## Criterion C: Development

The architecture of my project is an IOS Application coded in Swift with a MySQL database backend connected using a Node.js server. I also used the SwiftUI, UserDefaults, and UIKit frameworks.

**Techniques Used:**

- Model-View-ViewModel (MVVM) Design Pattern
- Networking and JSON Parsing
- Database Relations and Queries
- Observable/Environment Object
- Navigation and Routing
- Event-Based Programming and Event Handling
- Conditional Expressions and State Management
- Multidimensional Arrays
- Data Persistence
- User-Defined Methods
- UIKit Sharing

**Design Elements:**

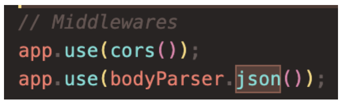
*Model-View-ViewModel (MVVM) Design Pattern*

I applied the Model-View-ViewModel (MVVM) architecture to my product. The architecture is divided into three components.

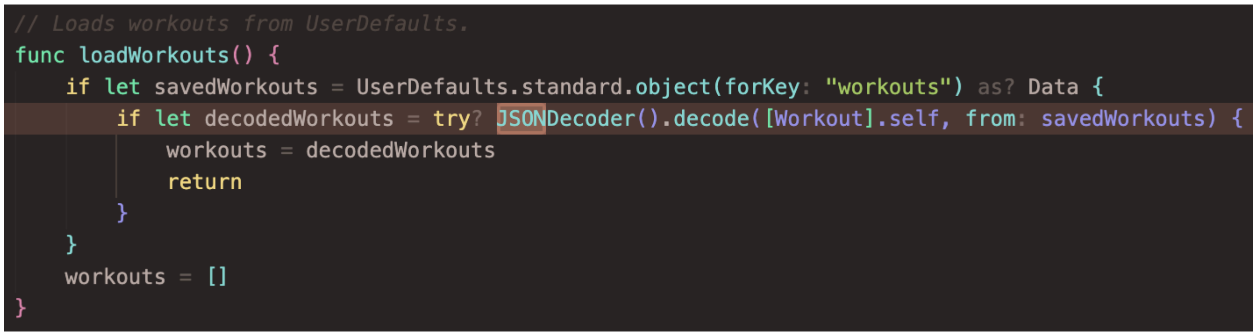
| |
|---|
| 1. The Model, which is the database I created in MySQL and the Node.js server I created to interact with the database, handles the data related to the climbs present in the gym and their properties. |
| 2. The View, the file 'ContentView' in my Swift project, allows the user to interact with the UI, selecting and viewing climbs. |
| 3. This is separate from the View Model (the file 'DataViewModel') in my project, which acts as a bridge between the Model and the View, handling data transmission from the server to the UI and managing user inputs. |

I chose MVVM for its clear separation of concerns, making the application easier to update, test, and bug fix. For example, any issues with the database will not adversely affect the view and vice versa. This architecture is highly related to the 'Flowchart of Information' shown in Criterion B.

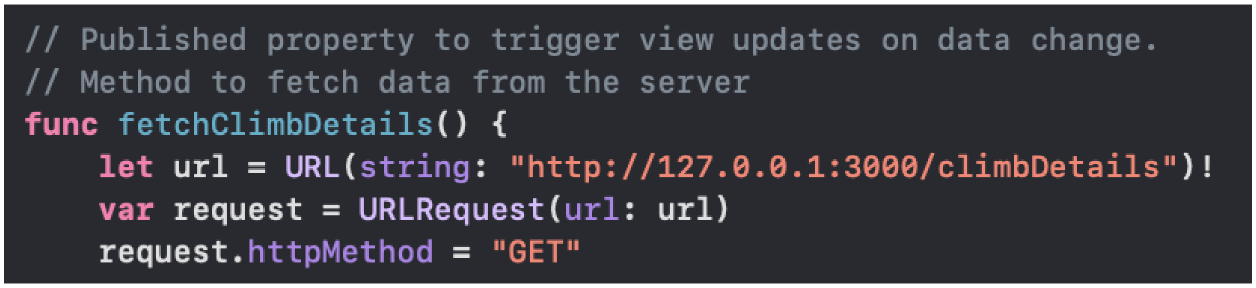
*Networking and JSON Parsing*



```
// Middlewares
app.use(cors());
app.use(bodyParser.json());
```



```
// Loads workouts from UserDefaults.
func loadWorkouts() {
    if let savedWorkouts = UserDefaults.standard.object(forKey: "workouts") as? Data {
        if let decodedWorkouts = try? JSONDecoder().decode([Workout].self, from: savedWorkouts) {
            workouts = decodedWorkouts
            return
        }
    }
    workouts = []
}
```

