University of the West of England
BRISTOL

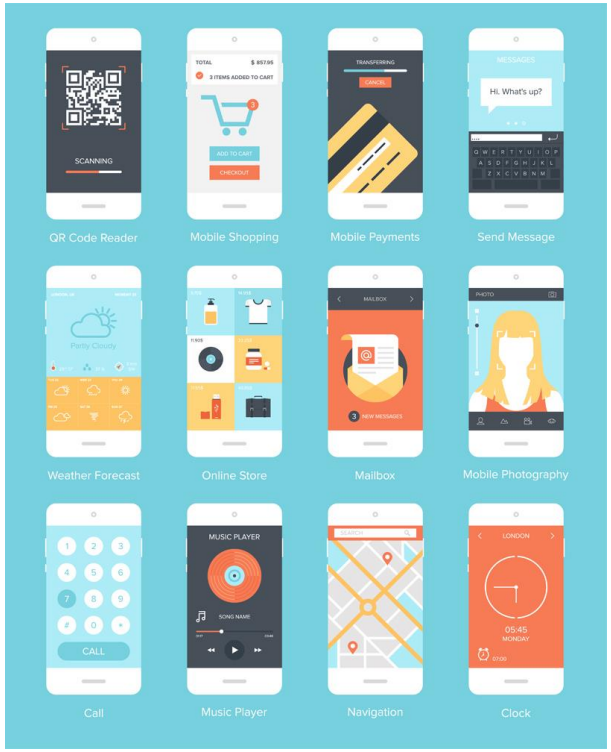
Week 7: Frameworks and Standards

UFCF7H-15-3 Mobile Applications

Dr Kun Wei

Essential factors in app quality

- With so many mobile apps available, especially between apps that claim to perform the same function, users are more sensitive to app quality as a differentiator.
- **If selecting between 5 similar apps, a user probably won't choose the app that crashes, is slow, and/or is not regularly updated with bug fixes and suffers performance improvements.**
- As such mobile app quality can be broken down into 4 equally important areas...



Nezar Mansour (2021)

1. Performance

Bad performance can make a mobile app unusable and users won't hesitate to delete it. Here are a few key aspects of performance:

- **Speed**: How fast things are executed by the mobile app. Screen transition speeds and how fast everything loads. If your app takes a longer than usual amount of time to start, users will notice.
- **UI performance**: The UI is what your users will be interacting with, so it's key that there are few to no hangs or other issues. If the app 'hangs', it will feel slow and frustrating.
- **Network performance**: One of the biggest factors affecting both speed & performance is network calls. Most applications rely on multiple services and network requirements. Any slow or failing network request hurts the overall performance of your app.

2. Stability

How reliable is your app? Is it behaving the way that it should? Commonly associated with crashes & errors, how often they occur is the measure of an app's stability, which affects app quality.

- Crashes: The worst-case scenario. A crash completely ends the user's session. Constant crashes are the biggest indicator of a poor quality app.
- Errors: Errors disrupt the user experience. If users experience too many errors or too often, they will probably delete your app.
- Resources: The most common cause of crashes & errors are resource constraints. Apps are often developed in a vacuum, but in the wild, an app would be running alongside a host of other apps. If an app consumes too many resources (CPU, memory, battery), it will negatively impact an app's stability & users (or OS) will delete it.

3. Testing

This is crucial in order to prevent unhandled errors & catch issues before they go into production. Integrating strong testing early on in an app's life cycle can save a lot of headaches & resources down the line.

- Functionality testing: Making sure your app is working as intended. Functionality testing helps eliminate any critical issues with the user journey and navigation flow of the app.

- Compatibility testing: Every user will have a different environment that your app will run in. Compatibility testing helps make sure the app is working as intended on the devices and in the different situations it was intended for.

- Performance testing: Performance issues disrupt the UX & lower the quality of your app. Performance testing helps eliminate issues such as excessive battery consumption or resource use.

4. Usability

How the user interacts with your app and if it is difficult to use:

- User-friendliness: How easily users can reach their goals. If user-friendly, the app is intuitive, easy to use, simple, & reliable for target users. This can be sometimes hard to quantify and can be very relative.

- The flow of critical path: Every app has at least one core function. The journey a user takes from launching an app to fulfilling its key purpose is known as the critical path. It's where the majority of users spend their time. How easy it is for users to get through this critical path is an important measure of quality.

