

```

import SwiftUI
import StreamChat
import FirebaseAuth
import JWTKit

class LoginViewModel: ObservableObject {

    // Existing properties...
    @Published var countryCode = ""
    @Published var phoneNumber = ""
    @Published var showAlert = false
    @Published var errorMsg = ""
    @Published var ID = ""
    @Published var isLoading = false
    @AppStorage("log_Status") var logStatus = false
    @AppStorage("userName") var storedUser = ""
    @Published var newUser = false

    // JWTSigners instance
    let signers = JWTSigners()

    init() {
        // Initialize the ChatClient here
        let config = ChatClientConfig(apiKey: .init(APIKey))
        ChatClient.shared = ChatClient(config: config)
        // Set up signers (if required)
    }

    func verifyUser() {
        // ... existing verification logic ...

        withAnimation{isLoading = true}

        // Undo this if testing with real devices or real ph Numbers...
        Auth.auth().settings?.isAppVerificationDisabledForTesting = false

        // Sending Otp And Verifying user...
        PhoneAuthProvider.provider().verifyPhoneNumber("+\(countryCode + phoneNumber)", uiDelegate: nil) { ID, err in

            if let error = err{
                self.errorMsg = error.localizedDescription
                self.showAlert.toggle()
                return
            }

            self.ID = ID!
            self.alertWithTF()
        }
    }

    func alertWithTF() {
        // ... existing alert logic ...

        let alert = UIAlertController(title: "Verification", message: "Enter OTP Code", preferredStyle: .alert)

        alert.addTextField { txt in
            txt.placeholder = "123456"
        }

        alert.addAction(UIAlertAction(title: "Cancel", style: .destructive, handler: nil))
        alert.addAction(UIAlertAction(title: "Ok", style: .default, handler: { _ in

            if let code = alert.textFields?[0].text{
                self.LoginUser(code: code)
            }
            else{

```

```

        self.reportError()
    }
}))

// presenting Alert View...
UIApplication.shared.windows.first?.rootViewController?.present(alert, animated: true, completion: nil)
}

func loginUser(code: String) {
    let credential = PhoneAuthProvider.provider().credential(withVerificationID: self.ID, verificationCode: code)

    Auth.auth().signIn(with: credential) { result, err in
        if let error = err {
            self.errorMessage = error.localizedDescription
            self.showAlert.toggle()
            return
        }

        guard let uid = Auth.auth().currentUser?.uid else {
            self.reportError()
            return
        }

        let payload = Payload(user_id: uid)

        do {
            let signers = JWTSigners()
            signers.use(.hs256(key: secretKey.data(using: .utf8)!))

            let jwt = try signers.sign(payload)

            let config = ChatClientConfig(apiKey: .init(APIKey))
            ChatClient.shared = ChatClient(config: config)

            let tokenProvider: TokenProvider = { completion in
                do {
                    let newJWT = try signers.sign(payload)
                    let token = try Token(rawValue: newJWT)
                    completion(.success(token))
                } catch {
                    print("Error creating token: \(error)")
                    completion(.failure(error))
                }
            }

            ChatClient.shared.connectUser(userInfo: .init(id: uid), tokenProvider: tokenProvider) { error in
                if let error = error {
                    print("Connection failed with: \(error)")
                    self.reportError()
                } else {
                    // User successfully connected
                    self.checkUserStatus()
                }
            }

        } catch {
            print(error.localizedDescription)
        }
    }
}

func checkUserStatus() {
    // Existing logic to check user status
    if let name = ChatClient.shared.currentUserController().currentUser?.name {
        withAnimation {

```

```
        self.storedUser = name
        self.logStatus = true
        self.isLoading = false
    }
} else {
    withAnimation {
        self.newUser = true
        self.isLoading = false
    }
}
}

func reportError() {
    // ... existing error reporting logic ...
    print ("Errorrrrr")
}
}

struct Payload: JWPayload, Equatable {
    var user_id: String

    func verify(using signer: JWTSigner) throws {
        // Verification logic if needed
    }
}
```

please explain this code bit by bit, in detail and super easy to understand

This Swift code defines a class `LoginViewModel` which acts as the view model for a login feature in a SwiftUI app. The app seems to be using Firebase for authentication and Stream Chat for chat functionality. Let's break down the key parts of the code:

Class Definition & Properties

```
```swift
class LoginViewModel: ObservableObject {
 // Published Properties...
}
```
```

`LoginViewModel` is a class that conforms to the `ObservableObject` protocol. This means instances of this class can be observed for changes, which is useful for updating the UI in SwiftUI.