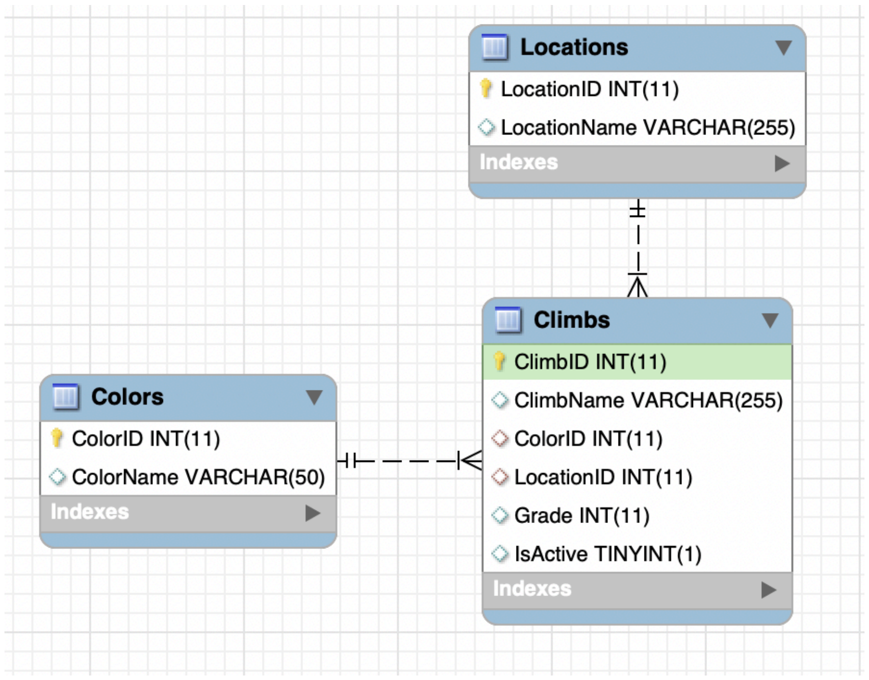


```
// Published property to trigger view updates on data change.
// Method to fetch data from the server
func fetchClimbDetails() {
    let url = URL(string: "http://127.0.0.1:3000/climbDetails")!
    var request = URLRequest(url: url)
    request.httpMethod = "GET"
```

To connect the different computing devices (i.e., the User's iPhone and the database server) I used the technique of Networking through a Node.js server. The Node.js server handles requests from the user on the iOS app, fetching the relevant data from the MySQL database, and sending it back to the app. This data exchange is done using JSON (JavaScript Object Notation). JSON parsing in my app involves converting the JSON data received from the server into Swift objects that can be easily managed and displayed in the app's UI.

I chose this approach because it efficiently handles data transfer between different application parts. The use of the Class JSONDecoder (as a part of the Foundation framework in all Swift applications) to parse the JSONs in the database is shown above. This is the implementation of the 'Flowchart of Information' shown in Criterion B.

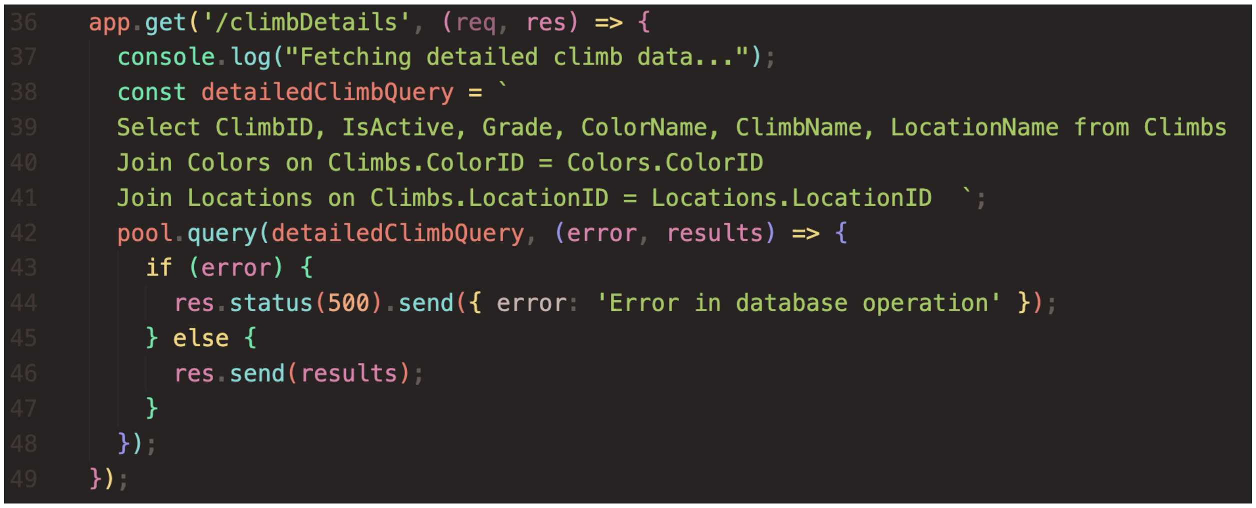
*Database Relations and Queries*



The database setup I used is shown above. Each Climb has an

- ID,
- Name (which describes the 'key concept' or skills practiced during the climb),
- Color,
- Location,
- Grade (the difficulty level of the climb, as determined by the staff),
- and a boolean for whether that climb is currently in session (IsActive).

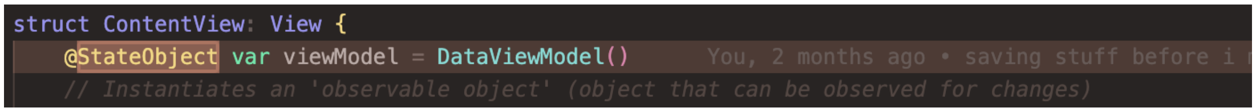
This is so that for every 'reset' of the gym where all old climbs are discontinued, staff can simply change whether the climb is active, meaning the user will not be able to search for it during their workout but still see it in their workout history. The relationship between both Colors and Locations to Climbs is one-to-many (one location or color can belong to several climbs).



```
app.get('/climbDetails', (req, res) => {
  console.log("Fetching detailed climb data...");
  const detailedClimbQuery = `
  Select ClimbID, IsActive, Grade, ColorName, ClimbName, LocationName from Climbs
  Join Colors on Climbs.ColorID = Colors.ColorID
  Join Locations on Climbs.LocationID = Locations.LocationID  `;
  pool.query(detailedClimbQuery, (error, results) => {
    if (error) {
      res.status(500).send({ error: 'Error in database operation' });
    } else {
      res.send(results);
    }
  });
});
```