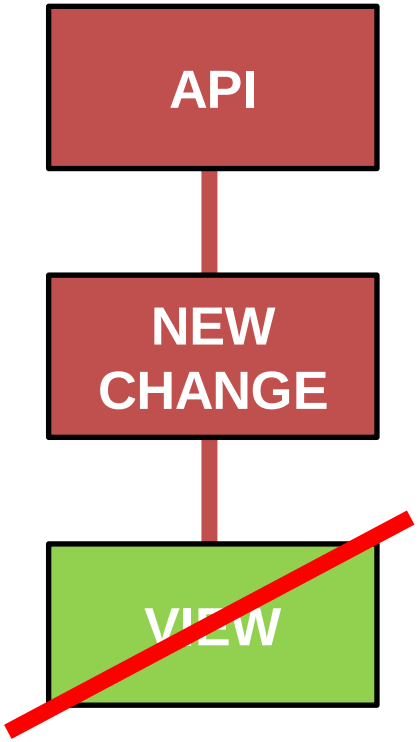
- Onboarding: Apps don't have a lot of time to impress new users. This is called bringing users onboard. The onboarding flow will be the first thing a new user interacts with, so it needs to be intuitive, easy to follow, & effective. This will take some amount of trial, error, testing, and most importantly getting first-time users' feedback to help guide your decision-making process.

Frameworks

- Frameworks or design patterns help us write app code that is structured, maintainable and can be easily tested.
- They help split large app into specific sections that all have their own purpose.
- Luckily, mobile development pretty much sets it up so you are forced to work in a framework.
- It is still best to understand what you are using and how they work.

Why do we use them

- **Understandable code:**
  - Others can read and more easily and this helps debugging and updating.
- **Code is more maintainable:**
  - Tightly coupled code is more dependant on other code.
- **Separation of concerns:**
  - Complex behaviours such as network calls, managing data etc. need separate attention to the UI requirements (a class should do one thing).
- **Supports the app lifecycle:**
  - Phone system needs means that if things are ordered incorrectly the app may crash. UI classes should work independently from network calls for example.



Options

- Essentially three types you need to be aware of:
  - MVC (Model View Controller)
  - MVVM (Model View View-Model)
  - MVP (Model View Presenter)
- Google (generally) recommend MVVM or MVP, but never actually say this – but everyone seems to use it.
- Apple recommend MVC (for UIKit) and now MVVM (for SwiftUI)

What is MVC?

- The Model View Controller is a software design pattern commonly used for developing user interfaces. For object-orientated designed apps.
- Apple adopted this pattern early, stating “it is central to good design in a mobile application” for UIKit.
- Considered the “classic framework” works great for small-scale apps
- There is a coined a term ”Massive View Controller” which is the opposite of what these frameworks are trying to achieve.

MVC in detail

Model

UI independent set of objects. Hold application data. They're usually structs or simple classes that help structure any received data but don't know what it is (the business bit). No view code!

