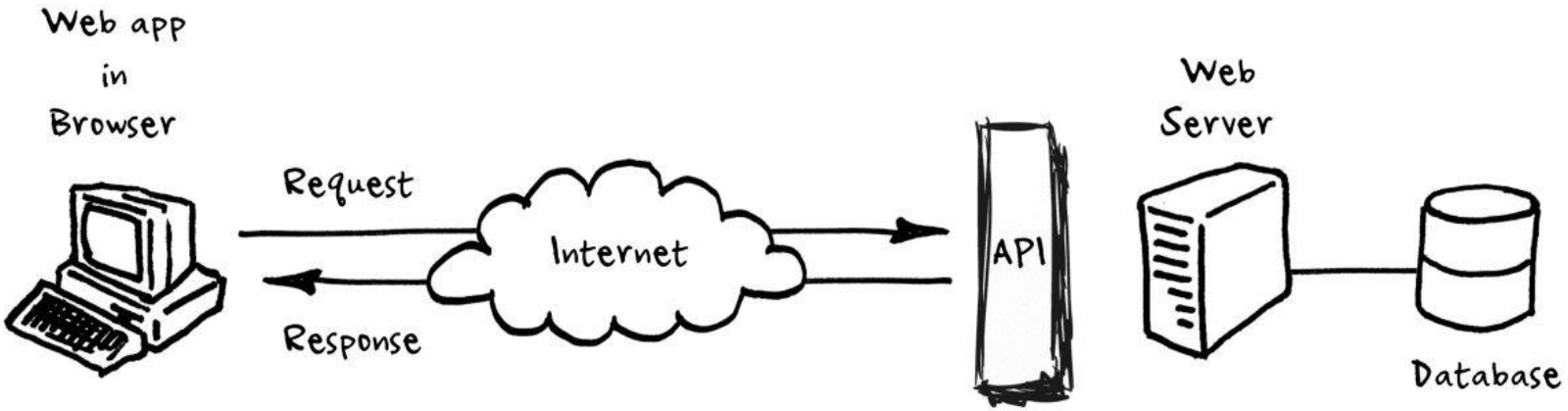
Working with Web Services

Retrieving Data from an API

A web API is usually a website that returns data that your application can use.

Examples:

- Pokemon API: https://pokeapi.co/
- Movies API: https://developers.themoviedb.org/3/movies
- More examples: https://github.com/public-apis/public-apis



API Responses

Generally speaking, APIs provide data in JSON format.

An API can respond to a request with:

- A single JSON object
- An array of JSON object

Example of a single JSON object:

- https://api.spacexdata.com/v4/launches/5eb87d42ffd86e000604b384

Example of an array of JSON objects:

- https://api.spacexdata.com/v4/launches

Retrieving Data from an API

1. Define a model class

- This class represents a **single item** from the API
- The class must conform to Codable protocol
    - The Codable protocol is a built in IOS class

In the ViewController.swift file,

2. Create an array of model objects

3. Make a network call to the API

4. Save results to the array of model objects

Model Class

Decide what keyw from the API you want to extract

Create 1 stored property per key

Using CodingKeys to create a mapping between the stored property and the API key name

Conform the struct/class to Codable

Implement the

- encode throws function
- init (from decoder:Decoder) throws

In the init(...):

- Use decodeIfPresent(...) to set default values for keys that don’t exist

Mandatory Codable Functions

The model class must implement 2 mandatory functions from the Codable Protocol

```
func encode(to encoder:Encoder) throws
{
        // do nothing
    }
```

```
init(from decoder:Decoder) throws {
        // 1. try to take the api response and convert it to
data we can use

        // 2. extract the relevant keys from that api
response

    }
```

Closure Functions (Completion Handlers)

.dataTask(...) Function

URLSession.shared.dataTask(with:api) is the function that connects to a URL endpoint and attempts to retrieve the results

This function occurs asynchronous (in the background)

```
URLSession.shared.dataTask(with: api) { (data, response, error) in
        // when data returned, do something
        print(data)
}.resume()
```

The function must run asynchronously because you don't know how long the API website will take to respond (you also don't know if it will even respond! 404 error!)

While the function is running, you do **NOT** want to "block" the user from doing other things.

That's why the the function runs on a background thread.

Closure (CallBack function)

The code within the { } symbols is called a “closure” or “completion handler function”

```
URLSession.shared.dataTask(with: api) { (data, response, error) in
        // when data returned, do something
        print(data)
}.resume()
```

A completion handler function is a function that is executed **after** a specific task finishes.

