DEVCON

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Table of Contents: Overview

Table of Contents: Extended

Prerequisites

Some tutorials and workshops at RWDevCon 2018 have prerequisites.

If you plan on attending any of these tutorials or workshops, **be sure to complete the steps below before attending the tutorial**.

If you don't, you might not be able to follow along with those tutorials.

> **Note**: All talks require **Xcode 9.2** installed, unless specified otherwise (such as in the ARKit tutorial and workshop).

4: Cloning Netflix

You will need:

- A raywenderlich.com account
- A reasonable network connection (conference WiFi recommended)
- To keep the top-level "shared" directory in the same relative location to the three starter projects

4: Cloning Netflix

A huge number of popular iOS apps are just front-end chrome built on top of some kind of back-end service, provided via a network API. Here at Razeware, we've been working on an app that fits that model perfectly—*emitron*.

emitron is an iOS app that serves as the front-end to *betamax*—our videos service, and as such uses the *betamax* API to obtain up-to-date details about the content available, and to allow playback of screencasts, courses and to track individual users' progress.

In this session we'll lift the lid on some of the approaches that we took to build *emitron*, and you'll leave with an understanding of the following topics:

- Using a 3rd-party tool (Moya) to abstract a RESTful API away to a wonderful interface.
- Using Swift Codable to interact with a JSON API.
- Using Realm as a local cache for JSON API responses.
- Making Realm objects play well with Codable.
- Building a sync-engine around Realm objects to ensure the remote API is consistent with the local cache.

As you've no doubt spotted, this session relies heavily on 3rd-party frameworks. A little knowledge of Realm might be useful, but is by no means required. Come with an open mind—leave your pseudo-religious objections to 3rd-party frameworks at the door.

Important Note

This project uses 3rd-party dependencies extensively. Rather than spend 20 minutes waiting for the Swift compiler during the session, you've been provided with pre-compiled binary libraries. To use these binaries it is imperative that you use the same version of the Swift compiler (i.e. Xcode) as was used to build them.

This means that this session is locked at **Xcode 9.3 (9E145)**. If you attempt to run the code with a different version of Xcode, it will not be able to link against the pre-compiled binaries.

If you need to use a different version of Xcode, then you can recompile the binaries using Carthage.

10

Cloning Netflix: Demo 1

by Sam Davies

In this demo, you will learn how to use a popular 3rd-party framework to model a RESTful API in Swift.

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 the starter project in **Demo1/starter**. Be sure to open the workspace named **BetamaxAPI.xcworkspace**.

Note: This project relies on pre-built binaries, so it is imperative that you use the version of Xcode that was used to compile them. This is **Xcode 9.3 (9E145)**.

1) Use Moya to download a list of all raywenderlich.com screencasts.

Open **BetamaxAPI.xcworkspace**, select **CarthageDependencies** from the scheme drop down, and then **clean (⌘⇧K)** and **build (⌘B)**. This will copy the pre-built binary frameworks into a location that the playground can use.

Note: Remember to check that you are using the correct version of Xcode, as specified above.

Open **BetamaxAPI.playground** from within the workspace, and add the following enum definition to the bottom:

```
enum BetamaxAPI {
  case showScreencasts
  case showVideo(id: Int)
}
```

This enum specifies the "shape" of the API—i.e. the list of endpoints you're modelling.

Add an extension on this enum, specifying conformance to the `TargetType` protocol:

```
extension BetamaxAPI: TargetType {
  // ...
}
```

The Swift compiler will inform you that `BetamaxAPI` does not conform to the `TargetType` protocol. Use the auto-fix functionality to create protocol stubs. This shows you all the methods and properties that you need to fill in.

