MetUCD Report Part 2

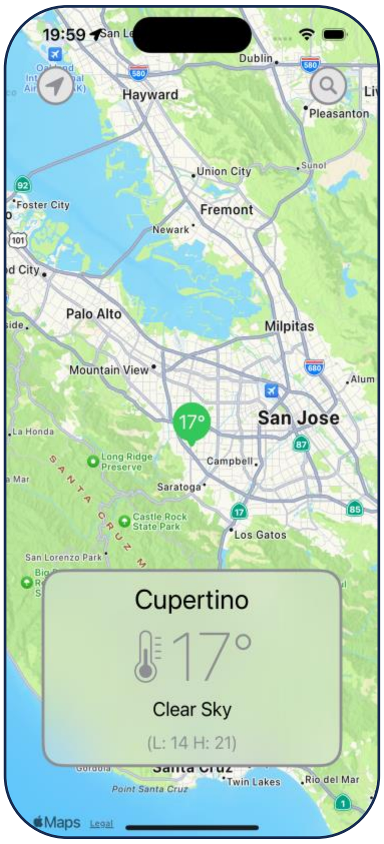
James Clackett
15732749
21st November 2023

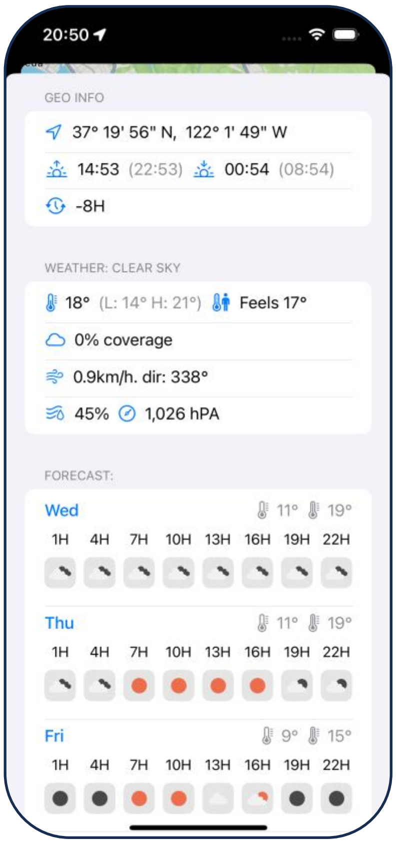
The objective of Part 2 was to improve upon the weather forecast application. The user would be presented with a Map of their current location. This map would allow them to select and search for new locations. For any given location, the user would see a summary of the weather and have the option to view more detailed info by clicking on a widget.
See code for more detailed notes on functionality.

Summary of actions taken:

1) Created MapViewModel view model.
2) Implemented the Core Location services, handling different device privacy scenarios.
3) Created a Map View with MapKit.
4) Updated weather and map view models to be able to work together.
5) Implemented a modal version of WeatherView using a SwiftUI Sheet
6) Modified the UI to suit different devices and orientations.
7) Refactored and Documented code.

Application UI:




20:50

GEO INFO

37° 19' 56" N, 122° 1' 49" W
14:53 (22:53) 00:54 (08:54)
-8H

WEATHER: CLEAR SKY

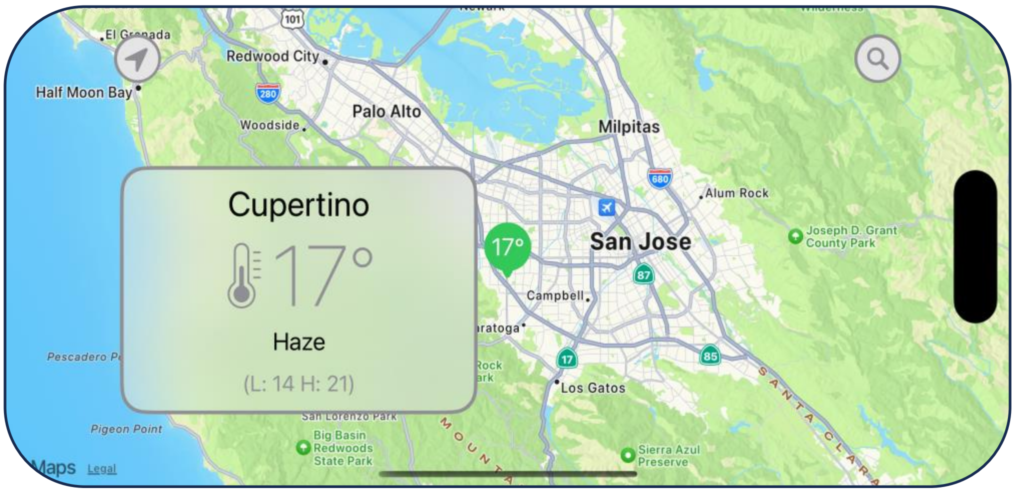
18° (L: 14° H: 21°) Feels 17°
0% coverage
0.9km/h. dir: 338°
45% 1,026 hPA

FORECAST:

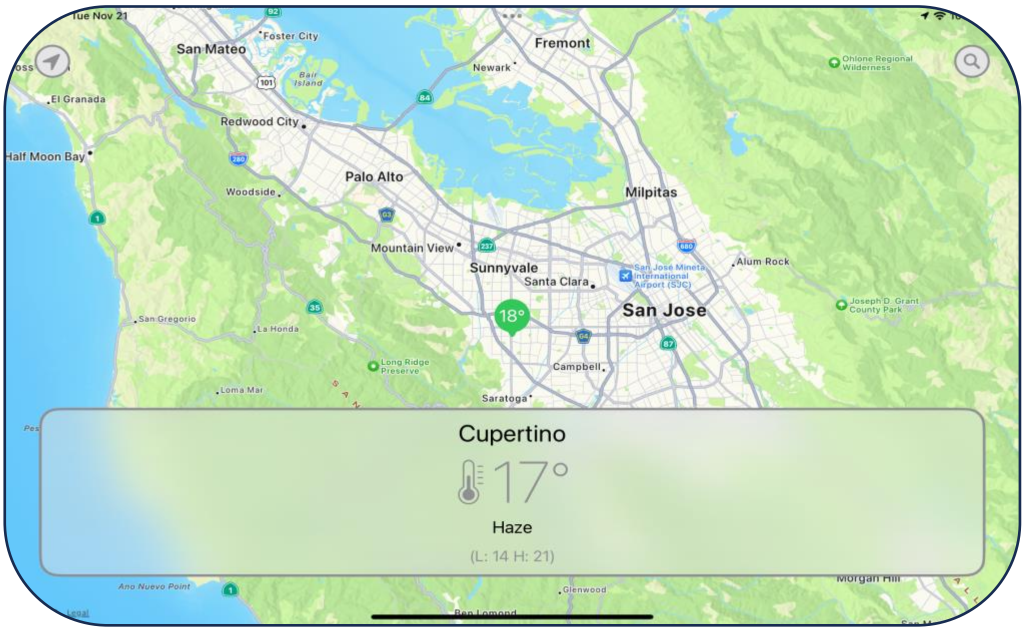
Wed 11° 19°
1H 4H 7H 10H 13H 16H 19H 22H

Thu 11° 19°
1H 4H 7H 10H 13H 16H 19H 22H

Fri 9° 15°
1H 4H 7H 10H 13H 16H 19H 22H



iPhone 15 Pro Landscape Orientation



iPad Mini Landscape Orientation

Important code:

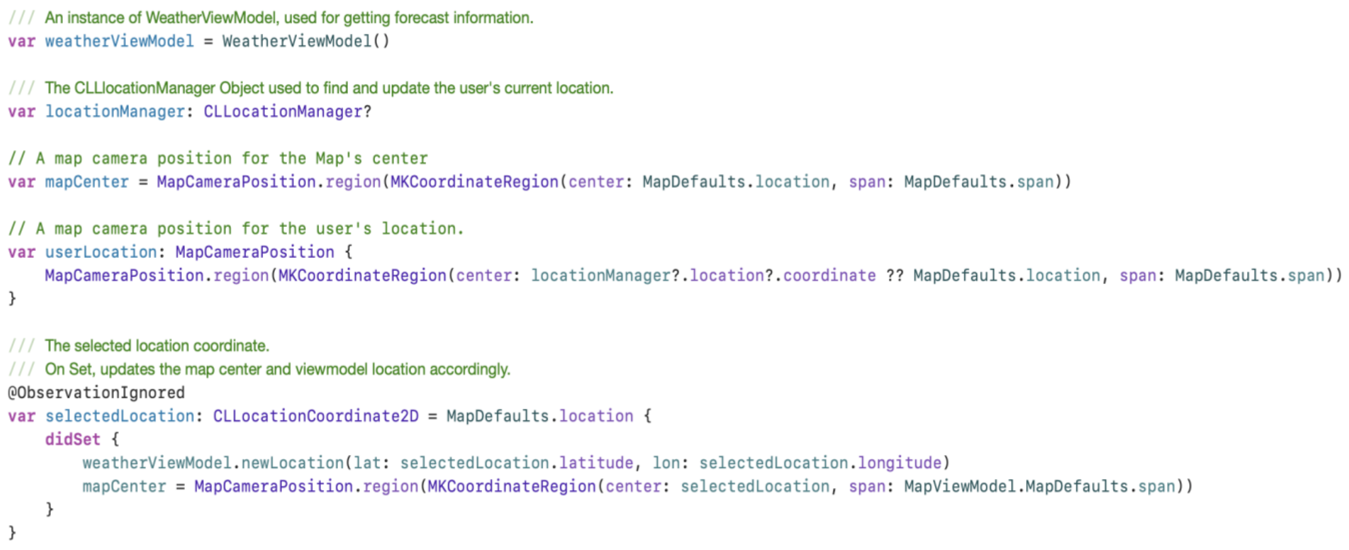
Search Location function:

```swift
/// A function that uses MapKit's LocalSearch to find a location.
/// Makes a search with a region of .world and result type of .address. This returns towns, cities, and countries.
/// If the search is successful (there are results), updates the State variable searchedLocation.
func searchLocation() async {
    let request = MKLocalSearch.Request()
    request.naturalLanguageQuery = searchText
    request.region = MKCoordinateRegion(.world)
    request.resultTypes = MKLocalSearch.ResultType.address

    let results = try? await MKLocalSearch(request: request).start()
    let foundLoc = results?.mapItems ?? []

    if !foundLoc.isEmpty {
        let locCoord = foundLoc.first?.placemark.coordinate
        self.searchedLocation = MapCameraPosition.region(MKCoordinateRegion(center: locCoord ?? MapViewModel.MapDefaults.location , span:
            MapViewModel.MapDefaults.span))
    }
}
```

