LambdaSchool / **ios-journal-coredata**

No description, website, or topics provided.

4 commits **4** branches **0** releases **1** contributor

Branch: **day4** View #28 Create new file Upload files Find file Clone or download

This branch is 3 commits ahead of master. #104 Compare

SpencerCurtis Update README.md Latest commit `03c936f` on Aug 16, 2018

README.md Update README.md 5 months ago

README.md

Journal (Core Data) Day 4

Introduction

For today's project, you will update Journal to update its Core Data data from the server in the background. This will allow you to practice more complex Core Data scenarios using multiple managed object contexts, as well as using concurrency with Core Data. You will be modifying an existing codebase to be more performant and correct. To prepare for many of these changes, you'll *refactor* your code, meaning you'll restructure it so that it's functionality can be updated without compromising its readability and maintainability.

The instructions for this project are intentionally somewhat less detailed that previous instructions this week. This project will require you to think about and understand how to architect your app to use multiple contexts and concurrency correctly. As always, follow the 20-minute rule, but don't be afraid to ask for help as you work.

Instructions

Use the Journal project you made yesterday. Create a new branch called `day4`. When you finish today's instructions and go to make a pull request, be sure to select the original repository's `day4` branch as the base branch, and your own `day4` branch as the compare branch.

Part 0 - Troubleshooting

Before starting, run your app with the `-com.apple.CoreData.ConcurrencyDebug 1` launch argument. Excercise all functions of the app and note whether any Core Data concurrency assertions are triggered. Were any triggered (ie. did the app crash)? If so, why? Today you'll fix these problems while simultaneously improving the overall performance of the app.

Part 1 - Refactor to Prepare for Multiple Contexts

Start by refactoring some of your code to be better prepared to switch to using a separate managed object context for syncing operations.

1. Make your `fetchSingleEntry` method accept a `context` argument that it uses to fetch from.

2. Change your convenience initalizer for creating an `Entry` from an `EntryRepresentation` to accept a context in which to create the new `Entry`.
3. Extract the code responsible for iterating through the array of fetched `EntryRepresentation`s and updating or creating corresponding `Tasks`. Put it in a function that takes a context.
4. Update code that calls these functions to supply a context. For now, that should continue to be the main context you've used so far.
5. Test your app to be sure your refactoring hasn't broken anything.

Part 2 - Use Concurrency APIs to Ensure Correctness

Even though you haven't yet updated your code to use multiple contexts, you can prepare for that by using Core Data's concurrency APIs to ensure that regardless of context, your code is concurrency-safe. Core Data is designed in such a way that you can write concurrency-correct code without having to keep track of and maintain dispatch queues, etc. yourself.

Remember that **any** use of managed objects or a managed object context must be done in a `perform` or `performAndWait` call for non-main-queue contexts. Even for main-queue contexts, it is safe and valid to use `perform` or `performAndWait`.

1. Go through each function that deals with managed objects. Decide whether it should ensure concurrency correctness itself, or whether responsibility for correctness should be left up to its caller.
2. Update each function to do its work using `perform()` or `performAndWait()` on the appropriate context.
3. Run your app with the `-com.apple.CoreData.ConcurrencyDebug 1` launch argument. Excercise all functions of the app and verify that no Core Data concurrency assertions are triggered (ie. the app shouldn't crash).

Part 3 - Use a Background Context for Syncing

1. Update your `fetchEntriesFromServer(...)` method so that it creates a new background context, and does all Core Data work on this context. It should update/create tasks from the fetched data on this context.
2. Save the context only after the update/creation process is complete. Remember that `save()` itself must be called on the context's private queue using `perform()` or `performAndWait()`.
3. In your `CoreDataStack`, after creating the container, set its `viewContext`'s `automaticallyMergesChangesFromParent` property to true. This is required for the `viewContext` (ie. the main context) to be updated with changes saved in a background context. In this case, the `viewContext`'s parent is the persistent store (coordinator), **not** another context.

Part 4 - Testing

Thoroughly test your app to be sure that all features continue to function correctly. From an end user perspective, the app should behave **exactly** as it did yesterday. While you're testing the app, be sure the `-com.apple.CoreData.ConcurrencyDebug 1` launch argument is set. Verify that no Core Data multithreading assertions are triggered.

If the app behaves correctly and doesn't trigger any assertions, you're done! Submit your pull request. If you have time left, try the suggestions in the Go Further section below.

Go Further

If you'd like a further challenge, try using a separate managed object context in the detail view controller. This managed object context can be used a scratchpad, so that operations on the task being created/edited occur in it, and are only saved to the main context after the user taps the Save button. You can make the detail view controller's context a child of the main context. You can also use one of the other multiple managed object context setups. Because you can not "switch" which context an object instance is in, you'll need to use `NSManagedObjectID` and related APIs to communicate to the detail view controller which task it should be displaying/editing.

Just like yesterday, try to solidify today's concepts by starting over and rewriting the project from where you started today. Or even better, try to write the entire project with both today and yesterday's content from scratch. Use these instructions as sparingly as possible to help you practice recall.