DEVCON

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Notice of Rights

Notice of Liability

Trademarks

18: Advanced Unidirectional Architecture

Technology and requirements are constantly changing, while at the same time, we are expected to build new features, fix bugs, and adopt new technologies faster and faster.

In this advanced tutorial we will combine all the cutting edge architecture design techniques such as reactive programming, dependency injection, protocol oriented programming, use cases, unidirectional data flow, and more in order to master the art of designing codebases that can easily change over time. We'll look at what causes code to change and how to use that insight to minimize the effort needed to build and maintain iOS apps.

Then, we'll practice the techniques and try fun things like easily switching from RxSwift to ReactiveSwift, from CoreData to Realm, from one view implementation to another. At the end, we'll discuss how to take these techniques home and easily apply them to your existing codebases regardless of what frameworks and architectures your apps use.

52 Advanced Unidirectional Architecture: Demo 1

By René Cacheaux

In this demo, you will build a simple view controller that's instantiated with an observable and control the view controller's behavior inside a playground.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

Note: Begin work with page **8. Putting it All Together** in **Demo 1** playground in the starter workspace, **Demo1/starter/Koober/App/KooberApp.xcworkspace**. The theory behind this demo is explained in the playground's pages 1 - 7.

If the playground fails to build and run, open the playground's overall **Source** directory and open **ReSwiftExtensions/ReSwiftRxSwift/RxStoreSubscriber.swift**. Then delete the `import RxSwift` line, save, and add the import back in. Go back to the playground, it should now build and run.

1) Settings VC Observable

Add the following stored property to `SettingsViewController`:

```
let stateObservable: Observable<SettingsViewState>
```

2) Inject Observable

Add the following initializer to `SettingsViewController`:

```
init(stateObservable: Observable<SettingsViewState>) {
    self.stateObservable = stateObservable
    super.init()
  }
```

3) Subscribe to State Changes

Call `subscribe(to:)` with the view controller's state observable in `viewDidLoad`:

```
  override func viewDidLoad() {
    super.viewDidLoad()
    subscribe(to: stateObservable)
  }
```

4) Add Settings VC Factory

Scroll down to `HomeViewController` and add the following stored property:

```
  let settingsViewControllerFactory: SettingsViewControllerFactory
```

5) Settings VC Injection

Add the following initializer to `HomeViewController`:

```
  init(settingsViewControllerFactory: SettingsViewControllerFactory) {
    self.settingsViewControllerFactory = settingsViewControllerFactory
    super.init()
  }
```

6) Implement goToSettings

Implement `goToSettings()` in `HomeViewController`:

```
  func goToSettings() {
    let settingsViewController =
      settingsViewControllerFactory.makeSettingsViewController()
    present(settingsViewController, animated: true)
  }
```

7) Dependency Container

That's it for the view controllers. Next is the `DependencyContainer`. For the `DependencyContainer` class, add conformance to `DependencyProvider`:

```
class DependencyContainer: DependencyProvider {
```

8) Redux Store

Add the following stored property to `DependencyContainer`:

```
let reduxStore: Store<AppState> = Store(reducer: reduce,
                                        state: nil)
```

9) Settings Observable Factory

Add the following method from the `DependencyProvider` protocol to `DependencyContainer`:

```
func makeSettingsViewStateObservable() ->
  Observable<SettingsViewState>
{

}
```

10) Make Observable

Make the settings view state observable inside `makeSettingsViewStateObservable()`:

```
let observable =
  reduxStore
    .makeObservable { appState in
      return appState.settingsViewState
    }.distinctUntilChanged()
```

11) Return the Observable

Return the new observable in `makeSettingsViewStateObservable()`:

```
return observable
```

12) Settings VC Factory

Add the `makeSettingsViewController()` method from the `DependencyProvider` protocol to `DependencyContainer`:

```
func makeSettingsViewController() -> UIViewController {

}
```

13) Make Settings VC

Make and return a new `SettingsViewController` inside `makeSettingsViewController()`:

```
let observable = makeSettingsViewStateObservable()
return SettingsViewController(stateObservable: observable)
```

14) Make Dependency Container

At the end of the playground, create a `DependencyContainer` as follows:

```
let dependencyContainer: DependencyProvider =
  DependencyContainer()
```

15) Make Home VC

On the next line, make a `HomeViewController`:

```
let homeViewController =
  HomeViewController(
    settingsViewControllerFactory: dependencyContainer
  )
```

16) Present Home VC

Present the `HomeViewController` in the playground live view:

```
PlaygroundPage.current.liveView = homeViewController
```

17) That's it!

