

MVVM IN SWIFTUI

MOHAMMAD AZAM

About the Author



Mohammad Azam is a veteran developer who has been professionally writing software for more than a decade. Azam has worked as a lead mobile developer for many fortune 500 companies including Valic, AIG, Dell, Baker Hughes and Blinds.com. Azam is also a top Udemy and LinkedIn instructor with more than 40K students. At present Azam is a lead instructor at DigitalCrafts, where he teaches software development.

Azam is also an international speaker and has been professionally speaking since 2006. In his spare time Azam likes to exercise and plan his next adventure to the unknown corners of the world.

Twitter: <https://twitter.com/azamsharp>

Udemy: <https://www.udemy.com/user/mohammad-azam-2/>

MVVM Design Pattern

The term MVVM is used fluently in the Microsoft community. MVVM stands for Model View ViewModel and it is one of the many design patterns for creating software. The real name of MVVM is [Presentation Model](#), which is inspired from Application Model in Smalltalk.

Design Patterns are platform independent. This means you can use design patterns with any language.

The three different components of MVVM are Model, View and ViewModel. Let's take a closer look at the different components of MVVM.

Model

The model represents the application domain. Models are classes in your code, which identify business objects. Consider implementing an application for a banking system. The domain objects in the banking system might consists of the following models:

Account
Customer
Transaction
Bank
AccountType
TransactionType

Each model represents a certain aspect of the business domain. A Customer object represents a customer. This may include the customer's name, address, ssn etc. The Account object represents the customer's account, which may include attributes like account number, balance, type, status, interest rate etc.

Model may also contain business rules, which are based on the domain of the application. For a banking applications these rules may include the following:

- Adding a penalty if the balance is less than a specific amount
- Activating the interest rates based on date time
- Processing transaction fees for wire transfers

View

The view represents the user interface in the MVVM design pattern. This includes webpages, screens of mobile devices, smart watches and even console/terminal screens. Anything that can be visually represented is a view. A view is the primary way a user can interact with your app i.e (pressing a button, scrolling, drag and drop etc).

A view is something you can see with your eyes

In iOS development, a view is created in a few different ways. This includes storyboards, programmatically or declaratively using SwiftUI framework.

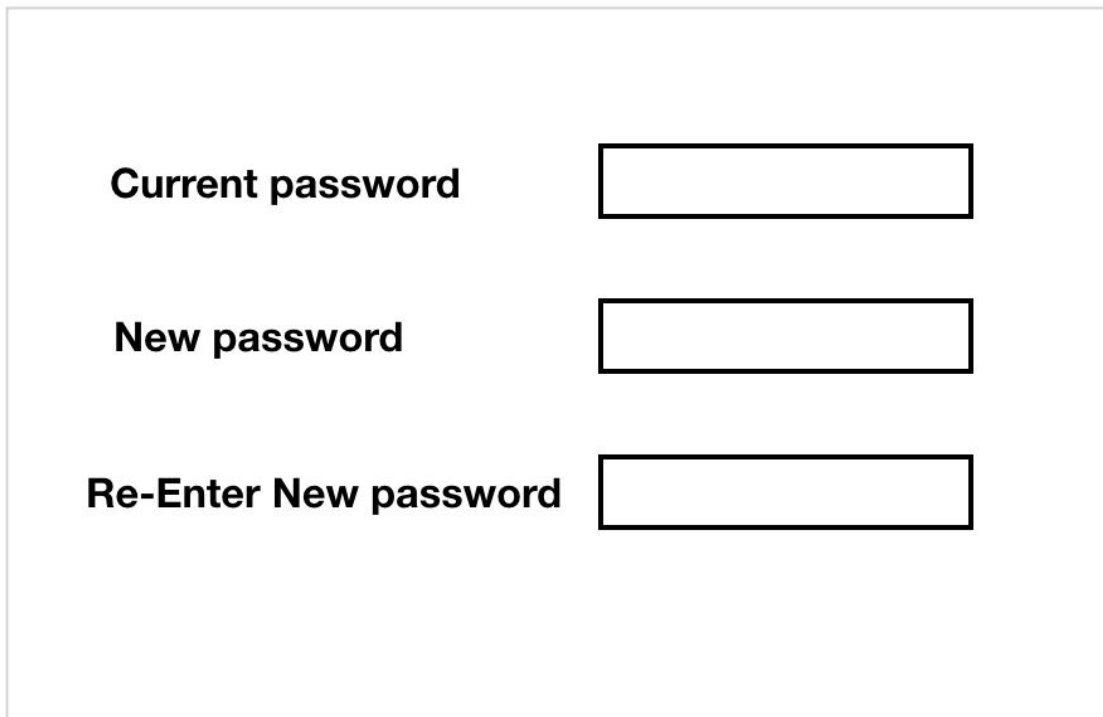
ViewModel

The primary role of a view model is to provide data to the view. A view model consists of properties which bind to the controls on the screen. Binding, simply means that data from a particular property of the view model will be displayed on a particular user interface element.

Binding can be bi-directional, meaning that the data from the view model is displayed on the view and if view changes the data then it is automatically updated in the view model. The whole idea behind binding is to keep the user interface (view) in-sync with the view model.

At this point, you might be thinking that why use a view model, why not simply bind the model to the view. Although you can do that, it is not advisable. Model contains the business rules, validations and domain logic. Model may even contain tons of other properties, which may make no sense to the view.

Consider a simple example of change password screen. The **view** is shown in **Figure 1**.



The figure shows a simple change password screen layout. It consists of three rows, each with a text label on the left and a rectangular input field on the right. The labels are 'Current password', 'New password', and 'Re-Enter New password'. The input fields are empty and have a thin black border.

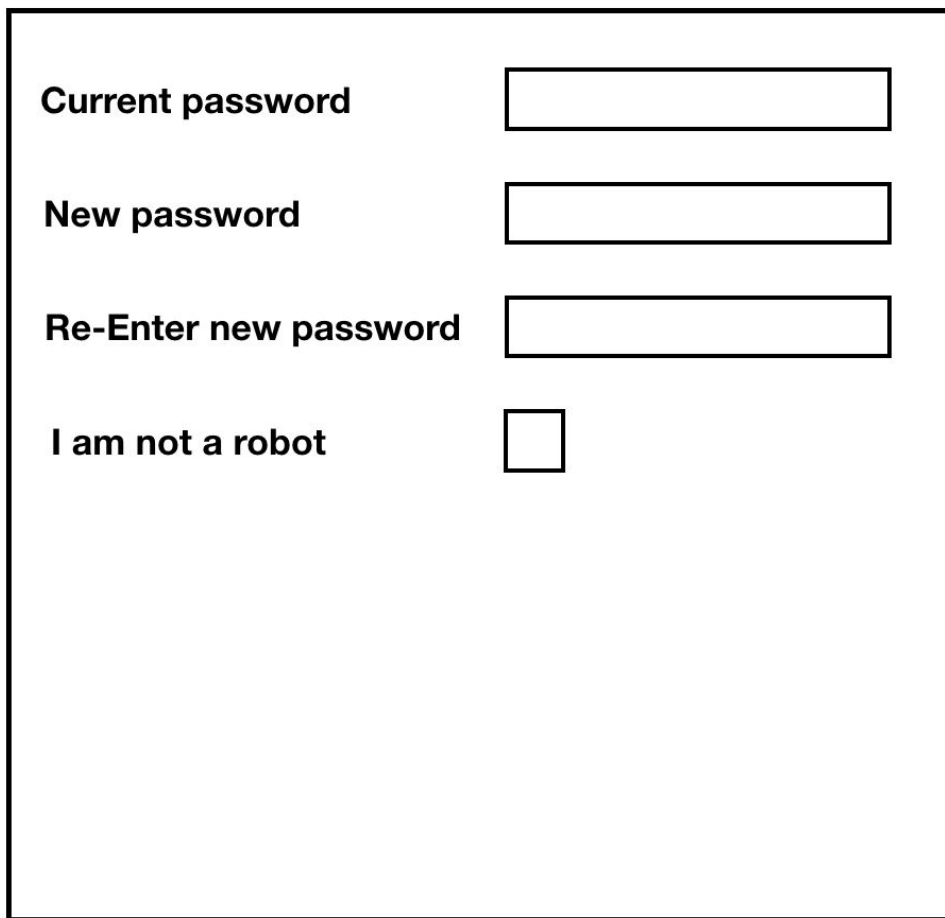
Figure 1: Change password screen

If you use a model to represent the change password screen, then you may end up with the implementation shown in **Listing 1**.

```
1. struct User {  
2.     let username: String  
3.     let password: String  
4.     let confirmPassword: String  
5. }
```

Listing 1: User model representing the password change screen

From a strictly technical point of view, the User model will fulfil your needs and your application will still work as expected. The problem will arise when the view changes and you need to accommodate new features. A change in the view will cause a change in your business model. Let's say we want to add an option to enable Captcha for our password change screen. Our view will update as shown in **Figure 2**.



The form is enclosed in a black rectangular border. It contains four labels on the left and their corresponding input fields on the right. The labels are 'Current password', 'New password', 'Re-Enter new password', and 'I am not a robot'. The first three labels are followed by rectangular text input fields. The last label is followed by a small square checkbox.

Current password	<input type="text"/>
New password	<input type="text"/>
Re-Enter new password	<input type="text"/>
I am not a robot	<input type="checkbox"/>

Figure 2: Captcha added to the change password screen

Now, we need to update our model to accommodate the captcha. This change is reflected in **Listing 2**.

```
1. struct User {  
2.     let username: String  
3.     let password: String  
4.     let confirmPassword: String  
5.     let isRobot: Bool  
6. }
```

Listing 2: isRobot property added to the User model

As you can see, the user interface behavior is slowly creeping into our business models. Soon it will have properties to represent the show/hide status of a particular view. The whole point of the model is to deal with business rules, not the view.

This is the main reason why view models exist. View models represent the data and the structure behind the view. Each view element can be bound to a property of the view model. This means if the view updates the property, it is automatically updated in the view model and vice versa. **Listing 3** shows the implementation of `ChangePasswordViewModel`, which is the view model behind the change password screen.

```
1. struct ChangePasswordViewModel {  
2.     let password: String  
3.     let confirmPassword: String  
4.     let isRobot: Bool  
5. }
```

Listing 3: Implementation of `ChangePasswordViewModel`

Now that you have the basic information about the MVVM Design Pattern, let's take a look at different ways view models can be structured in an application.

Structuring View Models

In the last chapter, you learned about the MVVM Design Pattern. We looked at different pieces of MVVM design patterns, which includes Model, View and ViewModel. In this section you are going to learn multiple techniques for structuring your view models.

View Models and Lists

When building mobile applications, quite often you need to display a list of items. This means you will have an array of objects, which you want to present on the view. There are a number of ways to structure your view models to accommodate the need to display a list.

You can always send an array of view models back to the view, but we found that creating a parent view model which contains a list of view models is a more flexible solution.

Consider a scenario, where you are building an app which is responsible for displaying list of dishes. The dishes view is shown in **Figure 3**.



Dishes
Filet Mignon
Fried Fish
Mac and Cheese
Spaghetti Meatballs
Grill Cheese Sandwich
Katakut

Figure 3: Dishes screen

One possible implementation of view model is shown in **Listing 4**.

```
1. import UIKit
2.
3. struct DishListViewModel {
4.     var dishes = [DishViewModel]()
5. }
6.
7. struct DishViewModel {
8.
9.     let name: String
10.    let course: String
11.    let price: Double
12.
13. }
```

Listing 4: DishListViewModel and DishViewModel

The DishListViewModel represents the entire view, which is responsible for displaying dishes on the screen. DishListViewModel contains a nested list, represented by DishViewModel objects. The benefit of this approach is that it gives you the flexibility to add additional attributes/properties to the view model. For example, maybe you want to add search functionality to your dishes screen. You can update the view to add a search bar and update the view model to capture the search term as shown in **Listing 5**.

```
1. struct DishListViewModel {
2.     var searchTerm: String = ""
3.     var dishes = [DishViewModel]()
4. }
5.
6. struct DishViewModel {
7.
8.     let name: String
9.     let course: String
10.    let price: Double
11.
12. }
```

Listing 5: DishListViewModel with searchTerm field

By adding a parent view model, you add the flexibility to accommodate future changes to the view much more easily.

Multi-Child View Models

As explained in the previous section, a parent view model can also accommodate multiple child view models. It all depends on the user interface of the application. Take a look at the **Figure 4** which shows a Stocks app interface.

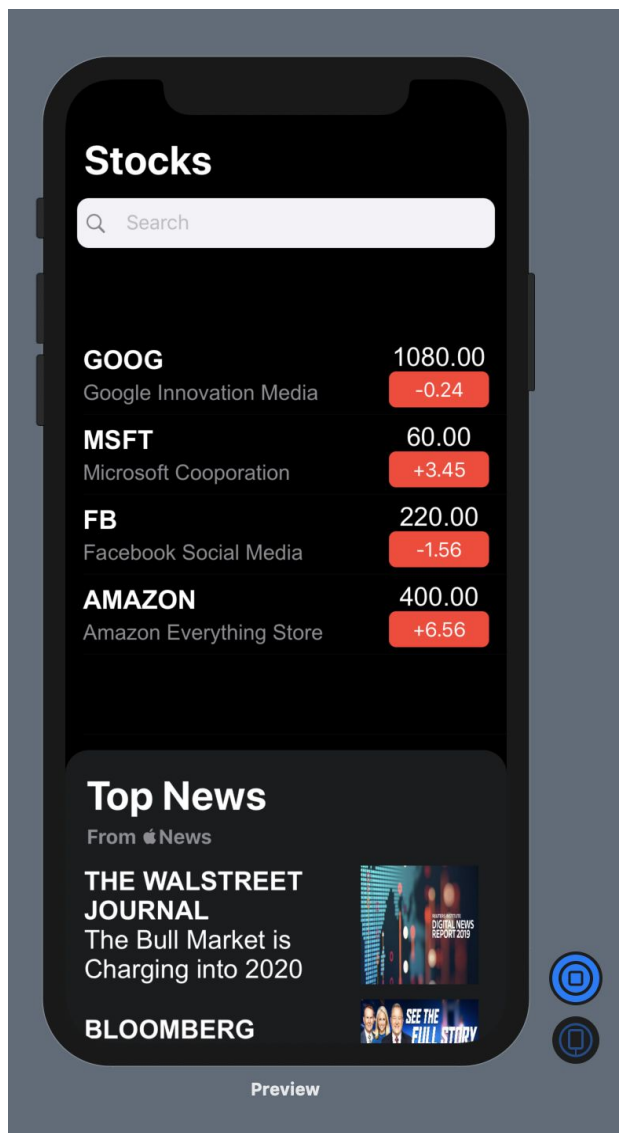


Figure 4: Stocks app interface

The stocks app displays two different types of information. It displays a list of stocks and top news. A single parent view model can represent the whole screen. The parent view model can be divided into multiple nested view models representing stocks and top news articles. One possible implementation of parent view model is shown in **Listing 6**.

```
1. struct HomeViewModel {  
2.     var stocks = [StockViewModel]()  
3.     var articles = [ArticleViewModel]()  
4. }  
5.  
6. struct StockViewModel {  
7.     let symbol: String  
8.     let price: Double  
9.     let company: String  
10.    let change: Double  
11. }  
12.  
13. struct ArticleViewModel {  
14.     let title: String  
15.     let publication: String  
16.     let imageURL: String  
17. }
```

Listing 6: Multi-Child View Model

The HomeViewModel represent the main model which controls the entire view. It is composed of two child view models, stocks and articles respectively. The HomeViewModel will be responsible for populating the child view models. This can be done by calling a webservice and fetching data from an API or populating it from the database.

