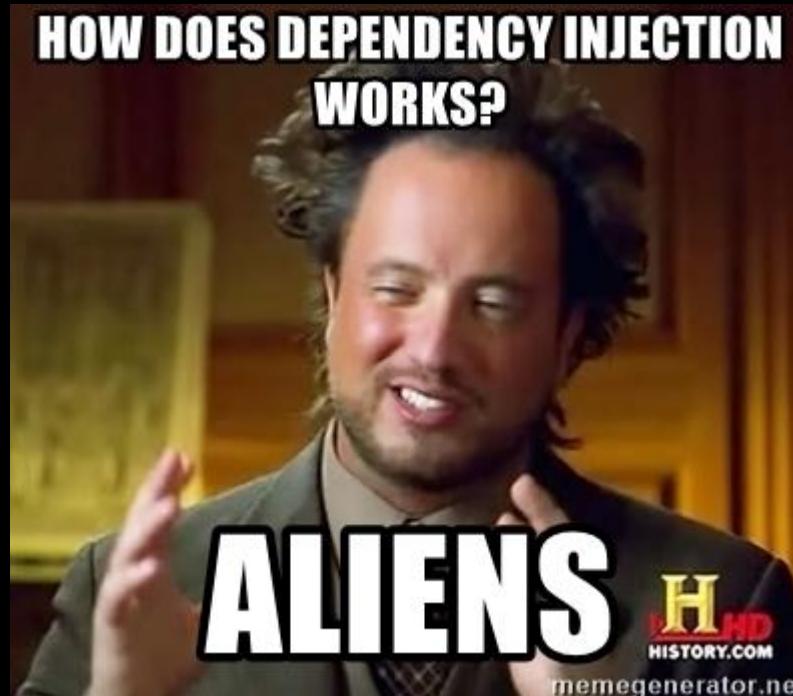


Injeção de Dependência com Swift



1. Apresentação

Sobre Mim:

Desenvolvo para iOS desde 2010

- Até 2016

- iOS principalmente como Hobby
- Já trabalhei com Backend (Java, C#), Windows Phone, Android, Infra (Unix), Front Web...

- Depois de 2016

- Entrei para trabalhar de backend, mas me pediram pra quebrar um galho de iOS... 😅
- Sou do interior de São Paulo, mas vivi alguns anos em Minas (BH e Uberlândia)
- Me mudei pra Amsterdam em 2021 para atuar como iOS Lead na adidas

Objetivo:

- Apresentar conceitos de Injeção de Dependência por meio de exemplos comuns

- Mostrar como combinar estratégias de Injeção por meio de exemplos comuns

 linkedin.com/in/bocato

 github.com/bocato

 bocato.medium.com

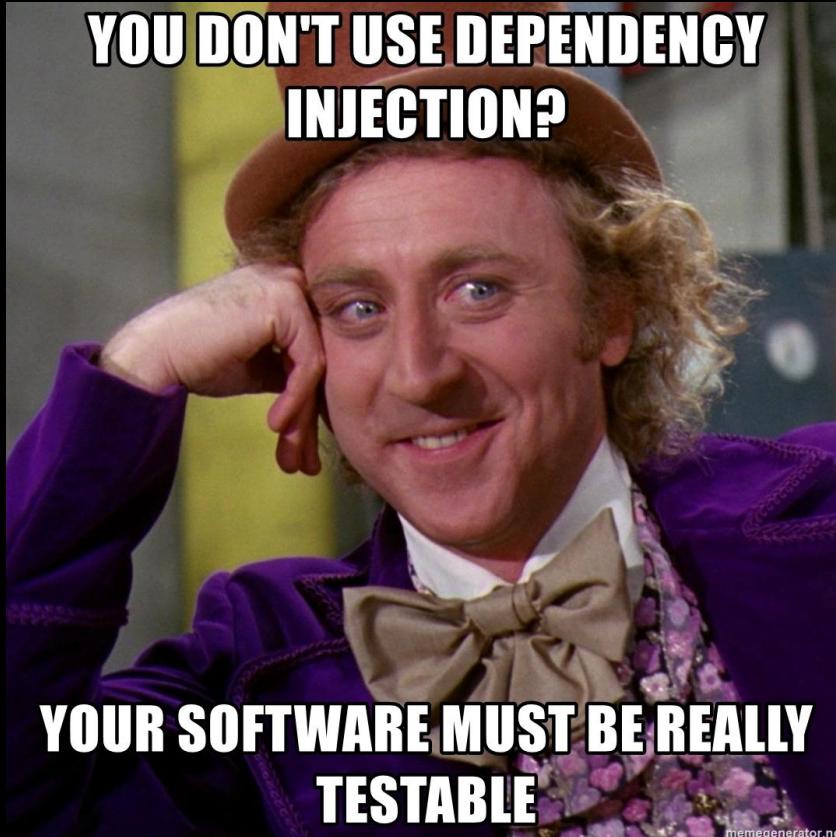
 [@dubocato](https://twitter.com/@dubocato)

ROTEIRO

- 1. Apresentação**
- 2. Injeção de Dependência Básica (DI 101)**
 - 2.1. O que é Injeção de Dependência?
 - 2.2. Tipos Básicos de Injeção:
 - 2.2.1. Pelo Inicializador (initializer based)
 - 2.2.2. Por Propriedades (property based)
 - 2.2.3. Por Parâmetros (parameter based / method Injection)
- 3. Injeção de Dependência Avançada**
 - 3.1. Singletons
 - 3.2. Factories
 - 3.3. Service Locator
- 4. Conclusão**
- 5. Perguntas**

Material da Talk: <https://bit.ly/3tA6sBs>

2. Injeção de Dependência Básica (DI 101)



2.1. O que é injeção de dependência?

Inversão de Controle (IoC)

O padrão de Inversão de Controle é sobre prover qualquer tipo de objeto (que implementa e/ou controla ações), ao invés de fazê-lo diretamente. Ou seja, é o ato de inverter e/ou redirecionar o controle para um agente/controlador externo.

Injeção de Dependência (DI)

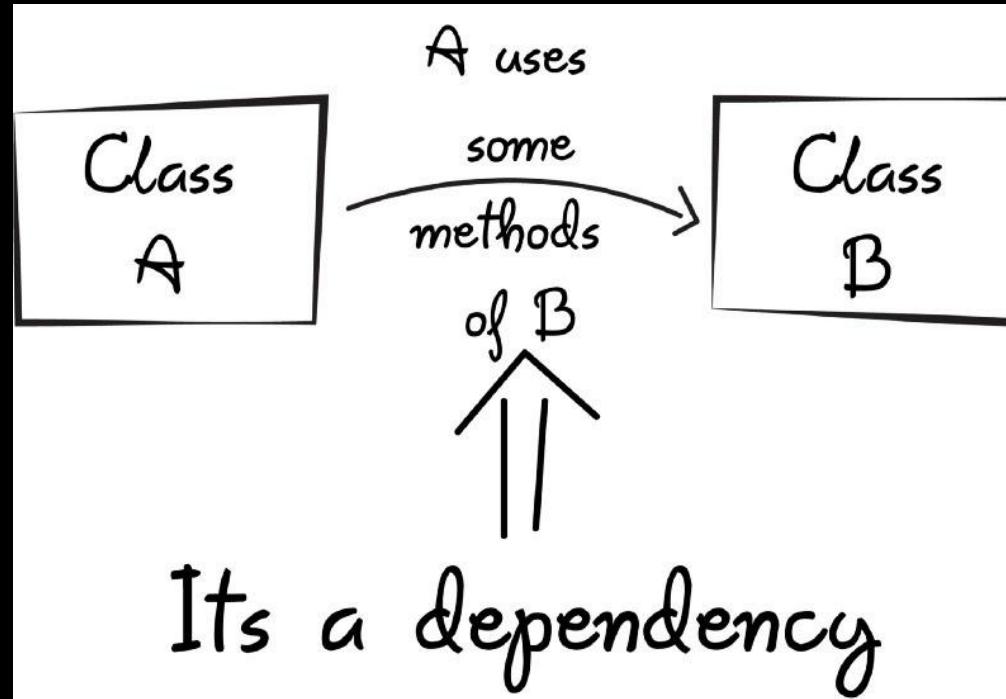
O padrão de Injeção de Dependência (DI) é um tipo mais específico do padrão de Inversão de Controle, onde implementações são passadas ao objeto de alguma forma (construtores, parâmetros, localizadores de serviços...), e este objeto passará a depender destas implementações para executar seu comportamento corretamente.

