DEVCON

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Table of Contents: Overview

Table of Contents: Extended

6: Clean Architecture on iOS

Architecture is the design of functional, safe, sustainable and aesthetically pleasing software. There are many ways to architect your applications like the common MVC, MVP and MVVM patterns. This session will get you comfortable with clean architecture and show you how to transform a basic application built using MVC to a clean architecture using VIPER that is scalable.

Clean Architecture on iOS: Demo 1

By Anthony Lockett

In the world of software engineering, there will come a time where you have to maintain software either created by yourself or someone else. Having a clean decoupled architecture will make lives for you and your collegues easier. Let's imagine you recieved a project that was created using the MVC architecture, and not very good at that. You decide it's best to fix this by making it more clean using VIPER.

In this demo, you will being your journey into the basics of clean architecture by converting and existing MVC todo list application to Clean Architechture using VIPER. You will start by creating Iteractors and Entities.

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) Creating the List Interactor Protocols

Before you get started creating the List Interactor, you should create the protocols. A protocol defines a blueprint of methods, properties and other requirements that suit a particular tasks or piece of functionality.

There are some protocols already defined for you.

Open **TodoListProtocols.swift** in the Protocols folder. Here you are going to create two protocols

`TodoListInteractorInputDelegate`

Add the following code

```
protocol TodoListInteractorInputDelegate: class {
    var dataManager: TodoListPersistenceManagerInputDelegate? { get set }

    func retrieveTodoList()
}
```

`TodoListInteractorOutputDelegate`

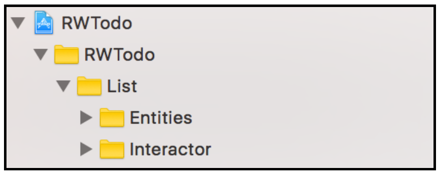
Add the following code

```
protocol TodoListInteractorOutputDelegate: class {
    func didRetrieveTodos(_ todos: [TodoItem])
    func onError()
}
```

The two protocols you defined above are used for conformance on retrieveing the todo list and what to do when the list is retrieved. They will be used in the Interactor and other classes to be defined later on.

2) Creating the List Interactor

First you should update the structure of your Xcode project. **Control-Click** on **RWTodo** and select **New Group**, call the group **List**. Continue this pattern for the following groups Entities and Interactor from their parent List.



Now that you started to get the Xcode folder structure in order create a new file inside the Interactor folder called **ListInteractor.swift**

```
// MARK: - TodoListInteractorInputDelegate

class ListInteractor: TodoListInteractorInputDelegate {

  weak var presenter: TodoListInteractorOutputDelegate?
  var dataManager: TodoListPersistenceManagerInputDelegate?

  func retrieveTodoList() {
    do {
      if let todoItems = try dataManager?.retrieveTodoList() {
        if !todoItems.isEmpty {
          presenter?.didRetrieveTodos(todoItems)
          } else {
            presenter?.didRetrieveTodos([])
          }
      }
    } catch {
      presenter?.didRetrieveTodos([])
```

```
        }
    }
}
```

Here you're creating an Interactor. The Interactor contains all of the application business logic as specified by a specific use case. Each use case is a protocol.

Now that the Interactor is created that conforms to the input protocol, create an extension that conforms to the output protocol.

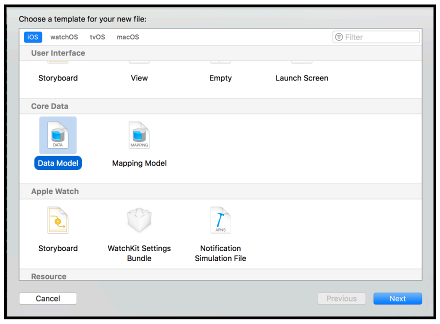
Add the following code

```
// MARK: - TodoListPersistenceManagerOutputDelegate

extension ListInteractor: TodoListPersistenceManagerOutputDelegate {

    func didRetrieveTodos(_ todos: [TodoItem]) {
        presenter?.didRetrieveTodos(todos)

        for todo in todos {
            do {
                try dataManager?.save(item: todo)
            } catch {
                presenter?.onError()
            }
        }
    }

    func onError() {
        presenter?.onError()
    }

}
```