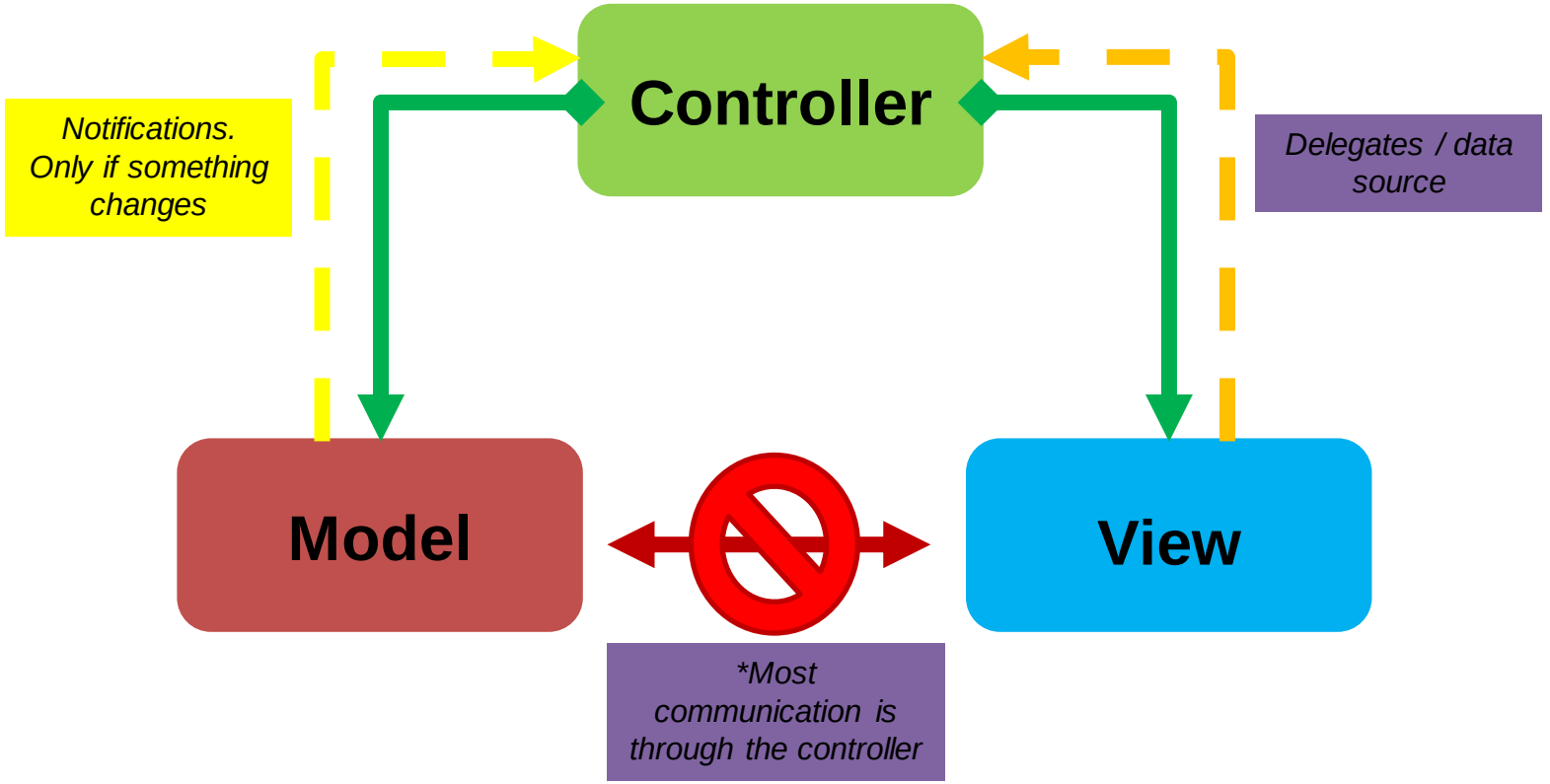
Controller

Performs requests. Never has to worry about data it sends and receives. Instead, it just tells the model what to do with it. Generic UI features such as buttons, labels etc. Knows nothing about the function of the app.