Exemplo:

Ao invés da aplicação chamar implementações de uma biblioteca diretamente (controladas por ela), ela chama um contrato (interface) que remete a à implementação concreta em um lugar controlado pela aplicação.

Caso queira se aprofundar nos conceitos: <https://martinfowler.com/articles/injection.html>

2.1. O que é injeção de dependência?



2.1. O que é injeção de dependência?

Prós

- Menor Acoplamento
- Testabilidade
- Reúso
- Legibilidade
- Separação de Responsabilidades (SoC)
- ...

Contras

- Complexidade
- Setups
- Mais código
- Verboso
- Alto acoplamento ao framework de injeção
- Isso depende muito de qual framework você está usando e principalmente de **COMO** está usando ele
- ...

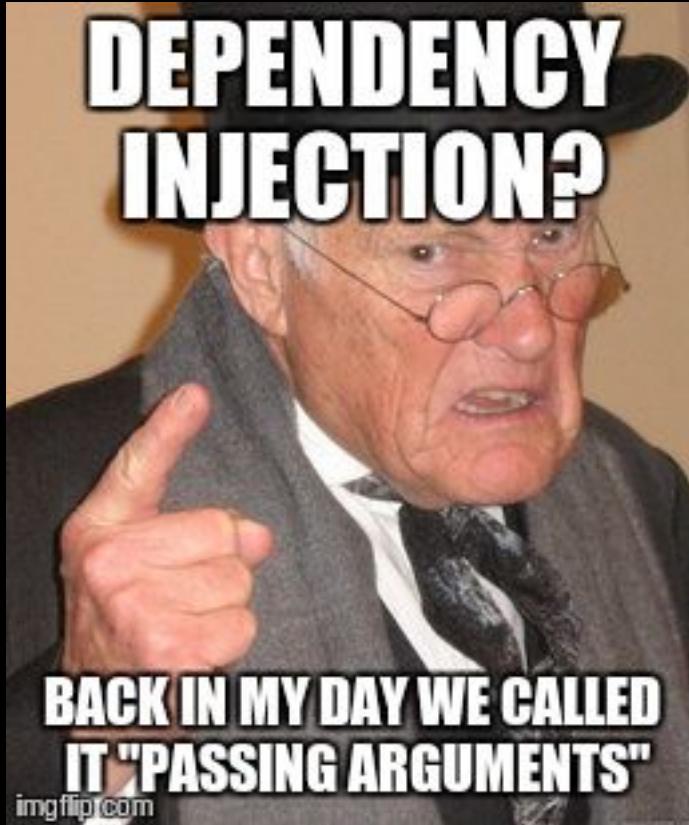
2.2. Tipos Básicos de Injeção



1. Pelo Inicializador (initializer based)
2. Por Propriedades (property based)
3. Por Parâmetros (parameter based / method injection)

2.2. TIPOS BÁSICOS DE INJEÇÃO

1. Por Inicializador (initializer based)



2.2. TIPOS BÁSICOS DE INJEÇÃO

1. Pelo Inicializador

As dependências são passadas para o objeto no seu método inicializador.

```
1 protocol NetworkProtocol {
2     func getData(from url: URL, completion: (Result<Data?, Error>) -> Void)
3 }
4
5 final class Network: NetworkProtocol {
6     static let shared = Network()
7     func getData(from url: URL, completion: (Result<Data?, Error>) -> Void) { /* ... */ }
8 }
9
10 struct User: Codable {
11     let name: String /* ... */
12 }
13
14 final class UserService {
15     private let network: NetworkProtocol
16     private let jsonDecoder: JSONDecoder
17
18     init(network: NetworkProtocol = Network.shared, jsonDecoder: JSONDecoder = .init()) {
19         self.network = network
20         self.jsonDecoder = jsonDecoder
21     }
22
23     func getUserData(_ completion: @escaping (Result<User, Error>) -> Void) {
24         let userServiceURL = URL(string: "www.someapi.com/userdata")!
25         network.getData(from: userServiceURL) { [jsonDecoder] result in
26             guard
27                 let data = try? result.get(),
28                 let user = try? jsonDecoder.decode(User.self, from: data)
29             else {
30                 completion(.failure(NSError(domain: "UserService", code: -999, userInfo: nil)))
31                 return
32             }
33             completion(.success(user))
34         }
35     }
36 }
```

2.2. TIPOS BÁSICOS DE INJEÇÃO

1. Pelo Inicializador

E como fica
no teste?

```
1 final class NetworkStub: NetworkProtocol {
2     var getDataResultToBeReturned: Result<Data?, Error> = .success(Data())
3     func getData(from url: URL, completion: (Result<Data?, Error>) -> Void) {
4         completion(getDataResultToBeReturned)
5     }
6 }
7
8 extension User {
9     func asData(using encoder: JSONEncoder = .init()) -> Data? {
10         try? encoder.encode(self)
11     }
12 }
13
14 final class UserServiceTests: XCTestCase {
15     func test_getUser_whenDataIsValid_thenItWillReturnSomeUser() throws {
16         // Given
17         let userMock: User = .init(name: "Random User")
18         let userDataMock: Data = try XCTUnwrap(userMock.asData())
19
20         let networkStub: NetworkStub = .init()
21         networkStub.getDataResultToBeReturned = .success(userDataMock)
22
23         let sut = UserService(network: networkStub)
24
25         // When
26         var userReturned: User?
27         let getUserDataExpectation = expectation(description: "UserService.getUserData was called.")
28         sut.getUserData { result in
29             userReturned = try? result.get()
30             getUserDataExpectation.fulfill()
31         }
32
33         // Then
34         wait(for: [getUserDataExpectation], timeout: 1.0)
35         XCTAssertNotNil(userReturned)
36     }
37 }
```

2.2. TIPOS BÁSICOS DE INJEÇÃO

2. Por Propriedades

A dependência é injetada por meio de uma propriedade, que pode ser modificada externamente.

- Pode ser uma opção válida quando você não tem total controle sobre a inicialização do objeto.