Make the `BetamaxAPI` extension match the following:

```
extension BetamaxAPI: TargetType {
  var baseURL: URL {
    return URL(string: "https://videos.raywenderlich.com/api/v1")!
  }

  var path: String {
    switch self {
    case .showScreencasts:
      return "/videos"
    case .showVideo(id: let id):
      return "/videos/\(id)"
    }
  }

  var method: Method {
    return .get
  }

  var sampleData: Data {
    return Data()
  }

  var task: Task {
    switch self {
    case .showScreencasts:
      return .requestParameters(parameters: ["format" : "screencast"],
encoding: URLEncoding.queryString)
    case .showVideo:
      return .requestPlain
    }
  }

  var headers: [String : String]? {
    return ["Content-type" : "application/json"]
  }
}
```

That's all the API modelling you need to do—next up is actually using it.

Create an API provider as follows:

```
let provider = MoyaProvider<BetamaxAPI>()
```

And finally, use the `.showScreencasts` endpoint to request a list of all raywenderlich.com screencasts:

```
provider.request(.showScreencasts) { (result) in
  switch result {
  case .success(let response):
    print(response.statusCode)
    print(String(bytes: response.data, encoding: .utf8)!)
  case .failure(let error):
    print(error)
  }
}
```

Hit the triangular play button below the code pane to execute the request.

You'll come across a problem—you haven't authenticated against the API, so it is refusing to serve you any data. You'll address that in the next section.

2) Authenticating against the API

The betamax API uses the standard bearer authentication approach—each request you send should include an `Authorization` header, with the content `Token <YOUR TOKEN HERE>`.

As you'll see, implementing this with Moya is easy, however, first you need to get hold of your authentication token:

1. Open your browser and visit https://videos.raywenderlich.com/
2. Login or signup for a free account.
3. Navigate to https://accounts.raywenderlich.com/debug
4. Copy the **User ID** and **Auth Token** from your browser into the playground, just above the `provider` definition:

```
let userId = "<USER ID>"
let authToken = "<AUTH TOKEN>"
```

Add a new extension to `BetamaxAPI`, specifying conformance to the `AccessTokenAuthorizable` protocol:

```
extension BetamaxAPI: AccessTokenAuthorizable {
  var authorizationType: AuthorizationType {
    return .bearer
  }
}
```

Finally, update the definition of the `provider` to use the `AccessTokenPlugin` to inject your auth token into each request:

```
let provider = MoyaProvider<BetamaxAPI>(
  plugins: [AccessTokenPlugin(tokenClosure: authToken)]
)
```

Run the playground again, and you'll see a long list of all the raywenderlich.com screencasts.

3) Download details on a specific screencast

Now that you've built the API, it's incredibly easy to switch to a different endpoint.

Update the first line of the request to match the following:

```
provider.request(.showVideo(id: 1419)) { (result) in
```

Run the playground again, and you'll then see details specific to a particular screencast.

4) Requesting user progress

An important part of a video streaming app is the ability to track each user's progress through the videos. The betamax API provides this functionality through the `/user/<id>/viewings` endpoint. Let's add the ability to query and update this endpoint:

Add the following cases to the `BetamaxAPI` enum:

```
case getProgress(userId: String)
case updateProgress(userId: String, progress: ProgressParams)
```

`ProgressParams` is a struct defined in the playground sources.

The compiler will now complain that you have some non-exhaustive `switch` statements—go through and fix them. First up, in the `path` property, add the following `case`:

```
case .getProgress(userId: let userId),
     .updateProgress(userId: let userId, progress: _):
```

```
    return "/users/\(userId)/viewings"
```

Next, replace the `method` property with the following:

```
  var method: Method {
    switch self {
    case .updateProgress:
      return .post
    default:
      return .get
    }
  }
```

Finally, update the final two cases of the `task` property:

```
  case .showVideo, .getProgress:
    return .requestPlain
  case .updateProgress(userId: _, progress: let progress):
    return .requestJSONEncodable(progress)
```

You can then update the request line to request progress for your user:

```
  provider.request(.getProgress(userId: userId)) { (result) in
```

If you've ever watched any videos on raywenderlich.com then you'll see a list of your progress through them. If not, you'll receive an empty response.