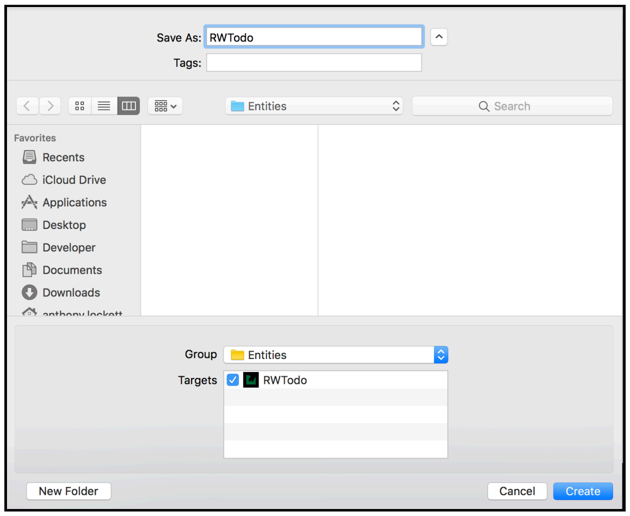
3) Creating the TodoItem Entity

The Entity are basic model objects that are used by the Interactor. In your app this is an Entity created by Core Data.

Currently if you run the application, you will notice no todo items save. You currently have a `TodoItem` model in your models folder. **Control-Click** on the `Entities` folder and select `New File`. Scroll down to the Core Data section, select the **Data Model** then select **Next**.

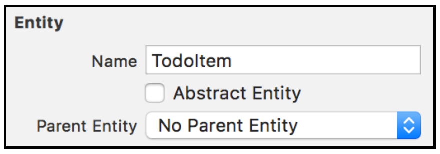


For Core Data to work, you need to name your model the same as your project, name it RWTodo then tap **Create**.


Save As: RWTodo
Tags:
Group: Entities
Targets: ✓ RWTodo

Click on the newly created **RWTodo.xcdatamodel** to open it. Click on the Add Entity button on the bottom. This will add an Entity under the Entities section on the left.

Click on the Entity, then in the Data Model Inspector on the right change the name to **TodoItem**

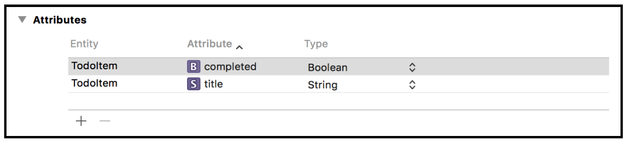

Entity
Name: TodoItem
☐ Abstract Entity
Parent Entity: No Parent Entity

Now you're ready to create the attributes for this Entity.

Under Attributes click on the **+** button. Name the attribute title and change the type to `String`.

Do this once again, but change this name to completed and the type to `Boolean`

When completed your attributes section should look like this.



Attributes

Entity	Attribute	Type
TodoItem	B completed	Boolean
TodoItem	S title	String

You're almost done setting up your Entity.

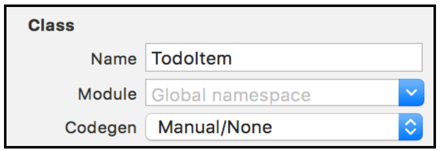
Select the **TodoItem** entity and in the Xcode menu select Editor then CreateNSManagedObject Subclass..

A new menu will appear asking which model and entity to create this from. The proper ones should already be selected.

Before the creation, click the Group dropdown and select your Entities folder. If you do not do this then Xcode will place these files outside your project. If so just drag them in :]

To finish setting up our entity, select the **RWTodo.xcdatamodel**.

Select the TodoItem entity and under the Data Model Inspector change the value of CodeGen under Class to Manual/None



Class

Name	TodoItem
Module	Global namespace
Codegen	Manual/None

You should see some errors, these are due to ambiguous models. You have two TodoItem models. GO ahead and delete the Model folder, since we no longer need it.

Currently, our project uses the older model when adding a TodoItem, we need to change a few lines of code in order to use our new Core Data models.

Open up **AddTodoItemTableViewController.swift** and inside the save() function add the following code in the else block above the `delegate.` function.

```
   let item = TodoItem(context: PersistenceService.context)
   item.title = text
   item.completed = false
```

Here you are getting the context from the PersistenceService and setting the title and completed status for the TodoItem on that context.