Exemplo: Quando usamos um ViewController inicializado por uma Storyboard.

```
1 protocol NetworkProtocol {
2     func getData(from url: URL, completion: (Result<Data?, Error>) -> Void)
3 }
4
5 final class Network: NetworkProtocol {
6     static let shared = Network()
7     func getData(from url: URL, completion: (Result<Data?, Error>) -> Void) { /* ... */ }
8 }
9
10 final class ViewController: UIViewController {
11     var network: NetworkProtocol = Network.shared
12
13     init() { super.init(nibName: "", bundle: .main) } // just to make it compile
14
15     required init?(coder aDecoder: NSCoder) {
16         fatalError("init(coder:) has not been implemented")
17     }
18
19     override func viewDidLoad() {
20         super.viewDidLoad()
21         loadData()
22     }
23
24     func loadData() {
25         network.getData(from: URL(string: "www.someapi.com/userdata")!) { result in
26             /* ... */
27         }
28     }
29 }
```

2.2. TIPOS BÁSICOS DE INJEÇÃO

2. Por Propriedades

E como fica no teste?

```
1 final class NetworkSpy: NetworkProtocol {
2     private(set) var getDataCalled = false
3     func getData(from url: URL, completion: (Result<Data?, Error>) -> Void) {
4         getDataCalled = true
5     }
6 }
7
8 final class ViewControllerTests: XCTestCase {
9     func test_networkIsCalled() {
10         // Given
11         let sut: ViewController = .init() // in real live, this can be different
12
13         let networkSpy: NetworkSpy = .init()
14         sut.network = networkSpy
15
16         // When
17         sut.loadData()
18
19         // Then
20         XCTAssertTrue(networkSpy.getDataCalled)
21     }
22 }
```

2.2. DEPENDENCY INJECTION 101

3. Por Parâmetros (method injection)

A dependência é injetada por meio de um parâmetro da função.

- Pode ser uma opção válida para quando estamos testando um método legado e queremos evitar o risco de quebrar as implementações já existentes.

```
1 protocol NetworkProtocol {
2     func getData(from url: URL, completion: (Result<Data?, Error>) -> Void)
3 }
4
5 final class Network: NetworkProtocol {
6     static let shared = Network()
7     func getData(from url: URL, completion: (Result<Data?, Error>) -> Void) {
8         /* ... */
9     }
10 }
11
12 extension UIImageView {
13     func setImageFromURL(
14         _ url: URL,
15         network: NetworkProtocol = Network.shared,
16         mainQueue: DispatchQueue = .main
17     ) {
18         network.getData(from: url) { result in
19             guard
20                 let data = try? result.get(),
21                 let remoteImage = UIImage(data: data)
22             else { return }
23             mainQueue.async { self.image = remoteImage }
24         }
25     }
26 }
```

2.2. DEPENDENCY INJECTION 101

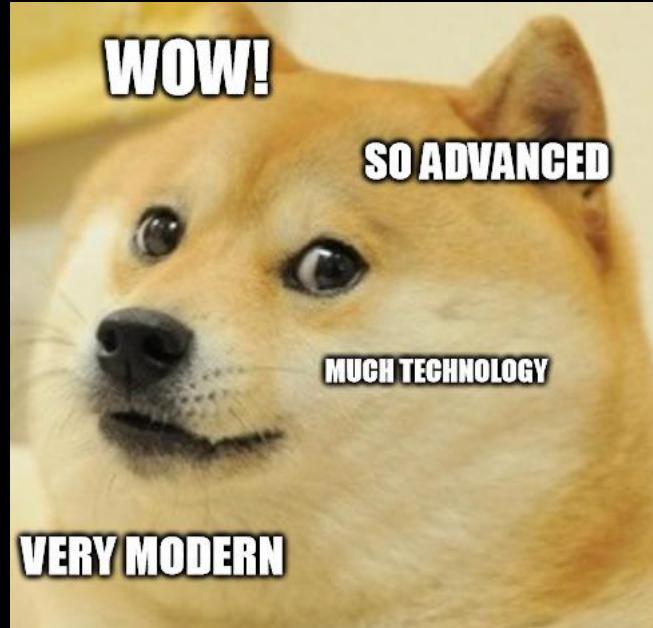
3. Por Parâmetros (method injection)

```
1 final class NetworkStub: NetworkProtocol {
2     var getDataResultToBeReturned: Result<Data?, Error> = .success(Data())
3     func getData(from url: URL, completion: (Result<Data?, Error>) -> Void) {
4         completion(getDataResultToBeReturned)
5     }
6 }
7
8 final class UIImageViewTests: XCTestCase {
9     func test_whenDataIsValid_thenItShouldReturnTheExpectedImage() throws {
10         // Given
11
12         let imageMock: UIImage = .add
13         let imageDataMock = try XCTUnwrap(imageMock.pngData())
14
15         let networkStub: NetworkStub = .init()
16         networkStub.getDataResultToBeReturned = .success(imageDataMock)
17
18         let sut: UIImageView = .init()
19
20         let dummyURL: URL = try XCTUnwrap(.init(string: "www.something.com/image.png"))
21
22         // When
23         sut.setImageFromURL(dummyURL, network: networkStub, mainQueue: .global())
24
25         // Then
26         XCTAssertNotNil(sut.image)
27     }
28 }
```

E como fica no teste?

3. Injeção de Dependência Avançada

1. Singletons
2. Factories
3. Service Locator



3. Injeção de Dependência Avançada

1. SINGLETONS

Singleton é um padrão de projeto criacional, que garante que uma classe terá somente uma instância provendo um ponto de acesso global para ela.

Implementações do padrão de projeto SINGLETON devem garantir que uma e somente uma instância dessa classe exista, e prover um ponto de acesso global a ela.

Prós:

- Acesso Global para uma única instância
- Estado Compartilhado

Contras:

- Acesso Global para uma única instância
- Estado Compartilhado

```
1 final class UserSession {
2     static let shared = UserSession()
3
4     private(set) var currentUser: LoggedUser?
5     var isValid: Bool { currentUser?.token.isEmpty == false }
6
7     private init() {}
8
9     func login(
10         username: String,
11         password: String,
12         then completion: @escaping (Result<Void, Error>) -> Void
13     ) {
14         dispatchLoginRequest(username: username, password: password) { [weak self] result in
15             do {
16                 let response = try result.get()
17                 self?.currentUser = .init(id: response.id, username: username, token:
18                     response.token)
19                 completion(.success(()))
20             } catch {
21                 self?.currentUser = nil
22                 completion(.failure(error))
23             }
24         }
25
26         private func dispatchLoginRequest(
27             username: String,
28             password: String,
29             then completion: @escaping (Result<LoginResponse, Error>) -> Void
30         ) { /* ... */ }
31 }
```

3. Injeção de Dependência Avançada

1. SINGLETONS

- OK, mas singleton não é um anti-padrão?
- Eles não são "intestáveis"?
- Eles não vão na contramão da idéia de Inversão de Controle e Injeção de Dependência?
- ...



3. Injeção de Dependência Avançada

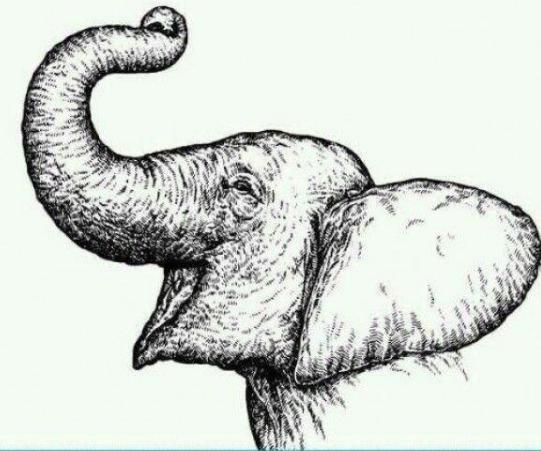
1. SINGLETONS

A resposta curta para os questionamentos do slide anterior é:

DEPENDE 😅

- Depende de **COMO**, **QUANDO** e **PORQUE** você está usando esse padrão.
- Se é uma aplicação pequena ou um caso específico dentro de um projeto grande, pode ser ok. Mas você sempre deve ter cuidado e compreender que o **COMO** tem ainda mais peso que o **PORQUE**, visto que você use vários deles (por simplicidade, por exemplo), ainda é possível manter seu código testável usando as técnicas que vimos nos slides anteriores.
- **Tenha muito cuidado para não torná-los uma "GOD CLASS", com muitas responsabilidades. Se vai utilizá-los, cuide para que seus singletons tenham um propósito único e/ou muito bem definido, tomando conta de contextos ou domínios específicos da aplicação.**

The answer to every programming question ever conceived



It Depends

The Definitive Guide

O RLY?