Congrats, at this time you should have a good understanding of decoupling your view controllers from the outside world by injecting Observables! It's time to move onto how to incorporate persistence into a Redux architecture.

53 Advanced Unidirectional Architecture: Demo 2

By René Cacheaux

In this demo, you will incorporate persistence into the Redux architecture from Demo 1.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with page **9. Putting it All Together** in **Demo 2** playground in the starter workspace, **Demo2/starter/Koober/App/KooberApp.xcworkspace**. The theory behind this demo is explained in the playground's pages 1 - 8.

If the playground fails to build and run, open the playground's overall **Source** directory and open **ReSwiftExtensions/ReSwiftRxSwift/RxStoreSubscriber.swift**. Then delete the `import RxSwift` line, save, and add the import back in. Go back to the playground, it should now build and run.

1) Use Case Factory

Add the following stored property to `HomeViewController`:

```
let loadPersistedStateUseCaseFactory:
  LoadPersistedStateUseCaseFactory
```

2) Use Case Factory Injection

Add the `loadPersistedStateUseCaseFactory` parameter to `HomeViewController`'s initializer and set the corresponding stored property:

```
init(settingsViewControllerFactory:
       SettingsViewControllerFactory,
     loadPersistedStateUseCaseFactory:
        LoadPersistedStateUseCaseFactory)
{
  self.settingsViewControllerFactory =
    settingsViewControllerFactory
  self.loadPersistedStateUseCaseFactory =
    loadPersistedStateUseCaseFactory
  super.init()
}
```

3) Restore State Method

Add the following `restorePersistedState` method to `HomeViewController`:

```
func restorePersistedState() {
  let useCase =
    loadPersistedStateUseCaseFactory
      .makeLoadPersistedStateUseCase()
  useCase.start()
}
```

4) Restore Persisted State

Call `restorePersistedState()` in `HomeViewController`'s `viewDidLoad()`:

```
override func viewDidLoad() {
  super.viewDidLoad()
  restorePersistedState()
}
```

5) Adding Subsystems

Ok, `HomeViewController` is complete. Time to add dependency provider methods to `DependencyContainer`. Add the following stored properties to `DependencyContainer`:

```
let userStore: PersistentUserStore = FakePersistentUserStore()
let statePersister: StatePersister
```

6) Initialize State Persister

Add the following initializer to `DependencyContainer`:

```
init() {
  statePersister = ReduxStatePersister(reduxStore: reduxStore)
}
```

7) Use Case Factory Method

Still in `DependencyContainer`, implement the load persisted state use case factory:

```
func makeLoadPersistedStateUseCase() -> UseCase {
  return
    LoadPersistedStateUseCase(userStore: userStore,
                              reduxStore: reduxStore,
                              statePersister: statePersister)
}
```

8) Inject Use Case Factory

At the bottom of the playground where `HomeViewController` is instantiated, add the `dependencyContainer` as the argument to the `loadPersistedStateUseCaseFactory` parameter:

```
let homeViewController =
  HomeViewController(
    settingsViewControllerFactory: dependencyContainer,
    loadPersistedStateUseCaseFactory: dependencyContainer //Add.
  )
```

9) That's it!

Congrats, at this time you should have a good understanding of persistence within a unidirectional architecture! The tutorial is now set up for the next step, adding networking.

54 Advanced Unidirectional Architecture: Demo 3

By René Cacheaux

In this demo, you will incorporate pull to refresh into the Redux architecture from Demo 2.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with page **10. Putting it All Together** in **Demo 3** playground in the starter workspace, **Demo3/starter/Koober/App/KooberApp.xcworkspace**. The theory behind this demo is explained in the playground's pages 1 - 9.

If the playground fails to build and run, open the playground's overall **Source** directory and open **ReSwiftExtensions/ReSwiftRxSwift/RxStoreSubscriber.swift**. Then delete the `import RxSwift` line, save, and add the import back in. Go back to the playground, it should now build and run.