Now switch over to **ListViewController.swift** and inside viewWillAppear() before setupUI() is called, add the call getSavedTodos()

You also have to add the following code to **ListViewController.swift** for ther ability to get all saved todos

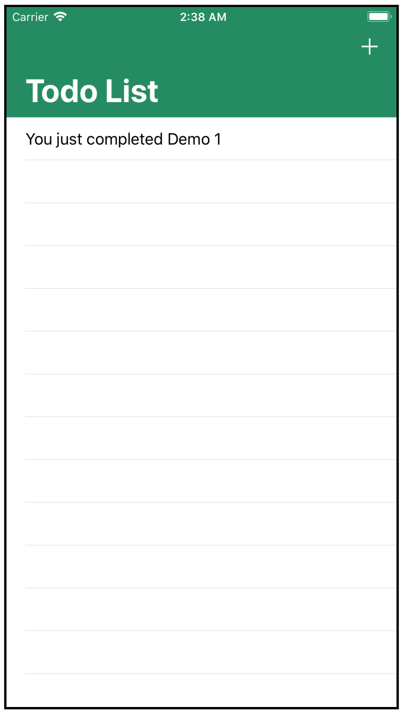
```
  private func getSavedTodos() {
    let fetchedRequest: NSFetchRequest<TodoItem> =
TodoItem.fetchRequest()
    do {
      let savedTodos = try
PersistenceService.context.fetch(fetchedRequest)
      self.items = savedTodos
    } catch {
      print("Error fetching todos")
    }
  }
```

Now you're done setting up your entity, you can start adding functionality to it. Currently your old model has a toggleCompleted() function, you need to move this to your new entity.

Open **TodoItem+CoreDataProperties.swift** and inside the function `fetchRequest()` add the following code.

```
func toggleCompleted() {
  completed = !completed
}
```

Build and Run you should now be able to add items that are saved and load in automatically when the app restarts.



You just completed Demo 1

4) That's it!

Congrats, at this time you should have a good understanding of Interactors and

Entities! It's time to move onto Presenters.

17 Clean Architecture on iOS: Demo 2

By Anthony Lockett

In this demo, you'll completely build out the P in VIPER the Presenter. Then you'll continue to expand the interactor and presenter connection.

The presenter is exaclty what it sounds like. It contains the view logic, it prepares the content that needs to be displayed. It works with the Interactor by asking for new data and presenting it what the data is ready.

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**.

1) Creating the List Presenter Protocol

You'll notice from Demo 1, there's some protocols already defined for you and some you created.

The presenter in this case requires a new protocol

Open **TodoListProtocols.swift** and add the following protocol

```
protocol TodoListPresenterDelegate: class {
  var view: TodoListDelegate? { get set }
  var interactor: TodoListInteractorInputDelegate? { get set }

  func viewDidLoad()
}
```

You declared two variables in the above protocol, the view and interactor. The view is the delegate used in the main todo list view. This delegate protocol has two methods.

1. `show(items)`
2. `error()`

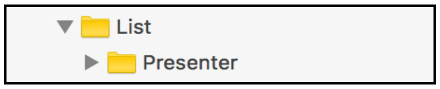
The interactor is the delegate to interface which is how the interactor gets the data. Here you have a variable and a function

1. `dataManager`
2. `retrieveTodoList()`

Finally a function called `viewDidLoad()` this function is called in the views `viewDidLoad()` lifecycle function after that super call.

2) Creating the List Presenter

Control-Click on **List** in the **Project Navigator** and select **New Group**, call the group **Presenter**.



Now you've expanded your folder structure, Create a new file inside the Presenter folder called **ListPresenter.swift**

Add the following code

```
class ListPresenter: TodoListPresenterDelegate {
  weak var view: TodoListDelegate?
  var interactor: TodoListInteractorInputDelegate?

  func viewDidLoad() {
    interactor?.retrieveTodoList()
  }
}
```

Now you've inherited the Presenter delegate, you should also deal with what happens when you receive the todos from the interactor. For this create an extension for `ListPresenter`

```
extension ListPresenter: TodoListInteractorOutputDelegate {

  func didRetrieveTodos(_ todos: [TodoItem]) {
    view?.show(items: todos)
  }

  func onError() {
    view.error()
  }
}
```

In this extension you are conforming to the `TodoListInteractorOutputDelegate` this delegate protocol determines what happens after you retrieve the todos. When you receive todos, you would want to show them.

