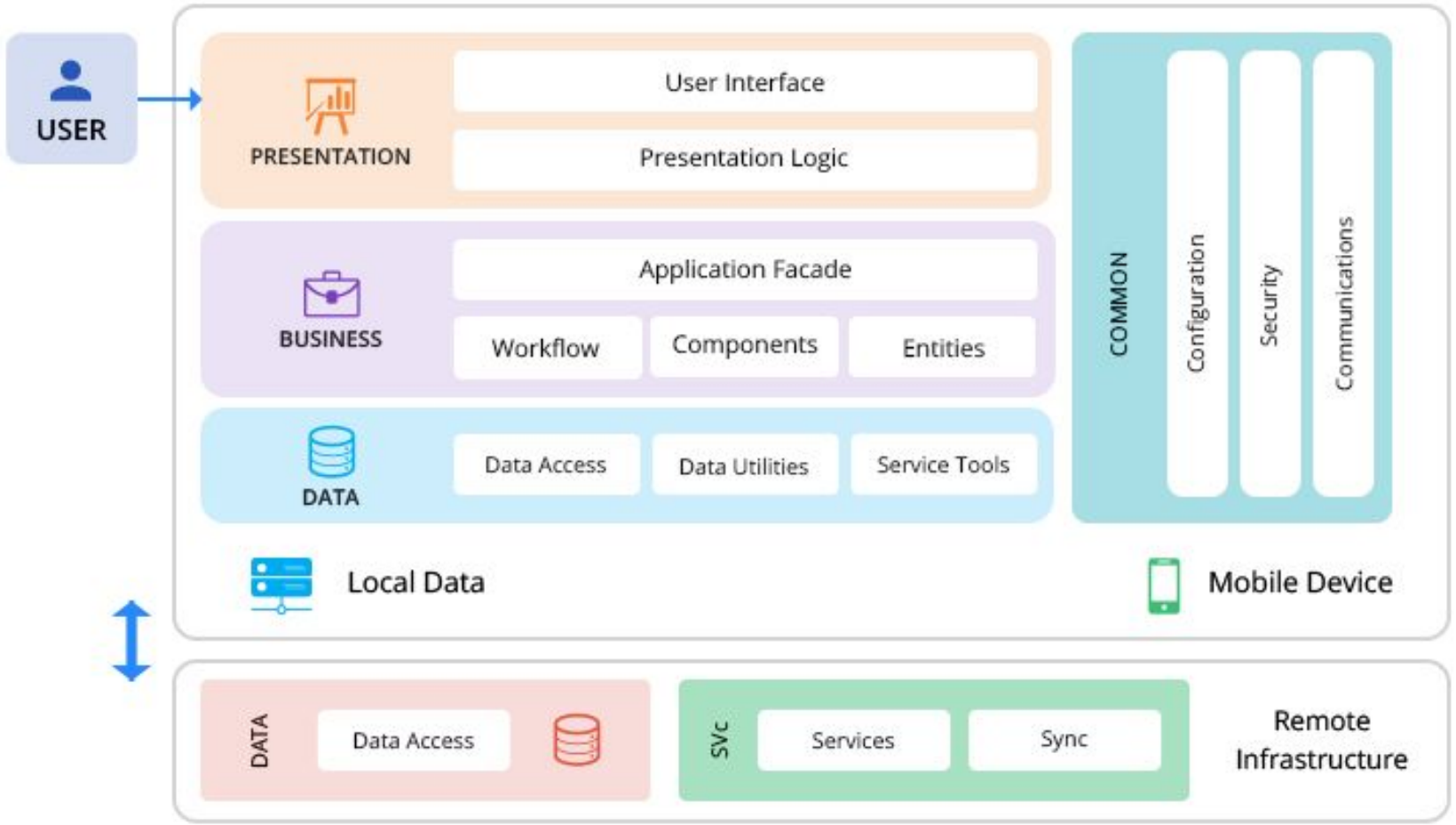
Case Study

- **DX design (Level 0)**
  - Ex: http://portal.minimum.apievangelist.com/



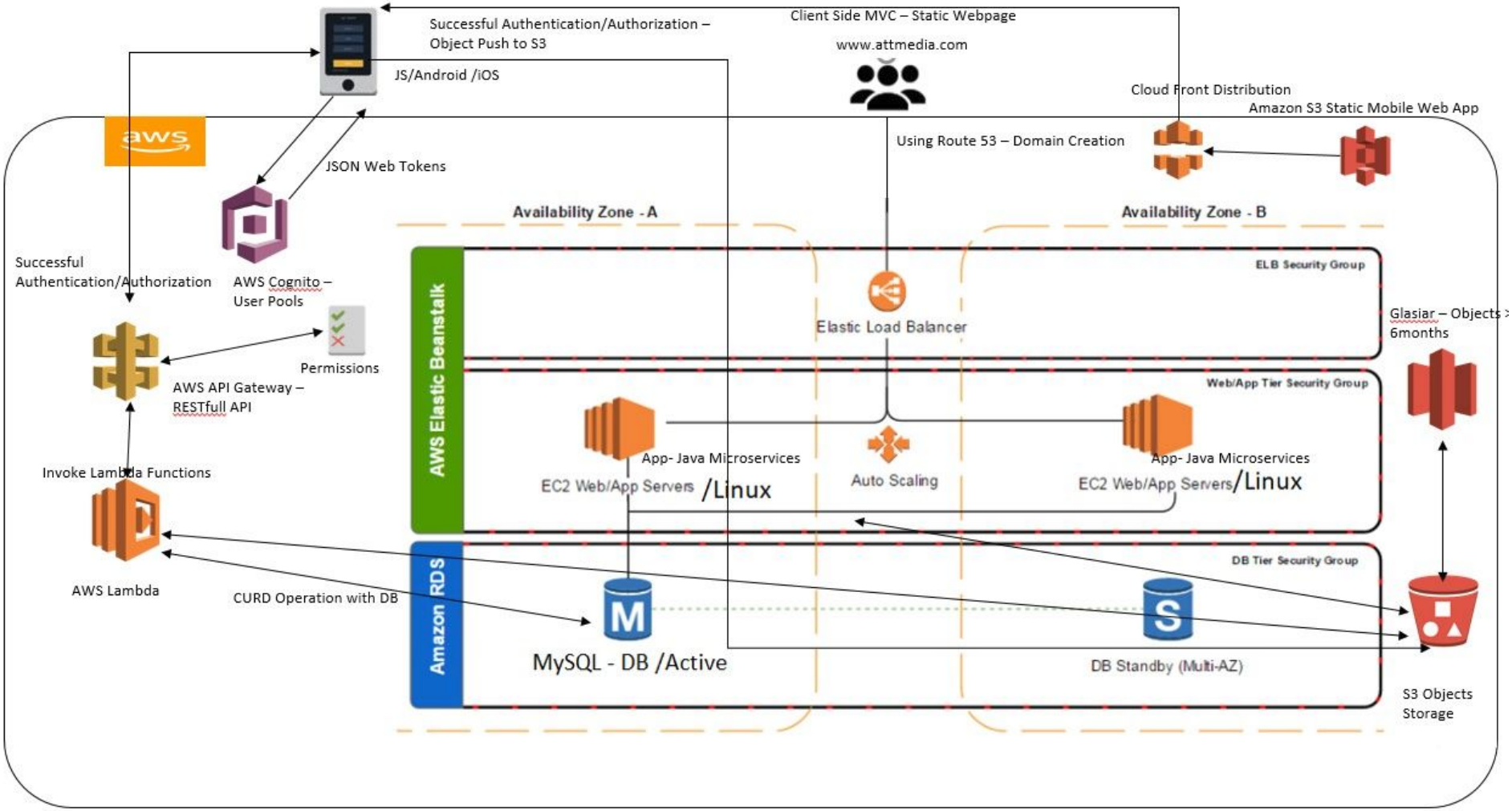
Case Study

- **Architecture (Level 1)**



Case Study

- **Architecture (Level 2)**



https://dzone.com/articles/mobile-based-architecture-on-existing-lamp-stack-e

Conclusion

- **Platform & frameworks**
  - Decide according to your business needs, ecosystem, maturity, existing know-how etc. Do not follow the trend blindly.
  - It is also valid for native / web / hybrid application dev.
- **UX & DX**
  - They are equally important if you have both B2C & B2B channels. Do not reinvent the wheel.
  - Follow practices like “Design Thinking”
  - Follow API design best practices
    - Ex: Zalando REST API Guidelines, Microsoft REST API Guidelines
- **Architecture**
  - Remember you need a “Level 3” if your solution is on-prem

Q/A