@ThePracticalDev

3. Injeção de Dependência Avançada

1. SINGLETONS

É possível tornar aquele singleton dos slides anteriores algo testável?

SIM 

E nem é complicado...

Basta extrair sua interface, conformar o singleton a esta interface e aplicar uma das técnicas básicas dos slides anteriores.

```
1 protocol UserSessionProtocol {
2     var currentUser: LoggedUser? { get }
3     var isValid: Bool { get }
4     func login(
5         username: String,
6         password: String,
7         then completion: @escaping (Result<Void, Error>) -> Void
8     )
9 }
10 extension UserSession: UserSessionProtocol {}
11
12 final class SomeViewModelThatNeedsUserSession {
13     let userSession: UserSessionProtocol
14     init(userSession: UserSessionProtocol) {
15         self.userSession = userSession
16     }
17 }
```

3. Injeção de Dependência Avançada

1. SINGLETONS:

Caso de Uso

Em um projeto não muito complexo ou biblioteca, podemos usar de singletons para construir um sistema de injeção simples, efetivo e de fácil entendimento.

- Esse cara vai resolver meus problemas pra sempre?

Não. Mas é uma opção simples de implementar que já desacopla seu código, e dependendo do tamanho seu projeto, é mais do que o suficiente.

- Quando esse cara pode passar a ser problemático?

Em um contexto onde há modularização, essa opção pode gerar acoplamento indesejado, dependendo de como for usada.

Sugestão de leitura:

[How to Control The World \(pointfree.co\)](#)

```
1 protocol HasURLSession {
2     var urlSession: URLSession { get }
3 }
4
5 protocol HasUserDefaults {
6     var userDefaults: UserDefaults { get }
7 }
8
9 protocol HasUserSession {
10    var userSession: UserSessionProtocol { get }
11 }
12
13 protocol AppDependenciesContainer: HasURLSession, HasUserDefaults, HasUserSession {}
14
15 final class AppDependenciesEnvironment: AppDependenciesContainer {
16     static let shared = AppDependenciesEnvironment()
17
18     private(set) var urlSession: URLSession
19     private(set) var userDefaults: UserDefaults
20     private(set) var userSession: UserSessionProtocol
21
22     private init() {
23         self.urlSession = .shared
24         self.userDefaults = .standard
25         self.userSession = UserSession.shared
26     }
27 }
28
29 final class SomeViewModel {
30     typealias Dependencies = HasUserSession & HasUserDefaults
31     private let dependencies: Dependencies
32
33     init(dependencies: Dependencies = AppDependenciesEnvironment.shared) {
34         self.dependencies = dependencies
35     }
36 }
```

3. Injeção de Dependência Avançada

1. SINGLETONS

- Há também uma outra forma de se implementar singletons chamada:
Singleton + ou Singleton Extended
- Esse padrão pode ser visto em classes do sistema como
FileManager, UserDefaults, URLSession e alguns outros, nos bibliotecas padrões da Apple.
- Eles são basicamente um singleton que não possui o inicializador privado e são criados de uma forma que nos permite um pouco mais de controle sobre seu estado compartilhado, mas ainda assim mantendo a opção de utilizar uma instância única e compartilhada.
- Também é comum que este tipo use o modificador de acesso **open**, para tornar possível que possamos herdar da classe base e controlar algumas funções ou propriedades ao sobrescrevê-las.

```
1 open class PersistencyManager {  
2     // MARK: - Single Instance  
3     static let shared = PersistencyManager(userDefaults: .standard)  
4  
5     // MARK: - Dependencies  
6  
7     private let userDefaults: UserDefaults  
8  
9     // MARK: - Public Properties  
10  
11    private(set) var values: [String] = []  
12  
13    // MARK: - Initialization  
14  
15    init(userDefaults: UserDefaults) {  
16        self.userDefaults = userDefaults  
17    }  
18  
19    // MARK: - Public Functions  
20  
21    func save(_ value: String) -> Bool {  
22        var newValues = values  
23        newValues.append(value)  
24        userDefaults.set(value, forKey: "valuesKey")  
25  
26        let sincronizationSucceeded = userDefaults.synchronize()  
27        if sincronizationSucceeded { values = newValues }  
28        return sincronizationSucceeded  
29    }  
30    /* ... */  
31 }
```

3. Injeção de Dependência Avançada

1. SINGLETONS

```
1 final class UserDefaultsSpy: UserDefaults {
2     private(set) var setValueCalled = false
3     private(set) var setValuePassed: Any?
4     private(set) var setValueKeyPassed: String?
5
6     override func set(_ value: Any?, forKey defaultName: String) {
7         setValueCalled = true
8         setValuePassed = value
9         setValueKeyPassed = defaultName
10    }
11
12    private(set) var synchronizeCalled = false
13    override func synchronize() -> Bool {
14        synchronizeCalled = true
15        return true
16    }
17 }
```

```
1 final class DefaultsManagerTests: XCTestCase {
2     func test_whenAddIsCalled_userDefaultsShouldReceiveValue_andSyncronize() {
3         // Given
4         let userDefaultsSpy = UserDefaultsSpy()
5         let sut = PersistenceManager(
6             userDefaults: userDefaultsSpy
7         )
8         let valueToAdd = "some value"
9         // When
10        let addSucceeded = sut.save(valueToAdd)
11        // Then
12        XCTAssertTrue(addSucceeded)
13        XCTAssertTrue(userDefaultsSpy.setValueCalled)
14        XCTAssertEqual(1, sut.values.count)
15        XCTAssertTrue(userDefaultsSpy.synchronizeCalled)
16    }
17
18 }
```

3. Injeção de Dependência Avançada

2. FACTORIES

IT'S DANGEROUS TO CODE
ALONE



TAKE THIS DESIGN PATTERN

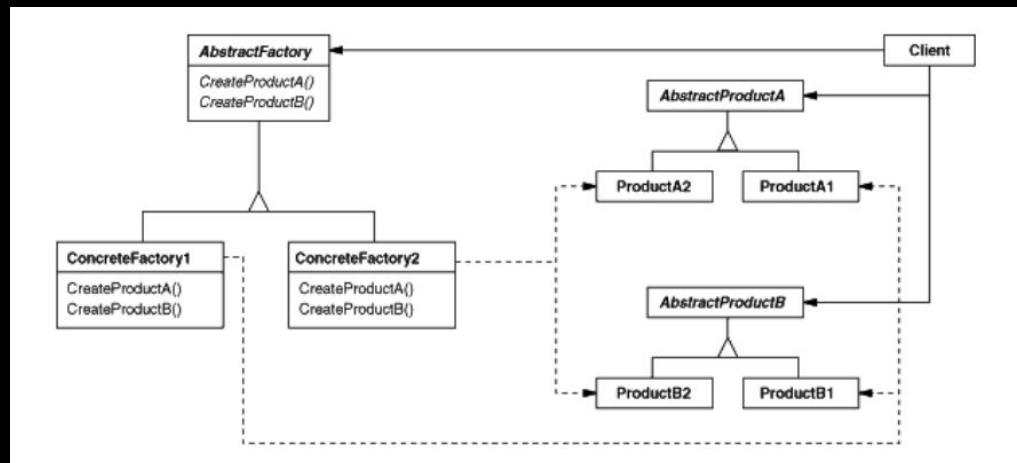
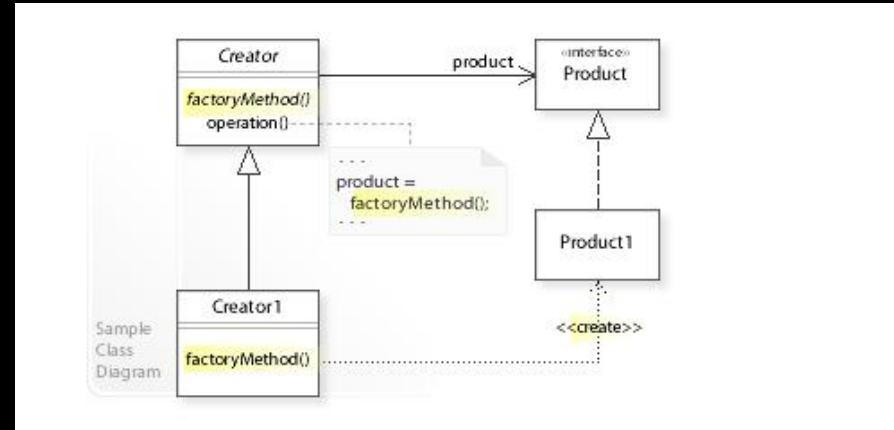
3. Injeção de Dependência Avançada