3) Completing the Interactor - Presenter Interaction

Now that the basic protocols and presenters are all set up, all that's left is for you to tie them into the rest of the code base.

Open **ListViewController.swift** and under the **MARK** for variables before items add the following code.

```
var presenter: TodoListPresenterDelegate?
```

Here you're setting up the presenter for use later on.

Next call the presenters `viewDidLoad()` function like discussed above.

```
override func viewDidLoad() {
  super.viewDidLoad()
  presenter?.viewDidLoad()

  self.title = navigationTitle
}
```

Now create an extension and conform to the `TodoListDelegate` by adding the following code.

```
extension ListViewController: TodoListDelegate {

  func show(items: [TodoItem]) {
    self.items = items
    rwTableView.reloadData()
  }

  func error() {
    view = errorView
  }
}
```

4) Cleaning up the code

Now that you have completed four of the five components of viper it is time to move onto routing, but before you do we should clean up the codebase a little bit, of course that is the whole reason for clean architecutre! :]

We can start by cleaning up our code a little bit. Currently in your

ListViewController you have some old functions for getting data called `getSavedTodos()`. Since `retrieveTodoList()` exists in the interactor we do not need to get any todos here anymore. Go ahead and remove this.

Now we can continue to clean up our code by moving our `configureCheckmark()` and `configureText()` functions to a new file.

Control + Click on the Views folder and add a new group, call this group **Cells**. Create a new file in the folder called **RWTableViewCell**

Add the following code

```
import UIKit

class RWTableViewCell: UITableViewCell {

  func configureCheckmark(forCell cell: UITableViewCell, withItem item:
TodoItem) {
    if item.completed {
      cell.accessoryType = .checkmark
    } else {
      cell.accessoryType = .none
    }

    PersistenceService.saveContext()
  }

  func configureText(for cell: UITableViewCell, with item: TodoItem) {
    cell.textLabel?.text = item.title
  }

}
```

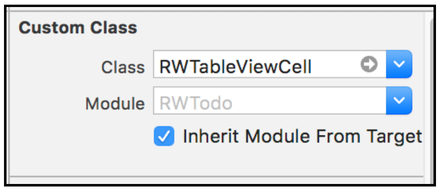
Now that you added the functionality for the cell, we need to connect the cell to the storyboard.

Open **Main.storyboard** and click on the ListViewController header

Mouse over the available options until you see RW Table View.

Click RWTable view and the table will appear above. Click on the cell in the table view above.

With the cell selected, change the class to **RWTableViewCell**


Custom Class
Class RWTableViewCell
Module RWTodo
[✓] Inherit Module From Target

Now back in the **ListViewController** you need to change the functionality of the cell so it is a **RWTableViewCell**

if you jump down to `updateItemInTableView(item:TodoItem)` on line **115** we need to change the cell type here, go ahead and cast the cell as shown below.

```
if let cell = rwTableView.cellForRow(at: indexPath) as? RWTableViewCell {
   cell.configureText(for: cell, with: item)
}
```

Actually you can should cast the cell again. Add this code to `didSelectRow` on line **163**

Inside the `cellForRow` method, cast the cell creation as well to a **RWTableViewCell**

```
let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier,
for: indexPath) as? RWTableViewCell

cell?.configureText(for: cell!, with: item)
cell?.configureCheckmark(for: cell!, with: item)

return cell!
```

If you run the application you will notice you can save todos and they appear in the list, but if you shut down and open the app, they no longer appear. Oh no! No worries, this is due to the routing aspect. Once you tie in the routing everything will fall into place!

5) That's it!

Congrats, at this time you should have a good understanding of Presenters! Take a break, and then it's time to move onto Routing.

Clean Architecture on iOS: Demo 3

By Anthony Lockett

In this demo, you will create the final protocols and router / wireframe to complete the VIPER architecture.

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**.

1) Creating the Wireframe protocol

Currently you have most of the architecture of VIPER laid out. Now you're only missing how to show and connect the peices. The wireframer or Router is the component that connects all the underlying components

Open **TodoListProtocols.swift** and replace `import Foundation` with the following:

```
import UIKit
```

Next, add the following protocol at the bottom of the file:

```
protocol TodoListWireframeDelegate: class {
  static func createTodoListController() -> UIViewController
}
```

You declared one function in the above protocol, the `createTodoListController`. This will later be used in the `AppDelegate` to create our TodoList View Controller

2) Fixing the protocols

Open **TodoListProtocols.swift**

Let's start with the `TodoListInteractorInputDelegate`, this protocol is missing a presenter.

