# Dependency 💉

To inject or not?

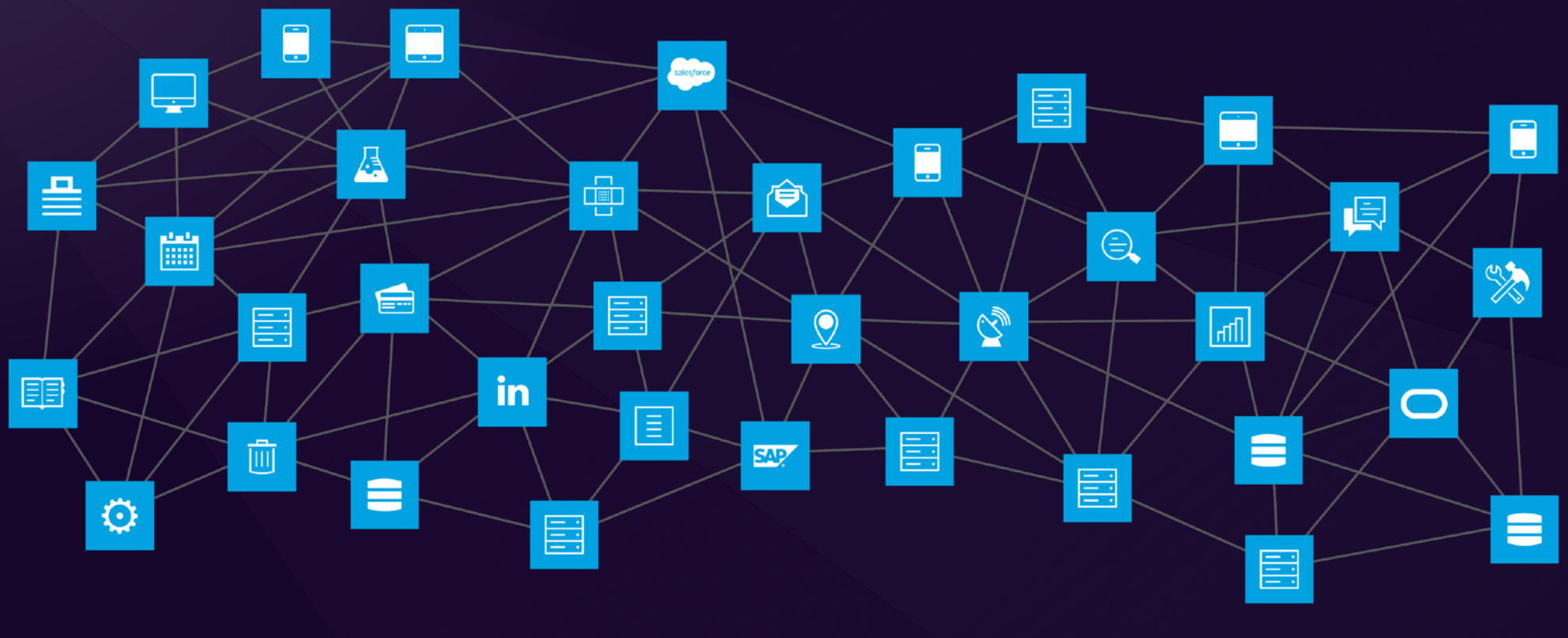
# Agenda

1) The problems in your code base
2) The solution (sort of) : Dependency Injection
3) What, why and how of Dependency Injection
4) Introduction to Swinject and Weaver
5) Wrap up