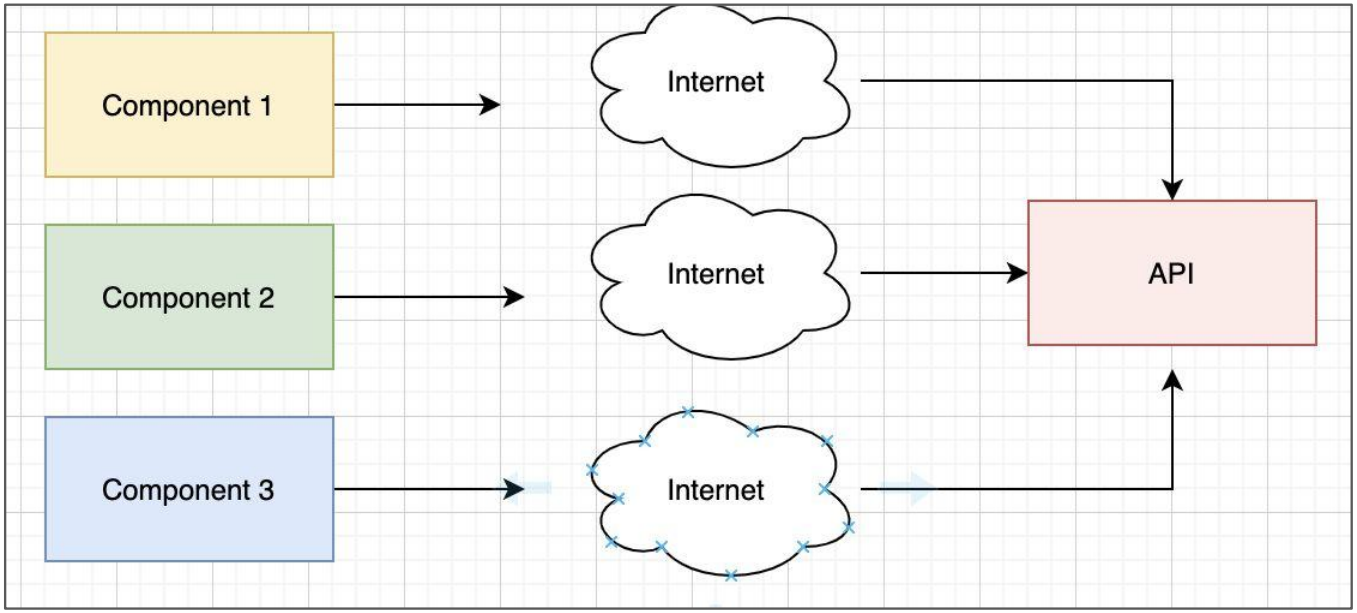
In this case, the *completion handler* runs **after** IOS receives a response from the API endpoint.

In other words: At some unknown time in the future, the **URLSession.shared.dataTask(...)** function will finish executing. When that occurs, run the **completion** handler**.**

Service Layer Architecture

OPTION 1: Every component makes an API Call

In this approach, each screen makes a separate call to the API.



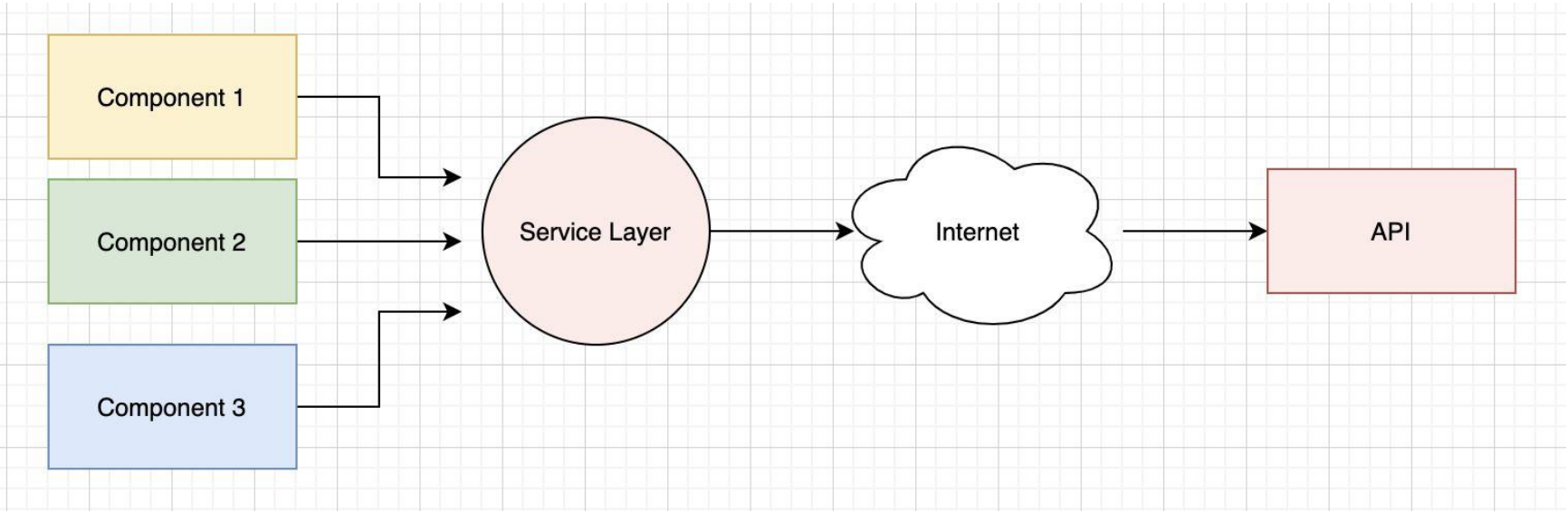
This approach is **not ideal** because each component has the same repeated “data fetching” code.

