# CSCI E65g: Mobile Application Development Using Swift and iOS

## Fall 2018

## Final Project

### Issued: 2018-11-13 Due: 2018-12-18 (LAST CLASS)

### Scoring

Graduate: 600 points

### Repository

Find through [Github Invitation](https://classroom.github.com/a/lRIBPm3l).

### Overview

This is the second phase of the final project, where you'll create a complete game App with networking capabilities and other features such as more navigation realistic game play, and history, and most interesting, a 2-player networking mode between different iPhones using a Google database service called Firebase.

### Reading

**Please note:** not all of the code referenced below as of 2018/10/17 has been published yet. Please be patient and start with reading and the non-persistence aspects first.

* (Carried over from previous assignment; was not included in actual Assignment 7 problems) UITabViewController These are fairly simple and don’t need much more than [the reference](https://developer.apple.com/documentation/uikit/uitabbarcontroller).
* [Overview of Firestore, the Cloud real-time database](https://firebase.google.com/docs/database/)
* [Getting started with Firestore on iOS](https://firebase.google.com/docs/database/ios/start)

### Setup

Follow [the setup guide](https://harvard-ios.github.io/csci-e65g-2018/assignments/Final-Project/setup.html).

### Requirements

#### Code Design

1. Different View Controllers will need access to a shared set of models. If you find this necessary, solve this using a global bootstrapping object, known as an *object broker* (in networked environments) or an *object factory* (in single App environments like ours). The factory will vend (supply) *Singleton* (only-one-of) using a *cooperative* design. Using classes can share references as long as they are obtained through the factory and not by instantiating their own models. However, the compiler does not prevent them from making that mistake. (See below in [Code Snippets](https://harvard-ios.github.io/csci-e65g-2018/assignments/Final-Project.html#code).)

#### Firebase Functionality

##### Document / Game representation

The game in progress should be stored as a move list in Firebase. This should be a property of the active game document. It can be implemented as a [map (Firebase version of Dictionary](https://firebase.google.com/docs/firestore/manage-data/data-types) of move numbers to move objects (where each move object is itself a map of properties to values), or as an explicit [sub-collection of “move documents”](https://firebase.google.com/docs/firestore/manage-data/structure-data).

The game should *not* be redundantly represented in two dimensions in Firebase.. Firebase is not well suited to store data like this, it would be redundant with respect to the move history, and the move history is important anyway for enforcing rules of some games or providing a foundation for long term game history storage.

The game in progress needs to have some minimal meta-data: each player's name, time started, and result. The player names allow the App instances to display their opponent's name properly. The rest may be useful as a double-check against App logic or for debugging.

As discussed below in functionality limits, only the game in progress needs to be represented, but keeping historical games is fairly easy with a few extra lines of code (e.g. requesting a new game document and populating it with the essential starting fields, rather than re-using the existing one.)

Firebase is *schemaless*, meaning you can add, mutate, and delete any data within any document at any time (subject to security restrictions). It is the responsibility of client code to impose order. This means there is no invokable template in Firebase for the root game document, or any document. The initial fields and values must be created by application code every time a new document is needed. (This was briefly explained incorrectly near the end of lecture 10.) When testing, you can interactively act as a Firebase agent through the web console.

#### App Functionality

The interface will have a few substantial changes. However this is not an interface-intensive project.

