DEVCON

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Notice of Rights

Notice of Liability

Trademarks

Table of Contents: Overview

Table of Contents: Extended

3: Architecting Modules

Modularity in code goes hand-in-hand with readability, testablity, reusability, scalability, and reliability, along with many other "ilities." The unit of modularity for a swift app is, obviously the module. This session covers how to make modules, figure out what code goes in a module, and then how to use those modules for maximum success. You'll learn best practices in code encapsulation and reuse, learn some programming buzz words, and level up your Xcode skills.

Architecting Modules: Demo 1

By Michael Katz

In this demo, you will split up an app into modules. For Illustrative purposes, we'll separate out the data layer. This gives a nice chunk that can be reused in future apps.

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**.

1) Create a new framework

1. In the project editor, click the "+" to add a new target.
2. Choose iOS > Framework & Library > Cocoa Touch Framework
3. Click Next
4. For **Product Name** type `TrainService`.
5. Make sure **Include Unit Tests** is selected.
6. Click Finish.

2) Move files over

Grab the following files and drag them to the new **TrainService** folder.

- API.swift
- Model.swift

- LinesResponses.swift
- TrainLine.swift
- TrainLineGeography.swift
- LineSchedule.swift
- Wallet.swift
- Ticket.swift
- Also the contents **Result folder** (don't grab the folder itself)

3) Fix imports

In

- **AppDelegate.swift**
- **MapViewController.swift**
- **ScheduleViewController.swift**
- **TicketsViewController.swift**
- **TicketViewController.swift**
- **PurchasedTicketTableViewController.swift**
- **LoginViewController.swift**
- **TrainLine+UIKit.swift**

add:

```
import TrainService
```

4) Fix access

In **TrainLine.swift**, first make the class `public`. Then also make its properties `public`.

Then in **Model.swift**, make the `var lines` `public` as well as `func line(forId:)` and `func buyTicket(line:completion:)`.

Build and run and nothing should have changed!

Congrats, all your hard work has had no effect at all! Don't worry, it'll pay off with cleaner, more modular code, even if the user never sees it.

5) Encapsulate Functionality

Separate out the train schedule functionality from ticketing functionality.

In **API.swift** change

```
extension API {
```

to

```
class UserAPI {
```

Next, separate out the model along the same functional lines.

6) Encapsulate the User Model

In **Model.swift**, change

```
extension Model {
```

to

```
public class UserModel {
```

This creates a new public class that models the user-specific data.

Then from `Model` move

```
public var wallet: Wallet?
```

to `UserModel`.

Next, add following property to the new class:

```
private let api = UserAPI()
public var wallet: Wallet?
```

Finally, add the following `init` so the class can be used by the application:

```
public init() {}
```

7) Fix the build errors

Fortunately this is a simple refactor, since the logic was already pretty neatly segregated.

In **Appdelegate.swift** add a new singleton for the user model:

```
static let sharedUserModel = UserModel()
```

Then in, **LoginViewController.swift**, **PurchasedTicketTableViewController.swift** and **TicketsViewController.swift** update `sharedModel` to `sharedUserModel`.

8) Add Unit Tests

Unit tests help define and enforce a module's API contract.

Add a new Unit Test file to `TrainServiceTests` target. Name it **ModelTests**.

Add the following to the imports at the top:

```
@testable import TrainService
```

Then, add the following test method:

```
func testLineForId() {
  let model = Model()

  let testLine = TrainLine(lineId: 300, color: "yellow", name: "Test
Line", fare: 30.0)
  model.lines.append(testLine)

  let expectedLine = model.line(forId: 300)!

  XCTAssertEqual(expectedLine.name, testLine.name)
}
```

This shows that `lineForId` will fetch a known train line.

9) Test the stack

Let's add a more interesting test.

Add a new Unit Test file to `TrainServiceTests` target. Name it **TicketTests**.