- If the API endpoint changes, breaks, or needs to be updated, you need to update your code in every screen

OPTION 2: The Better Way

In a service layer architecture, we create a *service layer* class. This class is responsible for connecting to the API and fetching data.

Then, any screen that requires data from the API make its request to the *service layer* (instead of the API)



Benefits of Service Layer Architecture

Separation of Concerns:

- Each “functional area” of an application should be responsible for its own tasks

Code maintainability:

- Clearer/more obvious what the code is doing
- Easier to debug: Issue relating to networking is in a single location, not spread across multiple view controllers

The Service Layer is a Singleton

The service layer should be implemented as a **Singleton**

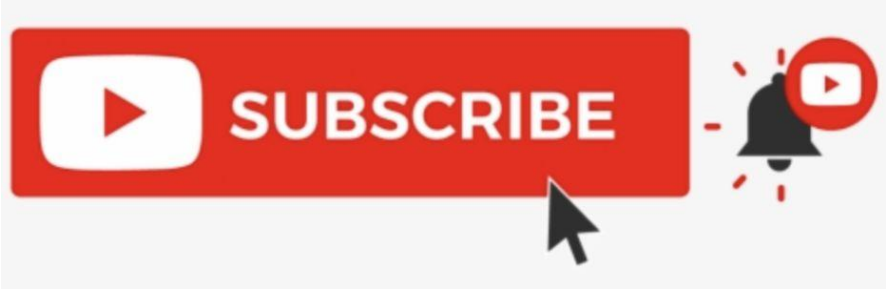
- The Singleton is a design pattern that only allows for 1 instance of the class to exist throughout the application

```
// singleton
private static var shared:APIFetcher?
static func getInstance() -> APIFetcher {
    if (shared != nil) {
        // we already created an instance of this class somewhere
        // so return that instance instead of making a new one
        return shared!
    }
    else {
        // this is the first time we're creating an instance of the APIFetcher
        shared = APIFetcher()
        return shared!
    }
}
```

Publish and Subscribe Models (Observer / Observables)

Publish Subscribe Model

How do we know when data has changed?

