Assignment V: Persistent History in CodeWord Breaker

Objective

You'll take your final step with your CodeWord Breaker app and make it remember permanently all the games you've played. This is all about learning how to use SwiftData and then integrating it into your UI.

Be sure to review the Hints section below!

Due

This assignment is due by the end of the quarter.

Materials

You will build directly on top of your Assignment 4 code.

As always, you are welcome to use any code from lecture.

Do not use AI. The code in your application must be entirely your own work (other than what was shown in lecture). **Do not import code from the internet or AI.** If you find yourself thinking you need to go to the internet learn how to do any of the Required Tasks in this assignment, you might be on the wrong track.

Required Tasks

1. Make your list of games persist between launchings by storing all games in SwiftData.
2. You must otherwise support all A4 Required Task functionality except that your Model will now obviously be using `@Model` instead of `@Observable`.
3. Make your list of games searchable by word. In other words, get a search string from the user and search for games that have an attempt or master code which contains the string. It's up to you whether you want the search string to match guesses (i.e. not-yet-attempted, in-progress guess input).
4. Provide an option to the user to filter your list of games by whether the game has been completed (i.e. successfully guessed) or not.
5. Allow the user to delete games from the list.
6. Don't forget that if you did not add settings to your application in A4, you must do it now for A5.

Hints

1. You might want to add a `created` `var` to your game and sort games with no attempts in reverse-`created` order (rather than some random order). This is just a Hint, not a Required Task.
2. Be careful to get your case-sensitivity right when searching. SQL searches are completely case-sensitive. In other words, be sure to convert whatever the user types in the search field to the same case as the case of your pegs in the database.
3. You might find it easier to store the "pegs" in the database as a `String` rather than as `[String]`. This might make the searching part easier. You could have the rest of your code continue to interact with the words as an `[String]` by creating a computed `var` that lets you get/set the word in that format (i.e. as an `[String]`). `missingPeg` might want to be _ or " " (space) or some such with this approach?
4. No other Hints this week. The idea is that you are now facile enough with SwiftUI that doing something new like SwiftData is something you can figure out how to do without too much assistance.

Things to Learn

Here is a partial list of concepts this assignment is intended to let you gain practice with or otherwise demonstrate your knowledge of.

1. SwiftData
2. Clean coding.

Evaluation

In all of the assignments this quarter, writing quality code that builds without warnings or errors, and then testing the resulting application and iterating until it functions properly is the goal.

Here are the most common reasons assignments are marked down:

- Project does not build.
- One or more items in the Required Tasks section was not satisfied.
- A fundamental concept was not understood.
- Project does not build without warnings.
- Code is visually sloppy and hard to read (e.g. indentation is not consistent, etc.).
- Your solution is difficult (or impossible) for someone reading the code to understand due to lack of comments, poor variable/method names, poor solution structure, long methods, etc.

Often students ask "how much commenting of my code do I need to do?" The answer is that your code must be easily and completely understandable by anyone reading it. You can assume that the reader knows the iOS API and knows how the CodeBreaker game code from the lectures works, but should not assume that they already know your (or any) solution to the assignment.

Challenge Mode

Unlike in previous assignments, it probably will be fun and not that much work to continue to support Challenge Modes from A4. Continuing to support 3, 4, 5 and 6 letter words might be zero work at all (depending on how you implemented this), but if you did the elapsed time Challenge Mode(s) in A4, you'll probably need to make `startTime` `@Transient` and update `elapsedTime` when the `modelContext` or `startTime` changes (as seen in lecture).

The primary Challenge Mode this week is to clean up your code. Time constraints may have left you with some gnarly code and now is the time to go back and clean that up.

1. 1-2 pts. Clean up your code to the maximum extent possible. You've learned a lot about what makes code look "clean" and of course you probably have a lot of experience outside of this course making clean code. Show us! The bar for earning even one point is very high here. Two points would only go to truly elegant, pristine codebases. At a minimum, you should address all of these things …

    Getting rid of crufty code (especially any created or left over from incorporating code from lecture that doesn't really apply to your application anymore).

    Really good variable and type names everywhere and clear comments, especially in-line during potentially confusing stretches of code.

    Appropriate decomposition, breaking up functions/computed `var`s with too many lines of code into more sensible, well-named sub-components, "helicopter" `View`s, view modifiers or `extension`s.

    Organizing your code files into sensible folders and using `//MARK` to organize your code within a file and to mark Data Flow.

    Eliminating as many magic numbers as possible and putting the remainder into well-named "constants" structs.

2. 1 pt. Make your settings persist between launches of your application. If you did this Challenge Mode in A4, you can still get a point for it here if you use SwiftData to accomplish the persistence. If you use SwiftData, then you'll no longer need `@Environment` to make your `Settings` available everywhere because you can get it with `@Query` anywhere you need to. Be careful to add `Settings.self` to the `modelContainer` view modifier. One of the trickier parts of using SwiftData to persist your Settings is making a `Color` persist. You can do it either by converting it to a `String` or by making it `Codable`. You are allowed to use AI to generate the code necessary to do either of these things. You'll also need to figure out where to insert your (probably one-and-only) Settings `class` instance into the database.