1. Embed the existing App in a [UITabBarController](https://developer.apple.com/documentation/uikit/uitabbarcontroller). One of the tab will be Game Play; the second, a scrollable HTML help file, and the third, a history of games played.
2. The HTML help file shold be be minimal formatted HTML to explain the rules of the game.
3. The game history file should be a read-only UITableView that shows the names of the players, date and time of game start (or end or both if you prefer), and the result or score. You are free to add statistics such as number of moves or captures as applicable. It should update immediately as soon as the game logic detects a game has finished. This history should be persisted using UserDefaults *or* Keyed Archiving or any other on-device mechanism of your choosing. It does not need to respond to any user touch events. It should sort by time. You can sort in forward or reverse order as you prefer.
4. Implement a real game. You can write your own, and this can be as simple as 3x3 tic-tac-toe, or integrate the (soon-to-be) supplied class logic for Connect Four, which will be more satisfying and fun to test.
5. The preferences need to have a setting to select the role of the player: 1 or 2. Choose your own UIView control for this. It should also be persisted in UserDefaultsand restored on App startup.
6. When a game starts, the other player’s name is learned from Firebase, and should be disabled from changing in Preferences while the game is in progress. The colors can remain fully settable and is not shared across Firebase.
7. When the App starts and there is a game history with no winner yet, and the Preferences are in Player 1 mode, the **New Game** button should become **Resume Game**. In that case, as Player 1 the App should upload the current move list to Firebase and update its internal (grid) model accordingly. Player 2 should only start their App (or switch their preference to Player 2, if already started) after Player 1 has completed this process, so that it can find the move list on Firebase immediately.
8. A **Reset storage** button should clear out the on-device pesisted model(s) so that the App can start from a known state. This will be very important for debugging and if Player 1 wants to start a new game rather than resuming current. Upon this reset, the **Resume Game** button should change back to **New Game**.
9. When a game starts, game-logic-affecting preferences such as grid size must also be locked down.
10. If firebase returns an error, it should propagate to the UI using a UIAlertController as well as to the console. You will find this very helpful for debugging. For the UI, you can truncate the message to 128 characters but does not need to be made end-user-friendly; make it test-engineer friendly. At that point, no further error-handling is necessary. It is acceptable to force-quit and restart the App after rectifying the situation (say, re-connecting to WiFi).
11. The status bar must now indicate when it is awaiting confirmation of a move upload and when it is waiting for the other user to move.
12. A countdown [timer](https://developer.apple.com/documentation/foundation/timer) should appear next to each player (you may want to rearrange the labels to be more separate, such as at opposite ends of the board, like in checkers or chess.) When the timer reaches zero without receiving a move, the game is forfeited and the waiting player should claim a win, to be shown in the status area, popped up in an alert, and to be entered in the game history. Again assume no slow network outages or network droppage that could make this claim false. An allowable timing alternative is to have a total time allotted for the entire game to be counted down when it is a player's turn to move. (The time limits can be hardcoded constants; making it a preference involves too much manual user coordination.)
13. The user should get a prominent indicator that it is their turn to move. It should be redundant, as in more than one visual: for example, greying out the other Player label background, setting the user's background color, *and* putting a large green dot next to their name.

#### App design implementation hints

##### State Design

There is now a new *third* model which represents the flow through the process of playing a game. The [State Model Design](https://harvard-ios.github.io/csci-e65g-2018/assignments/Final-Project/state-design.html) is so central to getting things working as to be a near-requirement. The storage is trivial: a single global state variable. All the challenge is in how various components *respond* to a change in state. If you are familiar with this sort of design already, you may implement an equivalent (or more fully-featured) variation. Otherwise stick to this.

##### Avoiding MVC errors

Your grasp of MVC has to be pretty solid to move onto networking and persistence. Avoid [these common errors](https://harvard-ios.github.io/csci-e65g-2018/assignments/Final-Project/common-errors.html).

##### Tab Bar Controller

To embed existing View Controllers in a new Tab Bar Controller, learn the manual way: Put a new TabBarController in Storyboard. Delete the automatically included pair of child controllers. Draw a *relationship* segue from parent (Tab VC) to child in the view navigator hierarchy.

Be sure to set the Item Titles to friendly names in the attribute inspector of the *child* view controllers. (It is intuitive to expect to find these settings in the Tab Bar as parent, but this is not the case.)

##### Notifications

Using Notifications can be verbose. Be alert for refactoring to make your life easier if you find repeated code for creating or receiving them.

You don't need to worry about ever calling removeObserver. That went away with iOS9. Observers are automatically freed up whenever the observing object is deinit'd.

##### Help View

Set the [NSAllowsArbitraryLoads](https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html) key to true in your Info.plist file to allow the WKWebView to display pages from non-HTTPS sources.

##### Persistence

* Resetting the local storage can be implemented by simply writing out a newly initialized model, rather than bothering to delete a file, but deleting is OK too.

##### Firebase

* Use [addSnapshotListener to listen to Firebase-driven updates. Note this works at the granularity of documents.](https://firebase.google.com/docs/firestore/query-data/listen)
* [Use](https://firebase.google.com/docs/firestore/query-data/listen)[setData](https://firebase.google.com/docs/firestore/manage-data/add-data) to upload data. Remember to set merge mode to avoid overwriting an existing move map!
* Avoid any polling loops to wait for expected data in Firebase. In practice, you'll be running two simulators right on the Mac, or one on your iPhone and one simulator. So all the manual coordination required, due to lack of automatic coordination, will not be too inconvenient.
* If you really want to implement notifications across the network as an extra credit effort, listen to designated Firebase nodes using [addSnapshotListener](https://firebase.google.com/docs/firestore/query-data/listen) in your document structure and use transactions when writes depend on previous reads. Always use listeners, which is *event-driven* design, not polling (repeatedly grabbing and checking data at an interval).
* The code outside of the View Controller and Views now falls into 4 categories. It would be a stretch to label all of this "the model" but for lack of a better term, we will, and refer to the model layers explicitly when we need to.
  1. The in-memory data and the basic mutations and queries (the well-specified "M" in MVC).
  2. The persistence layer that periodically translates it to an on-device representation and wakes it up from that representation when requested.
  3. The outbound network proxy that takes a move (or Player name) and uploads it to Firebase.
  4. The inbound network proxy that downloads initial Player and move data, and further listens for moves and mutates the in-memory data model, triggering appropriate notifications.