A better organized and structured view model can help to populate the view more efficiently. In the next section, we are going to take a look at how view models can be used to validate user's input and provide feedback.

MVVM and Validation

There is a famous saying in software development “**Garbage in, garbage out**”. It means that if you don’t validate user’s input, then you may end up with invalid or garbage data. In this chapter you will learn multiple ways of performing validation in SwiftUI applications. To keep things simple, we will be working on a registration screen which will allow users to create a new account. Let’s start with basic validation.

Basic Validation

The first step is to build the user interface for the app. There are countless ways to implement a registration view. We have used the Form view in SwiftUI to implement the registration view. The implementation of the view is shown in **Listing 6.1**.

```

1. import SwiftUI
2.
3. struct ContentView: View {
4.
5.     @State private var firstname: String = ""
6.     @State private var lastname: String = ""
7.     @State private var username: String = ""
8.     @State private var password: String = ""
9.
10.
11.     var body: some View {
12.         NavigationView {
13.
14.             Form {
15.                 VStack(spacing: 10) {
16.                     TextField("First name", text: $firstname)
17.
18.                     .textFieldStyle(RoundedBorderTextFieldStyle())
19.                     TextField("Last name", text: $lastname)
20.
21.                     .textFieldStyle(RoundedBorderTextFieldStyle())
22.                     TextField("Username", text: $username)
23.
24.                     .textFieldStyle(RoundedBorderTextFieldStyle())
25.                     SecureField("Password", text: $password)
26.
27.                     .textFieldStyle(RoundedBorderTextFieldStyle())

```

```
24.  
25.         Button("Register") {  
26.  
27.  
28.         }.padding(10)  
29.             .background(Color.blue)  
30.             .cornerRadius(6)  
31.             .foregroundColor(Color.white)  
32.  
33.  
34.  
35.         }  
36.  
37.     }  
38.  
39.     .navigationBarTitle("Registration")  
40.  
41. }  
42.  
43.  
44. }  
45. }
```

Listing 6.1: User interface using the Form view in SwiftUI

The output is shown in **Figure 4.1**.

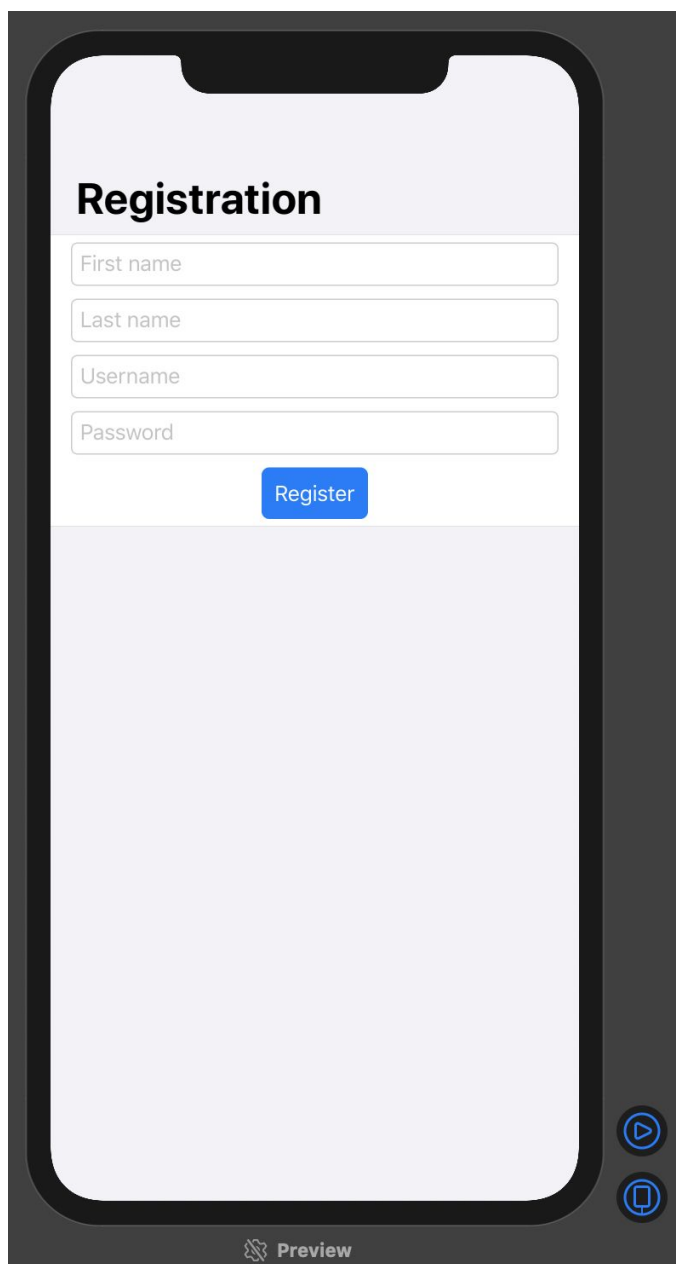


Figure 4.1: Registration view using SwiftUI Forms

At present our Register button does not perform any validation. We can write a very basic validation by checking the value of the properties as shown in **Listing 6.2**.

```
1. @State private var message: String = ""
2.
3. Button("Register") {
4.
5.         if self.firstname.isEmpty {
6.             self.message.append("Firstname
cannot be empty! \n")
7.         }
8.
9.         if self.lastname.isEmpty {
10.            self.message.append("Lastname
cannot be empty!")
11.        }
12.
13.            // do the same for username and
password
14.
15.
16.    }
```

Listing 6.2: Simple validation function

The result is shown in **Figure 4.2**.

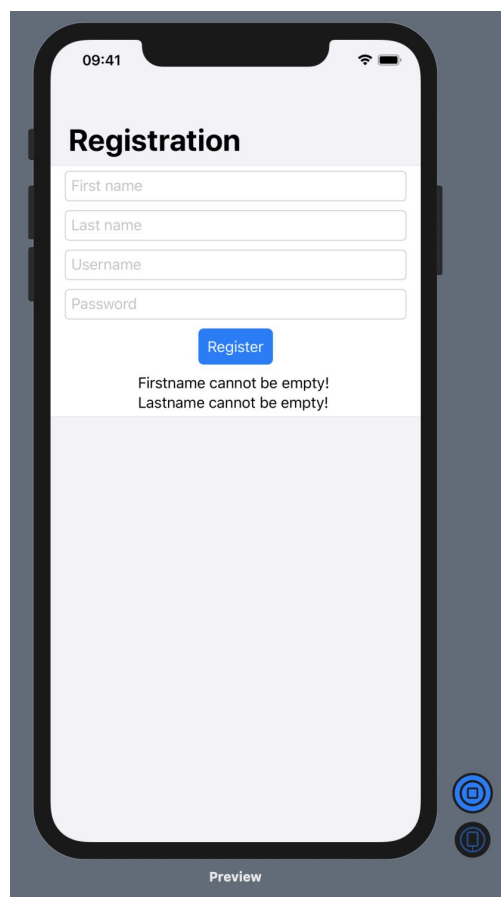


Figure 4.2: Simple validation

As you can see, it works but it is not a good solution. First of all, we are managing multiple independent state variables, which can be replaced by a single view model. The second reason is that our registration view is responsible for validating the user interface. As the complexity of the user interface grows, so does the validation code. In the next section we will look into building a custom validation engine, which will be responsible for performing validation for our view model.

Implementing Validation Engine

The first step is to get rid of individual properties and replace them with a single view model. We have implemented a view model called `RegistrationViewModel` which represents the entire screen. The implementation is shown in **Listing 6.3**.


```

1. import Foundation
2.
3. class RegistrationViewModel: ValidationBase {
4.     var firstname: String = ""
5.     var lastname: String = ""
6.     var username: String = ""
7.     var password: String = ""
8. }

```

Listing 6.3: RegistrationViewModel

RegistrationViewModel inherits from ValidationBase which will be discussed in a moment. Let's first replace our individual properties in the registration view (ContentView) with the newly created view model. This is shown in **Listing 6.4**.

```

1. struct ContentView: View {
2.
3.     @ObservableObject private var registrationVM =
        RegistrationViewModel()
4.
5.     var body: some View {
6.         NavigationView {
7.
8.             Form {
9.                 VStack(spacing: 10) {
10.                     TextField("First name", text:
                        $registrationVM.firstname)
11.                     .textFieldStyle(RoundedBorderTextFieldStyle())
12.                     TextField("Last name", text:
                        $registrationVM.lastname)
13.                     .textFieldStyle(RoundedBorderTextFieldStyle())
14.                     TextField("Username", text:
                        $registrationVM.username)
15.                     .textFieldStyle(RoundedBorderTextFieldStyle())
16.                     SecureField("Password", text:
                        $registrationVM.password)
17.                     .textFieldStyle(RoundedBorderTextFieldStyle())
18.

```

```

19.         Button("Register") {
20.
21.
22.
23.             }.padding(10)
24.                 .background(Color.blue)
25.                 .cornerRadius(6)
26.                 .foregroundColor(Color.white)
27.
28.
29.         }
30.     }
31.     .navigationBarTitle("Registration")
32.
33. }
34.
35. }
```

Listing 6.4: Replacing individual properties with view model

One thing to note is that we have decorated the view model instance with `@ObservableObject`. This means that `RegistrationViewModel` can publish events which can be observed by the view or anyone who wants to subscribe to it. To make our code modular and maintainable, we have created a `ValidationBase` class which provides base implementation of broken rules. The implementation of `ValidationBase` class is shown in **Listing 6.5**.

```

1. class ValidationBase: ObservableObject {
2.     @Published fileprivate (set) var brokenRules: [BrokenRule] =
3.         [BrokenRule]()
4. }
```

Listing 6.5: ValidationBase

`ValidationBase` class maintains an array of broken rules. A `BrokenRule` is simply a structure which represents a failed validation failure. The implementation is shown in **Listing 6.6**.

```

1. struct BrokenRule {
2.     let id = UUID()
```

```

3.     let name: String
4.     let message: String
5. }

```

Listing 6.6: Implementation of BrokenRule

The name represents the property, which failed the validation and the message represents a custom message to be displayed to the user.

We have also implemented a Validator protocol, which will be used by view models interested in providing validation behavior. The Validator protocol is shown in **Listing 6.7**.

```

1. protocol Validator: ValidationBase {
2.     func validate()
3. }
4.
5. extension Validator {
6.
7.     func addBrokenRule(_ rule: BrokenRule) {
8.         brokenRules.append(rule)
9.     }
10.
11.     func clearBrokenRules() {
12.         brokenRules = []
13.     }
14.
15. }

```

Listing 6.7: Validator protocol with default implementations

Now, we can update RegistrationViewModel to implement the validate function as shown in **Listing 6.8**.

```

1. class RegistrationViewModel: ValidationBase {
2.     var firstname: String = ""
3.     var lastname: String = ""
4.     var username: String = ""
5.     var password: String = ""
6. }
7.

```

```

8. extension RegistrationViewModel: Validator {
9.
10.     func validate() {
11.
12.         clearBrokenRules()
13.
14.         if firstname.isEmpty {
15.             addBrokenRule(BrokenRule(name: "firstname",
message: "Firstname should not be empty"))
16.         }
17.
18.         if(lastname.isEmpty) {
19.             addBrokenRule(BrokenRule(name: "lastname",
message: "Lastname should not be empty"))
20.         }
21.     }
22. }

```

Listing 6.8: Implementation of validate function

The validate function first clears out any existing broken rules and then performs validation on the individual properties of the view model. If the validation fails, a broken rule is added to a list of broken rules.

Since registrationVM is decorated with ObservableObject protocol and brokenRules with @Published property wrapper, any changes to the brokenRules will cause the publisher to publish new events. This is our opportunity to get the updated rules and display the broken rules on the user interface. In the next section we will implement BrokenRulesView, which will be responsible for displaying the validation errors on the screen.

Displaying Broken Rules

In the last section, we implemented the validate function, responsible for performing validation on the properties of the view model. In this section we will display the broken rules on the view.

Instead of polluting the registration view (ContentView), we will create a brand new view called BrokenRulesView. BrokenRulesView will take a list of broken rules as an

argument and then display them on the view. The implementation of BrokenRulesView is shown in **Listing 6.9**.

```

1. import SwiftUI
2.
3. struct BrokenRulesView: View {
4.
5.     let brokenRules: [BrokenRule]
6.
7.     var body: some View {
8.         List(brokenRules, id: \.id) { rule in
9.             Text(rule.message)
10.        }
11.    }
12. }
13.
14. struct BrokenRulesView_Previews: PreviewProvider {
15.     static var previews: some View {
16.         BrokenRulesView(brokenRules: [])
17.     }
18. }

```

Listing 6.9: Implementation of BrokenRulesView

The List view is used to iterate over the array of broken rules and then display them using a Text view. **Listing 6.10** shows how to use the BrokenRulesView in registration view (ContentView) to display the validation errors.

```

1. Form {
2.     ...
3.
4.     BrokenRulesView(brokenRules:
5.         registrationVM.brokenRules)
6. }
7. }

```

Listing 6.10: Using BrokenRulesView

Run the application and click the register button without filling out the TextFields. The validate function gets triggered and passes down the broken rules to BrokenRulesView where finally it gets displayed as shown in **Figure 4.3**.