Add the following to the imports at the top:

```
@testable import TrainService
```

Then, add the following test method:

```
func testPurchase() {
  let api = UserAPI()
  let testWallet = Wallet(username: "Tester Jones",
                          balance: 100.0,
                          tickets: [])

  let wait = expectation(description: "buy a ticket")

  api.buyTicket(lineId: 10,
                cost: 30,
                wallet: testWallet) { result in
    let newWallet = result.value!
    XCTAssertEqual(newWallet.balance, 70)

    let ticket = newWallet.tickets[0]
    XCTAssertEqual(ticket.cost, 30)
    XCTAssertFalse(ticket.activated)

    wait.fulfill()
  }

  waitForExpectations(timeout: 1, handler: nil)
}
```

10) That's it!

For a further challenge on your own, refactor `API` and `Model` to `TrainLineAPI` and `TrainLineModel`. Then separate these into separate frameworks.

Congrats, at this time you should have a good understanding of adding a new module and testing it! It's time to move onto what else you can get from a module.

8 Architecting Modules: Demo 2

By Michael Katz

In this demo, you will explore some of the benefits of modularization.

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**. This project has been improved by splitting up the code into a info framework and a user-centric framework, as suggested by the challenge in Demo 1.

First up, explore reusability by using the `InfoService` framework in both the main app and an Today app extension. The extension will show the next train to leave the station.

1) Create a "Next Up" Today Extension

1. In the project editor, click the **+** button under the *Targets* list.
2. In the popup, select **Today Extension** and click **Next**.
3. Name the extension `NextTrain` and click **Finish**.
4. Click **Activate** if Xcode asks to activate the "NextTrain" scheme. (This makes it easy to launch the simulator running the widget)

2) Build the widget UI

Since this is not a Today Extension tutorial, the widget itself will be very simple.

1. Open **MainInterface.storyboard** in the NextTrain folder.

2. Open the assistant editor, make sure it is showing **TodayViewController.swift**
3. Ctrl+drag the **Hello World** label on the code.
4. Name the outlet `label` and click **Connect**.

3) Make use of the framework

In **TodayViewController.swift** add to the `import` statements at the top:

```
import InfoService
```

Then add the following variables to the class:

```
let model = Model()
var nextRun: (TrainLine, LineSchedule.Run)? = nil
```

Add the following methods to find the next departing train and display it in the widget:

```
func updateLabel() {
  if let (line, run) = nextRun {
    let formatter = DateFormatter()
    formatter.dateStyle = .none
    formatter.timeStyle = .short
    let time = formatter.string(from: run.departs)
    let train = line.name
    self.label.text = "Next train: \(train) (\(run.train)): @ \(time)"
  } else {
    self.label.text = "No train information at this time."
  }
}
```

```
func updateOnMain() {
  DispatchQueue.main.async {
    self.updateLabel()
  }
}
```

```
func checkRunIfItsNext(_ run: LineSchedule.Run, line: TrainLine) {
  guard run.departs > Date() else { return }
  guard let latestRun = self.nextRun else {
    self.nextRun = (line, run)
    self.updateOnMain()
    return
  }

  if run.departs < latestRun.1.departs {
    self.nextRun = (line, run)
    self.updateOnMain()
  }
}
```

```
override func viewWillAppear(_ animated: Bool) {
  super.viewWillAppear(animated)
  model.lines.forEach { line in
    model.schedule(forId: line.lineId, completion: { schedule in
      schedule.schedule.forEach { run in
        self.checkRunIfItsNext(run, line: line)
      }
    })
  }
}
```

4) Fix some issues

Before being ready to build and run, a few fixes need to happen.

- In the project navigator, select the three `json` files under **TrainTime**.
- Then, in the *File Inspector*, click the check box next to **NextTrain** to add them to the extension target.

In a normal project, the data would come from a server, but here the data is loaded from included files. To access shared resources, they must be either duplicated between the main app and extension, or shared through a common container (out of scope for this tutorial).