Rather than impose any strict design on this, the only requirement is that *each of the four functionality groupings much go in their own source file* (further refactoring is allowed, but then you'll need to make Folder "Groups" in Xcode for a multi-file ipmlementation of a single category). It is still allowed that they are all members functions and properties of a single Swift class.

* The inbound proxy must use Notifications to broadcast arrival of data, which needs to be encoded into the userDict field then carefully unpacked by the receiver. There are important reasons for this. Most of the time when network data arrives in a callback function, that function *is not executed on the main thread*. By electing to receive Notifications on the main thread regardless of where they are broadcast from, we can safely stay on the main thread for all data and UI manipulation.
* Neither inbound nor outbound proxy should capture self, instead using local let declarations to capture only what's needed. This is subtle and I expect questions during implementation.
* (Relating to the above:) Any network-based arrival of data, or notification of successful upload of data, *absolutely must not mutate* the in-memory data model *or* the UI. It must be very simple and limited: using value types only, it should package up the information or new data in a convenient form and post it in a Notification.

#### Code Snippets

##### The Global Singleton factory

See [this distillation](https://harvard-ios.github.io/csci-e65g-2018/assignments/Final-Project/code/factory.swift) of a long discussion on Google Groups. Note it is still acceptable to use the private-initializer style Ben showed or the style shown here.

#### Functionality limits

Please note: several of these requirements are intentionally simplifying to provide a realistic path for all students to create a working two-player networked game by the end of the semester. Due to the different experience levels in programming, ranging from 2 to 15 years, some students may finish these requirements earlier. There are a number of features that would be necessary for a realistic App release. The following will compare commercially necessary features versus actual project requirements:

1. Switching between local and network mode. We will assume *networked mode only* for the project.
2. Handling an unreliable or slow network. We will assume it is reliable and fast (e.g. all communications happen well under the automatic 10-second timeout of the Firebase library).
3. Detecting or assigning who is Player 1 (first to move) and who is Player 2. Players will coordinate out-of-band (e.g. text message) and configure their own App instance to Player 1 mode or Player 2 mode via a selection view of your choosing in Preferences. If they do not coordinate properly before beginning the game, the result is unspecified (e.g. game data loss or corruption is allowed without penalty.)
4. Storing game history in Firebase. The move history will be persisted only to the local device. On receiving a **New Game** command, Player 1 will upload any game in progress to Firebase and Player 2 will download it. This means the App may always connect to, monitor, and mutate, the same active game root node every time.
5. Maintaing a user identity with which to authenticate to the system, and having only limited permissions thereto. For us, as long as the App has the requisiteGoogleServices.plist file, it will have full read/write access to the firestore data.
6. Establishing a *login session* (naturally, tied to the user identity) to allow browsing of other login sessions and challenging/accepting new game invites to one other. For us, assume only two players will connect to a particular firebase project at a time, as mentioned in the aforemetnioned out-of-band communication. Again if this is violated (say if 3 App instances connect) the results are unspecified.
7. Allowing multiple simultaneous games. This is closely related to the previous point. A commercial system would allow many simultaneous games and keep them separate. It might also limit the number of open games a particular user is allowed to have, or have other limits to create a paywall.
8. Allow the choice of different games. We will assume only one game logic shared by both App instances, and that if any game logic affecting preferences are possible such as grid size, that the users will also coordinate this out-of-band before starting a game.

If students *do* finish early, they can and should work on whatever features provide the most satisfaction and may discuss this freely on the forum. The most fundamental, meatiest theory to learn is *transactions* which occur in every distributed database system. A [firebase transaction](https://firebase.google.com/docs/firestore/manage-data/transactions) is critical piece to assign player 1 and 2 reliably based on a first-come first-served basis for the [test and set election algorithm.](https://harvard-ios.github.io/csci-e65g-2018/assignments/Final-Project/code/test-and-set.txt)