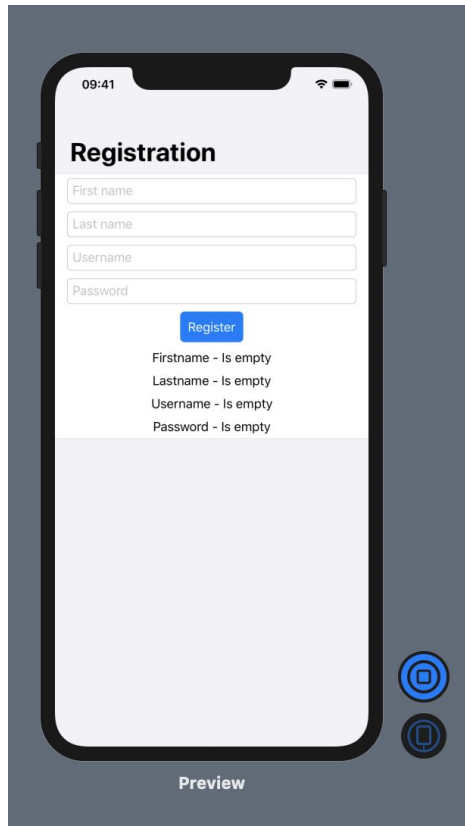


Figure 4.3: Broken rules displayed on the screen

This approach also gives you the flexibility to customize the display of validation errors by changing the user interface through the modification of BrokenRulesView. Although this approach works, we can definitely make it better. **Swift 5.1** introduced property wrappers which allows custom code to be triggered when evaluating properties.

In the next chapter you will learn how to use the **ValidatedPropertyKit** framework to perform validation in SwiftUI. ValidatedPropertyKit framework performs validation by decorating the properties with property wrappers.

Performing Validation Using Validated Framework

[ValidatedPropertyKit](#) is a Swift framework for performing validation in your application. It uses property wrappers feature introduced in Swift 5.1 to decorate properties with required validation attributes. You have already used several property wrappers including `@State`, `@Binding`, `@ObservableObject`.

In this section we are going to integrate ValidatedPropertyKit to our existing SwiftUI application and display the broken rules on the interface.

Integrating ValidatedPropertyKit

The first step is to integrate ValidatedPropertyKit framework in your SwiftUI application. Keep in mind that ValidatedPropertyKit is an independent library and has no dependency on SwiftUI framework.

The easiest way to integrate ValidatedPropertyKit is by using Swift Package Manager. In Xcode open the **File** menu and then select **Swift Packages** and then **Add Package Dependency**. This will open a dialog where you can enter the Github url for the dependency. This is shown in **Figure 4.4**.

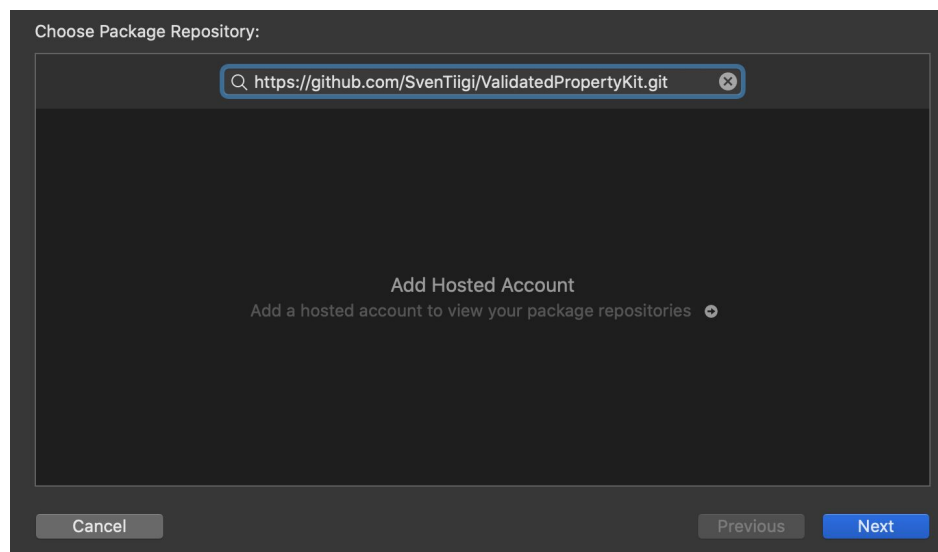


Figure 4.4: Adding dependency through Swift Package Manager in Xcode

Click on the Next button and complete the whole process. Once the dialog closes your dependency to ValidatedPropertyKit will be successfully added to the project. Next we will decorate our view models with property wrappers which will enable validation.

Adding Validation to RegistrationViewModel

As mentioned before, ValidatedPropertyKit revolves around property wrappers. In order to perform validation we will have to decorate our properties in the view model. This is shown in **Listing 6.11**.

```
1. import Foundation
2. import ValidatedPropertyKit
3.
4. class RegistrationViewModel: ObservableObject {
5.
6.     @Validated(.notEmpty)
7.     var firstname: String? = ""
8.
9.     @Validated(.notEmpty)
10.    var lastname: String? = ""
11.
12.    @Validated(.notEmpty)
13.    var username: String? = ""
14.
15.    @Validated(.notEmpty)
16.    var password: String? = ""
17.
18. }
```

Listing 6.11: RegistrationViewModel decorated with property wrappers

Please note that we have declared properties as optionals and initialized them as empty strings as ValidatedPropertyKit will only validate not nil values.

The implementation starts with importing ValidatedPropertyKit and then decorating property wrappers on the properties, which will be taking part in the validation. There are many different property wrappers available in ValidatedPropertyKit framework. Few of them are shown in the list below:

.notEmpty - Value cannot be null or empty

.isEmail - Value must be in email format

.range(8...) - Value must satisfy the range

.greaterOrEqual(1) - Value must satisfy the condition

You can also create your own custom validation wrappers for situations that are not covered in the default ValidatedPropertyKit.

The next step is to implement the validate function which will be responsible for evaluating the properties. The implementation is shown in **Listing 6.12**.

```

1. @Published private (set) var brokenRules: [BrokenRule] =
   [BrokenRule]()
2.
3.     func validate() {
4.
5.         brokenRules.removeAll()
6.
7.         let rules = [
8.             "Firstname": _firstname.validationError,
9.             "Lastname": _lastname.validationError,
10.            "Username": _username.validationError,
11.            "Password": _password.validationError
12.        ]
13.
14.        _ = rules.compactMap { pair -> Void in
15.
16.            guard let errorMessage = pair.1?.failureReason
17.            else { return }
18.            brokenRules.append(BrokenRule(name: pair.0,
19.            message: errorMessage))
20.        }
21.    }

```

Listing 6.12: Validate function of RegistrationViewModel

Inside the validate function we create a rules array, which contains the name of the property and the validation error associated with the property.

Next, we iterate through the rules and if we find any failure reason associated with the property we add that as a broken rule to the brokenRules array. The brokenRules is also marked with `@Published` property wrapper, which means that it will publish an event as soon as it is updated.

The final step is to display the broken rules on the user interface. This is exactly the same as before, we will be calling the validate function on the registrationVM and then using the BrokenRulesView to display the errors. The implementation is shown in **Listing 6.13**.

```

1. struct ContentView: View {
2.
3.     @ObservedObject private var registrationVM =
       RegistrationViewModel()
4.
5.     var body: some View {
6.         NavigationView {
7.
8.             Form {
9.                 VStack(spacing: 10) {
10.                    TextField("First name", text:
       $registrationVM.firstname.bound)
11.
12.                    .textFieldStyle(RoundedBorderTextFieldStyle())
13.                    TextField("Last name", text:
       $registrationVM.lastname.bound)
14.
15.                    .textFieldStyle(RoundedBorderTextFieldStyle())
16.                    TextField("Username", text:
       $registrationVM.username.bound)
17.
18.                    .textFieldStyle(RoundedBorderTextFieldStyle())
19.                    SecureField("Password", text:
       $registrationVM.password.bound)
20.
21.                    .textFieldStyle(RoundedBorderTextFieldStyle())
22.
23.                    Button("Register") {
24.
25.                        self.registrationVM.validate()
26.
27.                    }.padding(10)
28.                    .background(Color.blue)

```

```

25.                .cornerRadius(6)
26.                .foregroundColor(Color.white)
27.
28.
29.                BrokenRulesView(brokenRules:
registrationVM.brokenRules)
30.
31.
32.            }
33.        }
34.        .navigationBarTitle("Registration")
35.
36.    }
37.
38. }
```

Listing 6.13: BrokenRulesView added to the ContentView

For those of you paying attention, you will notice that we used a bound function when binding the values to the TextField view. The reason is that TextField binds to `Binding<String>` but not `Binding<String?>`. The bound extension allows us to unwrap the optional, which can then be used to bind to the views on the user interface. The implementation of the bound function is available as downloadable source code.

The result is shown in **Figure 4.5**.

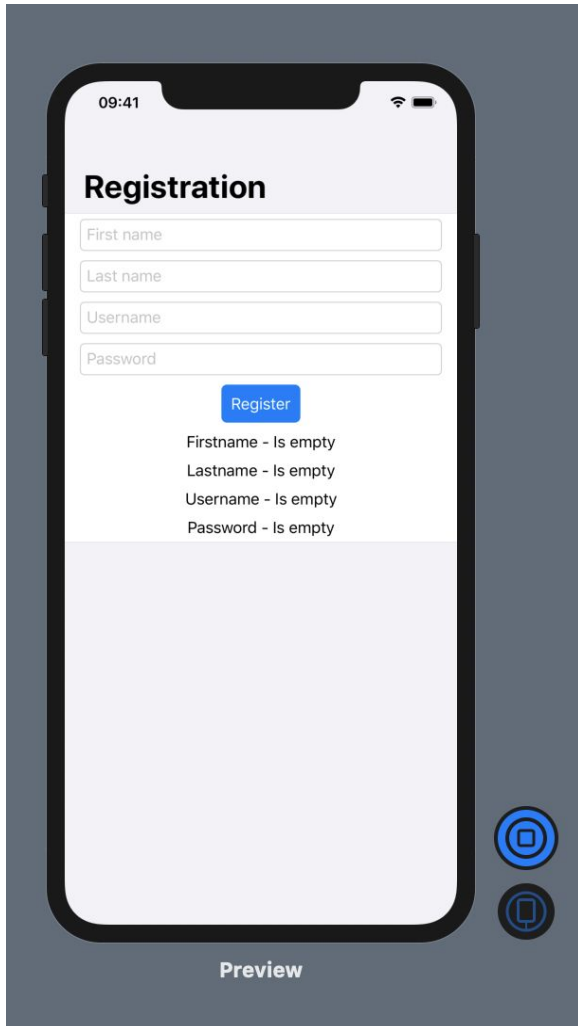


Figure 4.5: ValidatedPropertyKit showing errors

As you can see in **Figure 4.5**, validation is fired and the default error messages are displayed on the screen. This will work for some situations but it would be great if we can customize the error messages. In the next section, you will learn how to write a custom validation rule, which can provide a custom error message.

Customizing Validation Errors

In order to customize or add a new validation rule you will need to extend the Validation structure provided by ValidatedPropertyKit. Add a new file **“Validation+Extension.swift”** to your existing project and add a new static property called `required`. The implementation is shown in Listing **6.14**

```

1. import Foundation
2. import ValidatedPropertyKit
3.
4. extension Validation where Value == String {
5.
6.     static var required: Validation {
7.         return .init { value in
8.             value.isEmpty ? .failure("\(value) cannot be empty")
9.             : .success(())
10.        }
11.    }
12. }

```

Listing 6.14: Implementation of required validation property

The required property is our custom property, which will perform validation on String values. If the value is empty, failure closure is triggered with the error message, otherwise success closure is triggered.

Next, update your RegistrationViewModel and use the new required validation rule. This is shown in **Listing 6.15**.

```

1. class RegistrationViewModel: ObservableObject {
2.
3.     @Validated(.required)
4.     var firstname: String? = ""
5.
6.     @Validated(.required)
7.     var lastname: String? = ""
8.
9.     @Validated(.required)
10.    var username: String? = ""
11.
12.    @Validated(.required)
13.    var password: String? = ""

```

Listing 6.15: RegistrationViewModel using required validation rule

Run your app again and click on the register button without filling out the form. **Figure 4.6** shows the result.

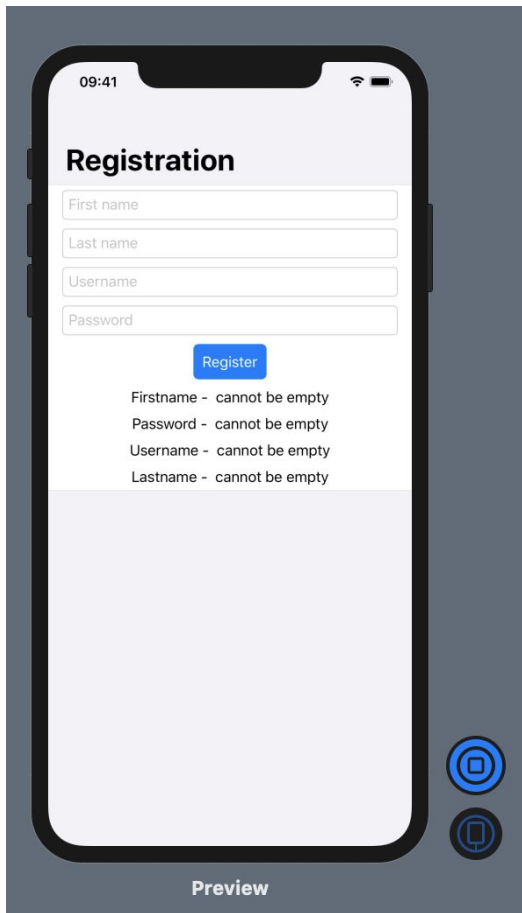


Figure 4.6: Custom validation messages

We can even take one step further and provide error messages right inside the property wrapper. For this to work we will update the custom required validator from property to a function. The implementation is shown in **Listing 6.16**.

```
1. static func required(errorMessage: String = "Is Empty") ->  
   Validation {  
2.     return .init { value in  
3.       value.isEmpty ? .failure(.init(message:  
   errorMessage)) : .success()  
4.     }  
5. }
```

Listing 6.16: Custom required validator

Now, you can update your `RegistrationViewModel` to utilize the new required function. This is shown in **Listing 6.17**.

```
1. class RegistrationViewModel: ObservableObject {  
2.  
3.     @Validated(.required(errorMessage: "First name cannot be  
empty"))  
4.     var firstname: String? = ""  
5.  
6.     @Validated(.required(errorMessage: "Last name cannot be  
empty"))  
7.     var lastname: String? = ""  
8.  
9.     @Validated(.required(errorMessage: "Username cannot be  
empty"))  
10.    var username: String? = ""  
11.  
12.    @Validated(.required(errorMessage: "Password cannot be  
empty"))  
13.    var password: String? = ""
```

Listing 6.17: Updated required validator to be a function

The result is shown in **Figure 4.7**.

