Dependency injection in practice

Workshop @ mDevCamp 2020

Welcome

Dependency injection in practice

Workshop @ mDevCamp 2020

- Checkout code

```
git clone git@github.com:Showmax/di-workshop.git
cd di-workshop/
xed .
```

- Build & Run
- Explore app and code
- Ask questions on Slack channel (invite)

Outline

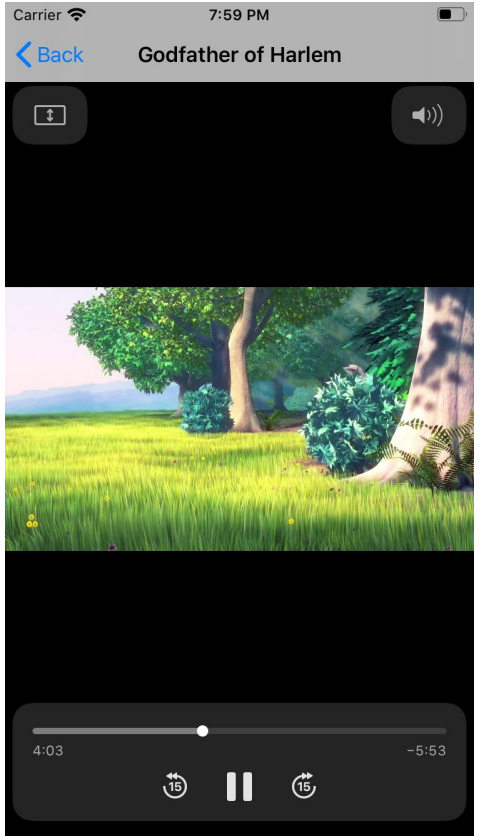
1. Checkout repository
2. Explore example app
3. Solve problem 1: scattered dependencies
4. Solve problem 2: not mockable dependencies
5. Solve problem 3: generate boilerplate mock code
6. Solve problem 4: inject dependencies
7. Discussion

Checkout repository

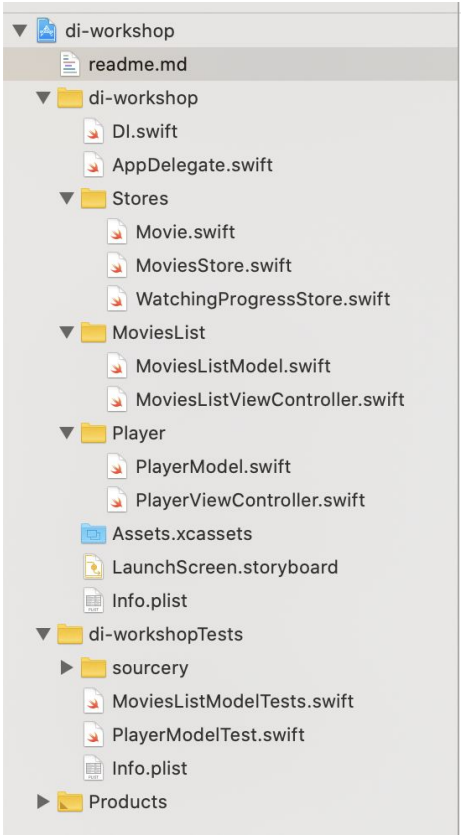
- ```
  git clone git@github.com:Showmax/di-workshop.git
  cd di-workshop/
  xed .
  ```
- Build & Run
- Explore app and code
  ```

Explore app


Carrier
7:58 PM
Movies + My Watching Progress
Brooklyn Nine-Nine 0%
Downton Abbey 0%
Godfather of Harlem 0%
Jason Bourne 0%
Paddington 0%
Ramy 0%
Rick and Morty 0%
Shrek 0%
The River 0%
Watchmen 0%
Younger 0%


Carrier
7:59 PM
Back
Godfather of Harlem
4:03
-5:53

Explore code and run tests



Problem #1: scattered dependencies

Why?

- ?

Solution

- ?

Problem #1: scattered dependencies

Why?

- Not testable (run tests)
- Cannot supply custom MoviesStore into model
  to fake expected movies in test
- Hard to get overview on what this object depends on

Solution

- ?

Problem #1: scattered dependencies

Why?

- Not testable (run tests)
- Cannot supply custom MoviesStore into model
  to fake expected movies in test
- Hard to get overview on what this object depends on

Solution

- For each model organise dependencies
  into `struct Dependencies`

Problem #2: not mockable dependencies

Why?

- ?

Solution

- ?

Problem #2: not mockable dependencies

Why?

- Still not testable
- We depend on details instead on abstraction
- We cannot influence what goes from store when testing

Solution

- ?

Problem #2: not mockable dependencies

Why?

- Still not testable
- We depend on details instead on abstraction
- We cannot influence what goes from store when testing

Solution

- Hide stores behind protocol
- Build -> see which methods should be in protocol
- Tests: mocks implements protocol

Problem #3: boilerplate code in mocks

Why?

- ?

Solution

- ?

Problem #3: boilerplate code in mocks

Why?

- Hard to manage, repetitive
- Change in protocol -> manually update mock
- Code not used in production

Solution

- Generate repetitive code with great Sourcery
- Drop in `di-workshopTests/sourcery` folder + add test target
- Mark protocols for AutoMockable
- (optional) install sourcery `brew install sourcery`
- Run `bash generateAutoMockable`

Problem #4: boilerplate code around dependencies

Why?

- Imagine more layers, dependencies, more complex code
- Soon will result in forwarding dependencies in lower layers
- Especially things like networking, feature flags, cache...

Solution

- Drop in DI.swift (based on great Lyft's talk)
- Define extensions for stores
- Use instance getter in `struct Dependencies`

Discussion

- Your experience with DI frameworks?
- Your approach for refactoring not-testable code?
- How do you manage dependencies?
- How do you update mocks?

Thank you!