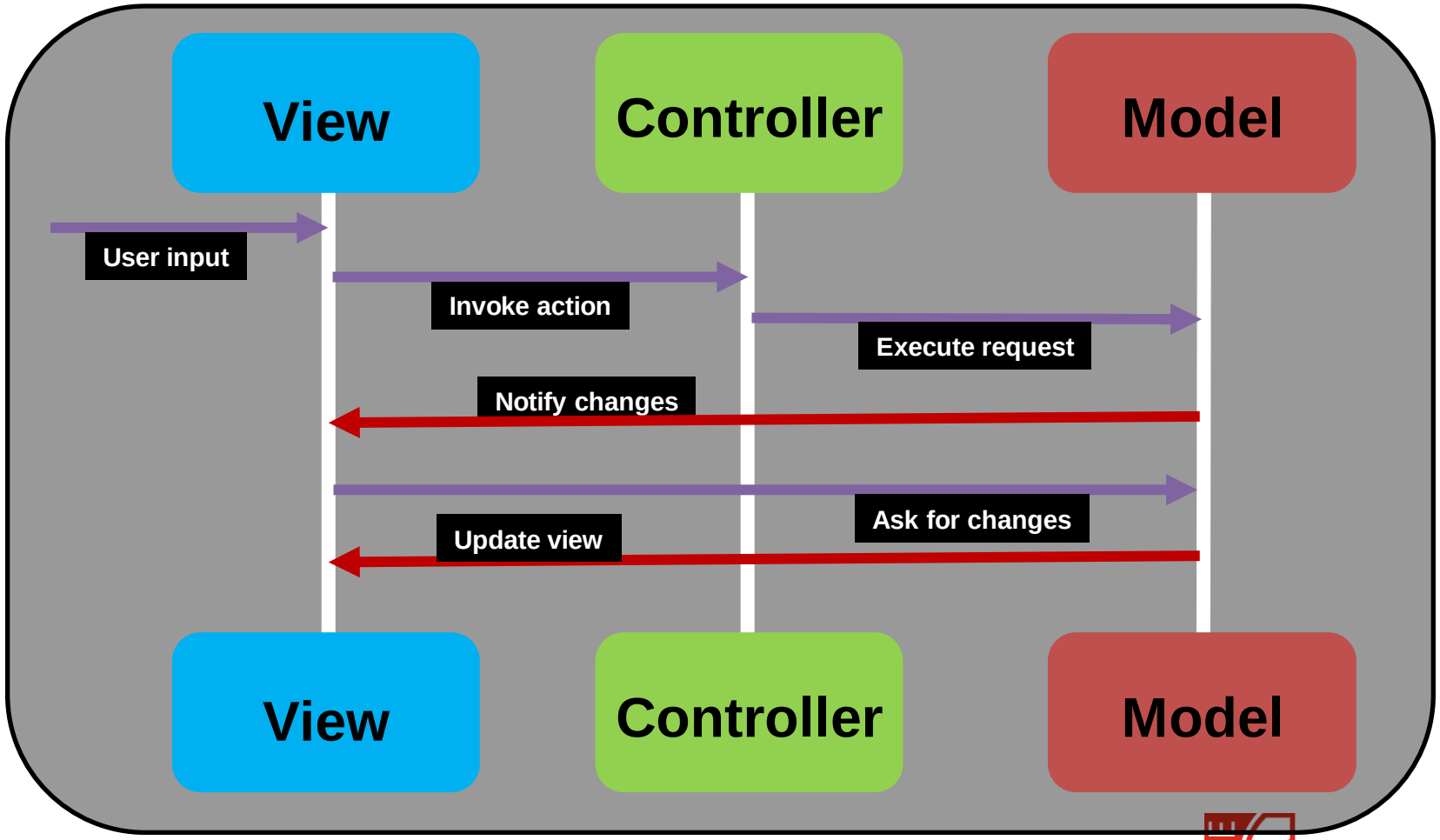
View

Handles data presentation. How your model presents data to the user (labels, views etc.) in the view. Dynamically renders the view for the user – could be considered more abstract such as a console window.

MVC Communications



MVC in practice



MVC Pros and Cons

Pros:

- Simplicity and Separation of Concerns
- No business logic in the UI
- Easier to unit test

Cons:

- Doesn't scale, separates UI but not the model
- Controller often grows to an unmanageable size
- Apps are growing in complexity so this is on the way out
- Violates Single responsibility, interface segregation

What is MVVM?

Model-View-View/Model

What we should all be using

This model has the benefits of easier testing and modularity and reduces the amount of "glue" code that is required to connect the **View** and **Model**.

The single different aspect is the **View-Model** that is responsible for wrapping the model and preparing observable data needed by the **View**.

MVVM in detail

Model

UI independent set of objects. hold application data. They're usually structs or simple classes that help structure any received data but don't know what it is (the business bit). No **View** code!