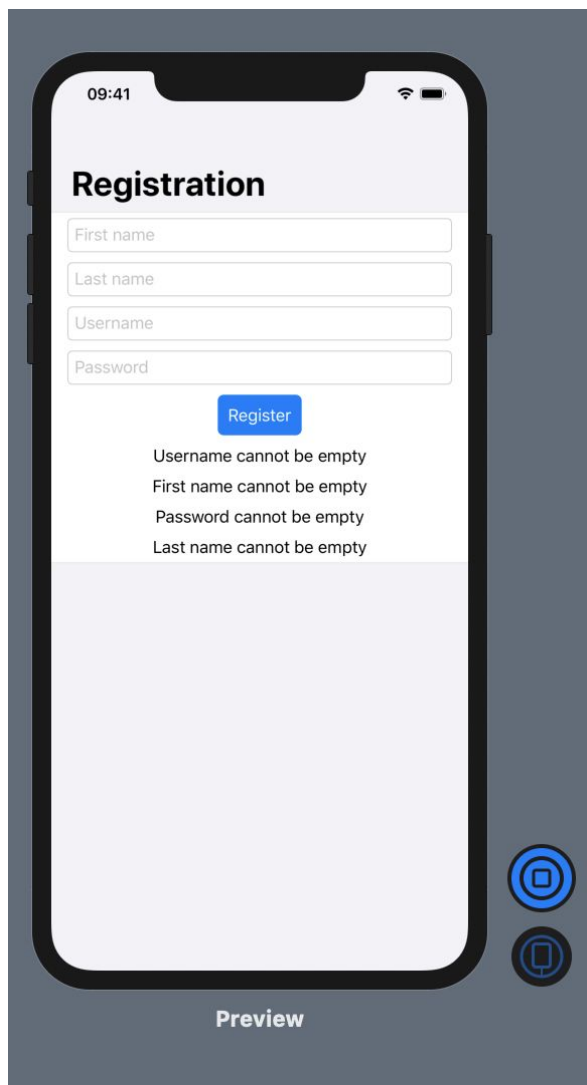


Figure 4.7: Error messages using custom validator

In this chapter you learned how to use ValidatedPropertyKit to validate your view model properties. ValidatedPropertyKit is an amazing framework, which uses the power of Swift 5.1 property wrappers to allow developers to add validation to their apps.

References

- 1) [ValidatedPropertyKit GitHub](#)
- 2) [NSScreencast ValidatedPropertyKit Demo Project](#)
- 3) [Implementation of bound function on StackOverFlow](#)

MVVM and Networking

One of the most common operations in iOS development is to perform a network request and fetch data from the server. In this chapter, we are going to take a look at how MVVM and networking fits together and how you can use smart view models to perform a network call.

When building applications using MVVM design pattern it is common to put a lot of functionality into the view models. A view model should not be responsible for making a network request. However, a view model can take help from a networking client to perform the request. **Figure 5** shows the flow.

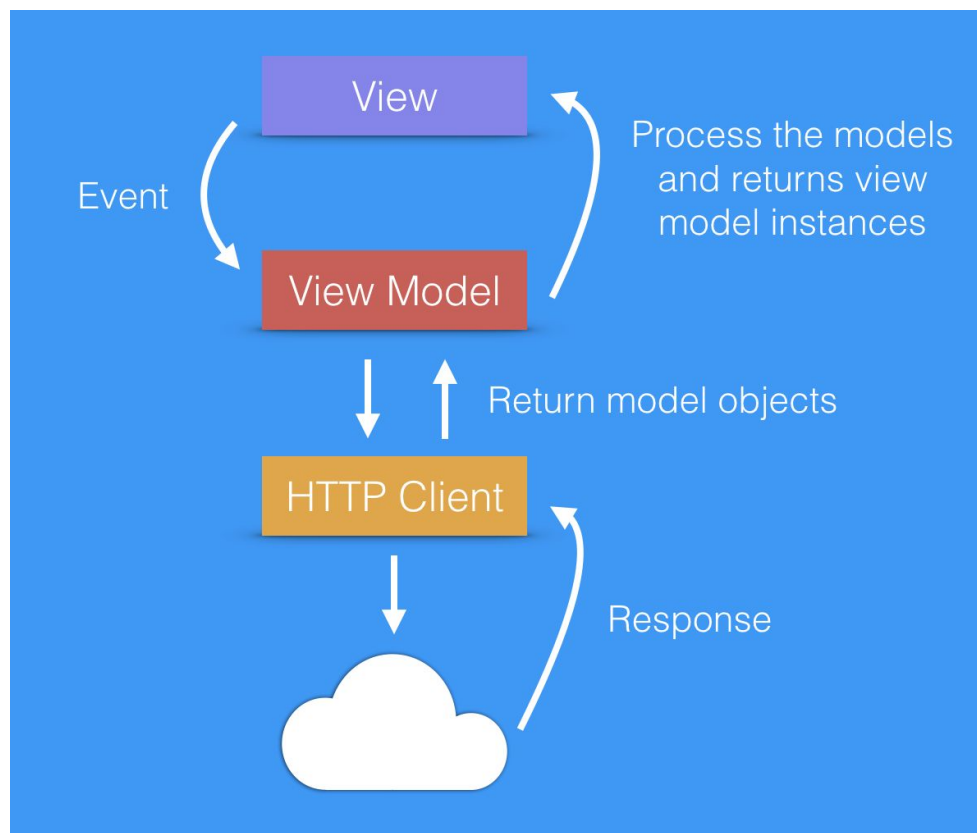


Figure 5: Roundtrip data flow between the view and the server

As shown in **Figure 5**, an event reaches the view model triggered from the view. The view model forwards the request to the HTTP client, which performs the actual request. The response is returned from the server to the HTTP client where it is passed to the

view model. The View model processes the model objects and returns the view model objects to the view. Finally, the view uses the view model to display the data on the screen.

For this chapter, we have built a very simple JSON API using Node and Express. The API is hosted on Glitch server at <https://silicon-rhinoceros.glitch.me> and it returns hard-coded data.

The response from the stocks endpoint is shown in **Listing 7**.

```
[{"symbol": "GOOG", "description": "Google Innovation Media", "price": 1080, "change": "-0.24"}, {"symbol": "MSFT", "description": "Microsoft Cooperation", "price": 60, "change": "+3.45"}, {"symbol": "FB", "description": "Facebook Social Media", "price": 220, "change": "-1.56"}, {"symbol": "AMAZON", "description": "Amazon Everything Store ", "price": 400, "change": "+6.56"}]
```

Listing 7: JSON response from stocks endpoint

The response consists of an array of stocks. Each stock contains attributes including **symbol**, **description**, **price** and **change**. In the next section we are going to implement `HTTPClient`, which will be responsible for fetching the response from the JSON API.

Implementing HTTP Client

The main purpose of `HTTPClient` is to send a request to an endpoint and then process JSON response. In **Listing 8** you can see the implementation of `HTTPClient`, which is responsible for returning list of stocks.

```
1. enum NetworkError: Error {
2.     case invalidURL
3.     case unknownError
4.     case decodingError
5. }
6.
7. class HTTPClient {
8.
9.     func getAllStocks(urlString: String, completion: @escaping
        (Result<[Stock], NetworkError>) -> Void) {
```

```

10.
11.         guard let url = URL(string: urlString) else {
12.             completion(.failure(.invalidURL))
13.             return
14.         }
15.
16.         URLSession.shared.dataTask(with: url) { data,
response, error in
17.
18.             guard let data = data, error == nil else {
19.                 completion(.failure(.unknownError))
20.                 return
21.             }
22.
23.             let stocks = try?
JSONDecoder().decode([Stock].self, from: data)
24.             stocks == nil ?
completion(.failure(.decodingError)) :
completion(.success(stocks!))
25.
26.             }.resume()
27.         }
28.
29.     }

```

Listing 8: Networking layer implementation in Swift

We have used the `Result` type feature of Swift 5 in our completion handler. This allows us to easily create handlers for success and error callbacks. One important point to note about `HTTPClient` is that it returns model objects and not view models. The model is implemented in **Listing 9**.

```

1. struct Stock: Decodable {
2.     let symbol: String
3.     let price: Double
4.     let description: String
5.     let change: String
6. }

```

Listing 9: Stock model implementation

The Stock model conforms to the Decodable protocol, making it easier to get populated by JSON response.

The property names of Stock model does not have to be the same as the JSON response. You can always use the CodingKey protocol to dictate your own custom mapping.

The StockViewModel is responsible for representing each stock in the view. The implementation is shown in **Listing 10**.

```
1. struct StockViewModel {  
2.  
3.     let stock: Stock  
4.  
5.     init(stock: Stock) {  
6.         self.stock = stock  
7.     }  
8.  
9.     var symbol: String {  
10.         return self.stock.symbol.uppercased()  
11.     }  
12.  
13.     var description: String {  
14.         return self.stock.description  
15.     }  
16.  
17.     var price: String {  
18.         return String(format: "%.2f", self.stock.price)  
19.     }  
20.  
21.     var change: String {  
22.         return self.stock.change  
23.     }  
24.  
25. }
```

Listing 10: StockViewModel implementation

As mentioned in earlier chapters, it is a good idea to create a parent view model which represents the entire screen. The implementation of StockListViewModel is shown in **Listing 11**.

```

1. class StockListViewModel {
2.
3.     var stocks = [StockViewModel]()
4.
5.     func fetchAllStocks() {
6.
7.         HTTPClient().getAllStocks(urlString:
8.             "https://silicon-rhinoceros.glitch.me/stocks") { result in
9.             DispatchQueue.main.async {
10.                 switch result {
11.                     case .success(let stocks):
12.                         self.stocks =
13.                         stocks.map(StockViewModel.init)
14.                     case .failure(let error):
15.                         print(error.localizedDescription)
16.                 }
17.             }
18.         }
19.     }
20. }
21.

```

Listing 11: Implementation of StockListViewModel

StockListViewModel consists of stocks property, which is an array of StockViewModel instances. It also consists of fetchAllStocks function, which is responsible for retrieving stocks using the HTTPClient object. Once the fetchAllStocks function returns the model Stock objects, they are converted into view models and sent to the view to be displayed.

The whole point of this flow is to build layers, where each layer is responsible for a certain aspect of the application. This will help us to keep the code cleaner and maintainable for future changes.

Before integrating MVVM design pattern into SwiftUI apps, it would be a good idea to learn about the SwiftUI framework. In the next few chapters you will learn about SwiftUI, Apple's new declarative framework. You will also learn about state management in SwiftUI, which is an integral subject when building SwiftUI applications.

Hello SwiftUI

At WWDC 2019 Apple announced SwiftUI framework. SwiftUI is a declarative framework for building user interfaces for all Apple devices. This means that instead of using Storyboards or dynamically creating your controls, you can use SwiftUI's fluent declarative syntax to build iOS interfaces.

If you have ever worked with React Native or Flutter then you will find a lot of similarities in SwiftUI framework. One of the main features of SwiftUI framework is Xcode previews, which enables the interface to update in real-time as you implement it. This feature is similar to **Hot Reload** in React Native and Flutter.

Getting Started

In order to build SwiftUI apps you need to have Xcode 11 Beta installed. Although Xcode previews feature is only available on macOS Catalina, we can still see live preview feature using Playgrounds on macOS Mojave. **For the most part of this book, we will be using macOS Catalina.**

Running SwiftUI on macOS Mojave

Apart from this small section this book explicitly uses macOS Catalina. But it would be beneficial to see how you can use macOS Mojave to run SwiftUI applications. In order to get a preview of our UI we will be using Xcode 11 Playgrounds. Check out the implementation in **Listing 12** which shows how to preview SwiftUI.

```
1. import UIKit
2. import PlaygroundSupport
3. import SwiftUI
4.
5. struct ContentView: View {
6.     var body: some View {
7.         Text("Hello SwiftUI from macOS Mojave")
8.     }
9. }
10.
11. let contentView = ContentView()
12. PlaygroundPage.current.liveView =
```

```
UIHostingController(rootView: contentView)
```

Listing 12: Preview in macOS Mojave

The most important part of the above code is the **UIHostingController**. UIHostingController makes it possible to host SwiftUI views in your applications. As mentioned earlier that although you can use SwiftUI framework in macOS Mojave running Xcode 11 but it is recommended that you use macOS Catalina since you will get the support of Xcode live previews.

Running SwiftUI on macOS Catalina

The main advantage of using Xcode 11 on macOS Catalina is the power of Xcode previews, which allows us to visualize user interface instantly. **From this point onwards we will be using macOS Catalina for the remainder of the book.**

Launch Xcode 11 and create a new Single View Application. Make sure to select “**SwiftUI**” as shown in **Figure 6**.