Add the following code to the `TodoListInteractorInputDelegate`:

```
var presenter: TodoListInteractorOutputDelegate? { get set }
```

Next, in the `TodoListPresenterDelegate`, you're missing the wireframe.

Add the following code to the `TodoListPresenterDelegate`:

```
var wireframe: TodoListWireframeDelegate? { get set }
```

Next, open **ListPresenter.swift** and add the following property to satisfy the `TodoListPresenterDelegate` conformance:

```
var wireframe: TodoListWireframeDelegate?
```

3) Creating the Router / Wireframe

Now we have the protocol we need in place, we can create the router / wireframe.

Under the **List** folder in the `Project Navigator` inside the `List` folder, create a new folder called **Wireframe**

Inside the Wireframe folder create a new file called **TodoListWireframe.swift** and replace its contents with the following:

```
import UIKit

//MARK: - TodoListWireframeDelegate
class TodoListWireframe: TodoListWireframeDelegate {

  let mainStoryboard = Utils.shared.mainStoryboard

  class func createTodoListController() -> UIViewController {
    let navController =
      mainStoryboard.instantiateViewController(
      withIdentifier: "TodoListNavController")

    guard let view = navController.childViewControllers.first
      as? ListViewController else {
        return UIViewController()
    }

    let presenter:
      TodoListPresenterDelegate &
      TodoListInteractorOutputDelegate = ListPresenter()
    let interactor:
      TodoListInteractorInputDelegate = ListInteractor()
    let localDataManager:
      TodoListPersistenceManagerInputDelegate =
        TodoListLocalDataManager()
    let wireframe: TodoListWireframeDelegate =
      TodoListWireframe()

    view.presenter = presenter
    presenter.view = view
    presenter.wireframe = wireframe
    presenter.interactor = interactor
    interactor.presenter = presenter
    interactor.dataManager = localDataManager

    return navController
  }

}
```

Here you're adding the basic wireframe, you conform to the `TodoListWireframeDelegate` that requires the creation of the createTodoListController class function.

In the createTodoListController you get an instance of the `navController` from the storyboard. This is the main navigation controller for the whole app.

Next you're getting the first child view controller of the nav controller which is the TodoList View Controller

Once you have that view, you can set the presenter, view, interactor, wireframe and data manager.

Now that the basics of the routing is set up you can fix some potential issues.

Now all you need to do it set up the `AppDelegate` to use the wireframe previously created.

4) Using the TodoListWireframe

The wireframe is the routing aspect of VIPER, It stores all of the logic needed to show views when they are needed.

Open **AppDelegate.swift** and add the following code:

```
func application(_ application: UIApplication,
                 didFinishLaunchingWithOptions launchOptions:
                 [UIApplicationLaunchOptionsKey : Any]? = nil)
                  -> Bool {
    let todoListController =
      TodoListWireframe.createTodoListController()
    window = UIWindow(frame: UIScreen.main.bounds)
```

```
      window?.rootViewController = todoListController
      window?.makeKeyAndVisible()
      return true
  }
```

Here you're creating the `TodoListController` from the wireframe created previously.

Then you're setting the window and properties of the window like the `rootViewController`.

Build and Run and you should see the application works as normal, the only difference now is you seperated the distinct layers and are using `CoreData`.

You can test this by setting breakpoints in specific areas. Go ahead and set a breakpoint at `line 40` and `line 42` in the `ListInteractor.swift`, `line 38` in the `ListPresenter.swift` and `line 55` in the `ListViewController.swift`

Build and Run Again

You see that you can follow this down. You first call `viewDidLoad()` on the presenter, then you call `retrieveTodoList()` from the interactor and finally you check if the `dataManager` returns some todo items, if it does then you call `didRetrieveTodos(todoItems)` and pass in those todos to the presenter, which presents them on the view.

Carrier
9:03 AM
+
Todo List

Completed Demo 3!

5) That's it!

Congrats! You should now have a basic understanding of how to create a clean architecture using VIPER. As you can tell there is a bunch more that could be done to make this even more clean like adding the Add/Edit view to using VIPER as well.