Next, open the **InfoService** target in the project editor. Under *Deployment Info* click the **App Extensions** checkbox to enable "Allow app extension API only". Do the same for the **Helpers** framework as well.

This will silence the warning about including an unsafe framework in the extension.

Finally, the frameworks need to be linked against the extension.

1. Open the Project Editor and select the **NextTrain** target.
2. Under *Linked Frameworks and Libraries*, click the **+** button to add new frameworks.
3. Select Helpers.framework, and InfoService.framework and click **Add**.

5) Build and run

Build run the **NextTrain** scheme. The simulator should launch to the Today screen, showing the next departing train.

> The simulator is finicky. If the widget fails to load, you may have to delete the app from the simulator and run again.

Now the app makes use of the same module, `InfoService` across two "apps". Re-use achievement unlocked!

6) Take a look at launch times

Modularization isn't all sunshine and roses. One side effect of breaking up an app into many frameworks is that each dynamic framework adds a cost to the startup time of the app as the libraries have been loaded and addressed. One way to mitigate that is to use static libraries.

If you skipped the previous part, you can launch with the `intermediate` project.

To inspect launch times, open the **TrainTime** scheme.

Under **Run > Arguments > Environment Variables** add a new variable. Set the **name** to `DYLD_PRINT_STATISTICS` and **Value** to `1`.

Build and run the app.

After the app launches you should see statics like the following in the console:

```
Total pre-main time: 1.2 seconds (100.0%)
         dylib loading time: 187.18 milliseconds (14.4%)
        rebase/binding time: 997.14 milliseconds (77.2%)
            ObjC setup time:  54.70 milliseconds (4.2%)
           initializer time:  52.12 milliseconds (4.0%)
           slowest intializers :
               libSystem.dylib :  15.57 milliseconds (1.2%)
```

One way to knock down the `dylib loading time` is by replacing dynamic libraries with static ones.

7) Create a static User Service Library

You'll next convert the dynamic frameworks to static libraries, one-by-one.

1. In the Project Editor, create a new target.
2. Select **Cocoa Touch Static Library** under **iOS > Framework & Library**.
3. Name it `UserServiceStatic` and click **Finish**.

This will create a new target for the library. Next we have to move the project over to it.

4. Select all the swift files in **UserService**. Drag them to the **UserServiceStatic** folder in Project navigator. This will add all the same files to the new library. Double-check that the **Target Membership** in the file inspector, has changed.

5. Back in the project editor, delete the `UserService` target.
6. Select **UserServiceStatic** target, and in the **Build Phases** tab, add a **Target Dependencies** to the `Helpers` framework.
7. Select the **TrainTime** target, and in **General > Linked Frameworks and Libraries** add `libUserServiceStatic.a`.
8. In **AppDelegate.swift**, **TicketViewController.swift**, **TicketsViewController.swift**, and **PurchasedTicketTableViewController.swift** change `import UserService` to `import UserServiceStatic`.
9. Also in **AppDelegate.swift**, update this line: `static let sharedUserModel = UserModel(api: UserServiceStatic.API())`.

Clean and Build and run. Since the launch times vary significantly between individual launches and because these frameworks are so small and contain very few symbols, it's unlikely you'll actually notice a difference, if any. This work only pays off when there are a large number of frameworks or if the frameworks contain many classes/functions.

> If there are any build errors, try following the error message for the issue, there may be a missing update to an `import`. Also try clearing `Derived Data` or deleting the app from the simulator, or device.

8) Create a static Info Service library

Steps #8 and #9 repeats this conversion to static library process for the info service and then for the helpers library.

1. Create a new static library target and call it `InfoServiceStatic`.
2. Move the **InfoService** files over to **InfoServiceStatic** and delete the old target.
3. Link `libInfoServiceStatic.a` to both the **TrainTime** and **NextTrain** targets.
4. Add the `Helpers` framework to the dependencies of **InfoServiceStatic**.
5. In **TodayViewController.swift** under **NextTrain** update the `import InfoService` statement to `import InfoServiceStatic`.
6. Do the same in the main target for: **AppDelegate.swift**, **LoginViewController.swift**, **MapViewController.swift**, **TicketsViewController.swift**, **ScheduleViewController.swift**, and **TrainLine+UIKit.swift**.