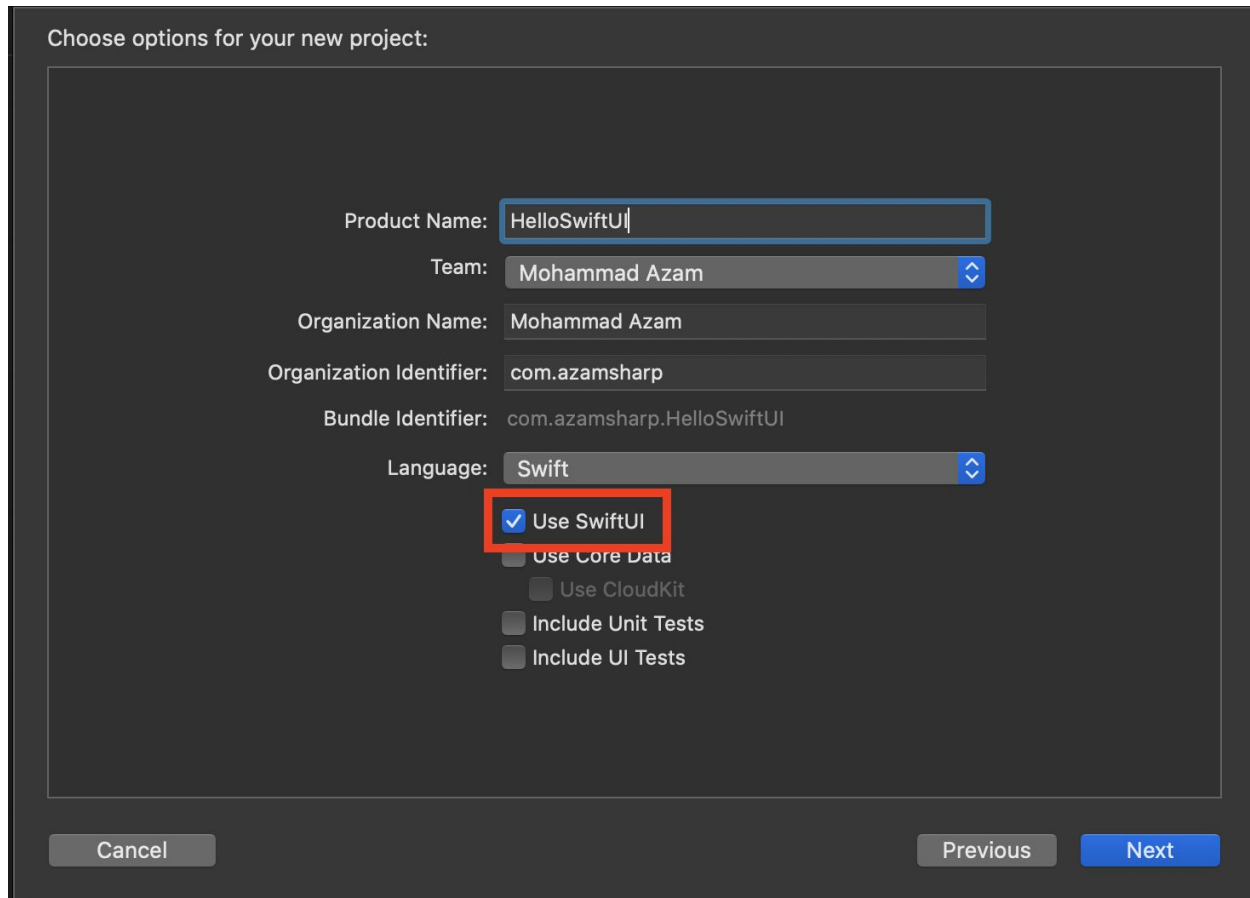


Figure 6: Creating a new SwiftUI project

Press “**Next**” and choose the location to create the project. Xcode 11 will create the project and open up the editor showing multiple panes. **Figure 7** explains the purpose of each pane.



Figure 7: Xcode 11 SwiftUI panes

One thing you will immediately notice is that there are no view controllers. Although you can use view controllers in SwiftUI application, it is recommended that you use MVVM Design Pattern instead. In this book we will not cover MVVM Design Pattern but you can definitely check out my [course](#) to learn more.

Let's try to dissect the code in **Listing 13** and understand how it works.

```

1. struct ContentView : View {
2.     var body: some View {
3.         Text("Hello World")
4.     }
5. }
  
```

Listing 13: Content View SwiftUI

The **ContentView** is the name for your view. It conforms to the protocol View, which only has a single requirement to provide implementation of the body property. The body property returns **some** kind of View. The keyword some indicates that it is an opaque type being returned. This means that even though you are returning a type of View,

compiler will know the exact type. Inside the body property we return a Text view which is responsible for displaying text on the screen. The output is shown in **Figure 8**.



Figure 8: ContentView rendered

Go ahead and change the “Hello World” text in Text view to something different. You will notice that the Xcode preview automatically updates and reflect your changes.

Xcode previews are amazing and going to help speed up the development of your interfaces drastically!

The next piece of code is used to create Xcode previews as shown in **Listing 14**.

```

1. #if DEBUG
2. struct ContentView_Previews : PreviewProvider {
3.     static var previews: some View {
4.         ContentView()
5.     }
6. }
7. #endif

```

Listing 14 - The code to generate the Xcode previews

The **ContentView_Previews** conforms to the **PreviewProvider** which enables to create previews. The only requirement for the PreviewProvider protocol is to provide implementation of static property called **previews**. The previews property is going to return the view whose preview is meant to be created.

Please note that Xcode previews construct the actual view by calling the view in the previews property. It is not an emulation of the view, it is the actual view.

Going back to the implementation of ContentView. If you try to add another Text view then it will give you an error indicating that this is not a valid operation as shown in **Listing 15**.

```

1. struct ContentView : View {
2.     var body: some View {
3.         Text("Hello World")
4.         Text("Bye World") // causes an error
5.     }
6. }

```

Listing 15: Error when trying to return multiple items from body

The reason is that two or more elements cannot be at the same parent level. In order to accommodate multiple childs inside a view we will take help from Stack. Stacks are layout container views which means that their primary purpose is to help with the layout of the view. They are also container views, which means they can contain other child views. In SwiftUI framework, stacks comes in three different flavours. This includes **ZStack**, **HStack** and **VStack**.

HStack is also known as horizontal stack. The main purpose of HStack is to arrange elements horizontally. **Figure 9** shows the stacking order of HStack.



Figure 9: Horizontal stacks

VStack or vertical stack is used to stack items vertically. **Figure 10** shows the stacking order of VStack.

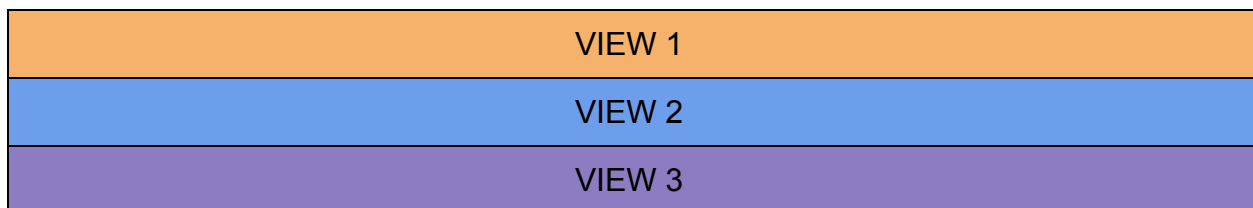


Figure 10: Vertical stacks

Finally, **ZStack** is used to stack items behind or in front of each other. This is demonstrated in **Figure 11**.

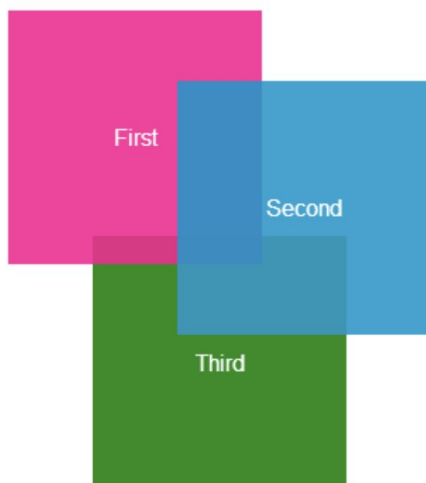


Figure 11: ZStack is used to stack views in front or behind each other

Now, that you have a basic understanding of stacks let's go ahead and use it in our SwiftUI application. **Listing 16** shows how we have stacked two Text views on top of each other.

```
1. struct ContentView : View {  
2.     var body: some View {  
3.         VStack {  
4.             Text("Hello World")  
5.             Text("Bye World")  
6.         }  
7.     }  
8. }
```

Listing 16: Placing views in vertical stack

The output is shown in **Figure 12**.

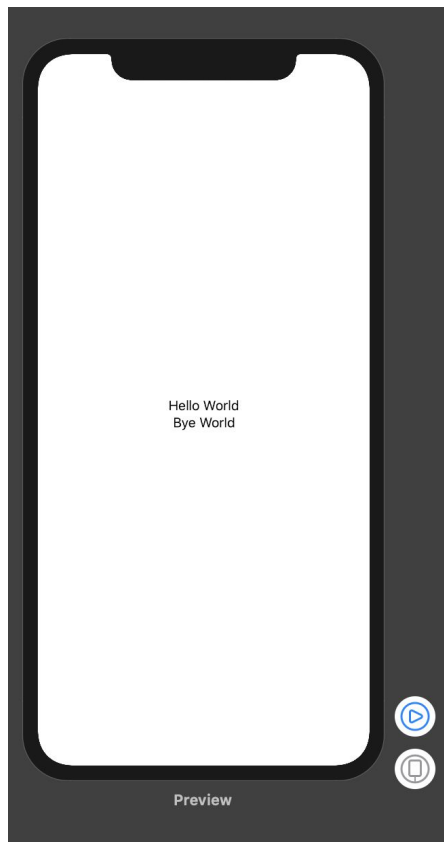


Figure 12: Text views embedded inside in a VStack

You can also command click on a particular view and embed it inside different kinds of controls as shown in **Figure 13**.

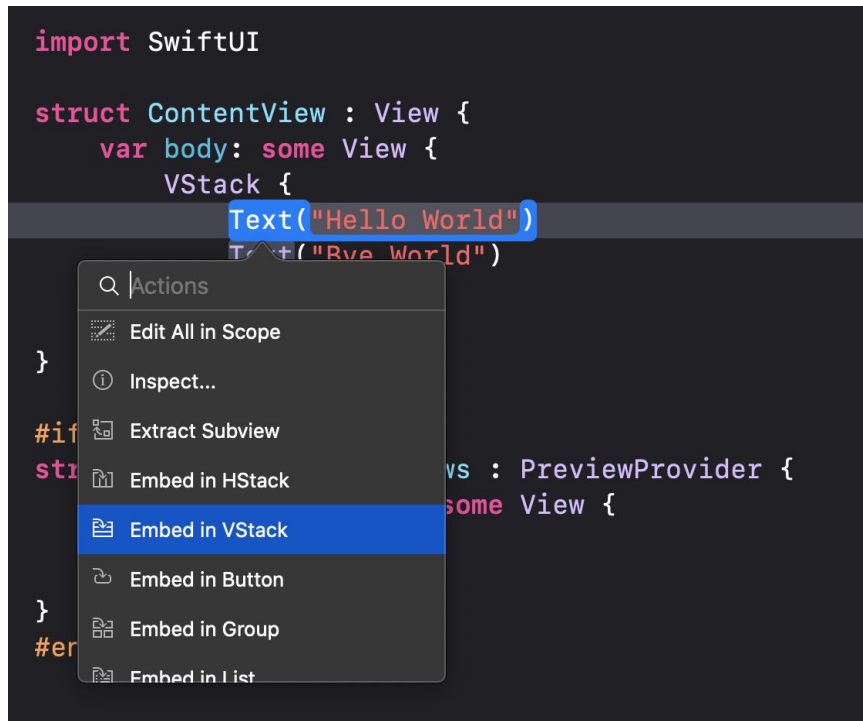


Figure 13: Embedding views in stacks from within Xcode

Similarly, you can use `HStack` to embed elements in horizontal direction as shown in Listing 17.

```

1. struct ContentView : View {
2.     var body: some View {
3.
4.         HStack {
5.             Text("First Item")
6.             Text("Second Item")
7.         }
8.     }
9. }

```

Listing 17: Embedding views inside HStack

The result is shown in **Figure 14**.



Figure 14: Text views embedded inside horizontal stack

ZStack is a little different because it allows to layer items on top of each other. Take a look at **Listing 18**, which shows how to add three Button views stacked on top of each other.

```
1. struct ContentView : View {
2.     var body: some View {
3.
4.         ZStack {
5.
6.             Button("VIEW 1") {
7.
8.             }.padding(100)
9.             .background(Color.orange)
10.
11.             Button("VIEW 2") {
```

```
12.  
13.         }.padding(100)  
14.           .background(Color.blue)  
15.           .offset(y: 50)  
16.  
17.         Button("VIEW 3") {  
18.  
19.         }.padding(100)  
20.           .background(Color.yellow)  
21.           .offset(y: 100)  
22.  
23.  
24.         }  
25.  
26.     }  
27. }
```

Listing 18: Stacking views on top of each other using ZStack

The result is shown in **Figure 15**.

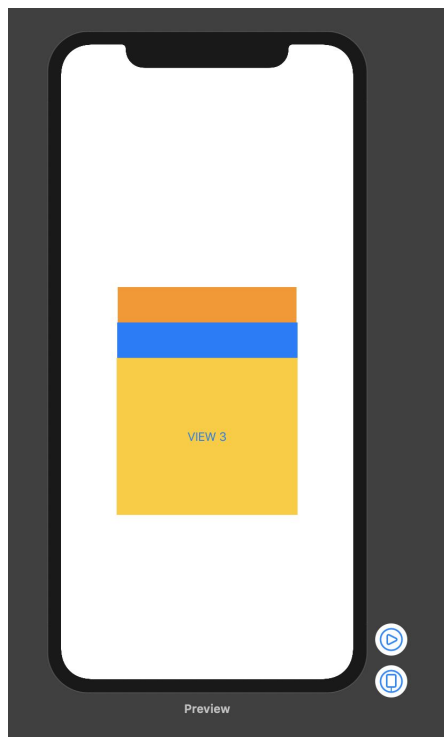


Figure 15: Stacking views on top of each other using ZStack

Stacks serves as an essential layout view in SwiftUI and It allows to structure and align other parts of your app in a seemingly easy way. In the next chapter you will learn about List view, which is used for creating and displaying multiple items in a view. You will also realize how stacks become an important and integrate layout containers for designing and structuring your iOS applications.

Lists and Navigation

One of the most common operations in iOS development is to display a scrolling list of items. In UIKit this was accomplished by using the UITableView control. In SwiftUI, this is performed by List view.

Similar to SwiftUI Stack, List serves as a container view, which means it can have child views. Having worked with UIKit's UITableView control, you will find working with List a great experience. Let's start our journey by displaying a list of numbers in a List.

```
1. struct ContentView: View {  
2.     var body: some View {  
3.  
4.         List(1...10, id: \.self) { index in  
5.             Text("Item Number - \(index)")  
6.         }  
7.     }  
8. }
```

Listing 19: Implementing a simple list to display items based on range

The result is shown in **Figure 16**.