2. FACTORIES

- **Factories** são padrões de projeto criacionais, que buscam resolver o problema de criar objetos sem explicitar suas classes concretas.
- Objetivo: encapsular os detalhes de implementação sobre como um objeto é criado.
- Vantagens: nos provê uma interface comum para construção de objetos onde o consumidor não precisa saber os detalhes sobre a lógica de como o objeto é construído

IMPORTANTE:

A idéia aqui é resumir o conceito de Factory, para mais detalhes busque: Factory Method e Abstract Factory



3. Injeção de Dependência Avançada

2. FACTORIES

Objetivo:

Controlar a criação do PokemonTradeViewController e suas dependências.

```
1 final class PokemonListViewController: UITableViewController {
2     // MARK: - Dependencies
3
4     private let userToTradeWithID: String
5     private let pokemonsService: PokemonsServiceProtocol
6
7     // MARK: - Properties
8
9     private var pokemons: [Pokemon] = []
10
11    // MARK: - Initialization
12
13    init(
14        userToTradeWithID: String,
15        pokemonsService: PokemonsServiceProtocol
16    ) {
17        self.userToTradeWithID = userToTradeWithID
18        self.pokemonsService = pokemonsService
19        super.init(nibName: nil, bundle: nil)
20    }
21
22    // MARK: - Lifecycle
23
24    required init?(coder: NSCoder) {
25        fatalError("init(coder:) has not been implemented")
26    }
27
28    override func viewWillAppears(_ animated: Bool) {
29        super.viewWillAppears(animated)
30        pokemonsService.loadPokemons(for: userToTradeWithID) { [weak self] result in
31            do {
32                let pokemons = try result.get()
33                self?.reloadTableView(with: pokemons)
34            } catch { print(error) }
35        }
36    }
37 }
```

```
1 extension PokemonListViewController {
2     // MARK: - Private API
3
4     private func reloadTableView(with pokemons: [Pokemon]) {
5         /* ... */
6     }
7
8     override func tableView(
9         _ tableView: UITableView,
10         didSelectRowAt indexPath: IndexPath
11     ) {
12        let pokemonIWant = pokemons[indexPath.row]
13        let tradeViewController: PokemonTradeViewController = .init(
14            pokemonIWant: pokemonIWant,
15            userToTradeWithID: userToTradeWithID,
16            tradingManager: TradingManager(),
17            pokemonsService: pokemonsService
18        )
19        navigationController?.pushViewController(
20            tradeViewController,
21            animated: true
22        )
23    }
24 }
```

3. Injeção de Dependência Avançada

2. FACTORIES

Podemos criar uma factory que conhece o container de dependências e sabe como criar os controllers

```
1 protocol ViewControllersFactoryProtocol {
2     func makePokemonTradeViewController(
3         for pokemonIWant: Pokemon,
4         userToTradeWithID: String
5     ) -> UIViewController
6 }
7
8 struct ViewControllersFactory: ViewControllersFactoryProtocol {
9     typealias Dependencies = HasPokemonsService & HasTradingManager
10    private let dependencies: Dependencies
11
12    init(dependencies: Dependencies = AppDependenciesEnvironment.shared) {
13        self.dependencies = dependencies
14    }
15
16    func makePokemonTradeViewController(
17        for pokemonIWant: Pokemon,
18        userToTradeWithID: String
19    ) -> UIViewController {
20        let viewController: PokemonTradeViewController = .init(
21            pokemonIWant: pokemonIWant,
22            userToTradeWithID: userToTradeWithID,
23            tradingManager: dependencies.tradingManager,
24            pokemonsService: dependencies.pokemonsService
25        )
26        return viewController
27    }
28 }
```

3. Injeção de Dependência Avançada

2. FACTORIES

```
1 final class PokemonListViewController: UITableViewController {  
2     // MARK: - Dependencies  
3  
4     private let pokemonsService: PokemonsServiceProtocol  
5     private let userToTradeWithID: String  
6     private let viewControllersFactory: ViewControllersFactoryProtocol  
7  
8     // MARK: - Properties  
9  
10    private var pokemons: [Pokemon] = []  
11  
12    // MARK: - Initialization  
13  
14    init(  
15        pokemonsService: PokemonsServiceProtocol,  
16        userToTradeWithID: String,  
17        viewControllersFactory: ViewControllersFactoryProtocol  
18    ) {  
19        self.pokemonsService = pokemonsService  
20        self.userToTradeWithID = userToTradeWithID  
21        self.viewControllersFactory = viewControllersFactory  
22        super.init(nibName: nil, bundle: nil)  
23    }  
24  
25    // MARK: - Lifecycle  
26  
27    override func viewWillAppears(_ animated: Bool) {  
28        super.viewWillAppears(animated)  
29        pokemonsService.loadPokemons(for: userToTradeWithID) { [weak self] result in  
30            do {  
31                let pokemons = try result.get()  
32                self?.reloadTableView(with: pokemons)  
33            } catch { print(error) }  
34        }  
35    }  
36 }
```