Clean and Build and run. If done properly, nothing should change to the user.

9) Create a static Helpers library

Things start to get a little more hairy when making static libraries out of a whole dependency chain.

Start off the same way by:

1. Create a new static library target and call it `HelpersStatic`.
2. Move the **Helpers** files over to **HelpersStatic** and delete the old target.
3. In the **InfoServiceStatic** target, add a **Target Dependencies** for `HelpersStatic`.
4. Change the import at the top of **API.swift** and **Model.swift** to `import HelpersStatic`.
5. Then add the `HelpersStatic` dependency for the **UserServiceStatic** target.
6. Change the import at the top of **API.swift** to `import HelpersStatic`.
7. Finally to satisfy the transitive dependency and make sure the `HelpersStatic` code is included in the app, open the **TrainTime** target in the project editor. Add a **Linked Frameworks and Libraries** for `libHelpersStatic.a`.
8. Do the same for the **NextTrain** target as well.

Build and run again. The functionality should be unchanged. Once again since these libraries are so small, it's unlikely to make a difference for this app's load times.

10) That's it!

Congrats, at this time you should have a good understanding of reusability and static libraries! Take a break, and then it's time to move onto architecting around 3rd party modules.

If you're looking for a challenge, fix up the test target so it now builds and tests using the static libraries.

9

Architecting Modules: Demo 3

By Michael Katz

In this demo, you'll use modules to create an insulating layer around a third-party local persistence library. The app uses a separate (potentially 3rd party) library called `Clio` for local persistence, this will be hidden behind a facade, so another library can be substituted.

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**. The elves have been busy since the last demo, and **TrainTime** now has local persistence.

1) What's Changed

Check out `Model.swift` in the **InfoService** folder. This now uses the **ClioDB** framework to provide local caching of the "server" data.

2) Create a new facade framework:

1. In the Project Editor, add a new target.
2. Choose 'iOS > Framework & Library > Cocoa Touch Framework'
3. Name it `PersistenceFacade`
4. Click Finish

This facade will be the insulating layer between the persistence implementation and the services layer.

3) Create the facade class:

Add a new **Swift File** to the `PersistenceFacade` framework called `Persistence.swift`.

Open, `Persistence.swift` and replace its contents with the following:

```
import Foundation

public class Persistence {

}
```

This class will provide the facade to the "3rd-party" persistence framework.

4) Add the bridge

In order to separate the individual APIs of different libraries, create a protocol to abstract the persistence implementation by functionality. Put the following code at the top of `Persistence.swift` below the `import Foundation` statement:

```
public protocol PersistenceLayer {
  func save<T: Encodable>(key: String, items: [T])
  func load<T: Decodable>(key: String) -> [T]?
}
```

This will enable arrays of values to be saved and loaded from this layer.

5) Create the wrapper

Create a new **Swift File** inside `PersistenceFacade` named `ClioPersistenceLayer` and replace its contents with the following:

```
import Foundation
import ClioDB

public class ClioPersistenceLayer: PersistenceLayer {
  private let clioDb = ClioBase()

  public init() {
  }

  public func save<T: Encodable>(key: String, items: [T]) {
    do {
      try clioDb.saveTable(table: key, rows: items)
    } catch {
      print("experienced error saving \(error)")
    }
  }

  public func load<T: Decodable>(key: String) -> [T]?  {
    do {
      return try clioDb.loadTable(table: key)
    } catch {
      print("experienced error loading \(error)")
    }
    return nil
  }
}
```

This supplies the **ClioDB** implementation for the facade protocol.

