

Week 3 Tutorial

Implementing Data Persistence and Navigation



Do complete previous week's tutorial, solution is available to enable you to continue with this week's activities.

By the end of this tutorial, you will be able to:

- Save calculation data so it's available even after closing the app
- Build a History view that lists previous calculations (newest first)
- Use TabView for smooth navigation between screens
- Use @AppStorage to automatically reload the last calculation

Introduction



This week, you'll upgrade the **Parcel App Calculator (Version 3)** by adding:

- **Data persistence** (saving calculations)
- **A history view** to display past calculations
- **Tab-based navigation** to switch between Calculator and History views
- **Automatic loading** of the last calculation when the app restarts



Two Ways to Save Data in iOS

We'll compare **two common approaches** to persistence:

1. Traditional Approach: UserDefaults + Codable – [Demonstrated in the lecture next week](#)

- **Data Model**
 - A Parcel struct that conforms to Codable (properties: weight, dimensions, cost, date).
- **How It Works**
 - **Encode:** Convert the array of Parcel objects to JSON with JSONEncoder.
 - **Store:** Save JSON to UserDefaults (key-value storage).
 - **Retrieve:** On app launch, load JSON, decode it back into [Parcel], and display it in History.
- **Pros & Cons**
 -  Great for learning encoding/decoding basics -covered later
 -  Manual, not efficient for larger datasets, and no built-in observation.

2. Modern Approach: SwiftData + @Model

- **Data Model**
 - A Parcel class annotated with @Model. No need for Codable.
- **How It Works**
 - **Storage:** Managed by ModelContainer.
 - **Operations:** Insert, fetch, delete via ModelContext.
 - **Retrieve:** Use @Query in HistoryView to load and observe all parcels, sorted by date.
- **Pros & Cons**
 -  Efficient, automatic updates, handles relationships, minimal code.
 -  Requires iOS 17+ (not backward compatible).

Step-by-Step Implementation

Step 1: Add a Date Picker

First, add a date property and a DatePicker to ContentView to timestamp each calculation.

```
@State private var postDate: Date = Date()
```

1. Add a DatePicker view below your Toggle view:

```
DatePicker(  
    "select Date", selection: $postDate, in: ...Date(),  
    displayedComponents: .date  
)
```

Step 2: Create the SwiftData Model

1. Create a new Swift file (File → New → File → Swift File)
2. Name it "ParcelDataModel.swift"
3. Annotate the Parcel class with @Model macro

```
import Foundation
import SwiftData

@Model
class ParcelDataModel {
    var weight: String
    var volume: String
    var cost: String
    var postDate: Date
    init(weight: String, volume: String, cost: String, postDate: Date) {
        self.weight = weight
        self.volume = volume
        self.cost = cost
        self.postDate = postDate
    }
}
```

Step 3: Set Up the ModelContainer

In ParcelAppCalculatorApp.swift file, set up the ModelContainer for your app.

```
import SwiftUI
import SwiftData

@main
struct ParcelAppCalculatorApp: App {
    var body: some Scene {
        WindowGroup {
            MainTabView()
        }
        .modelContainer(for: ParcelDataModel.self)
    }
}
```

Step 4: Create Tab Navigation

Create a new SwiftUI View (MainTabView.swift) to host your tabs.

```
import SwiftUI

struct MainTabView: View {
    var body: some View {
        TabView {
            ContentView()
                .tabItem {
                    Label("Calculator", systemImage: "function")
                }

            HistoryView()
                .tabItem {
                    Label("History", systemImage: "clock")
                }
        }
    }
}

#Preview {
    MainTabView()
}
```

The compiler may throw an error if the HistoryView() does not exist.

1. Update app's main file to use MainTabView() instead of ContentView():

```
import SwiftUI
import SwiftData

@main
struct ParcelAppCalculatorApp: App {
    var body: some Scene {
        WindowGroup {
            MainTabView()
        }
        .modelContainer(for: ParcelDataModel.self)
    }
}
```

Step 5: Save Calculations to Database

1. In ContentView, access the model context:

```
struct ContentView: View {
    // TODO: Add @State properties for height, width, depth, weight and cost, modelContext
    @Environment(\.modelContext) private var modelContext
    @State private var height = ""
    @State private var width = ""
```