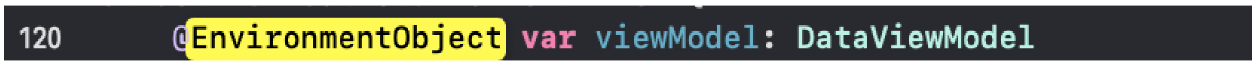
The query I used to select the climbs is shown above. It simply sends all of the data to the app, and logic within the app is used to determine which climbs are shown to users.

*Observable and Environment Objects*

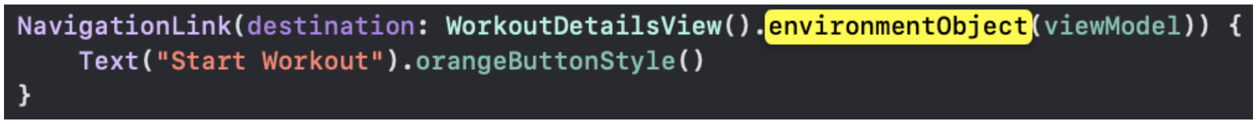


```
struct ContentView: View {
    @StateObject var viewModel = DataViewModel()        You, 2 months ago • saving stuff before i
    // Instantiates an 'observable object' (object that can be observed for changes)
```

In this project, the DataViewModel class serves as an Observable Object, meaning it's capable of notifying our views about changes to the data it holds, such as the list of climbing routes. When new data is fetched or existing data is updated, SwiftUI re-renders the affected parts of the UI to reflect these changes, ensuring our user interface is always up-to-date.



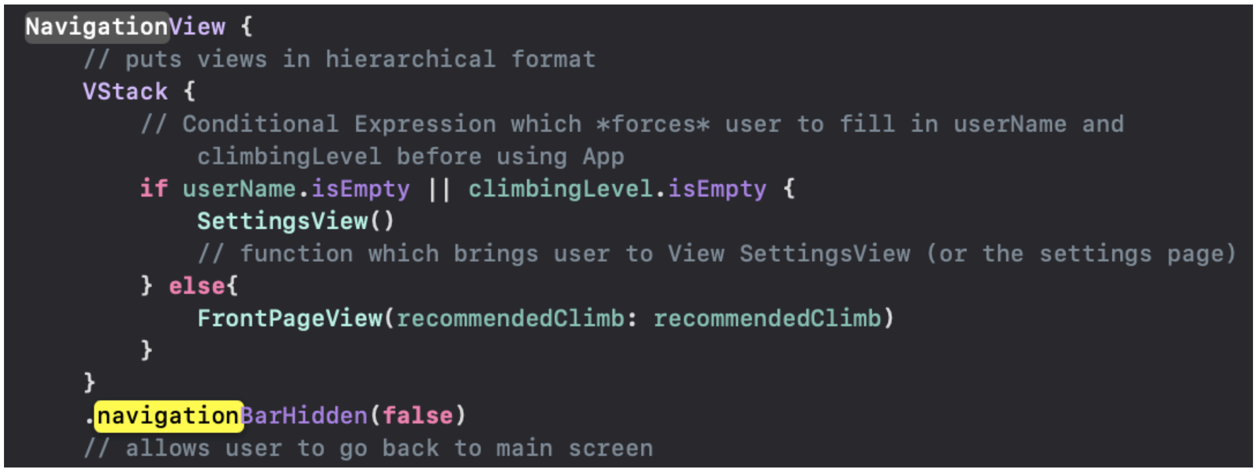
```
120         @EnvironmentObject var viewModel: DataViewModel
```



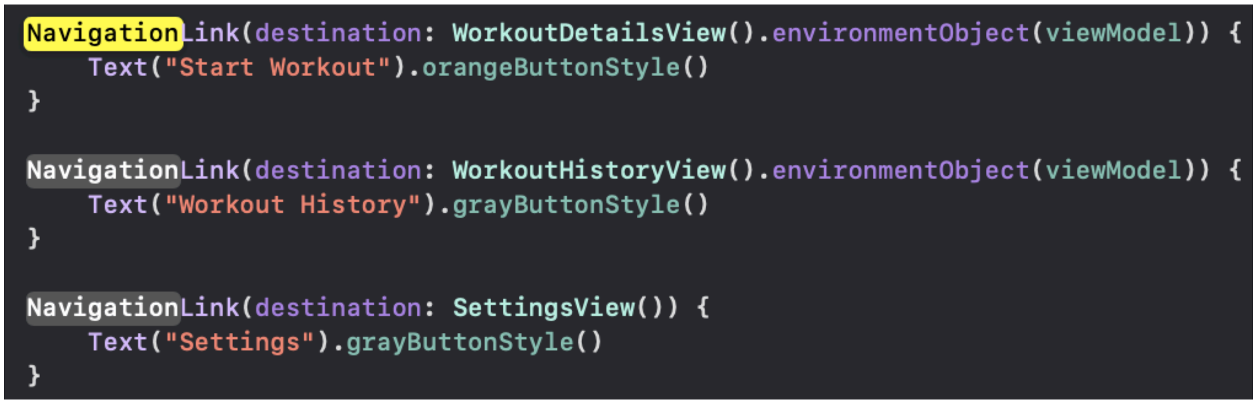
```
NavigationLink(destination: WorkoutDetailsView().environmentObject(viewModel)) {
    Text("Start Workout").orangeButtonStyle()
}
```

Observable Objects are used to monitor data changes within a specific view, while Environment Objects are Observable Objects passed down the view hierarchy, allowing multiple views to access shared data. By adopting @EnvironmentObject, we share viewModel across the app's various views.