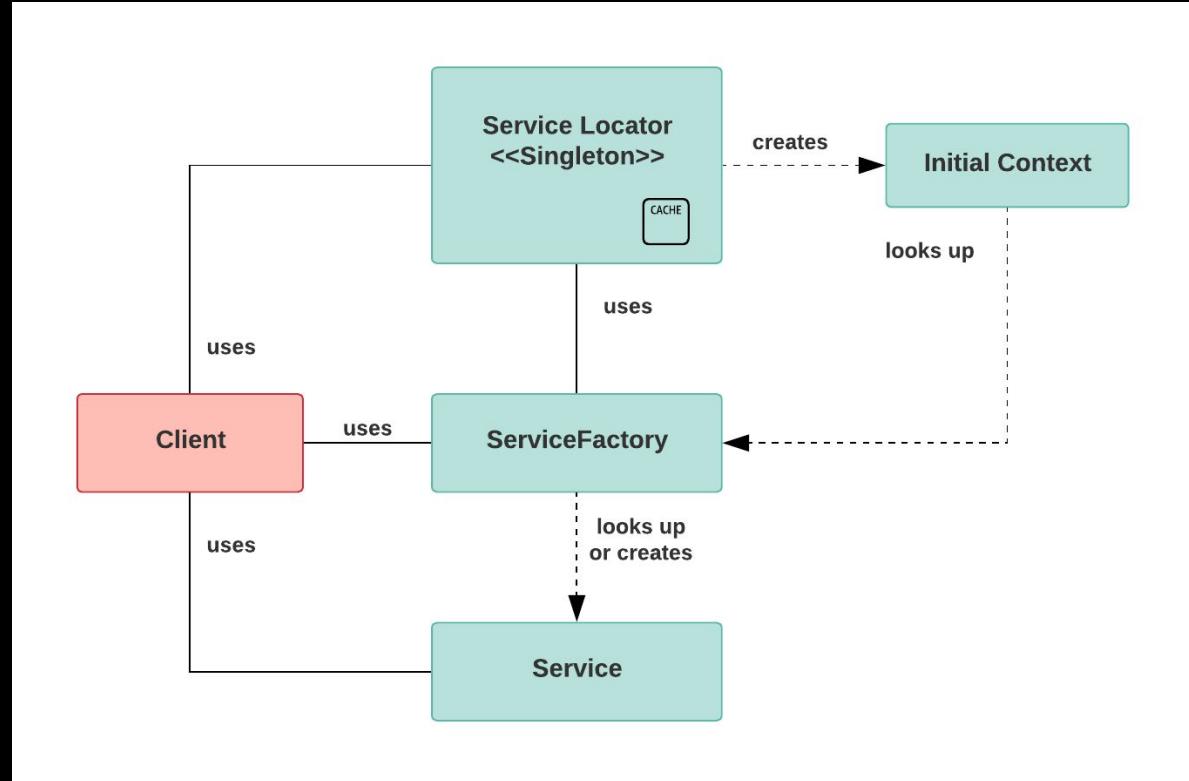
Agora refatoramos o ViewController para receber as nova forma de injeção e a factory

```
1 extension PokemonListViewController {  
2     // MARK: - Private API  
3  
4     private func reloadTableView(with pokemons: [Pokemon]) { /* ... */ }  
5  
6     override func tableView(  
7         _ tableView: UITableView,  
8         didSelectRowAt indexPath: IndexPath  
9     ) {  
10        let pokemonIWant = pokemons[indexPath.row]  
11        let tradeViewController = viewControllersFactory.makePokemonTradeViewController(  
12            for: pokemonIWant,  
13            userToTradeWithID: userToTradeWithID  
14        )  
15        navigationController?.pushViewController(tradeViewController, animated: true)  
16    }  
17 }
```

3. Injeção de Dependência Avançada

3. SERVICE LOCATOR

Service Locator é padrão que retém e provê acesso a serviços ou instâncias de dependências.



3. Injeção de Dependência Avançada

3. SERVICE LOCATOR

```
1 public typealias ServiceLocatorInterface = Resolver & Container
2 public final class ServiceLocator {
3     public static let shared: ServiceLocatorInterface = ServiceLocator()
4
5     var instances: [String: Any] = [:]
6     var lazyInstances: NSMapTable<NSString, LazyInstanceWrapper> = .init(
7         keyOptions: .strongMemory,
8         valueOptions: .weakMemory
9     )
10
11    typealias LazyDependencyFactory = () -> Any
12    var factories: [String: LazyDependencyFactory] = [:]
13
14    final class LazyInstanceWrapper {
15        let instance: Any
16        init(instance: Any) {
17            self.instance = instance
18        }
19    }
20
21    private func getKey<T>(for metaType: T.Type) -> String {
22        let key = String(describing: T.self)
23        return key
24    }
25 }
```

```
1 public protocol Resolver {
2     func resolve<T>(_ metaType: T.Type) -> T
3     func autoResolve<T>() -> T
4 }
5
6 public protocol Container {
7     func register<T>(instance: T, forMetaType metaType: T.Type)
8     func register<T>(
9         factory: @escaping (Resolver) -> T,
10        forMetaType metaType: T.Type
11    )
12 }
13 extension Container {
14     func register<T>(
15         factory: @escaping () -> T,
16         forMetaType metaType: T.Type
17     ) {
18         self.register(
19             factory: { _ in factory() },
20             forMetaType: metaType
21         )
22     }
23 }
```

<https://developer.apple.com/documentation/foundation/nsmaptable>

3. Injeção de Dependência Avançada

3. SERVICE LOCATOR

```
1 extension ServiceLocator: Container {
2     public func register<T>(instance: T, forMetaType metaType: T.Type
3     ) {
4         let key = getKey(for: metaType)
5         guard instances[key] == nil else {
6             fatalError("You must not register something twice!")
7         }
8         instances[key] = instance
9     }
10
11    public func register<T>(factory: @escaping (Resolver) -> T, forMetaType metaType: T.Type) {
12        let key = getKey(for: metaType)
13        guard factories[key] == nil else {
14            fatalError("You must not register something twice!")
15        }
16        factories[key] = { factory(self) }
17    }
18 }
```

3. Injeção de Dependência Avançada

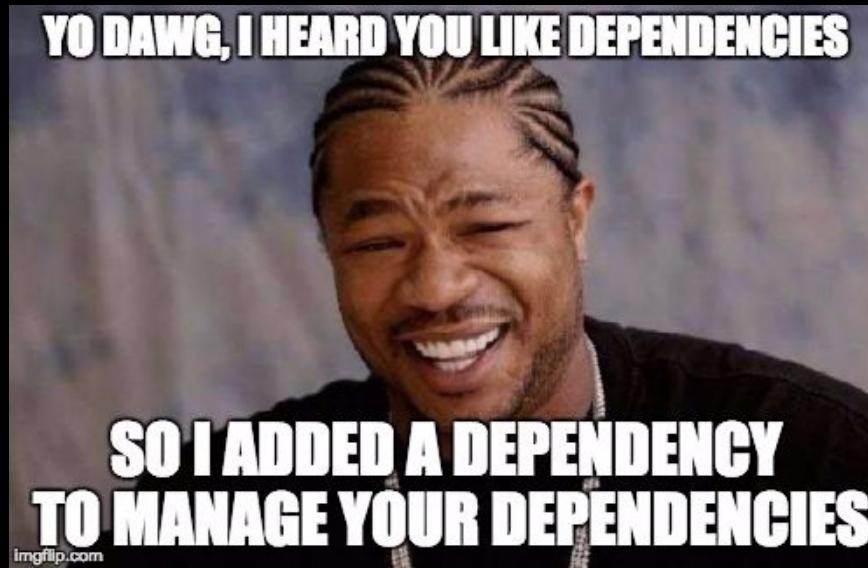
3. SERVICE LOCATOR

```
1 extension ServiceLocator {
2     private func getInstance<T>(forMetatype: T.Type) -> T? {
3         let key = getKey(for: T.self)
4         if let instance = instances[key] as? T {
5             return instance
6         } else if let lazyInstance = getLazyInstance(for: T.self, key: key) {
7             return lazyInstance
8         } else {
9             return nil
10    }
11 }
12
13 private func getLazyInstance<T>(for _: T.Type, key: String) -> T? {
14     let objectKey = key as NSString
15
16     if let instanceInMemory = lazyInstances.object(forKey: objectKey)?.instance as? T {
17         return instanceInMemory
18     }
19
20     guard
21         let factory = factories[key],
22         let newInstance = factory() as? T
23     else { return nil }
24
25     let wrappedInstance = LazyInstanceWrapper(
26         instance: newInstance
27     )
28     lazyInstances.setObject(
29         wrappedInstance,
30         forKey: objectKey
31     )
32
33     return newInstance
34 }
35 }
```

```
1 extension ServiceLocator: Resolver {
2     public func resolve<T>(_ metaType: T.Type) -> T {
3         guard let instance = getInstance(forMetatype: T.self) else {
4             fatalError("There is no instance registered for `\\(\(getKey(for: T.self))`!")
5         }
6         return instance
7     }
8
9     public func autoResolve<T>() -> T {
10        guard let instance = getInstance(forMetatype: T.self) else {
11            fatalError("There is no instance registered for `\\(\(getKey(for: T.self))`!")
12        }
13        return instance
14    }
15 }
```