5) Decoding the response

The progress response is an array of objects that can be decoded in to `ProgressParams` structs. Moya provides a nice way of doing this.

Update the `.success` branch of the `result` switch to the following:

```
  print(response.statusCode)
  //print(String(bytes: response.data, encoding: .utf8)!)
  let progress = try! response.map([ProgressParams].self, atKeyPath:
  "viewings")
  print(progress.filter { $0.videoId == 471 })
```

This uses the fact that `ProgressParams` is `Decodable` to map the JSON response into an array of `ProgressParams` structs.

We then filter the array to search for your progress through a particular video.

6) Updating the progress for a video

All of your API requests so far have been `GET` requests, but updating the progress for a video requires a `POST` request. Because of the way in which Moya abstracts the API modelling away from you, the request is as simple as a `GET` request.

Add the following just before the existing moya request:

```
let updatedProgress = ProgressParams(videoId: 471, time: 440, duration:
447, finished: true)

provider.request(.updateProgress(userId: userId, progress:
updatedProgress)) { (result) in
  switch result {
  case .success(let response):
    print(response.statusCode)
  case .failure(let error):
    print(error)
  }
}
```

This creates a new `ProgressParams` struct that we want to persist to the API, and then makes the request.

Run the playground to make the request, and see the value of the response.

7) That's it!

That concludes the whirlwind tour of Moya, and how the abstraction over APIs is really nice.

In the next part of the session you'll discover how you can use Moya with Realm and Codable to form the basis of your apps data layer.

11

Cloning Netflix: Demo 2

by Sam Davies

In this demo, you will discover how to use Realm, Moya and Swift Codable in sweet harmony.

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 the starter project in **Demo2/starter**.

Note: This project relies on pre-built binaries, so it is imperative that you use the version of Xcode that was used to compile them. This is **Xcode 9.3 (9E145)**.

1) Using a custom JSON decoder with Moya

If you build and run the app in the starter directory, once you've logged in you'll be presented with a blank screen, and some errors in the console.

These are due to the fact that the default JSON decoder used by Moya is unable to parse the JSON it has received from the betamax API.

Open **Services/Betamax.swift** and locate the private lazy var `jsonDecoder`. Update the definition to match the following:

```
let dateFormat = "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'"
let decoder = JSONDecoder()
let dateFormatter = DateFormatter()
dateFormatter.dateFormat = dateFormat
decoder.dateDecodingStrategy = .formatted(dateFormatter)
return decoder
```

This creates a custom date formatter that can parse the dates provided by the JSON API, and assigns it to the appropriate property on the `JSONDecoder`.

Moya is already configured to use this decoder in most places, but there is one left for you to fix. Find the "TODO: Update to use custom JSON decoder" comment in the `getScreencasts()` method, and update the `screencasts` line to match the following:

```
let screencasts = try response.map([Video].self, atKeyPath: "videos",
using: self.jsonDecoder)
```

This does 2 things:

1. Tells Moya to use the custom JSON decoder you just configured.
2. Sets the key path for the start of the data.

Build and run again, and you'll see the courses page populate itself, having successfully parsed the downloaded JSON.

2) Customising Decodable behaviour with Realm objects

Navigate into one of the courses displayed in emitron, and you'll see its details. However, at the bottom of the screen, there's a big space where the individual videos should be.

This is caused by a problem in the JSON decoding. The API supplies details of videos as an array in the `videos` array, but the `Collection` model is not currently attempting to decode this property.

Open **Models/Collection.swift** to find the model object that represents a video course. This is a subclass of `Content`, which contains properties shared between courses and videos. The `Course` object conforms to `Decodable`, but obviously does not handle properties that only exist in the `Collection` subclass.

Replace the "// TODO: Add Decodable conformance" comment with the following:

```
private enum CodingKeys: String, CodingKey {
  case videos
  case videoCount
}
```

This defines an enum that the Swift compiler uses to generate the code required for the `Decodable` protocol. However, since this is a subclass of Realm's `Object`, the compiler is unable to auto-generate the required code.