*Navigation and Routing*



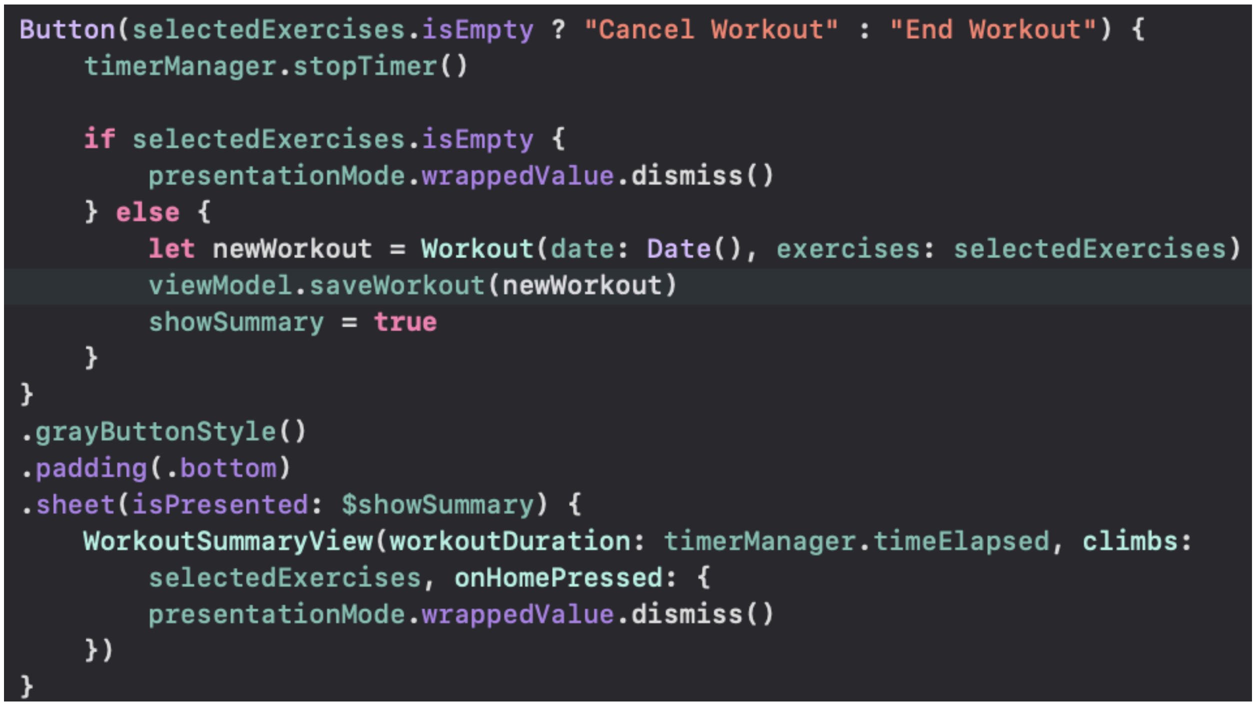
```
NavigationView {
    // puts views in hierarchical format
    VStack {
        // Conditional Expression which *forces* user to fill in userName and
            climbingLevel before using App
        if userName.isEmpty || climbingLevel.isEmpty {
            SettingsView()
            // function which brings user to View SettingsView (or the settings page)
        } else{
            FrontPageView(recommendedClimb: recommendedClimb)
        }
    }
    .navigationBarHidden(false)
    // allows user to go back to main screen
```



```
NavigationLink(destination: WorkoutDetailsView().environmentObject(viewModel)) {
    Text("Start Workout").orangeButtonStyle()
}

NavigationLink(destination: WorkoutHistoryView().environmentObject(viewModel)) {
    Text("Workout History").grayButtonStyle()
}

NavigationLink(destination: SettingsView()) {
    Text("Settings").grayButtonStyle()
}
```

I use navigation and routing in this project to guide users through different functionalities of the app. In the screenshots shown above, structures such as ‘NavigationView’ and routing features such as ‘NavigationLink’ transition users between different screens. These screens include starting a new workout, viewing workout history, viewing their settings, and more. The view hierarchy typically gives a user a ‘back’ symbol in the top left, allowing them to return to the main page.