1) Use Case Factory

Add the following stored property to `SettingsViewController`:

```
let loadUserProfileUseCaseFactory: LoadUserProfileUseCaseFactory
```

2) Use Case Factory Injection

In `SettingsViewController`'s initializer, add a parameter for `loadUserProfileUseCaseFactory` and set the corresponding stored property:

```
init(stateObservable:
       Observable<SettingsViewState>,
     loadUserProfileUseCaseFactory:
       LoadUserProfileUseCaseFactory) //Add.
{
  self.stateObservable =
    stateObservable
  self.loadUserProfileUseCaseFactory =
    loadUserProfileUseCaseFactory  //Add.
  super.init()
}
```

3) Implement IX Responder

Add the following extension to `SettingsViewController`:

```
extension SettingsViewController: SettingsIXResponder {
  func loadUserProfile() {
    let useCase =
      loadUserProfileUseCaseFactory
        .makeLoadUserProfileUseCase()
    useCase.start()
  }
}
```

4) Inject IX Responder

Update `SettingsViewController`'s `loadView()` method:

```
override func loadView() {
  view = SettingsRootView(ixResponder: self)
}
```

5) Remote API Factory

That's it for the `SettingsViewController`. Now, in `DependencyContainer` add the following factory method:

```
func makeUserRemoteAPI() -> UserRemoteAPI {
  return KooberUserRemoteAPI()
}
```

6) Use Case Factory

Add the following factory method to `DependencyContainer`:

```
func makeLoadUserProfileUseCase() -> UseCase {
  let remoteAPI = makeUserRemoteAPI()
  return LoadUserProfileUseCase(remoteAPI: remoteAPI,
                                reduxStore: reduxStore)
}
```

7) Update VC Factory

Update `DependencyContainer`'s `SettingsViewController` factory as follows:

```
func makeSettingsViewController() -> UIViewController {
  let observable = makeSettingsViewStateObservable()
  return
    SettingsViewController(stateObservable: observable,
                           loadUserProfileUseCaseFactory: self)
}
```

8) That's it!

Congrats, at this time you should have a good understanding of incorporating networking side effects and user interaction into a unidirectional architecture!

55

Advanced Unidirectional Architecture: Demo 4

By René Cacheaux

In this demo, you will add user interaction to the Redux architecture from Demo 3.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with page **8. Putting it All Together** in **Demo 4** playground in the starter workspace, **Demo4/starter/Koober/App/KooberApp.xcworkspace**.

If the playground fails to build and run, open the playground's overall **Source** directory and open **ReSwiftExtensions/ReSwiftRxSwift/RxStoreSubscriber.swift**. Then delete the `import RxSwift` line, save, and add the import back in. Go back to the playground, it should now build and run.

1) Use Case Factory property

In `SettingsViewController`, add the following stored property:

```
let updateClawedUseCaseFactory: UpdateClawedUseCaseFactory
```

2) Update Initializer

Update `SettingsViewController`'s initializer by adding a parameter for `updateClawedUseCaseFactory` and set the corresponding stored property inside the initializer:

```
init(stateObservable:
        Observable<SettingsViewState>,
      loadUserProfileUseCaseFactory:
        LoadUserProfileUseCaseFactory,
      updateClawedUseCaseFactory:
        UpdateClawedUseCaseFactory) //Add.
  {
    self.stateObservable =
      stateObservable
    self.loadUserProfileUseCaseFactory =
      loadUserProfileUseCaseFactory
    self.updateClawedUseCaseFactory =
      updateClawedUseCaseFactory //Add.
    super.init()
  }
```

3) Add Update Method

Add the following method in `SettingsViewController`'s `IXResponder` extension:

```
  public func update(clawed: Bool) {
    let useCase =
      updateClawedUseCaseFactory
        .makeUpdateClawedUseCase(clawed: clawed)
    useCase.start()
  }
```

4) Add Use Case Factory

That's it for `SettingsViewController`. In `DependencyContainer`, add the following factory method:

```
  func makeUpdateClawedUseCase(clawed: Bool) -> UseCase {
    let userRemoteAPI = makeUserRemoteAPI()
    return UpdateClawedUseCase(clawed: clawed,
                               remoteAPI: userRemoteAPI,
                               reduxStore: reduxStore)
  }
```

5) Update VC Instantiation

Update the following `SettingsViewController` instantiation in `DependencyContainer'sn makeSettingsViewController()` method:

```
  return
    SettingsViewController(stateObservable: observable,
                           loadUserProfileUseCaseFactory: self,
                              updateClawedUseCaseFactory: self)
```

6) That's it!

Congrats, at this time you should have a good understanding of integrating user interaction into a unidirectional architecture!

Conclusion

We hope you enjoyed the RWDevCon 2018 Tutorial Video Vault!

We also hope that our team's passion for iOS, Swift, Android, and Kotlin development has spread to you, and that you can take what you've learned here and put it into practice.

And thanks for connecting with us! As Tammy said in the keynote, “we are all better together.” We hope to see you at the next RWDevCon!

— Ray, Vicki, and the entire RWDevCon Team