Properties

Inside the class, several properties are defined with `@Published` and `@AppStorage` wrappers, which are used to observe and persist changes, respectively.

Initializer

```
```swift
init() {
 let config = ChatClientConfig(apiKey: .init(APIKey))
 ChatClient.shared = ChatClient(config: config)
}
```
```

When an instance of `LoginViewModel` is created, the initializer sets up the shared `ChatClient` with a configuration.

verifyUser() Function

```
```swift
func verifyUser() {
 // Sends OTP and Verifies user...
}
```
```

This function sends an OTP (One-Time Password) to the user's phone number and verifies it using Firebase Authentication. If an error occurs, it updates the `errorMsg` property and shows an alert.

alertWithTF() Function

```
```swift
func alertWithTF() {
 // Presents an alert with a text field for OTP code...
}
```
```

This function presents an alert with a text field where the user can enter the received OTP code. If the user presses "Ok", it calls the `loginUser(code: String)` function with the entered code.

loginUser(code: String) Function

```
```swift
func loginUser(code: String) {
 // Signs in the user with the received credential...
}
```
```

This function signs in the user using the received OTP code. If successful, it creates a JWT (JSON Web Token) for the user, initializes the `ChatClient` with this token, and connects the user to the Stream Chat.

checkUserStatus() Function

```
```swift
func checkUserStatus() {
 // Checks the user's status...
}
```
```

This function checks if the user is already signed up in the chat system. If the user has a name, it updates the `storedUser` and `logStatus` properties and stops loading. If not, it sets the `newUser` property to true and stops loading.

```
### reportError() Function
```swift
func reportError() {
 print ("Errorrrrr")
}
```
```

This function is a placeholder to handle error reporting. Currently, it just prints an error message to the console.

```
### Payload Struct
```swift
struct Payload: JWTPayload, Equatable {
 var user_id: String

 func verify(using signer: JWTSigner) throws {}
}
```
```

This struct represents the payload of a JWT token. It conforms to the `JWTPayload` protocol and contains a `user_id` property.

In summary, this `LoginViewModel` class provides functionalities to verify the user's phone number through OTP, log in the user, and connect them to the Stream Chat using a JWT token. The properties marked with `@Published` and `@AppStorage` are used to reflect changes in the UI and store user data, respectively.

more in detail, especially the auth and token part

Sure, let's go into more detail on the authentication and token parts.

1. Firebase Phone Authentication

Firebase Phone Authentication is being used to verify the user's phone number by sending an OTP (One-Time Password).

```
```swift
PhoneAuthProvider.provider().verifyPhoneNumber("+\\(countryCode + phNumber)", uiDelegate: nil) { ID, err in
 // ... handle the result ...
}
```
```

Here, `PhoneAuthProvider` sends an OTP to the provided phone number. When the OTP is received, it is expected to be input by the user in the presented alert. This OTP is then used to create a credential object:

```
```swift
let credential = PhoneAuthProvider.provider().credential(withVerificationID: self.ID, verificationCode: code)
```
```

This credential is used to sign in the user using Firebase Authentication:

```
```swift
Auth.auth().signIn(with: credential) { result, err in
 // ... handle the sign-in result ...
}
```
```

Once the user is signed in, their unique Firebase `uid` is retrieved:

```
```swift
guard let uid = Auth.auth().currentUser?.uid else {
 self.reportError()
 return
}
```
```

2. JWT Token Generation

Once the user is successfully authenticated, a JWT (JSON Web Token) is created for the user to authenticate with the Stream Chat service.

```
```swift
let payload = Payload(user_id: uid)
```
```

A `Payload` struct is created, which contains the user's Firebase `uid` as the `user_id`.

Then, a `JWTSigners` object is created and a signing method is used:

```
```swift
let signers = JWTSigners()
signers.use(.hs256(key: secretKey.data(using: .utf8)!))
```
```

This specifies that the JWT should be signed using the HS256 algorithm and a `secretKey`.

Next, the JWT is signed with the payload:

```
```swift
let jwt = try signers.sign(payload)
```
```

This JWT token (`jwt`) will be used to authenticate the user with the Stream Chat service.