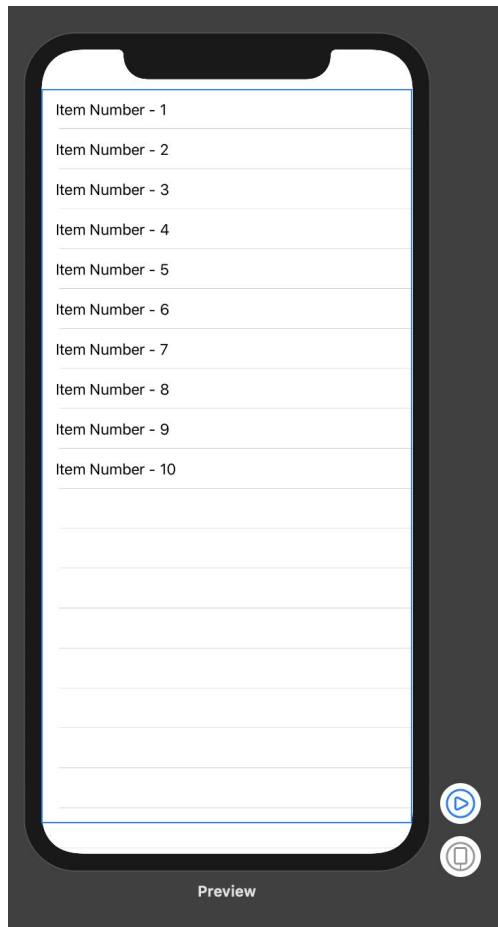


Figure 16: Displaying a simple using List view

Wow! We were able to achieve all that with just a few lines of code without having to setup any delegates or data sources.

The List view requires two arguments, which includes the collection and the keypath to uniquely identify them. For the first argument, we passed a closed range and for the second argument we passed `.\self` to identify each value based on their hash. In the next example let's check out how we can create a List based on custom objects.

Populating List Using Custom Objects

Custom model objects are used to represent the domain of your application. In this example we will populate our List with custom objects.

The first task is to create a class or struct to represent our custom object. Let's create a class called Country which will represent a particular country in the world as shown in **Listing 20**.

```
1. import Foundation
2.
3. struct Country {
4.
5.     let flag: String
6.     let continent: String
7.     let name: String
8.
9. }
```

Listing 20: Implementing of Country model

Since, we are not connected to any database and not consuming any web service we will have to come up with country instances ourselves. Inside the Country struct we have added a static function called “all” which will be responsible for returning hard-coded Country objects as shown in **Listing 21**.

```
extension Country {
    static func all() -> [Country] {
        return [
            Country(flag: "🇵🇰", continent: "Asia", name: "Pakistan"),
            Country(flag: "🇺🇸", continent: "North America", name:
"USA"),
            Country(flag: "🇧🇷", continent: "South America", name:
"Brazil"),
            Country(flag: "🇨🇳", continent: "Asia", name: "Chine")
        ]
    }
}
```

Listing 21: Returning a list of countries

Finally, a good use of emojis ;)

Now, let's jump into ContentView and see how we can iterate through all the countries and display them in a List control. The implementation is shown in **Listing 22**.

```
1. struct ContentView: View {  
2.  
3.     var countries = Country.all()  
4.  
5.     var body: some View {  
6.  
7.         List(self.countries, id: \.name) { country in  
8.             HStack {  
9.                 Text(country.flag)  
10.                Text(country.name)  
11.                Spacer()  
12.                Text(country.continent)  
13.            }  
14.        }  
15.    }  
16. }
```

Listing 22: Displaying all countries in a List

One important thing to note is that we used the key path “**name**” to uniquely identify each country. You can replace “`\\.name`” with “`\\.self`” but then you will have to make sure that your country objects confirms to Hashable protocol.

Inside the List we used the HStack to arrange our items. We also used the Spacer view to add flexible space between the views. **Figure 17** shows the List view populated with countries.

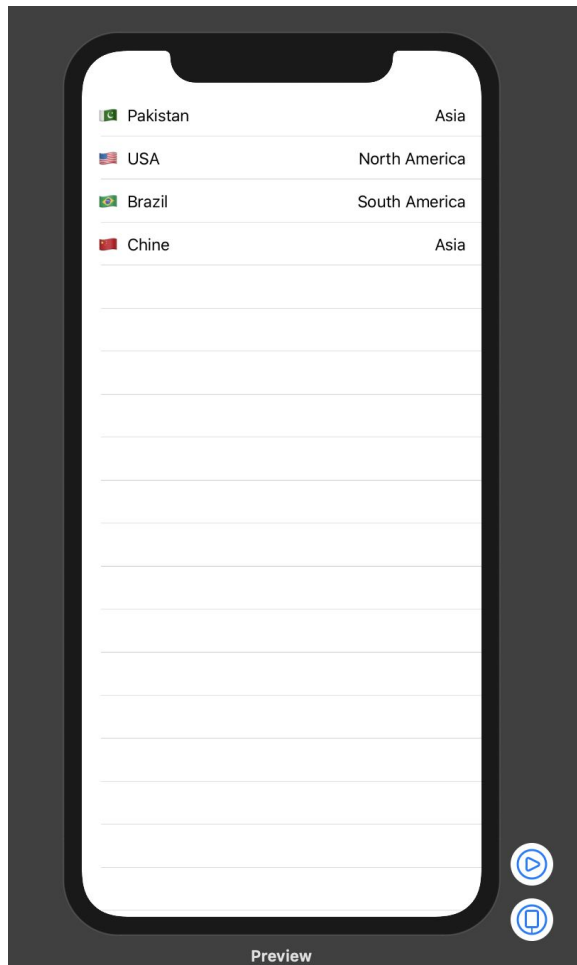


Figure 17: List populated with countries

Sweet!

Although this works great and solves our needs but what if we needed to display a header before the List. This header can be anything but for the sake of simplicity we will display an image for the header. The problem is that List itself cannot accommodate headers and footers on its own.

But we can use a ForEach inside the List, which will help us create views dynamically. ForEach computes the views on demand based on the collection. The implementation is shown in **Listing 23**.

```
1. struct ContentView: View {  
2.  
3.     var countries = Country.all()
```

```
4.
5.     var body: some View {
6.
7.         List {
8.
9.             Image("un").resizable()
10.                .frame(height: 300)
11.
12.             ForEach(self.countries, id: \.name) { country in
13.
14.                 HStack {
15.                     Text(country.flag)
16.                     Text(country.name)
17.                     Spacer()
18.                     Text(country.continent)
19.                 }
20.             }
21.
22.         }
23.     }
24. }
```

Listing 23: ForEach to compute views dynamically based on the collection

The result is shown in **Figure 18**.

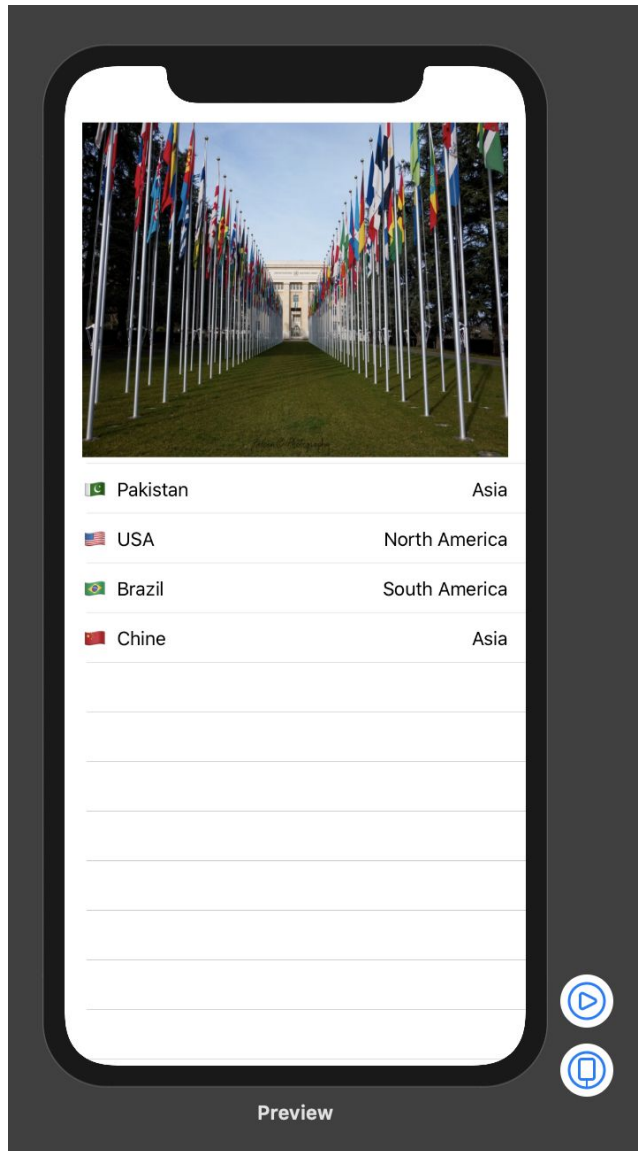


Figure 18: Displaying a header in List using ForEach

Not complicated at all right!

This is a great starting point for our app. Unfortunately most of the apps don't end on one single screen. It would be great if we can tap on the cell and see more details about the country. Let's do that in the next section.

Adding Navigation

SwiftUI uses `NavigationView` to perform navigation in your apps. `NavigationView` is also responsible for setting the title of the view. In most cases `NavigationView` becomes the root view of your application. The code in **Listing 24** shows the implementation of a `NavigationView` along with the title.

```

1. struct ContentView: View {
2.
3.     var countries = Country.all()
4.
5.     var body: some View {
6.
7.         NavigationView {
8.
9.             List {
10.
11.                 Image("un").resizable()
12.                     .frame(height: 300)
13.
14.                 ForEach(self.countries, id: \.name) { country in
15.
16.                     HStack {
17.                         Text(country.flag)
18.                         Text(country.name)
19.                         Spacer()
20.                         Text(country.continent)
21.                     }
22.                 }
23.
24.             }
25.             .navigationBarTitle("Countries")
26.         }
27.     }
28. }
```

Listing 24: NavigationView in SwiftUI

As you can see we added `NavigationView` as the root view of our app. We also added the title for your view by using `.navigationBarTitle`.

Figure 19 shows the result.

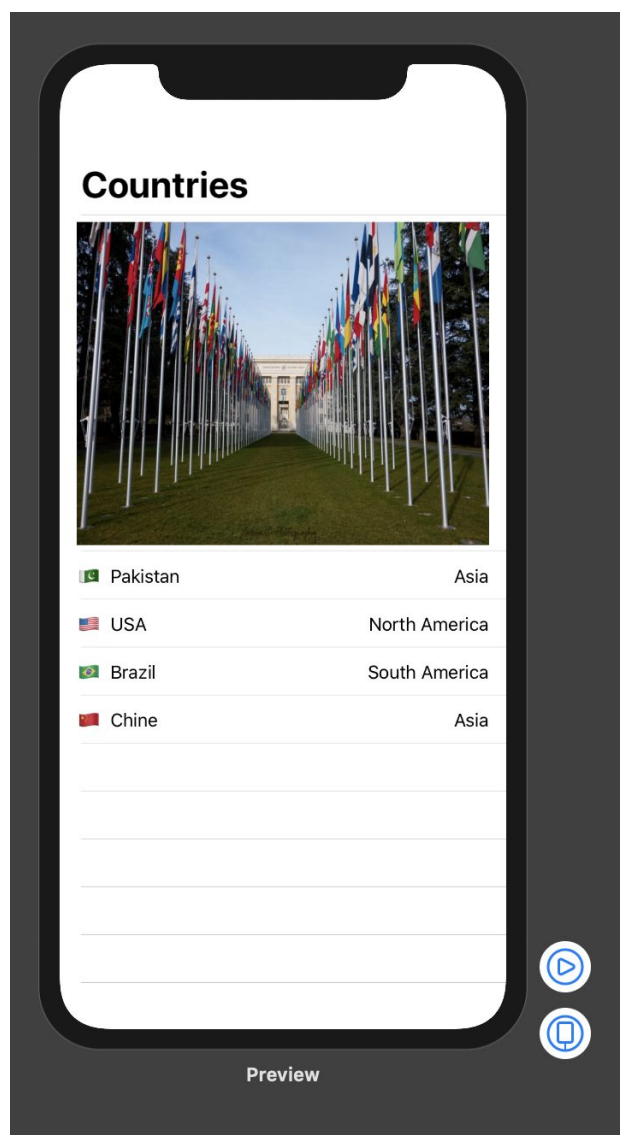


Figure 19: Navigation view with large title

If you are not interested in large titles then you can pass the display mode and set the value inline as shown in **Listing 25**.

```
.navigationBarTitle("Countries", displayMode: .inline)
```

At present we cannot tap on the cells. Before making the cells tappable, let's perform a refactoring of our existing code.

Command click on the HStack and select “**Extract Subview**” option. Extract subview is going to extract a subview and put it underneath the current view. Unfortunately, our application will not compile and return a lot of errors. The reason is simple! The extracted view does not have access to the country object. This can be easily fixed by introducing the country property on the extracted subview. We will also rename our extracted view to “**CountryView**” as shown in **Listing 26**.

```

1. struct ContentView: View {
2.
3.     var countries = Country.all()
4.
5.     var body: some View {
6.
7.         NavigationView {
8.
9.             List {
10.
11.                 Image("un").resizable()
12.                     .frame(height: 300)
13.
14.                 ForEach(self.countries, id: \.name) { country in
15.
16.                     CountryView(country: country)
17.                 }
18.
19.             }
20.             .navigationBarTitle("Countries", displayMode:
21. inline)
22.         }
23.     }
24.
25. struct CountryView: View {
26.
27.     let country: Country
28.
29.     var body: some View {
30.         HStack {
31.             Text(country.flag)
32.             Text(country.name)
33.             Spacer()

```

```

34.         Text(country.continent)
35.     }
36. }
37. }

```

Listing 26: CountryView extracted

Next step is to add navigation to our cells. Simply wrap the HStack with `NavigationLink` so we can tap on the cell and proceed to the destination screen. **Listing 27** shows the implementation of `NavigationLink`.

```

1. struct ContentView: View {
2.
3.     var countries = Country.all()
4.
5.     var body: some View {
6.
7.         NavigationView {
8.
9.         List {
10.
11.             Image("un").resizable()
12.                 .frame(height: 300)
13.
14.             ForEach(self.countries, id: \.name) { country in
15.                 NavigationLink(destination: Text(country.name)) {
16.                     CountryView(country: country)
17.                 }
18.             }
19.
20.         }
21.         .navigationBarTitle("Countries", displayMode:
22.             .inline)
23.     }
24. }

```

Listing 27: NavigationLink to link pages

`NavigationLink` embeds the `CountryView` because we want tap anywhere on the `CountryView` and proceed to the destination. The first argument is the destination, which is simply a `Text` view. The second argument is the `CountryView` which indicates that we need to tap on the `CountryView` to proceed to our destination.

