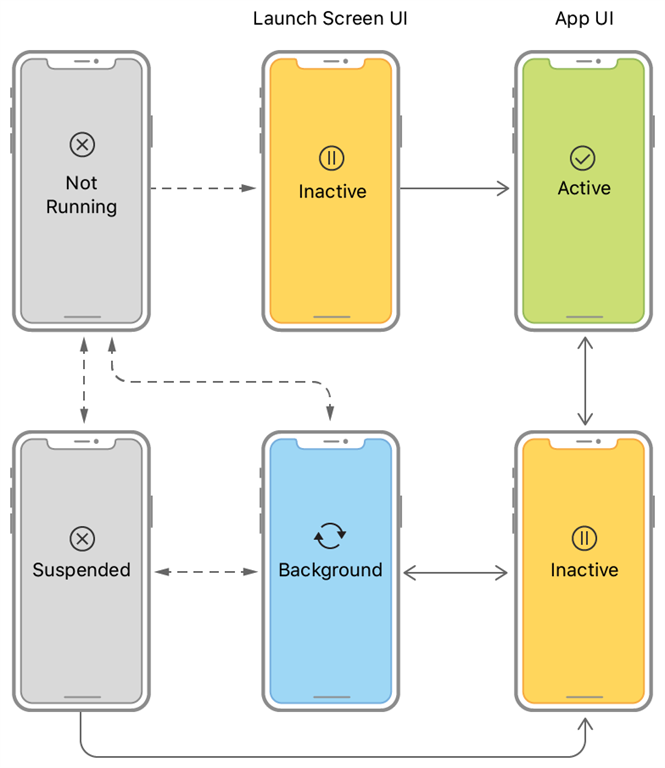
Alex Banning

November 3, 2024

**HW3: State Management Document**

**Possible States in an iOS App:**

There are several different lifecycle states an iOS can be in: active, inactive, background, suspended, and not running. The following image comes from Apple Developer documentation regarding the various states and flow between them:



In the active state, the app is in the foreground and can receive events. This is the main running state of the app’s lifecycle and where all the normal SwiftUI code is executed. In this state, the app is visible on the screen and responding to interaction and sensor input.

In the inactive state, an app is in the foreground but not receiving events. This is typically used as a transition state between active and background. An iOS app briefly enters an inactive state when closing and opening.

In the background state, an app has now moved totally to the background and is not receiving events and is not visible. In this state, an iOS app can still run code, but with the caveat that the operating system can free memory and other resources at any time, killing the app. This is a good state to run code to save critical data before it is lost by the OS freeing memory.

In the suspended state, the app is in the background and not running any code. This means it is still in memory and easily accessible to start up again, but it can’t perform any tasks in the meantime and it’s basically in a waiting state either to be killed or started again.

If an app is not running, then it is not in memory in the background or foreground, either by being freed by the OS or not being started at all.

A lot of the state management (e.g. leaving an app and coming back to the same spot/data entered) can be handled in SwiftUI using @State wrappers for variables. This tells the app to retain that data in a special place and to redraw any views that rely on the data whenever it is changed. We can see an example of this in tab views 2 and 3 of “HW3/StateManagement” XCode project.

iOS manages a lot of the states of an app on its own, and there are limited things a developer can do to manage these states using SwiftUI. One possible tool is to use an environment variable called “scenePhase.” The scenePhase environment variable keeps track of the main scene of the app and developers can use the onChange function to detect changes in the scene (i.e. active, inactive, and background states) and code accordingly (see tab view 1 of “HW3/StateManagement” XCode project). Another possible solution is to use a UIApplicationDelegate, but this feature has generally been deprecated in more modern versions of SwiftUI mostly because iOS has become increasingly powerful and manages even more of an app’s state due to new features like keeping commonly-used apps in memory for longer to quickly reload them.

**State Management in MeetMeHalfway:**

State management is important in this application because people may very well be switching between phone calls, text messages, Google and other apps when planning a trip to meet up with someone. Most importantly, I need to handle background states appropriately.

The key information a user needs to input for my app to work is Location 1 and Location 2. All the subsequent calculation and views rely on that data. Location 1 and 2 are state variable strings that update as the user types, so I am confident these are being stored in memory correctly; however, in the event of an interrupt I am worried about this memory being freed, so after hitting the MeetMeHalfway button and beginning calculation, these Locations should be stored in a temporary place on the device during the transition to background, perhaps .cachedDocuments in the default directory tree for the app.

Additionally, after the user has begun a search, meeting point selection is also important. MeetingPoint is being stored as a state variable as well, but again, if the user leaves for an extended period of time (perhaps talking on the phone to the friend they’re trying to meet), this data will be lost. We could load the Location 1 and 2 data from the cachedDocuments folder and re-search, or we could just store the meeting point selected in the cachedDocuments as well to speed up the process.

Finally, during the process of looking at nearby results for a particular meeting point, a user can select favorites. If the user selects “SaveTrip” at the top of this screen, a data structure consisting of location1, coordinate1, location2, coordinate2, midpointname, midpointcoordinate, and favoritePlaces[] is stored in a JSON blob. However, if there are interrupts during this process, I need to ensure the user can pick up wherever they left off last and all the favorites are saved correctly.

To reiterate, I am mostly concerned about the inactive and background states of my app. It is relatively simple overall, but relies on some key data that needs to remain secure. When the inactive state happens, I can use it as a trigger to begin storing data while my app is in the background, then when inactive triggers again I can pull any data if necessary (i.e. my app has been in background so long the OS freed some memory, and I need to recheck the critical values). Generally, iOS manages this stuff pretty well automatically, so I don’t anticipate needing to do much more than I already have, but I am going to rigorously test the app to ensure all of the data is being held and processed correctly.