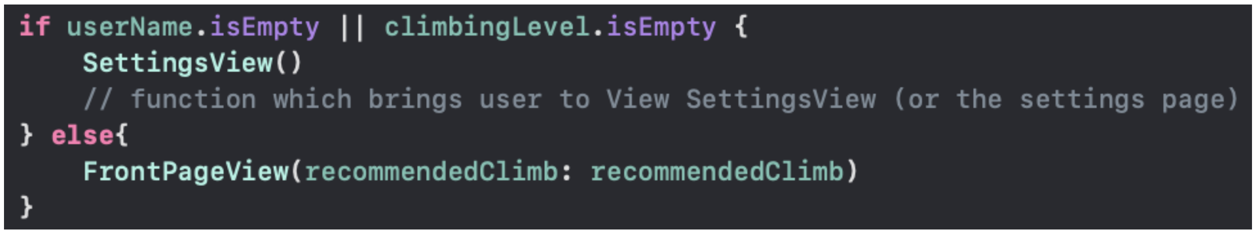
*Event-Based Programming and Event Handling*



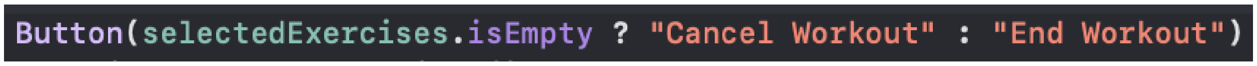
```swift
Button(selectedExercises.isEmpty ? "Cancel Workout" : "End Workout") {
    timerManager.stopTimer()

    if selectedExercises.isEmpty {
        presentationMode.wrappedValue.dismiss()
    } else {
        let newWorkout = Workout(date: Date(), exercises: selectedExercises)
        viewModel.saveWorkout(newWorkout)
        showSummary = true
    }
}
.grayButtonStyle()
.padding(.bottom)
.sheet(isPresented: $showSummary) {
    WorkoutSummaryView(workoutDuration: timerManager.timeElapsed, climbs:
        selectedExercises, onHomePressed: {
        presentationMode.wrappedValue.dismiss()
    })
}
```

The “End Workout” (or “Cancel Workout”) button is a great example of Event-Based Programming and Event Handling in my application. Upon tapping, it triggers the workout timer to stop, saves the workout, and transitions to a summary view (or the main page). Taps of buttons in my application can often lead other functions to take place (besides just navigating), dynamically updating my app.

*Conditional Expressions and State Management*

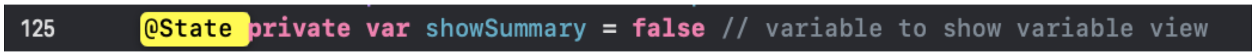


```
if userName.isEmpty || climbingLevel.isEmpty {
    SettingsView()
    // function which brings user to View SettingsView (or the settings page)
} else{
    FrontPageView(recommendedClimb: recommendedClimb)
}
```

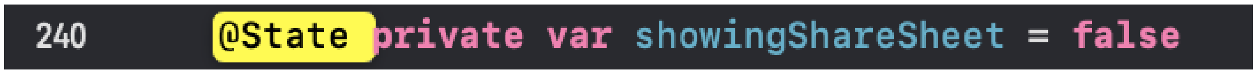


```
Button(selectedExercises.isEmpty ? "Cancel Workout" : "End Workout")
```

I typically used Conditional Expressions to make quality-of-life improvements for end-users. The first shown above essentially forces the user to input their name and climbing level before accessing the rest of the features of the application. The second feature conditions the workout a user is currently in based on whether they have selected any exercises, preventing any empty workouts from being logged into workout history.



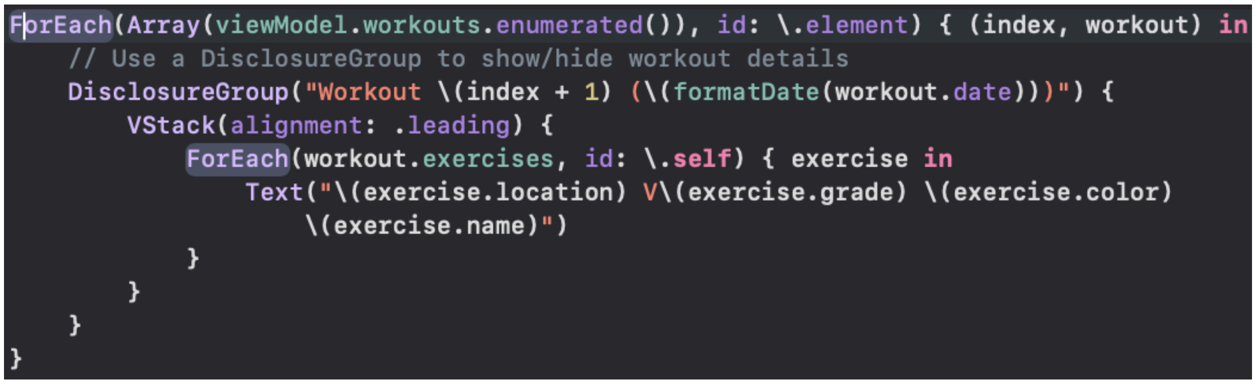
```
@State private var showSummary = false // variable to show variable view
```



```
@State private var showingShareSheet = false
```

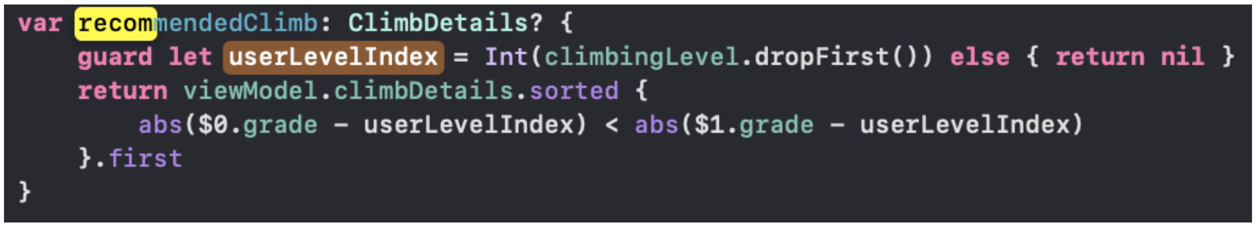
Above are two examples of state management in my application. The '@State' property wrapper is used to declare state variables for a view– when these variables are changed, the view re-renders with the updated information.  In the examples above, other conditional statements and button presses will change the values of these state variables, contextually changing an end-user's experience.