Go ahead and run the application and tap on the cell. You will notice that a push navigation takes place and shows you a destination view with the name of the country. This is great but we can make it even better. Instead of showing a Text view in the destination, we can create a detail screen.

Add a new SwiftUI file to your project and name it “**CountryDetailView**”. The implementation of the detail screen is shown in **Listing 28**.

```

1. import SwiftUI
2.
3. struct CountryDetailView: View {
4.
5.     var country: Country
6.
7.     var body: some View {
8.         VStack {
9.             Text(country.flag)
10.                .font(.custom("Arial", size: 100))
11.
12.             Text(country.name)
13.         }
14.     }
15. }
16.
17. #if DEBUG
18. struct CountryDetailView_Previews: PreviewProvider {
19.     static var previews: some View {
20.         CountryDetailView(country: Country(flag: "🇵🇰",
21.             continent: "Asia", name: "Pakistan"))
22.     }
23. #endif

```

Listing 28: CountryDetailView

The CountryDetailView displays the flag of the country and the name of the country using the Text view. In order to create an instance of CountryDetailView you need to pass the country instance to the view. **Listing 29** shows how to use the CountryDetailView as a destination view from within the ContentView.

```

1. struct ContentView: View {

```

```

2.
3.     var countries = Country.all()
4.
5.     var body: some View {
6.         NavigationView {
7.
8.         List {
9.
10.            Image("un").resizable()
11.                .frame(height: 300)
12.
13.            ForEach(self.countries, id: \.name) { country in
14.                NavigationLink(destination:
15.                    CountryDetailView(country: country)) {
16.                        CountryView(country: country)
17.                    }
18.            }
19.
20.        }
21.        .navigationBarTitle("Countries", displayMode:
22.        .inline)
23.    }
24. }

```

Listing 29: Using CountryDetailView as the destination

Go ahead and run the application again! This time you will notice that when you tap on a cell, you are taken the CountryDetailView which shows the flag of the country along with the name of the country. **Figure 20** shows the CountryDetailView as the destination view.

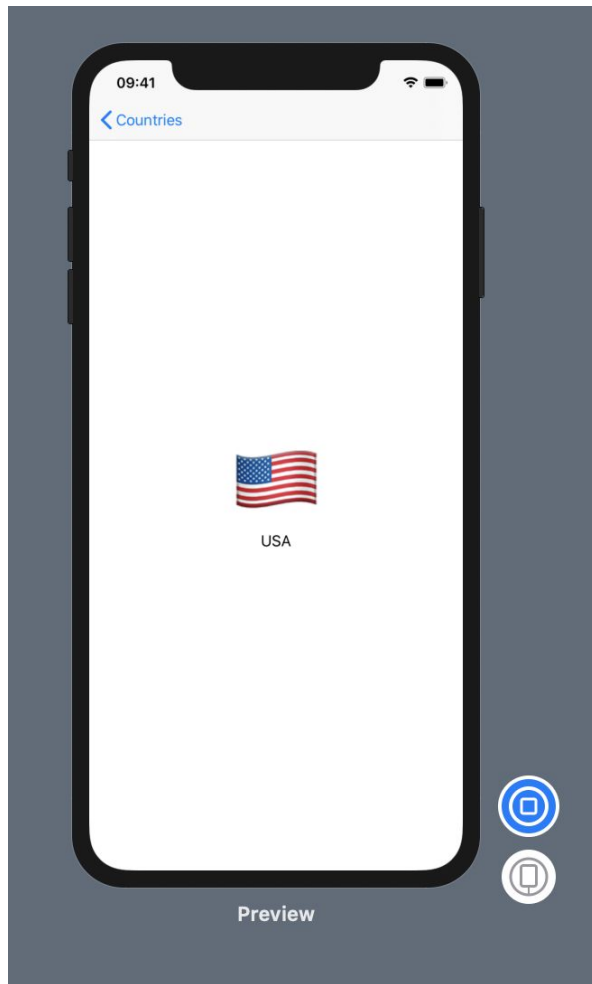


Figure 20: CountryView as the destination view

And that's how simple it is to implement navigations in SwiftUI framework. SwiftUI makes it easy to perform navigation without the need of segues or navigation controllers.

Please note that under the hood SwiftUI does use UINavigationController but all of that complexity is hidden when using SwiftUI framework.

At present we have used controls which can display values on the view. But what about interactive controls. Controls that changes and maintain states. In the next chapter we are going to take a deep dive into the concept of data flow in SwiftUI applications.

State Management in SwiftUI

State represents the data associated with the view. It can be represented by a primitive type like Boolean, String, Int etc or a complex object like a view model. In SwiftUI, changing a state value causes the view to re-render allowing the view to sync with the data.

Similar concepts are available in ReactJS and Flutter, where changing the state causes render and build functions to fire respectively.

Let's take a look at a simple example of state in SwiftUI in **Listing 30**. In this example we will allow the user to toggle a switch. Based on the status of the switch the icon will change between night and day.

```
1. struct ContentView: View {
2.
3.     @State private var isOn: Bool = false
4.
5.     var body: some View {
6.
7.         VStack {
8.
9.             Toggle(isOn: $isOn) {
10.                 Text("")
11.             }.labelsHidden()
12.
13.             Text(self.isOn ? "☀️" :
"🌙").font(.custom("Arial", size: 100))
14.
15.         }
16.
17.     }
18. }
```

Listing 30: Toggling State in SwiftUI

The important part in this example is the use of **@State** property wrapper on the `isOn` boolean property. We have also marked it with `private` keyword indicating that this state is a local/private state for the `ContentView` component.

The `Toggle` view takes in a `Binding<Bool>`, which is satisfied by passing the `isOn` state property. When `Toggle` changes the state between on or off the state value gets updated, rendering the view again.

The result is shown in **Figure 21**:



Figure 21: Toggle states ON and OFF

Great!

Now that you have some basic understanding of state, let's see how we can capture multiple pieces of information using state.

Adding More Variables

The Register view is implemented in **Listing 31**. It includes all the basic elements for registering a new user. To keep this example simple we are using basic SwiftUI elements like TextField etc. In your actual application you can use the power of Form view available in SwiftUI framework.

```

1. struct Register: View {
2.
3.     @State private var firstName: String = ""
4.     @State private var lastName: String = ""
5.     @State private var username: String = ""
6.     @State private var password: String = ""
7.
8.
9.     var body: some View {
10.
11.         NavigationView {
12.
13.             VStack {
14.                 TextField("First Name", text:
self.$firstName)
15.                 TextField("Last Name", text:
self.$lastName)
16.                 TextField("User Name", text:
self.$username)
17.                 SecureField("Password", text:
self.$password)
18.                 Button("Register") {
19.                     // register the user
20.                 }
21.
22.             }.padding()
23.
24.             .navigationBarTitle("Register")
25.         }
26.
27.     }
28. }

```

Listing 31: Register view

The state of the Register view is controlled by four independent variables. Each variable captures a slice of the state. As the user types in the TextFields, the state variables gets updated.

This mostly works great! But we have to deal with four independent variables, which represent a single model. This can be simplified by introducing a view model which represents the state of the view.

SwiftUI and MVVM

Although, you can use any design pattern to build your SwiftUI application but the recommended pattern is [Presentation Model](#), also known as MVVM. In MVVM design pattern you will start by creating a view model which will be responsible for providing data to the view and managing the state associated with the view.

The implementation of RegistrationViewModel is shown in **Listing 32**.

```
1. struct RegistrationViewModel {  
2.  
3.     var firstName: String = ""  
4.     var lastName: String = ""  
5.     var username: String = ""  
6.     var password: String = ""  
7.  
8. }
```

Listing 32: RegistrationViewModel

Now, we can update the Register view and utilize our newly created RegistrationViewModel as shown in Listing 32.1.

```

1. import SwiftUI
2.
3. struct RegistrationViewModel {
4.     var firstName: String = ""
5.     var lastName: String = ""
6.     var username: String = ""
7.     var password: String = ""
8. }
9.
10.
11. struct Register: View {
12.
13.     @State private var registrationVM: RegistrationViewModel
    = RegistrationViewModel()
14.
15.     var body: some View {
16.
17.         NavigationView {
18.
19.             VStack {
20.                 TextField("First Name", text:
self.$registrationVM.firstName)
21.                 TextField("Last Name", text:
self.$registrationVM.lastName)
22.                 TextField("User Name", text:
self.$registrationVM.username)
23.                 SecureField("Password", text:
self.$registrationVM.password)
24.                 Button("Register") {
25.                     // register the user
26.                 }
27.
28.             }.padding()
29.
30.             .navigationBarTitle("Register")
31.         }
32.
33.     }
34. }

```

Listing 32.1: RegistrationViewModel used by Register view

As you can see our Register view is now much simpler and the RegistrationViewModel is responsible for maintaining the state of the view. We have

removed the individual slices of state variables and replaced it with a single view model. Now anytime a user updates any of the `TextField`, the view model gets updated automatically. The MVVM design pattern allows to cleanly structure your SwiftUI application and at the same time making it easier to write unit tests.

But what if you wanted to change the state of the parent from a child view. This is performed by `@Binding` property wrapper, which is explained in the next section.

@Binding

A single view in SwiftUI may be composed of multiple child views. Sometimes you want to allow the child to change the state of the parent view. Binding allows you to pass state from the parent view to the child view. Once the child view alters the state, the parent automatically gets the updated copy and re-renders the view. Let's implement the same day/night example as before but this time we will put the Toggle view into a child view called "DayNightView".

The implementation of DayNightView is shown in **Listing 33**.

```
1. import SwiftUI
2.
3. struct DayNightView: View {
4.
5.     @Binding var isOn: Bool
6.
7.     var body: some View {
8.         Toggle(isOn: $isOn) {
9.             Text("")
10.            }.labelsHidden()
11.        }
12.    }
13.
14. struct DayNightView_Previews: PreviewProvider {
15.     static var previews: some View {
16.         DayNightView(isOn: .constant(false))
17.     }
18. }
```

Listing 33: DayNightView using @Binding

The property wrapper `@Binding` indicates that the `isOn` property will be passed to the `DayNightView`. Once the `DayNightView` changes the `isOn` property then the original sender will be notified by re-rendering the view.

Assigning the `isOn` property will cause render to get called on the parent view

The parent view's implementation is shown in **Listing 34**:

```

1. struct ContentView: View {
2.
3.     @State private var isOn: Bool = false
4.
5.     var body: some View {
6.
7.         return VStack {
8.
9.             Text(self.isOn ? "☀️" : "🌙")
10.                .font(.custom("Arial", size: 100))
11.
12.             DayNightView(isOn: $isOn)
13.
14.         }
15.
16.     }
17. }
```

Listing 34: Updating parent view through `@Binding`

`DayNightView` will be responsible for displaying the Toggle switch. `DayNightView` takes a binding as an argument. We have passed the `@State` property `isOn` to the `DayNightView`. This means when the `DayNightView` updates the bindable property, state property `isOn` in the parent view also gets updated.

The main purpose of `@Binding` is to pass the state to a child view where it can be updated. This gives child view(s) an opportunity to communicate with the parent and update the parent.

`@Observable` and `@Observed`

Most of the apps fetch their data from an outside source, mainly using JSON Web API. Once the data is downloaded it is populated in a DTO (Data Transfer Object) and later mapped to the view models and then displayed on the screen.

One common issue with consuming asynchronous requests is to notify the user interface that data has been downloaded so the view can display fresh data. SwiftUI solves this problem by introducing `Observable` and `Observed` property wrappers.

Before jumping into `Observable` and `Observed` property wrappers we must find a way to perform asynchronous requests to fetch data. For the sake of simplicity we are going to make a fake request and get a list of posts in an asynchronous manner as shown in **Listing 35**.

```
1. struct Post {
2.     let id = UUID().uuidString
3.     let title: String
4.     let body: String
5. }
6.
7. class Webservice {
8.
9.     func fetchPosts(completion: @escaping ([Post]) -> Void) {
10.
11.         DispatchQueue.main.asyncAfter(deadline: .now() + 2.0)
12.         {
13.             // fetch from a web api and then populate the
14.             Post array
15.             let posts = [
16.                 Post(title: "Hello SwiftUI", body: "Learn to
17.                 create your first SwiftUI App!"),
18.                 Post(title: "Getting started with Combine",
19.                 body: "Introduce reactive programming using Combine framework")
20.             ]
21.             completion(posts)
22.         }
23.     }
24. }
```

Listing 35: Webservice returning fake posts

The Webservice simply waits for 2 seconds and then sends hardcoded list of posts back to the user in a completion handler.

The Webservice class is used by PostListViewModel to perform the request. The implementation of PostListViewModel is shown in **Listing 36**.

```

1. class PostListViewModel: ObservableObject {
2.
3.     let webservice = Webservice()
4.     @Published var posts = [PostViewModel]()
5.
6.     func fetchPosts() {
7.         self.webservice.fetchPosts { posts in
8.             self.posts = posts.map(PostViewModel.init)
9.         }
10.    }
11.
12. }
13.
14. struct PostViewModel {
15.
16.     let id = UUID().uuidString
17.     let post: Post
18.
19.     var title: String {
20.         return self.post.title
21.     }
22.
23.     var body: String {
24.         return self.post.body
25.     }
26.
27. }
```

Listing 36: Calling webservice through view model

The PostListViewModel represents the data, which will be displayed on the post listing screen. The most important thing to notice is the use of ObservableObject protocol.

The ObservableObject protocol allows the class to publish events. The posts property

is also decorated with `@Published` property wrapper, which means it acts like a publisher. When a value is assigned to the `posts` property, it publishes an event indicating that it has been changed.

Finally, the `PostListView` uses the `PostListViewModel` to fetch and display the posts in a view as implemented in **Listing 37**:

```

1. struct PostListView: View {
2.
3.     @ObservedObject private var postListVM = PostListViewModel()
4.
5.     init() {
6.         self.postListVM.fetchPost()
7.     }
8.
9.     var body: some View {
10.         List(postListVM.posts, id: \.id) { post in
11.             VStack(alignment: .leading, spacing: 10) {
12.                 Text(post.title).font(.headline)
13.                 Text(post.body)
14.             }
15.         }
16.     }
17.
18.     struct PostListView_Previews: PreviewProvider {
19.         static var previews: some View {
20.             PostListView()
21.         }
22.     }
23. }

```

Listing 37: Displaying posts in PostListView

If you run the application, `PostListView` is going to use `PostListViewModel` and populate the list of posts in a `List` view. The MVVM design pattern along with the ability to publish and notify the changes makes syncing the view with the view model much easier.