Add the following require initialiser below the enum definition:

```
required init(from decoder: Decoder) throws {
  let container = try decoder.container(keyedBy: CodingKeys.self)
  videoCount = try container.decode(Int.self, forKey: .videoCount)
  if let videos = try? container.decode([Video].self, forKey: .videos) {
    self.videos.append(objectsIn: videos)
  }

  try super.init(from: decoder)
}
```

This first decodes the `videoCount` property as an `Int`. Decoding the `videos` property is a bit different since this is a Realm object—where the collection of videos is represented as `List<Video>` instead of `[Video]`. You attempt to decode them, and if successful, add them to the Realm list. Finally, it defers to the equivalent initialiser on `Content`

The compiler will now be complaining that you've implemented one required initialiser, but not all of them. Use the "Fix" tooltip to create the three missing initialisers, and update them to match the following:

```
required init() {
  super.init()
}

required init(realm: RLMRealm, schema: RLMObjectSchema) {
  super.init(realm: realm, schema: schema)
}

required init(value: Any, schema: RLMSchema) {
  super.init(value: value, schema: schema)
}
```

These are inherited from the Realm `Object` class. For reference, overriding these requires inclusion of the `Realm` framework, not just `RealmSwift`.

Build and run emitron again, and click on a course. You'll now see the videos collection view at the bottom of the screen populated with the videos that make up that collection.

3) Realm's reactive paradigm

Click onto the **Screencasts** tab and you'll see that the screencasts aren't displayed. They are being downloaded from the API, but not displayed.

This is an easy fix—open **Controllers/ScreencastsViewController.swift** and find the `viewDidLoad()` method.

Replace the "// TODO: Add screencasts observer" comment with the following:

```
changeToken = videos.observe { [weak self] (change) in
```

```
    self?.collectionView.reloadData()
  }
```

The `videos` property is a lazy loaded Realm `Results<Video>` object using the `getScreencasts()` method on the `Betamax` store. Even though this will initially be empty, once the API call has succeeded, and the screencast video objects are saved into Realm, this `observe` callback will be invoked, which in turn reloads the collection view.

Build and run, and select the Screencasts tab. You'll now see a list of all the screencasts available on raywenderlich.com.

4) That's it!

Nice work. You now have an understanding of how you can use Moya, Realm and Swift Codable to build a reactive data layer with ease. Next up, you're going to see how you can expand on this knowledge to build a local cache that auto-syncs with the remote API.

Cloning Netflix: Demo 3

by Sam Davies

In this demo, you will discover how to build a local cache using Realm, which automatically synchronises to the web API.

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 the starter project in **Demo3/starter**.

> **Note:** This project relies on pre-built binaries, so it is imperative that you use the version of Xcode that was used to compile them. This is **Xcode 9.3 (9E145)**.

1) Update progress bars

The UI already contains progress bars for each of the videos, which will update as the appropriate `Viewing` objects update. However, these objects aren't updating as the user watches the video.

Open **Services/VideoPlayer.swift** and in `playStream(_:with:)` update the "TODO: Add a time observer" comment to match the following:

```
let interval = CMTime(seconds: 5, preferredTimescale:
CMTimeScale(NSEC_PER_SEC))
timeObservationToken = player.addPeriodicTimeObserver(forInterval:
interval, queue: DispatchQueue.main) { [weak self] time in
  self?.store.updateProgressFor(viewing: viewing, time:
Int(time.seconds))
}
```

This adds a periodic observer to the `AVPlayer`, which uses the `updateProgressFor(viewing:time:)` method provided by the `Betamax` store.

Further up in the same method, replace the "TODO: Remove observer" comment with the following:

```
player.removeTimeObserver(timeObservationToken)
self.timeObservationToken = .none
```

This ensures that if there is an existing player, we cancel notifications when the video stream changes.