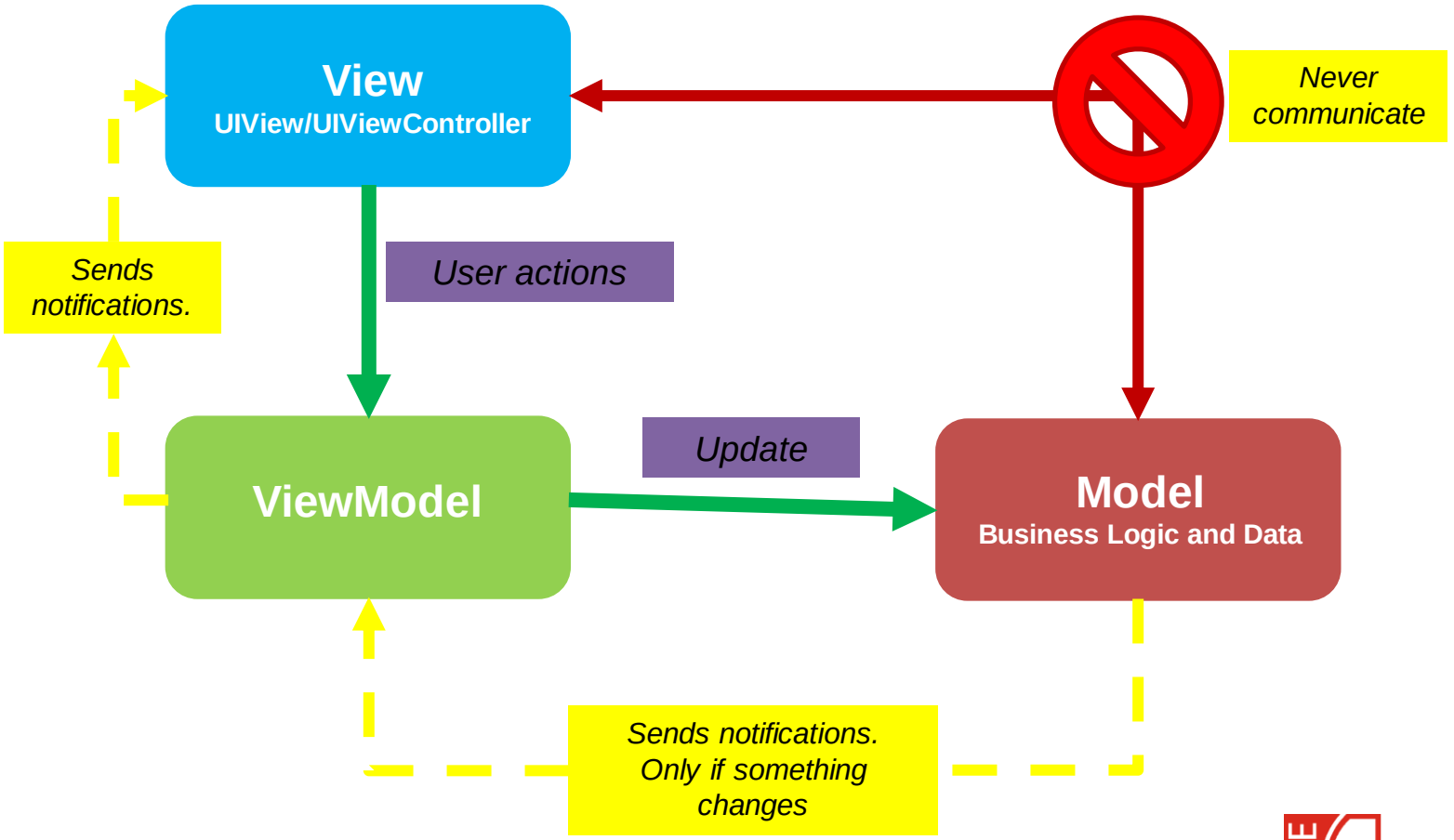
View-Model

Acts as an intermediate between **Model** and **View**. They transform model information into values that can be displayed. Usually classes, that can be passed around as references. They receive data, do some JSON and create a list, then pass to the lovely view you have created.

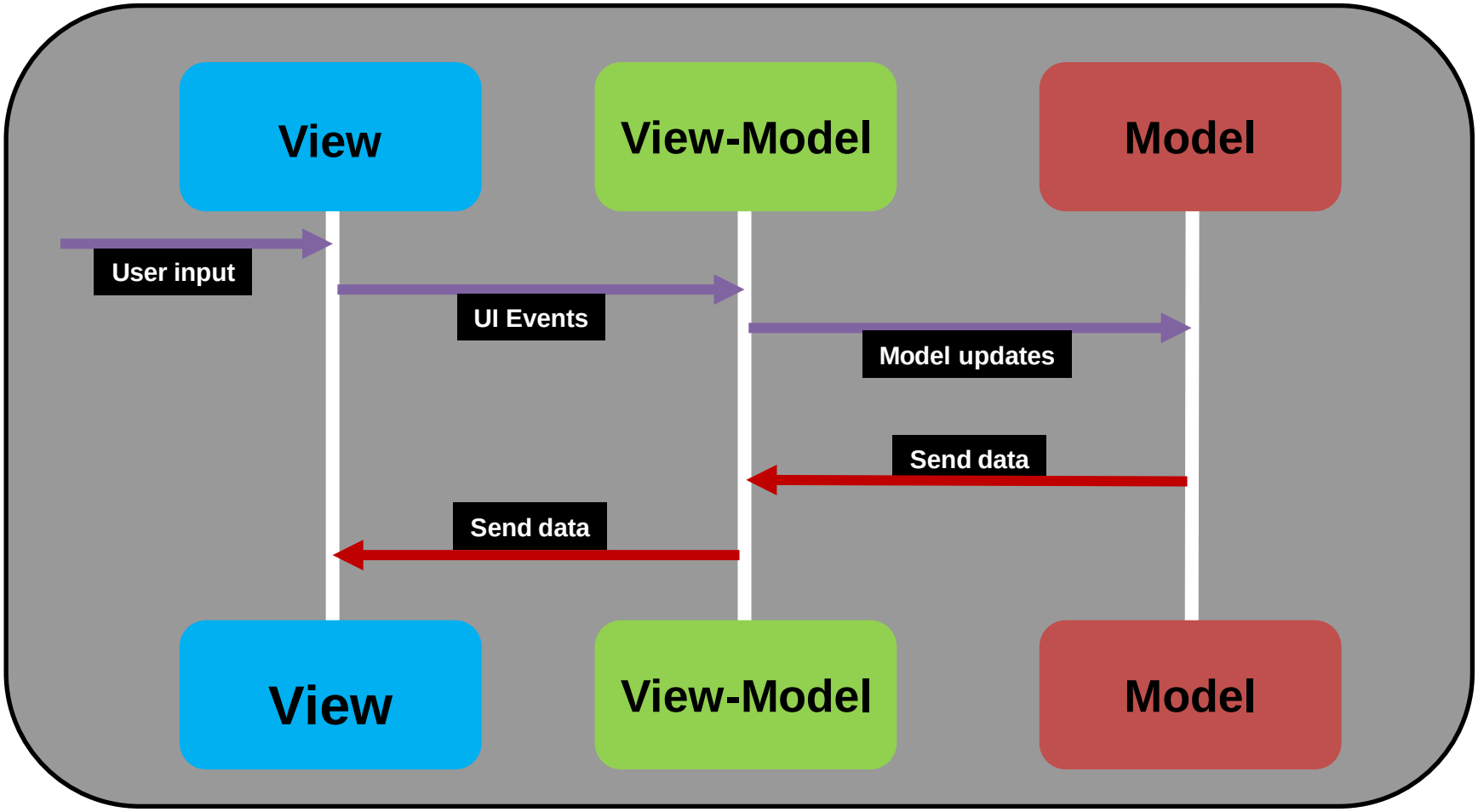
View

Handles data presentation. How your **Model** presents data to the user (labels, views etc.) in the **View**. Dynamically renders the view for the user – could be considered more abstract such as a console window.