3. Injeção de Dependência Avançada

3. SERVICE LOCATOR



3. Injeção de Dependência Avançada

3. SERVICE LOCATOR

```
1 final class AppDelegate: UIResponder, UIApplicationDelegate {
2     var serviceLocator: ServiceLocatorInterface = ServiceLocator.shared
3     func application(
4         _ application: UIApplication,
5         didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?
6     ) -> Bool {
7         registerDependencies()
8         return true
9     }
10
11    private func registerDependencies() {
12        serviceLocator.register(instance: URLSession.shared, forMetaType: URLSession.self)
13        serviceLocator.register(
14            factory: { resolver in
15                let session: URLSession = resolver.autoResolve()
16                return LoginService(urlSession: session)
17            },
18            forMetaType: LoginServiceProtocol.self
19        )
20        serviceLocator.register(
21            factory: UserSession.init,
22            forMetaType: UserSessionProtocol.self
23        )
24    }
25 }
```

```
1 protocol LoginServiceProtocol { /* ... */ }
2 struct LoginService: LoginServiceProtocol {
3     private let urlSession: URLSession
4     init(urlSession: URLSession) {
5         self.urlSession = urlSession
6     }
7 }
8
9 protocol UserSessionProtocol { /* ... */ }
10 final class UserSession: UserSessionProtocol {
11     init() {}
12 }
```

3. Injeção de Dependência Avançada

3. SERVICE LOCATOR

```
1 final class LoginViewModel {  
2     private let loginService: LoginServiceProtocol  
3     private let userSession: UserSessionProtocol  
4  
5     init(  
6         loginService: LoginServiceProtocol? = nil,  
7         userSession: UserSessionProtocol? = nil  
8     ) {  
9         // if you don't want to expose the ServiceLocator...  
10        self.loginService = loginService ?? ServiceLocator.shared.autoResolve()  
11        self.userSession = userSession ?? ServiceLocator.shared.resolve(UserSessionProtocol.self)  
12    }  
13  
14    // ...  
15 }
```

```
1 final class OtherLoginViewModel {  
2     private let loginService: LoginServiceProtocol  
3     private let userSession: UserSessionProtocol  
4  
5     // If you don't mind exposing it...  
6     init(  
7         loginService: LoginServiceProtocol = ServiceLocator.shared.autoResolve(),  
8         userSession: UserSessionProtocol = ServiceLocator.shared.resolve(UserSessionProtocol.self)  
9     ) {  
10        self.loginService = loginService  
11        self.userSession = userSession  
12    }  
13  
14    // ...  
15 }
```

3. Injeção de Dependência Avançada

3. SERVICE LOCATOR



3. Injeção de Dependência Avançada

3. SERVICE LOCATOR

O que são property wrappers em Swift?

Property Wrapper é uma estrutura genérica para encapsular acessos de leitura e escrita em uma propriedade, também adicionando comportamentos a ela.

```
1 /* ----- */
2 /* Definition */
3 /* ----- */
4
5 @propertyWrapper
6 struct Capitalized {
7     var wrappedValue: String {
8         didSet { wrappedValue = wrappedValue.capitalized }
9     }
10
11    init(wrappedValue: String) {
12        self.wrappedValue = wrappedValue.capitalized
13    }
14 }
15
16 struct User {
17     @Capitalized var firstName: String
18     @Capitalized var lastName: String
19 }
20
21 /* ----- */
22 /* Example */
23 /* ----- */
24
25 var user = User(firstName: "eduardo", lastName: "bocato")
26
27 // Prints -> "Eduardo Bocato"
28 print(user.firstName, user.lastName)
```

3. Injeção de Dependência Avançada

3. SERVICE LOCATOR

Primeiro passo

- Pequeno refator no Resolver

Depois...

- @propertyWrapper para encapsular a lógica necessária para resolver uma instância
- Controlar o acesso às propriedades dessa estrutura
- Mensagens de erro para indicar quando algo não ocorreu como esperado

```
1 @propertyWrapper
2 public final class Dependency<T> {
3     private let resolver: Resolver?
4     private let failureHandler: (String) -> Void
5     private(set) var resolvedValue: T!
6
7     public var wrappedValue: T {
8         resolveIfNeeded()
9         return resolvedValue!
10    }
11
12    public convenience init() {
13        self.init(
14            resolvedValue: nil,
15            resolver: ServiceLocator.shared,
16            failureHandler: { preconditionFailure($0) }
17        )
18    }
19
20    fileprivate init(
21        resolvedValue: T?,
22        resolver: Resolver?,
23        failureHandler: @escaping (String) -> Void
24    ) {
25        self.resolvedValue = resolvedValue
26        self.resolver = resolver
27        self.failureHandler = failureHandler
28    }
29
30    private func resolveIfNeeded() {
31        guard resolvedValue == nil else {
32            failureHandler("\(type(of: self)) shouldn't be resolved twice!")
33            return
34        }
35        guard let instanceFromContainer = resolver?.resolve(T.self) else {
36            failureHandler("Could not resolve \(type(of: self)), check it it was registered!")
37            return
38        }
39        resolvedValue = instanceFromContainer
40    }
41 }
```

```
1 public protocol Resolver {
2     func resolve<T>(_ metaType: T.Type) -> T?
3     func autoResolve<T>() -> T?
4 }
5
6 // ...
7
8 extension ServiceLocator: Resolver {
9     public func resolve<T>(_ metaType: T.Type) -> T? {
10         getInstance(forMetatype: T.self)
11     }
12
13     public func autoResolve<T>() -> T? {
14         getInstance(forMetatype: T.self)
15     }
16 }
17 // ...
```

3. Injeção de Dependência Avançada

3. SERVICE LOCATOR

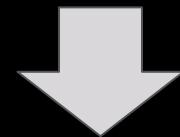
Vantagens

- Simplificar o acesso a propriedades registradas sem que precisemos resolvê-las explicitamente ✓

- Código mais limpo, sem boilerplate ✓

- Mais moderno e "Swifty" ✓

```
1 final class ViewModelWithoutPropertyWrappers {
2     private let loginService: LoginServiceProtocol
3     private let userSession: UserSessionProtocol
4
5     init(
6         loginService: LoginServiceProtocol? = nil,
7         userSession: UserSessionProtocol? = nil
8     ) {
9         self.loginService = loginService ?? ServiceLocator.shared.autoResolve()!
10        self.userSession = userSession ?? ServiceLocator.shared.resolve(UserSessionProtocol.self)!
11    }
12
13    // ...
14 }
```



Problemas

- E os testes, como ficam? 🤔

```
1 final class ViewModelWithPropertyWrappers {
2     @Dependency var loginService: LoginServiceProtocol
3     @Dependency var userSession: UserSessionProtocol
4     // ...
5 }
```

3. Injeção de Dependência Avançada

3. SERVICE LOCATOR

1. No mesmo arquivo (pois o initializer é fileprivate), crio uma extensão para iniciar um @Dependency com um valor logo de cara.