# What is the problem in your code base? 🤔

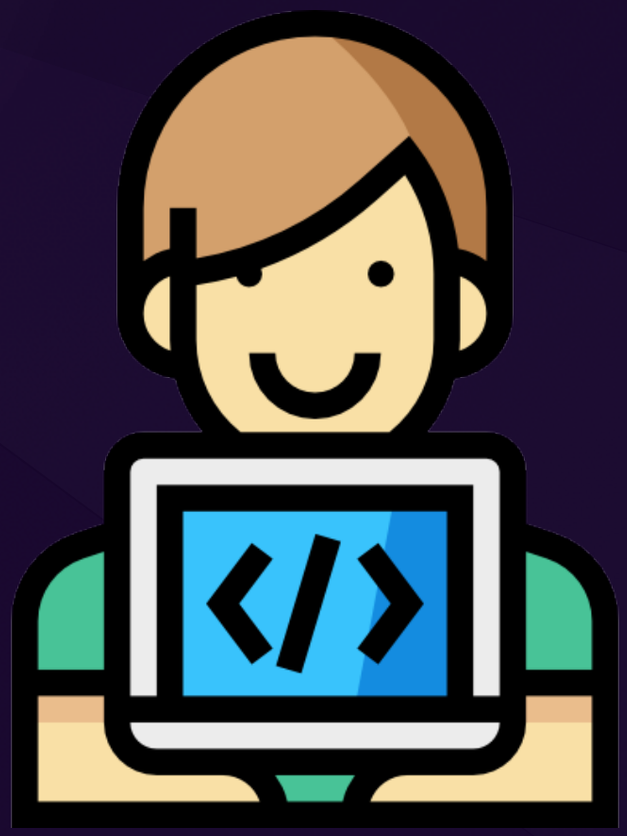
# “Spaghetti Code”

Spaghetti code is a phrase for unstructured and difficult to maintain source code. It can be caused by volatile project requirements, and insufficient abilitv or experience.




# “Unmaintainable Codebase”

# “Non testable code”



“Difficulty for fellow peers”

# Nobody wants that right? 

# Enter Dependency Injection! 🤩

# DI what? 😐

Dependency injection is a technique whereby one object supplies the dependencies of another object. An injection is the passing of a dependency to a dependent object. The service is made part of the client's state.

# Fact

Dependency Injection is a 25 dollar term for a 5 cent concept

**You've already done Dependency Injection, unknowingly 🤫**

Passing data through Segues, is Dependency Injection

# What DI is NOT ❌

- a library
- a framework
- a tool

# What DI is ✅

- a way of thinking
- a way of designing code
- general guidelines

```
class TextEditor {
  var checker: Spellchecker

  init() {
    self.checker = Spellchecker()
  }
}
```

```
class TextEditor {
  var checker: IoCSpellChecker

  init(checker: IoCSpellChecker) {
    self.checker = checker
  }
}
```

# Current Implementations

- Initialiser Based
- Property Based
- Parameter Based

# Initialiser Based

The idea is that an object should be given the dependencies it needs when being initialised. The benefit is that it guarantees that our objects have everything they need in order to work the right way.

This is easy af. 🤫

```swift
class FileLoader {
  private let fileManager : FileManager
  private let cache : Cache

  init(fileManager: FileManager = .default, cache: Cache = .init()) {
    self.fileManager = fileManager
    self.cache = cache
  }
}
```

## Property Based

Instead of injecting an object's dependencies in the initialiser, properties can simply be assigned afterwards. This can also reduce boilerplate code, especially when there is stuff that doesn't need to be injected.

```swift
extension UIViewController {
  weak public var transitioningDelegate: UIViewControllerTransitioningDelegate?
}
```

## Parameter Based

Sometimes we need a specific dependency once, or we need to mock it under certain conditions. Instead of having to change an object's init or expose properties, we open up an API to accept dependency as a parameter.

```swift
class NoteManager {
  func loadNotes(query: String, completionHandler: @escaping ([Note]) -> Void) {
    Dispatch.global(qos: .userInitiated).async {
      let database = self.loadDB()
      let notes = database.filter { note in
        return note.matches(query: query)
      }
    }
    completionHandler(notes)
  }
}
```

```
class NoteManager {
  func loadNotes(query: String, on queue: DispatchQueue = .global(qos: userInitiated),
    completionHandler: @escaping ([Note]) -> Void) {
    queue.async {
      let database = self.loadDB()
      let notes = database.filter { note in
        return note.matches(query: query)
      }
    }
    completionHandler(notes)
  }
}
```

# Dependency Container 

A Dependency Injection Container (or DI Container) is basically an object able to instantiate, retain, and resolve other objects' dependencies for them.