MVVM Communications



MVVM in practice



MVVM Pros and Cons

Pros:

- Both OSs support this first (at least now)
- Compile time checking (Kotlin and Swift)
- Testable
- Less code
- Separation of concerns and Scalability

Cons:

- Data binding not always appropriate
- View-Model is still the middle-man

Pros and Cons compared

MVVM

Pros:

- Both OSs support this first (at least now)
- Compile time checking (Kotlin and Swift)
- Testable
- Less code
- Separation of concerns and Scalability

Cons:

- Data binding not always appropriate
- View-Model is still the middle-man

MVC

Pros:

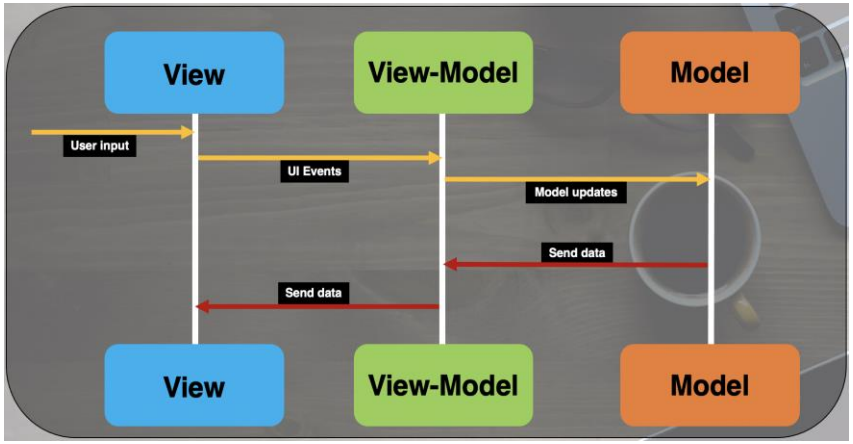
- Simplicity and Separation of Concerns
- No business logic in the UI
- Easier to unit test

Cons:

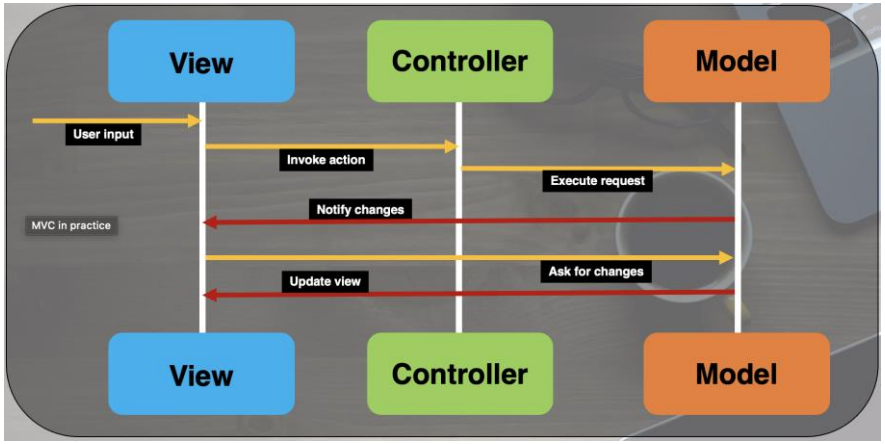
- Doesn't scale, separates UI but not the model
- Controller often grows to an unmanageable size
- Apps are growing in complexity, so this is on the way out
- Violates Single responsibility, interface segregation

MVVM vs MVC

MVVM



MVC



- Less back and forth
- Allows for larger, more complex apps without becoming bloated
- Considers actual app design and a mobile interface

What is an API?

APIs Make Life Easier for Developers

An API is a set of **definitions** and **protocols** for building and integrating application software.

APIs let your app, product or service **communicate** with other products and services without having to know how they're implemented.

This can simplify app development, saving **time** and **money**.

Mobile development consists mostly of working with API's.

What do we work with?