*Multidimensional Arrays*

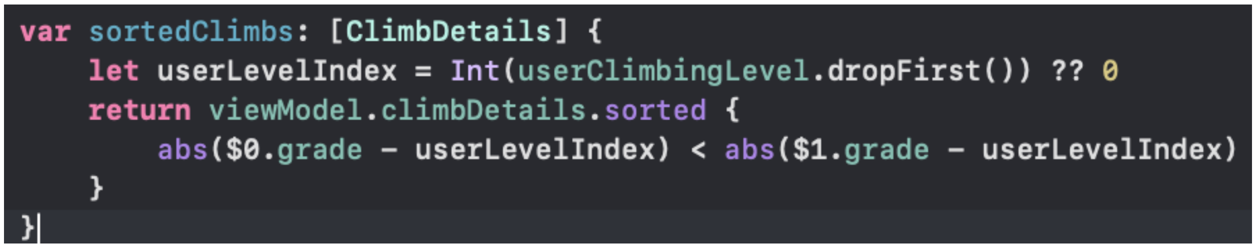


```
ForEach(Array(viewModel.workouts.enumerated()), id: \.element) { (index, workout) in
    // Use a DisclosureGroup to show/hide workout details
    DisclosureGroup("Workout \(index + 1) (\(formatDate(workout.date)))") {
        VStack(alignment: .leading) {
            ForEach(workout.exercises, id: \.self) { exercise in
                Text("\(exercise.location) V\(exercise.grade) \(exercise.color)
                    \(exercise.name)")
            }
        }
    }
}
```

I used Arrays and Arraylists, often multidimensional, to store workout data. Multidimensional arrays were the best tool here as I would like each workout to have multiple climbs (referred to as exercises in the code), and each climb to have multiple data points that I can extract, such as location grade, color, name, etc. The code above is an example of iterating through the array and displaying all of the exercises by their respective location, grade, color, and name.



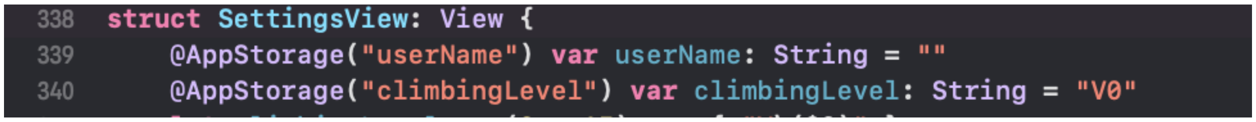
```swift
var recommendedClimb: ClimbDetails? {
    guard let userLevelIndex = Int(climbingLevel.dropFirst()) else { return nil }
    return viewModel.climbDetails.sorted {
        abs($0.grade - userLevelIndex) < abs($1.grade - userLevelIndex)
    }.first
}
```



```swift
var sortedClimbs: [ClimbDetails] {
    let userLevelIndex = Int(userClimbingLevel.dropFirst()) ?? 0
    return viewModel.climbDetails.sorted {
        abs($0.grade - userLevelIndex) < abs($1.grade - userLevelIndex)
    }
}
```

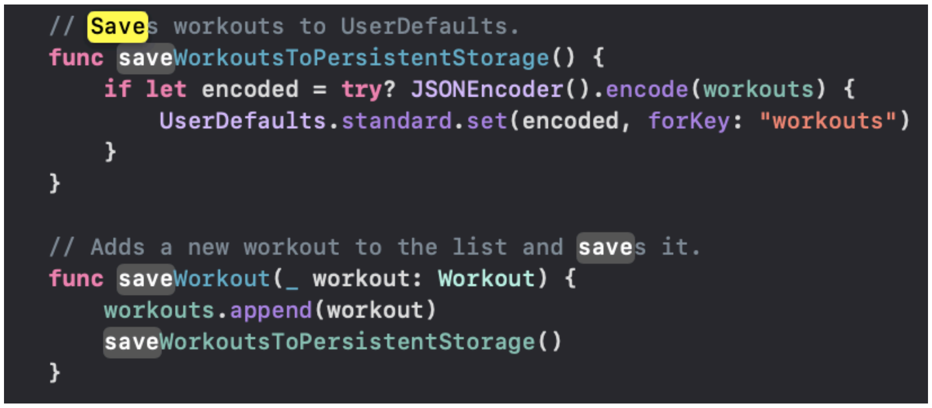
Above is an implementation of the pseudo-code describing the ‘Recommended Climb Algorithm’ shown in Criterion B.