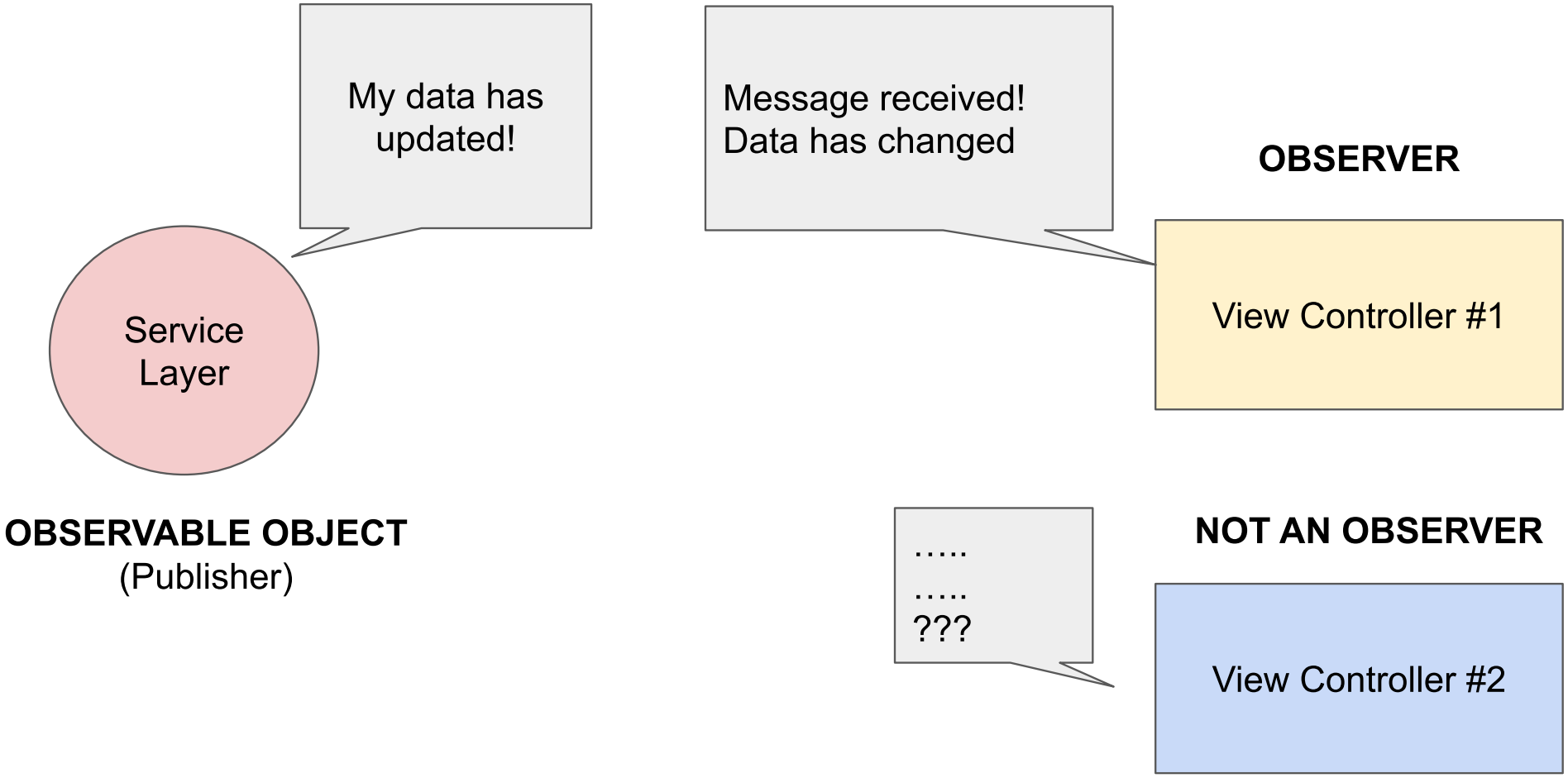


In the publish/subscribe (observable/observer) model:

- A class can “publish” updates about itself
- Other classes can “subscribe” to updates
- When the publisher class changes, the other subscriber classes are notified

Observe / Observable

This model is also known as the **Observable** and **Observer** model



Observable Object

An ObservableObject is an class that will **publish notifications** when values within the class change.

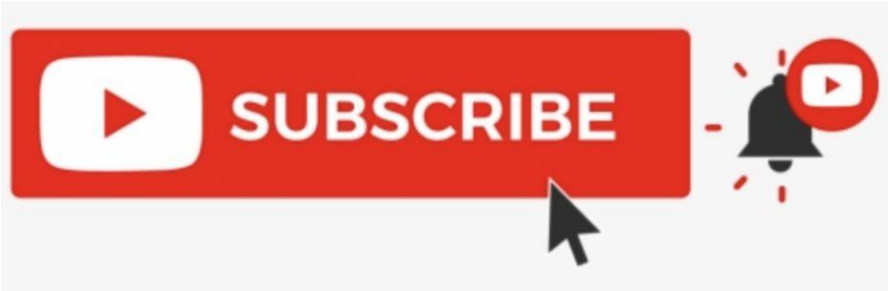
Other classes can then **subscribe** to the changes.

Within the ObservableObject:

- Use the @Published annotation to mark a property as “listenable”

Within another class:

- Create a “cancellables” variables
- Use the .receive(...).sink {...}.store(&cancellables) function to subscribe to changes



Code Snippet: Publishing Changes

```
class APIFetcher:ObservableObject {
    ...
    //stored property (data you want to notify others about)
    @Published var launchList:[SpaceshipLaunch] = [SpaceshipLaunch]()

     func fetchData() {
        ...
        URLSession.shared.dataTask(with: apiURL) { (data, response, error) in
            ...
            let decoder = JSONDecoder()
            let decodedItem:[SpaceshipLaunch]
                    = try decoder.decode([SpaceshipLaunch].self, from: jsonData)

            self.launchList = decodedItem
        }.resume()
    }
}
```

2. Mark the launchList with @Published so it can be "observed" by other classes

1. After the API data is fetched and decoded, update the launchList variable

Code Snippet: Subscribing to Changes

Import Combine framework

```
import Combine

class ViewController: UIViewController {

      private var cancellables:Set<AnyCancellable> = []

      private func watchForChanges() {
        print("Watching for changes")
        // subscribe / watch for changes in the launchList from apiFetcher class

        self.apiFetcher.$dataToWatch.receive(on: RunLoop.main).sink { (changedData) in
            print("We saw a change in the api fetcher's launch list")
            // what do we want to do when we see a change in the api fetcher's launch list?
        }.store(in: &cancellables)
    }

}
```

Create cancellables variable

Use cancellables with receive.sink(..) to subscribe to changes