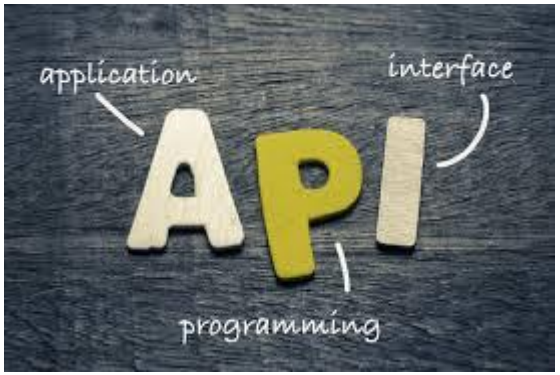
As mobile developers, there are two we need to be aware of:

Application Programming Interface

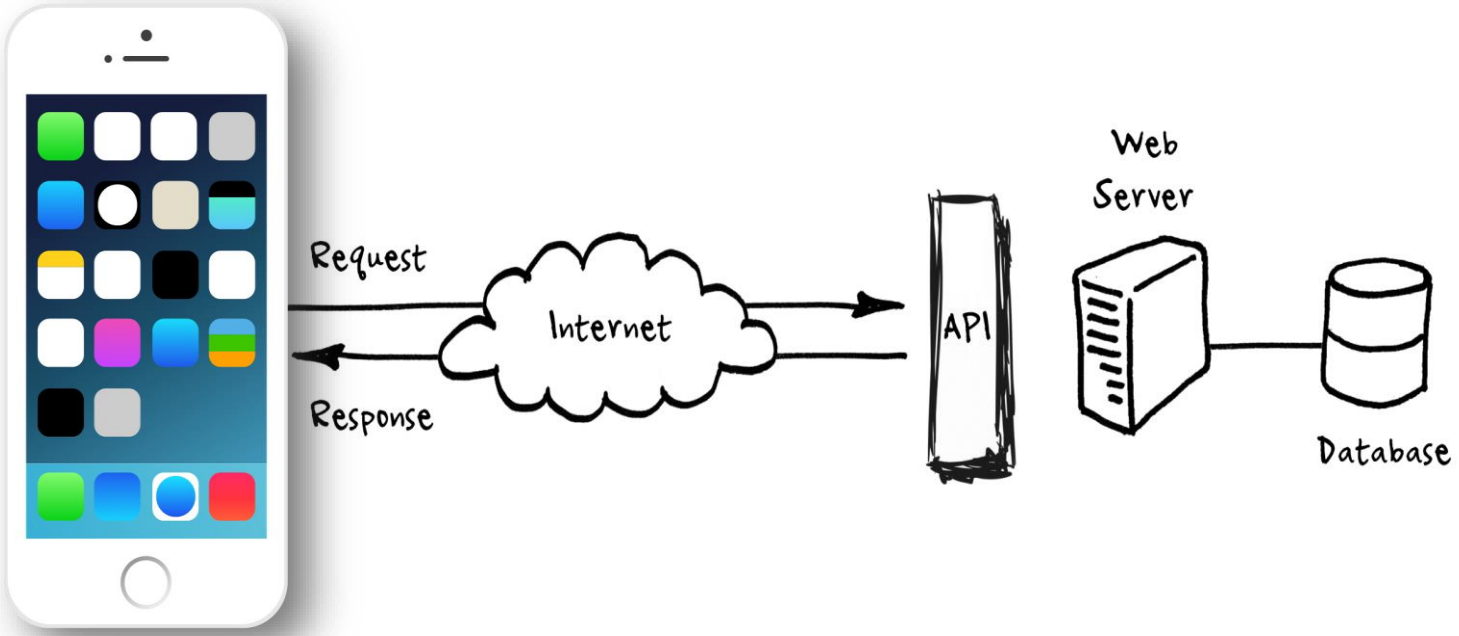
- A specification for how various applications can interact with a set of **software** components. Applications can include APIs for external use by other software.

Device API

- A mobile platform-specific API that lets applications access specific mobile **hardware** functionality.