*Data Persistence*



```swift
struct SettingsView: View {
    @AppStorage("userName") var userName: String = ""
    @AppStorage("climbingLevel") var climbingLevel: String = "V0"
```

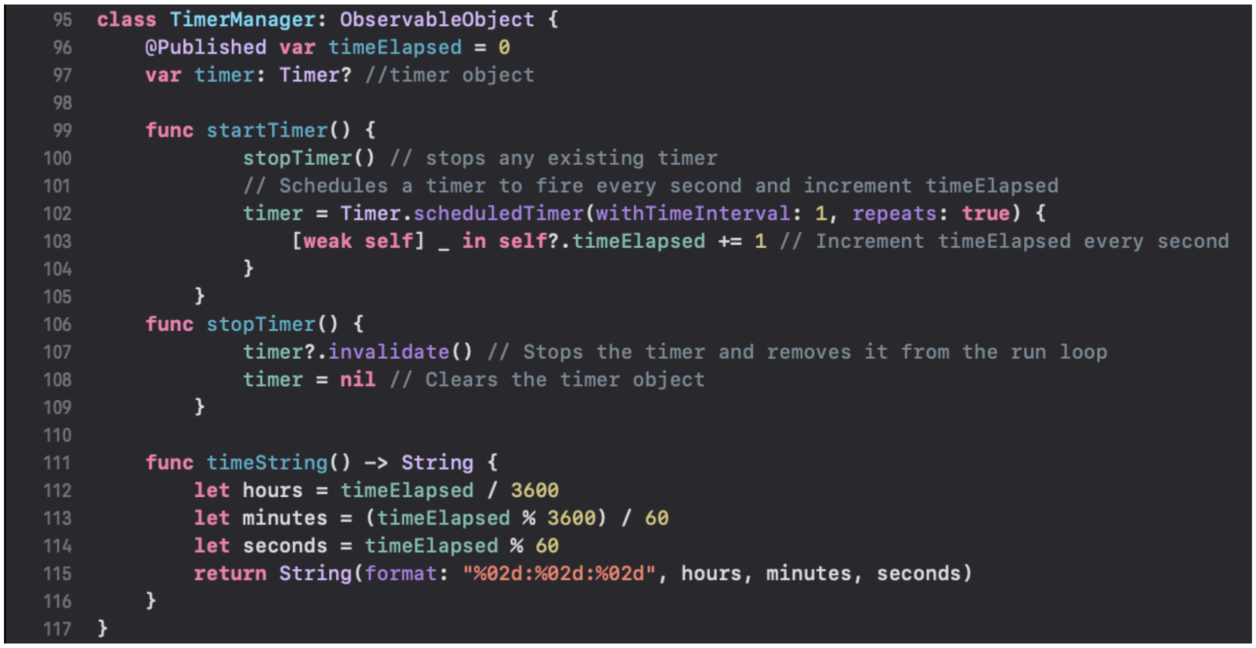
AppStorage for small, often infrequently changing quantities. This is shown in the first picture above, where I used AppStorage to keep the userName and climbing level data points from the user’s settings.



```swift
// Saves workouts to UserDefaults.
func saveWorkoutsToPersistentStorage() {
    if let encoded = try? JSONEncoder().encode(workouts) {
        UserDefaults.standard.set(encoded, forKey: "workouts")
    }
}

// Adds a new workout to the list and saves it.
func saveWorkout(_ workout: Workout) {
    workouts.append(workout)
    saveWorkoutsToPersistentStorage()
}
```

I also used UserDefaults, which is more robust in saving and loading data. The code above is the functionality for saving workouts to persistent storage, taking a current climb (workout), and appending it to the historical ‘workouts’ array shown in the User’s history.