At present we have only discussed local state, which represents the state maintained and available to a particular view. If you need to change the state of the view from another view you can pass it as an argument and using the `@Binding` property wrapper. This works great if you are passing the state between few views but quickly become a

hassle when several views are involved or when you need to pass the state to a deeply nested view in the hierarchy.

In the next section we are going to look at global state, which can be accessible from any SwiftUI view.

@EnvironmentObject

The concept of an `EnvironmentObject` is very similar to **Redux**. The main purpose of an `EnvironmentObject` is to maintain global state. Global state is a state that can be accessed from any view. The `EnvironmentObject` is usually injected at the top level view making the global state to be available to all child views.

To keep the example simple we are going to create a class called `UserSettings`, which will be shared between multiple views. We will implement three different views namely Facebook, Twitter and `TotalLikes`. The Facebook and Twitter views will allow the user to increment likes and `TotalLikes` view will be responsible for displaying the total likes.

The implementation of `UserSettings` class is shown in **Listing 38**.

```
1. import Foundation
2.
3. class UserSettings: ObservableObject {
4.     @Published var likes: Int = 0
5. }
```

Listing 38: Implementation of `UserSettings` object

The `UserSettings` class is using the `ObservableObject` protocol, which means it can publish events. The only property in `UserSettings` class is `likes` which is decorated with `@Published` property wrapper indicating that it will act as a publisher and will notify the subscribers when the value changes.

Before using `UserSettings`, we need to inject it into the parent view. Open `SceneDelegate.swift` and implement the code shown in **Listing 39**.

```
1. if let windowScene = scene as? UIWindowScene {
```

```

2.         let window = UIWindow(windowScene: windowScene)
3.         let userSettings = UserSettings()
4.         let contentView =
ContentView().environmentObject(userSettings)
5.         window.rootViewController =
UIHostingController(rootView: contentView)
6.         self.window = window
7.         window.makeKeyAndVisible()
8.     }

```

Listing 39: Injecting global object in ContentView

Once injected in the ContentView, the UserSettings object will be available to the ContentView and all the views inside ContentView. Next we will implement Facebook and Twitter view as shown in **Listing 40**.

```

1. import SwiftUI
2.
3. struct Facebook: View {
4.
5.     @EnvironmentObject var userSettings: UserSettings
6.
7.     var body: some View {
8.
9.         VStack {
10.            Text("Facebook")
11.            Button("👍") {
12.                self.userSettings.likes += 1
13.            }
14.        }
15.    }
16. }
17.
18. struct Twitter: View {
19.
20.     @EnvironmentObject var userSettings: UserSettings
21.
22.     var body: some View {
23.
24.         VStack {
25.            Text("Twitter")
26.            Button("👍") {
27.                self.userSettings.likes += 1

```

```
28.         }  
29.     }  
30. }  
31. }
```

Listing 40: Implementation of Twitter and Facebook view

The interesting thing to note is the usage of `@EnvironmentObject` property wrapper. The `userSettings` instance will be automatically populated from the parent view. When you increment the `likes` property then it will be incremented globally for all the views interested in the `UserSettings` global state.

The updates to the global state will also cause the views to render again automatically

The `TotalLikes` view is responsible for displaying the value of `likes` property.

Finally, the Twitter and Facebook views are used inside the `ContentView` as shown in **Figure 22**.

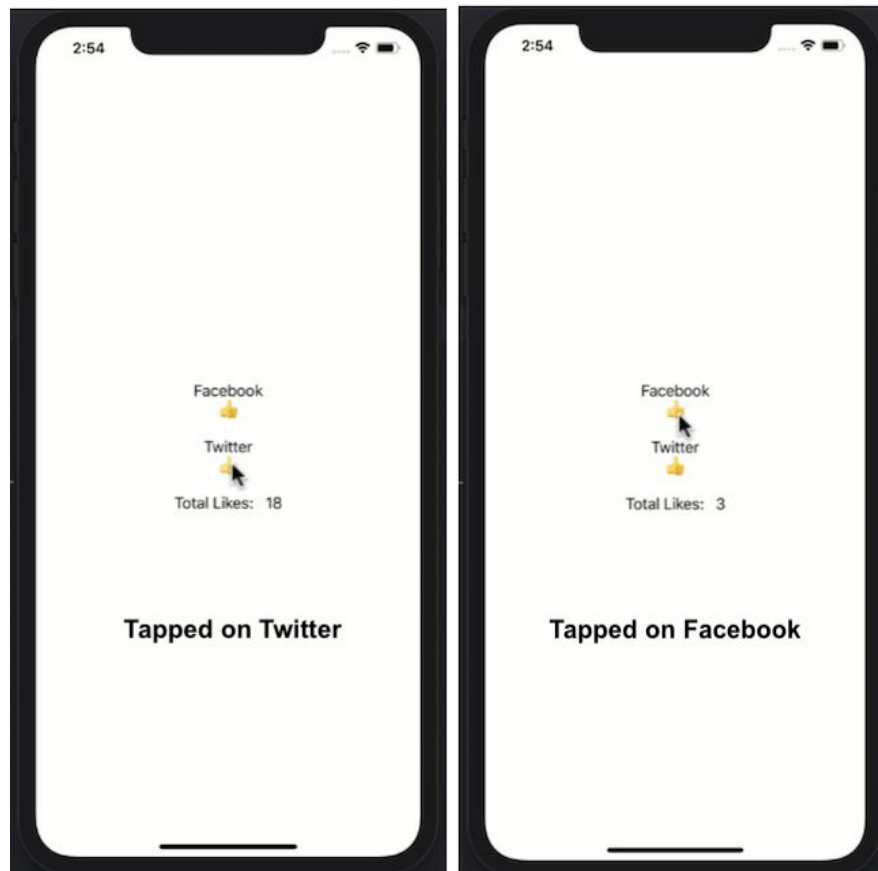


Figure 22: Global state updated from different views

Awesome!

You were able to share the state between multiple views by using global state in SwiftUI. `EnvironmentObject` is ideal when you want to share data with multiple views, especially if those views are nested deep into the view hierarchy.

Getting Started with SwiftUI and MVVM

In the previous chapters you learned about the MVVM Design Pattern, SwiftUI and state management in SwiftUI. Now it is time to look at how MVVM pattern can be integrated with a SwiftUI application. In this chapter we will start with a very basic counter application using SwiftUI and MVVM Design Pattern. The point of this chapter is to get comfortable with different components of MVVM and how it fits with the SwiftUI framework.

View

The view for our counter app is pretty simple. It consists of a single Text label and two Button views. One button is to increment the counter and the other is to decrement the counter. The complete code for the view is shown in **Listing 41**.

```
1. struct ContentView: View {  
2.  
3.     var body: some View {  
4.         VStack {  
5.             Text("Counter will be displayed here")  
6.  
7.             HStack {  
8.                 Button("Increment") {  
9.  
10.                     }  
11.                 Button("Decrement") {  
12.  
13.                     }  
14.             }  
15.         }  
16.     }  
17. }
```

Listing 41: The SwiftUI code for counter view

ContentView when rendered is shown in **Figure 23**.

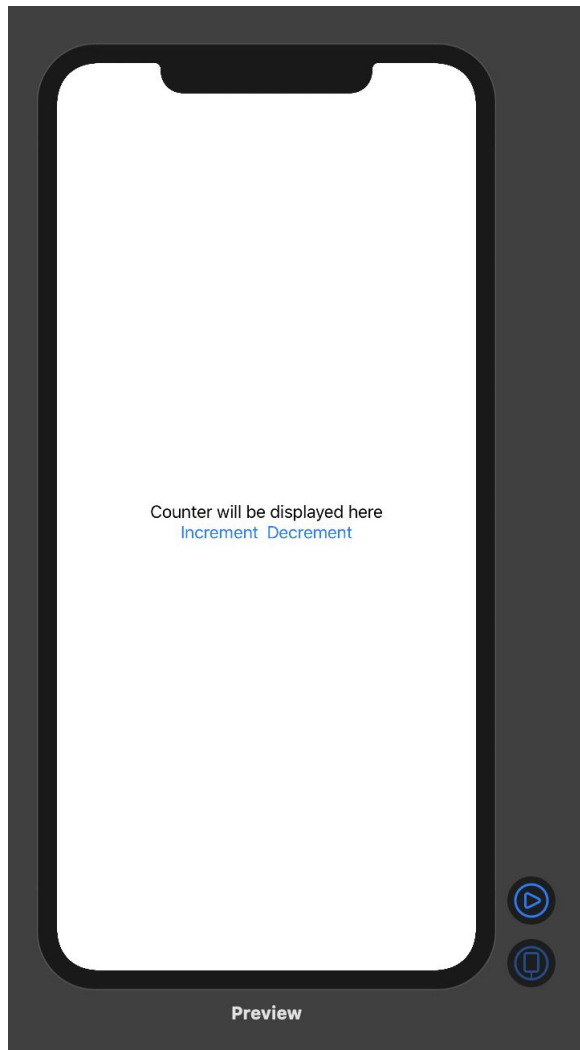


Figure 23: ContentView

Next step is to create a model, which will be responsible for providing the updated counter value.

Implementing Counter Model

The Counter model consists of separate functions for increment and decrement operations. The value property represents the updated/current value of the counter. The complete implementation of the Counter model is shown in **Listing 42**.


```

1. import Foundation
2.
3. struct Counter {
4.
5.     var value: Int
6.
7.     mutating func increment() {
8.         value += 1
9.     }
10.
11.     mutating func decrement() {
12.         value -= 1
13.     }
14.
15.
16. }

```

Listing 42: Counter model implementation

If you want to add any business rules that needs to validate the counter value then you can add those in the increment and decrement functions. The Counter model will be used by the CounterViewModel, which will be responsible for updating the values on the view.

Implementing CounterViewModel

The primary task of CounterViewModel is to provide updated data to the view. CounterViewModel will also expose increment and decrement functions which can be called from the view.

The implementation of CounterViewModel is shown in **Listing 43**.

```

1. import Foundation
2.
3. class CounterViewModel: ObservableObject {
4.
5.     @Published var counter: Counter = Counter(value: 0)
6.
7.     var value: Int {

```

```

8.         return self.counter.value
9.     }
10.
11.     func increment() {
12.         self.counter.increment()
13.     }
14.
15.     func decrement() {
16.         self.counter.decrement()
17.     }
18. }

```

Listing 43: Implementation of CounterViewModel

There are a couple of interesting things about CounterViewModel. First, it is decorated with ObservableObject protocol. The ObservableObject protocol is new in iOS 13 and it allows an object to dispatch events. This means other parts of the code can observe instances of CounterViewModel. The counter object is marked with @Published property wrapper, which means that anytime the counter value is altered, it will publish an event. That event can be handled by the view to keep the user interface in-sync with the data.

The view has been updated to make use of the CounterViewModel as shown in **Listing 44**.

```

1. struct ContentView: View {
2.
3.     @ObservedObject private var counterVM: CounterViewModel =
CounterViewModel()
4.
5.     var body: some View {
6.         VStack {
7.             Text("\(self.counterVM.value)")
8.
9.             HStack {
10.                 Button("Increment") {
11.                     self.counterVM.increment()
12.                 }
13.                 Button("Decrement") {
14.                     self.counterVM.decrement()

```

```
15.         }  
16.     }  
17. }  
18. }  
19. }
```

Listing 44: ContentView Using CounterViewModel

The first thing to note is the `@ObservedObject` property wrapper. The `@ObservedObject` property wrapper is going to make sure that when the `CounterViewModel` publishes an event then the `ObservedObject` gets the updated value. This also causes the view to be rendered again, thus giving you an opportunity to update and sync the view with the underlying data.

Once the user clicks the increment or decrement button, it updates the counter value in `CounterViewModel`. Since the counter property is marked with `@Published`, it publishes an event which is handled in the `ContentView`. Finally, the `Text` view displays the updated value of the counter. The result is shown in **Figure 24**:

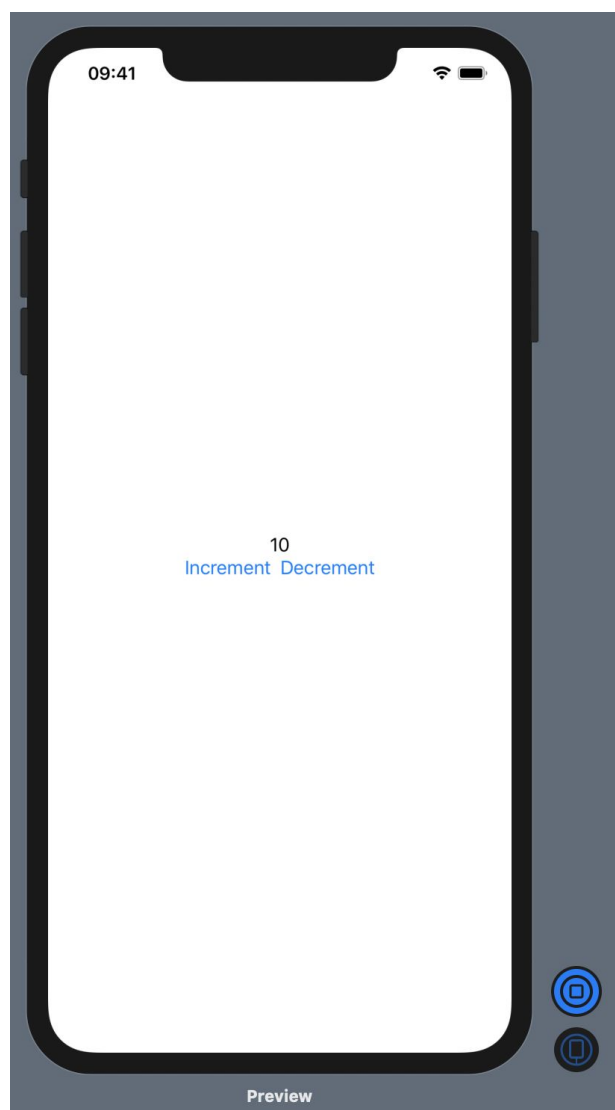


Figure 24: Counter value updated

In this chapter, you learned how to implement MVVM design patterns in a simple SwiftUI application. In the next chapter, we are going to combine everything we have learned to build a real world Notes application using SwiftUI, MVVM and Networking.