1. Services API

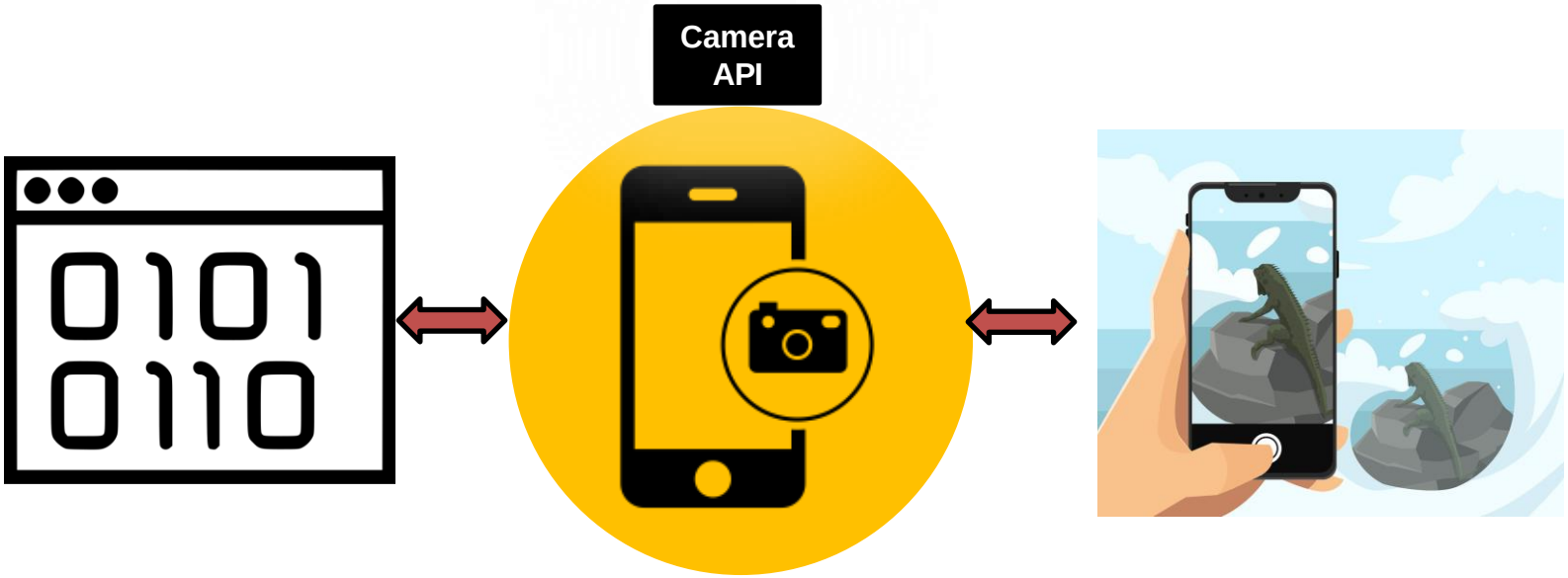


You need information from somewhere and there is a service that can provide it securely to advertisers, apps websites and more.

They use an API like a doorway. You make calls/requests to access and retrieve information and for that you need a key to communicate back and forth.

Example

2. Device API



If you want to take photos from the phone's camera, you don't have to write your own camera interface. You use the camera API to embed the phone's built-in camera in your app.

For each OS developers have done the hard work so the developers can just use the camera API to embed a camera, and then get on with building their app. Also, when the OS improves the camera API, all the apps that rely on it will take advantage of that improvement automatically.

Often need to use both

A good example of the use of both device and external resource API's is a "weather app".

a. App needs to know your **location** (this is device/hardware dependant)

b. App needs to request **weather data** from an external site using the device location.

Hardcoding the data

If you are unable to find a data set or API that you are able to use *or* have a bespoke idea that would need data created for it, you have two options:

1. Hardcode it (we can assess)
2. Create your own database / API (we will not assess)

For the purpose of this assignment, you can dummy the data and hardcode it in, but remember we are looking for **scalability** in your work.

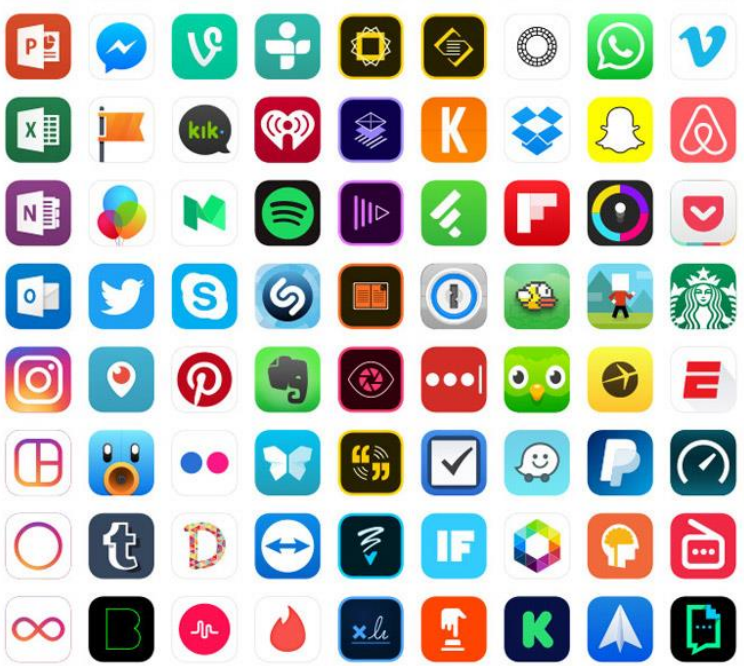
App Icons

Just a note to remind you all: Please include them!

Easy marks that will produce a more polished app (and easy to do).

Lots of tools available – process is pretty simple. You can use any images you wish (you do not have to design your own).

You should use .png format.



Final reminder…

A mobile app should do one thing and do it well.

A mobile app should be as simple as possible, but no simpler.

Don’t be afraid to iterate and change as you learn new things.