2. Empacoto isso com #if DEBUG para garantir que esse código não vá para produção...

```
1 #if DEBUG
2 extension Dependency {
3     // To enable mocking from outside
4     static func resolved(_ value: T) -> Self {
5         .init(
6             resolvedValue: value,
7             resolver: nil,
8             failureHandler: { _ in }
9         )
10    }
11 }
12 #endif
```

3. Injeção de Dependência Avançada

3. SERVICE LOCATOR

Organizo as dependências do ViewModel...

```
1 struct LoginViewEnvironment {
2     @Dependency var loginService: LoginServiceProtocol
3     @Dependency var userSession: UserSessionProtocol
4 }
5
6 final class LoginViewModel {
7     private let environment: LoginViewEnvironment
8     init(environment: LoginViewEnvironment) {
9         self.environment = environment
10    }
11    // ...
12 }
```

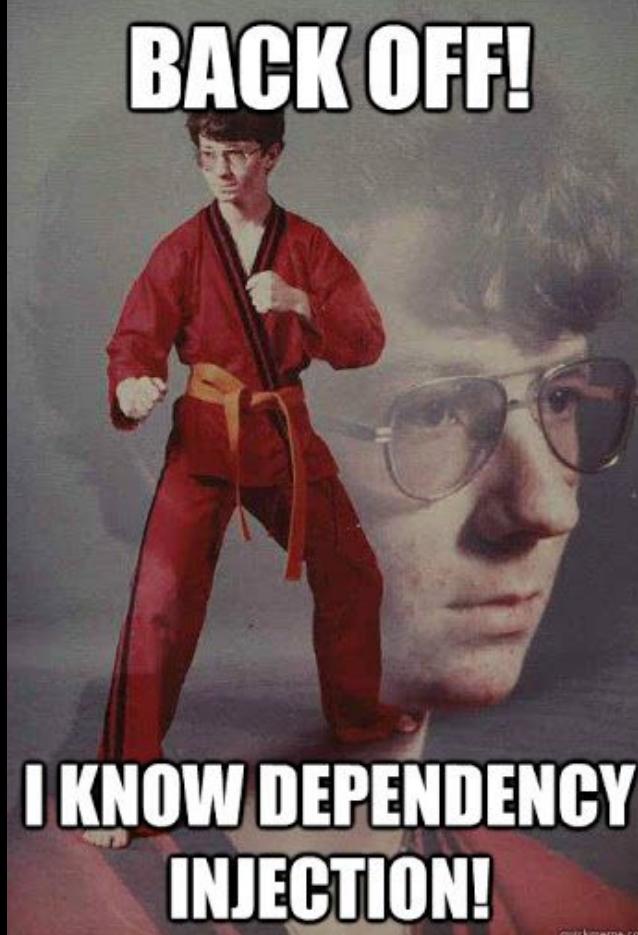
Agora criamos um construtor para as dependências...

```
1 #if DEBUG
2 extension LoginViewEnvironment {
3     static func mocking(
4         loginService: LoginServiceProtocol,
5         userSession: UserSessionProtocol
6     ) -> Self {
7         .init(
8             loginService: .resolved(loginService),
9             userSession: .resolved(userSession)
10        )
11    }
12 }
13 #endif
```

Pronto, agora é só testar!

```
1 final class LoginViewModelTests: XCTestCase {
2     func test_something() {
3         // Given
4         let sut: LoginViewModel = .init(
5             environment: .mocking(
6                 loginService: LoginServiceMock(),
7                 userSession: UserSessionMock()
8             )
9         )
10        _ = sut // ... Test your stuff!
11    }
12 }
```

BACK OFF!



**I KNOW DEPENDENCY
INJECTION!**

quickmeme.com

4. Conclusão

Mas Injeção de Dependência é só isso?

- Não, tem outras técnicas, bibliotecas e por aí vai...
- Os conceitos apresentados aqui, combinados ou não, são no mínimo um bom ponto de partida para resolver vários problemas comuns ou no mínimo te ajudar a escolher uma biblioteca ou outra técnica.

Se eu fizer isso tudo é certeza que meu código vai ficar testável?

- Também não.
- Mas se você aplicar os conceitos corretamente em conjunto com outras boas práticas e uma boa modelagem de dependências, provavelmente vai.

4. Conclusão

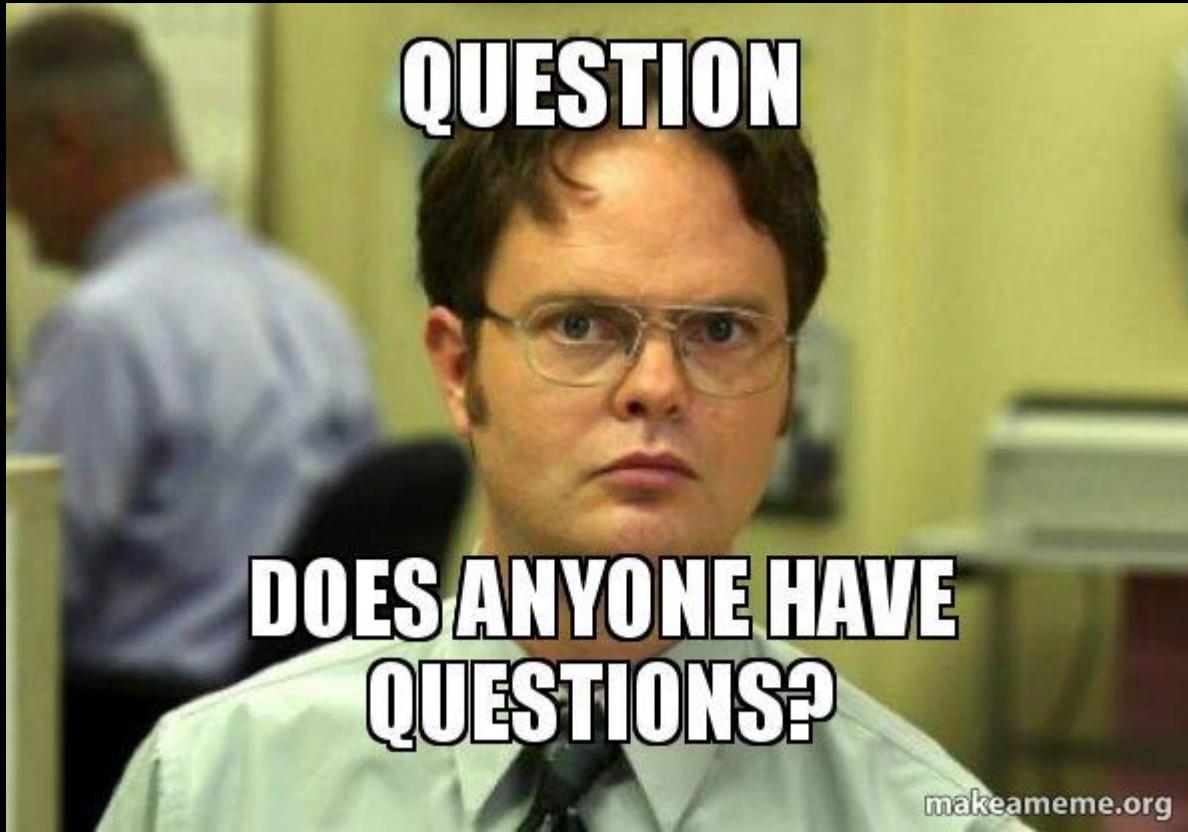
Mas Injeção de Dependência é só isso?

- Não, tem outras técnicas, bibliotecas e por aí vai...
- Os conceitos apresentados aqui, combinados ou não, são no mínimo um bom ponto de partida para resolver vários problemas comuns ou no mínimo te ajudar a escolher uma biblioteca ou outra técnica.

Se eu fizer isso tudo é certeza que meu código vai ficar testável?

- Também não.
- Mas se você aplicar os conceitos corretamente em conjunto com outras boas práticas e uma boa modelagem de dependências, provavelmente vai.

5. Perguntas?



makeameme.org