6) Add the framework dependency

Go to the project editor, select the **PersistenceFacade** target and add `ClioDB.framework` as a **Linked Frameworks and Libraries**.

While on this screen, select the **Allow app extension API only** checkbox under **Deployment Info**, so this can be used by the NextTrain extension without warnings.

7) Wire up the facade

Open **Persistence.swift**, add the following code to `Persistence`.

```
  let implmentationLayer: PersistenceLayer

  public init(layer: PersistenceLayer) {
    self.implmentationLayer = layer
  }

  public func save<T: Encodable>(key: String, items: [T]) {
    implmentationLayer.save(key: key, items: items)
  }

  public func load<T: Decodable>(key: String) -> [T]? {
    return implmentationLayer.load(key: key)
  }
```

This forwards the requests from the client caller to the individual implementation wrappers. This is the layer that insulates the main app from the third party library. Libraries can be swapped out by changing the `PersistenceLayer` implementation independently of changing any application-level code.

8) Update the application dependencies

Open the **InfoService** target in the project editor. Add `PersistenceFacade.framework` as a **Linked Frameworks and Libraries**.

9) Update the application

Finally, the application has to be refactored to use this new module.

1. Open **Model.swift** inside the **InfoService** folder
2. Replace `import ClioDB` with `import PersistenceFacade`
3. Replace `private let db: ClioBase` with `private let db: Persistence`
4. Change `init(api:)` to the following:

```
public init(api: API = API()) {
  self.db = Persistence(layer: ClioPersistenceLayer())

  self.lines = db.load(
    key: PersistenceKeys.trainLine.rawValue) ?? []

  self.api = api
  api.loadTrainLines { [weak self] result in
    guard let strongSelf = self,
      let lines = result.value else {
      return
    }

  strongSelf.lines = lines
  strongSelf.db.save(key: PersistenceKeys.trainLine.rawValue,
    items: lines)
  }
}
```

Finally, update the rest of the usages of `db`.

In `schedule(forId:completion:)` change the `try? ...` line to

```
strongSelf.db.save(key: PersistenceKeys.schedule.rawValue,
                   items: lines)
```

Do the same for `loadGeographyFromAPI(_:)`

```
strongSelf.db.save(key: PersistenceKeys.geography.rawValue,
                   items: geography)
```

Next, change `loadGeography(completion:)` to:

```
private func loadGeography(
  completion: @escaping ([TrainLineGeography]) -> ()) {

  if geographyReloadedFromServer {
    if let geography: [TrainLineGeography] =
       self.db.load(key: PersistenceKeys.geography.rawValue) {
        completion(geography)
    } else {
      loadGeographyFromAPI(completion)
    }
  } else {
    loadGeographyFromAPI(completion)
  }
}
```

Much cleaner!

If you build and run, once again nothing has changed!

10) Swap implementations

The promise of the facade layer is to change the 3rd party dependency. There is an alternate persistence layer in the project, **Munin**.

Create a new file in **PersistenceFacade**. Name it `MuninPersistenceLayer`. This is going to look almost exactly like the Clio layer, so implement it like so:

```
import Foundation
import Munin

public class MuninPersistenceLayer: PersistenceLayer {
  let db = MuninBase()

  public init() {

  }

  public func load<T>(key: String) -> [T]? where T : Decodable {
    do {
      return try db.loadTable(key)
    } catch {
      print("error \("error")")
    }
    return nil
  }

  public func save<T>(key: String, items: [T]) where T : Encodable {
    do {
      try db.saveTable(table: key, rows: items)
    } catch {
      print("error \(error)")
    }
  }
}
```

Now the only code that has to change further up to swap implementations is in **Model.swift**. Replace the first line of `init(api:)` with:

```
self.db = Persistence(layer: MuninPersistenceLayer())
```

and now there is a new database framework powering the cache and the application code doesn't have to change!

10) That's it!

Congrats, at this time you should have a good idea on how to refactor an app to create an insulating facade layer between the app a third-part (or first-party) module.