Map View Model:



```
/// An instance of WeatherViewModel, used for getting forecast information.
var weatherViewModel = WeatherViewModel()

/// The CLLlocationManager Object used to find and update the user's current location.
var locationManager: CLLocationManager?

// A map camera position for the Map's center
var mapCenter = MapCameraPosition.region(MKCoordinateRegion(center: MapDefaults.location, span: MapDefaults.span))

// A map camera position for the user's location.
var userLocation: MapCameraPosition {
    MapCameraPosition.region(MKCoordinateRegion(center: locationManager?.location?.coordinate ?? MapDefaults.location, span: MapDefaults.span))
}

/// The selected location coordinate.
/// On Set, updates the map center and viewmodel location accordingly.
@ObservationIgnored
var selectedLocation: CLLocationCoordinate2D = MapDefaults.location {
    didSet {
        weatherViewModel.newLocation(lat: selectedLocation.latitude, lon: selectedLocation.longitude)
        mapCenter = MapCameraPosition.region(MKCoordinateRegion(center: selectedLocation, span: MapViewModel.MapDefaults.span))
    }
}
```

Issues:

1) The TextField issues from Part 1 are still present.
2) There is no easy way to tailor views based on device orientation. I have used sizeClasses to do the best I can. In landscape, WeatherWidget is pushed to the side in on compact devices to allow users to view the map more clearly. Unfortunately this creates a small alignment issue on some devices in Portrait mode (widget slightly off-center).
3) The biggest issue with Part 2 was with the Map Annotation. It was working perfectly with my initial messy and un-refactored code. When I cleaned the application and implemented Environment Objects, it began to misbehave. Somewhere I am misunderstanding/incorrectly using Observables. Currently, when the user selects a new location by touch, *the temperature updates but then quickly reverts back to the previous value*. This does not happen when searching/pressing UserLocationButton from start-up. I believe the issue is related to either WeatherMapView being reloaded too many times, or being reloaded in a strange order. There are three observed variables being changed as a result of onTapGesture (mapCenter, selectedLocation, weatherViewModel), and one or all of these must be causing the unexpected behaviour.
4) I was unable to decide the best place for the searchLocation and moveToSearchLocation functions (see SearchLocationField). I am likely not adhering to MVVM architecture by including them inside a view. However I felt that their purpose is too tied with the SearchLocationField to be placed anywhere else.

Discussion:

Building on part 1, MetUCD part2 introduces Core Location and MapKit. The application now contains a Map that allows the user to view a weather forecast for their current location, or a location they have selected. Users can select news location via a search bar or by touching the map. A summary of the current forecast is shown on a widget, as well as a small temperature annotation. More detailed weather information can be found by tapping on the widget, which uses a SwiftUI sheet to display a WeatherView.

A new view model, MapViewModel was created. This view model handles the location services of the application, providing a way for Views to access the device's current location. It also maintains three important properties: mapCenter, weatherViewModel, and selectedLocation. No model was created as there is very little work being done to get/set these properties.

The entry point for the application is now MapView, as WeatherView is displayed Modally. The view is given an instance of MapViewModel as an environment object to ensure that all views are working with the same instance, and to avoid needing to pass the reference down through the chain of views. MapView is also given another environment object of helper class MapViewModel.Toggles. This class acts as a mean to keep track of whether the app is in 'searchMode' or not.

Inside MapView are three main views, WeatherMap, NavigationBar, and WeatherWidget. WeatherMap contains a MapReader and Map, NavigationBar contains controls for the app, and WeatherWidget gas been described earlier.

Most of the work in Part 2 was UI related, as opposed to part 1 where the majority of my time was spent working on the model and view model. This is mainly due to the increased level of user interaction, but also due to the increased styling required. One of the drawbacks of SwiftUI is the level of nesting that takes place as the UI grows. This became very apparent, and refactoring was essential.

The most difficult aspect was managing and dealing with state. SwiftUI views are designed to observe properties and reload themselves when said properties change. This can get confusing when you need to change multiple observed properties. As the complexity of the UI grew, debugging became increasingly difficult, and I experienced many unexpected behaviours. In fact I spent many hours trying to fix one of these behaviours (see issues section) to no avail.