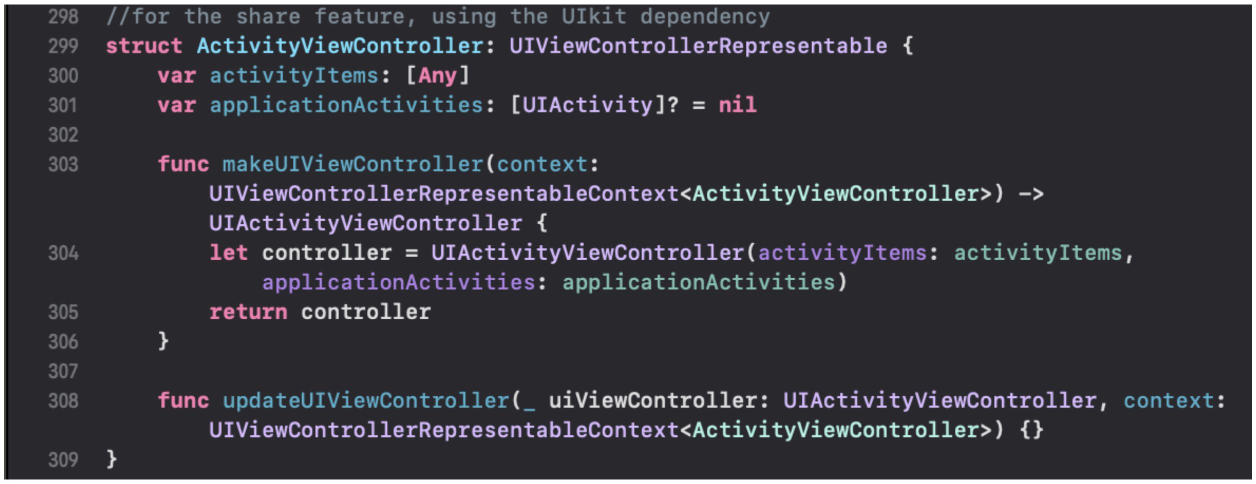
*User-Defined Methods*



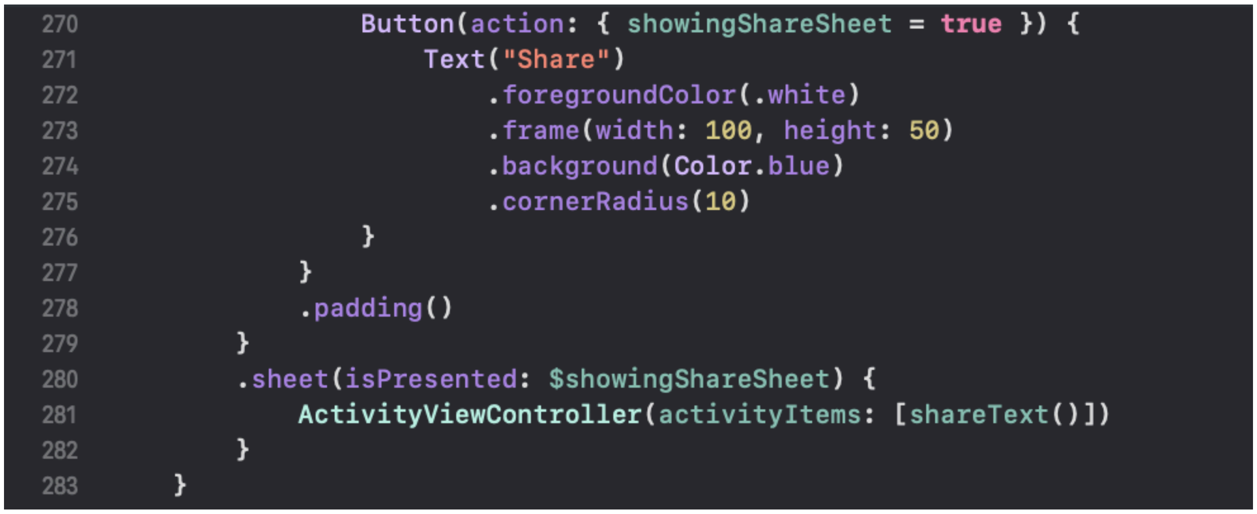
```swift
class TimerManager: ObservableObject {
    @Published var timeElapsed = 0
    var timer: Timer? //timer object

    func startTimer() {
            stopTimer() // stops any existing timer
            // Schedules a timer to fire every second and increment timeElapsed
            timer = Timer.scheduledTimer(withTimeInterval: 1, repeats: true) {
                [weak self] _ in self?.timeElapsed += 1 // Increment timeElapsed every second
            }
        }
    func stopTimer() {
            timer?.invalidate() // Stops the timer and removes it from the run loop
            timer = nil // Clears the timer object
        }

    func timeString() -> String {
        let hours = timeElapsed / 3600
        let minutes = (timeElapsed % 3600) / 60
        let seconds = timeElapsed % 60
        return String(format: "%02d:%02d:%02d", hours, minutes, seconds)
    }
}
```

I defined methods in places where they would crowd out the purpose of the code, increasing readability and adaptability. I used this in the case of the timer (as shown above), so all of the code related to the timer (starting it, stopping it, and converting the time to a readable string for users) would be stored in one class.

*UIKit Sharing*



```swift
//for the share feature, using the UIkit dependency
struct ActivityViewController: UIViewControllerRepresentable {
    var activityItems: [Any]
    var applicationActivities: [UIActivity]? = nil

    func makeUIViewController(context:
        UIViewControllerRepresentableContext<ActivityViewController>) ->
        UIActivityViewController {
        let controller = UIActivityViewController(activityItems: activityItems,
            applicationActivities: applicationActivities)
        return controller
    }

    func updateUIViewController(_ uiViewController: UIActivityViewController, context:
        UIViewControllerRepresentableContext<ActivityViewController>) {}
}
```



```swift
                    Button(action: { showingShareSheet = true }) {
                        Text("Share")
                            .foregroundColor(.white)
                            .frame(width: 100, height: 50)
                            .background(Color.blue)
                            .cornerRadius(10)
                    }
                }
                .padding()
            }
            .sheet(isPresented: $showingShareSheet) {
                ActivityViewController(activityItems: [shareText()])
            }
        }
```

I implemented UiKit (another Framework integrated closely with SwiftUi) to enrich user interaction within the app, specifically through its sharing functionality. I created the struct ‘ActivityViewController’ to use iOS’ native share sheet, meaning that they will see various options to share a workout summary text.

Wordcount: 1009

**Citations**

“AppStorage.” *Apple Developer Documentation*, Apple Inc., developer.apple.com/documentation/swiftui/appstorage. Accessed 10 Feb. 2024.

“EnvironmentObject.” *Apple Developer Documentation*, Apple Inc., developer.apple.com/documentation/swiftui/environmentobject. Accessed 10 Feb. 2024.

“NavigationView.” *Apple Developer Documentation*, Apple Inc., developer.apple.com/documentation/swiftui/navigationview. Accessed 10 Feb. 2024.

“ObservableObject.” *Apple Developer Documentation*, Apple Inc., developer.apple.com/documentation/combine/observableobject. Accessed 10 Feb. 2024.

"OpenAI ChatGPT." Personal conversation on the implementation of Observable and Environment Objects in Swift/SwiftUI applications. 9 Feb. 2024, 5:00 AM.

“UIKit.” *Apple Developer Documentation*, Apple Inc., developer.apple.com/documentation/uikit. Accessed 10 Feb. 2024.

“UserDefaults.” *Apple Developer Documentation*, Apple Inc., developer.apple.com/documentation/foundation/userdefaults. Accessed 10 Feb. 2024.