# Swinject

Swinject is a lightweight dependency injection framework for Swift. Swinject helps your app split loosely-coupled components. Swinject is powered by the Swift generic type system and first class functions to define dependencies of your app simply.

```swift
protocol Animal {
    var name: String? { get }
}

class Cat: Animal {
    let name: String?

    init(name: String?) {
        self.name = name
    }
}
```

```swift
protocol Person {
    func play()
}

class PetOwner: Person {
    let pet: Animal

    init(pet: Animal) {
        self.pet = pet
    }

    func play() {
        let name = pet.name ?? "someone"
        print("I'm playing with \(name).")
    }
}
```

```swift
let container: Container = {
        let container = Container()
        container.register(Animal.self) { _ in Cat(name: "Mimi") }
        container.register(Person.self) { r in
            PetOwner(pet: r.resolve(Animal.self)!)
        }
        container.register(PersonViewController.self) { r in
            let controller = PersonViewController()
            controller.person = r.resolve(Person.self)
            return controller
        }
        return container
    }()

```

```swift
func application( _ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [
        UIApplicationLaunchOptionsKey : Any]? = nil) -> Bool {

        // Instantiate a window.
        let window = UIWindow(frame: UIScreen.main.bounds)
        window.makeKeyAndVisible()
        self.window = window

        // Instantiate the root view controller with dependencies injected by the container.
        window.rootViewController = container.resolve(PersonViewController.self)

        return true
    }

```

# Why containers? 🤓

1) Inject “N” dependencies with one parameter
2) Different init logic for different layers
3) With the right interfaces, Unit tests are easy
4) Implements Inversion of control more effectively

# Why not Containers? 😨

1) Can crash at runtime

2) Unit tests need their own containers

3) Not easy, very conceptual

# Weaver to the Rescue!! 🚀

# What is Weaver?

Weaver is a lightweight Dependency Injection framework that is able to generate the necessary boilerplate code to inject dependencies into Swift types, based on annotations.

# Quick Wins 🏆

- Works compile time
- Container Auto Generation
- Type and Thread safe
- ObjC Support

# How does it work? ⚒️

- Scans your code for annotations
- Generates an AST*
- Generates a Dependency Graph
- Performs safety checks
- Generates Boilerplate code using the graph. Generate one dependency container/ struct or class with injectable dependencies.

* Abstract Syntax Tree

# Implementing a MoviesListController

3 notice worthy objects

1) AppDelegate, registering the dependencies
2) MovieManager, providing the movies
3) MoviesListVC , showing the movies

# Registering Dependencies

```swift
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    private let dependencies = AppDelegateDependencyContainer()

    // weaver: testController = TestController <- UIViewController

```

# Defining Scope

```
// weaver: testController.scope = .container

```

# Building Root View Controller

```
let rootVC = dependencies.homeViewController

```

```swift
final class HomeViewController: UIViewController {

    private let dependencies: HomeViewControllerDependencyResolver

    // weaver: testController <- TestViewController

    required init(injecting dependencies: HomeViewControllerDependencyResolver) {
        self.dependencies = dependencies
        super.init(nibName: nil, bundle: nil)
    }

    override func viewDidLoad() {
        super.viewDidLoad()

        //use testController
        dependencies.testController. //

    }
}

```

# Further Reading 👨🏻‍💻

1) Medium Article
https://medium.com/@JoyceMatos/dependency-injection-in-swift-87c748a167be

2) Podcast by Cocoacasts
https://cocoacasts.com/nuts-and-bolts-of-dependency-injection-in-swift

3) Swinject
https://github.com/Swinject/Swinject

4) Weaver
https://github.com/scribd/Weaver

# Thanks! 👻

🤝


## Amanjeet Singh

@droid_singh

Bobble Keyboard

## Bhagat Singh

@soulful_swift

Zomato

# Fin.