Finally, you need to start the video at the most recently viewed point. Update the "TODO: Jump to the correct start point" comment to match the following:

```
let startPoint = CMTime(seconds: Double(viewing.time),
preferredTimescale: CMTimeScale(NSEC_PER_SEC))
player.seek(to: startPoint)
```

Now, build and run the app. Choose a video to watch and jump to point in the middle. Let it play for a few seconds, before then closing it. You'll see the progress bar for that video has updated, and when you click to play the video again it'll start from the point you exited.

2) Import progress from the betamax API

There are two parts to synchronising the local progress records with the remote API —first you'll tackle the downloading of viewing records.

Open **Services/SyncEngine.swift** and find the private `syncFromApi()` utility method. This is currently configured to request all the `Viewing` objects for the current user from the betamax API, and then loop through each of them in turn.

Locate the `// TODO` comment in the `viewings` loop and add the following conditional statement:

```
// Does a local equivalent exist?
if let localViewing = localViewings.filter("videoId == %@",
viewing.videoId).first {
  // TODO
} else {
  // TODO
}
```

This takes the current remote `Viewing` and tries to find a local equivalent (i.e. has the same `videoId`).

Add the following to the first branch of this conditional:

```
// If the local one is dirty, keep the most recent
if localViewing.dirty && localViewing.updatedAt > viewing.updatedAt {
  // NO-OP—it'll get synced to the API later
} else {
  // TODO
}
```

Here you're checking whether or not the local viewing has been marked as `dirty`—i.e., it has been updated locally. If it is dirty, and has a more recent update time than the one received from the API then we just ignore it—the local one will be automatically synced to the remote API at some point in the future.

Add the following to the `else` branch you just created:

```
// Update the local one with the details from the remote on
localViewing.updateFrom(remote: viewing)
```

This uses a utility method on `Viewing` to update the attributes on the local object to match those from the API—i.e., to sync the remote progress to the local version.

Finally, jump out a level to the `else` branch of the first conditional you created. Add the following:

```
// Not seem this before—make a local version
realm.add(viewing)
```

This covers the case that there is no local equivalent to the `Viewing` downloaded from the API. In this situation we add the object to the local realm.

Build and run the app, and you should see the progress bars displayed on videos updating to match the progress through videos you've watched on raywenderlich.com. Check it's working by watching a video on the website, and then restarting the app. Your progress will be sync'ed from the API to the app.

3) Syncing local changes back to the API

The final part of the synchronisation puzzle is to push the local changes back to the API. Realm's reactivity is especially helpful here—you'll set up a filter that will trigger a sync each time the results change.

Open **Services/SyncEngine.swift** and find the private `configureListeners()` utility function. Replace the content with the following:

```
let result = realm.objects(Viewing.self).filter("dirty == TRUE")
dirtyNotificationToken = result.observe { (_) in
  for viewing in result {
    self.syncToApi(viewing: viewing)
  }
}
dirtyViewings = result
```

This first creates a Realm `Results` object, which is an auto-updating list of all `Viewing` objects that have been marked as dirty. It then creates an observer that loops through the viewings, passing each of them to the `syncToApi(viewing:)` method.

`syncToApi(viewing:)` already uses the appropriate method on the `Betamax` store to upload the changes to the API, and now needs to update the local version. Replace the "// TODO" comment in the `syncToApi(viewing:)` method with the following:

```
do {
  try self.realm.write {
    viewing.dirty = false
  }
} catch (let error) {
  print(error)
}
```

This marks the `Viewing` object as no-longer dirty, meaning it'll be removed from the sync engine list.

You've got one final thing to do, and that is to stop the automatic sync process in the `stopSync()` method.

Replace the "// TODO" comment in the `stopSync()` method with the following:

```
dirtyNotificationToken?.invalidate()
dirtyNotificationToken = .none
```

You can now choose a video in the app, watch it for a while before then navigating to that same video in a browser. When you play the video on the site, you'll see that it starts from the same place you left off in the app.

4) That's it!

With that you're done. You've discovered how you can use Realm to create a local API cache, and to automatically sync that cache back to the remote API.