3. Connecting to Stream Chat

Once the JWT is created, the user can be connected to the Stream Chat service:

```
```swift
ChatClient.shared.connectUser(userInfo: .init(id: uid), tokenProvider: tokenProvider) { error in
 // ... handle the connection result ...
}
```
```

Here, `ChatClient.shared.connectUser` is used to connect the user to Stream Chat. The `userInfo` parameter is initialized with the user's Firebase `uid`, and the `tokenProvider` is a closure that generates a new JWT token when needed.

The `tokenProvider` looks like this:

```
```swift
let tokenProvider: TokenProvider = { completion in
 do {
 let newJWT = try signers.sign(payload)
 let token = try Token(rawValue: newJWT)
 completion(.success(token))
 } catch {
 print("Error creating token: \(error)")
 completion(.failure(error))
 }
}
```
```

This closure tries to sign a new JWT token with the payload and calls the `completion` handler with the result. If signing is successful, it passes the new token to `completion`. If there is an error, it prints the error and passes the error to `completion`.

Finally, the completion handler of `connectUser` checks for errors and calls `checkUserStatus()` if the connection is successful, or reports an error if the connection fails.

4. Checking User Status

The `checkUserStatus` function checks whether the connected user is a new user or an existing user by checking if the user has a name:

```
```swift
if let name = ChatClient.shared.currentUserController().currentUser?.name {
 // ... user exists ...
} else {
 // ... new user ...
}
```
```

If the user has a name, it updates `storedUser` and `logStatus`, and stops loading. If not, it sets `newUser` to `true` and stops loading.

In summary, this code authenticates the user with Firebase using their phone number, generates a JWT token for the authenticated user, connects the user to Stream Chat using this token, and checks the user's status in the chat system.

How will we implement the same code if there was user as a model . struct User: Codable, Identifiable, Hashable { var id: UUID = UUID() var phone: String? var name: String? var city: String? var status: String? var photo: String? }

To adapt the existing code to use the `User` struct, you would have to make several modifications. Below are the changes that you can incorporate:

1. Update `LoginViewModel`:

Incorporate a property `currentUser` of type `User` in `LoginViewModel`.

```
```swift
@Published var currentUser: User?
```
```

2. Modify `LoginUser` method:

After successfully signing in with Firebase, you can create a `User` instance and assign it to `currentUser`. Also, instead of using `uid` for Stream Chat user ID, you would use `currentUser.id`.

```
```swift
func loginUser(code: String) {
 // ... existing code ...

 Auth.auth().signIn(with: credential) { result, err in
 if let error = err {
 self.errorMessage = error.localizedDescription
 self.showAlert.toggle()
 return
 }

 guard let firebaseUser = Auth.auth().currentUser else {
 self.reportError()
 return
 }

 // Create a User instance
 self.currentUser = User(phone: firebaseUser.phoneNumber, name: firebaseUser.displayName)

 let payload = Payload(user_id: self.currentUser!.id.uuidString)

 // ... rest of the code ...

 ChatClient.shared.connectUser(userInfo: .init(id: self.currentUser!.id.uuidString), tokenProvider: tokenProvider) { error in
 // ... existing code ...
 }
 }
}
```
```

3. Update `checkUserStatus` method:

Modify the `checkUserStatus` method to update the `currentUser` property instead of `storedUser`.

```
```swift
func checkUserStatus() {
 if let name = ChatClient.shared.currentUserController().currentUser?.name {
 withAnimation {
 self.currentUser?.name = name
 self.logStatus = true
 self.isLoading = false
 }
 } else {
 withAnimation {
 self.newUser = true
 }
 }
}
```
```



```
        self.isLoading = false
    }
}
...

```

4. Storing `User` data:

If you need to persist the user's data, you may consider using a local database like CoreData or a simple solution like UserDefaults (not recommended for sensitive data). Codable conformance will allow easy encoding/decoding of the `User` struct.

5. Update `Payload`:

Ensure that `Payload` accepts `String` user_id since `UUID` will be converted to `String`.

```
```swift
struct Payload: JWTPayload, Equatable {
 var user_id: String

 func verify(using signer: JWTSigner) throws {
 // Verification logic if needed
 }
}
...

```

These changes should allow you to incorporate a `User` model into your existing logic. Keep in mind that this adaptation assumes that the `photo`, `city`, and `status` properties of the `User` will be handled elsewhere in your application logic or set with default values.