2. When a cost is calculated, create and save a new ParcelDataModel object:

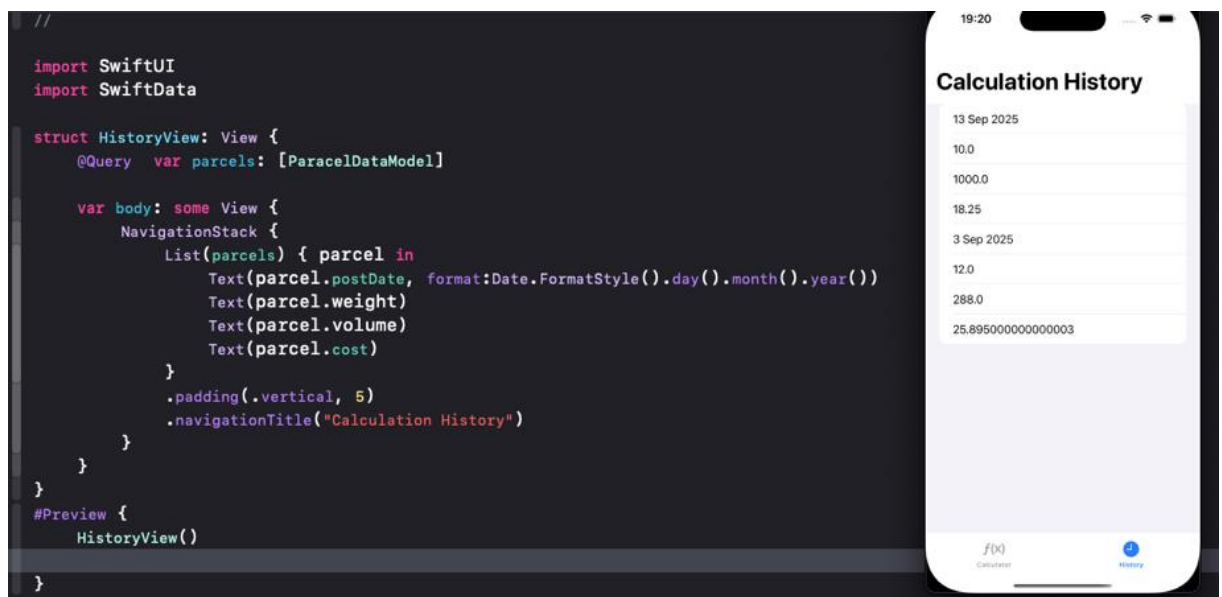
Currently this will happen in either of the calculator functions and needs to be coded in both places, but later we can optimise and do it in one place:

```
// add record to database
let newRecord = ParcelDataModel(
    weight: String(weight),
    volume: String(volume),
    cost: String(totalCost),
    postDate: postDate
)
modelContext.insert(newRecord)
```

Do note the need to cast values from Double to String to save the record.

Step 6: Build the HistoryView

1. Create a new SwiftUI View file named "HistoryView.swift"
2. Implement the HistoryView with @Query:



Step 7: Save and Load the Last Calculation Using @AppStorage

1. In ContentView, add @AppStorage properties to save the last calculation:

```

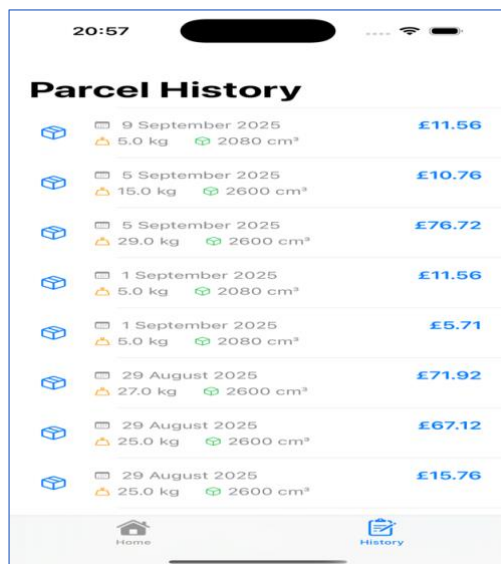
struct ContentView: View {

    // TODO: Add @State properties for height, width, depth, weight and cost, modelContext
    @Environment(\.modelContext) private var modelContext
    @AppStorage("height") private var height = ""
    @AppStorage("width") private var width = ""
    @AppStorage("depth") private var depth = ""
    @AppStorage("weight") private var weight = ""
    @AppStorage("cost") private var cost = ""
    @AppStorage("errorMessage") private var errorMessage: String = "" // is this correct?
    @AppStorage("postDate") private var postDate: Date = Date()
}

```

Independent Task Build the HistoryView

Using your knowledge of SwiftUI containers like HStack , VStack and SF symbols like *shippingbox*, *calendar* and *scalemass*, re-code HistoryView so that it renders as image shown below:



This is a good exercise in building a complex interface, you are advised to plan it on paper and build in stages.