audreywelch / **ios-journal-performance**
forked from LambdaSchool/ios-journal-performance

No description, website, or topics provided. Edit

Manage topics

4 commits **1** branch **0** releases **2** contributors

Branch: **master** New pull request | Create new file | Upload files | Find file | Clone or download

This branch is even with LambdaSchool:master. Pull request Compare


SpencerCurtis Update EntryController with new (working) baseURL Latest commit cd9178e on Dec 4, 2018

JournalCoreData.xcodeproj	Add starter project	5 months ago
JournalCoreData	Update EntryController with new (working) baseURL	2 months ago
.gitignore	Initial commit of README	5 months ago
README.md	Update README.md	5 months ago

README.md

Journal Performance Tuning

The goal of this project is to use Instruments to improve the performance of Journal with large data sets. You'll also practice using Core Data in an efficient manner. This project will help you practice the concepts learned in the Debugging II - Instruments module of Sprint 8. After completing the lesson material and this project, you should be able to:

- understand and explain the purpose of Instruments and the problems it helps solve
- use Instruments' Time Profiler instrument to understand slowness in an application
- use Instruments' Allocations instrument to determine the cause of high memory usage

Part 0 - Fork and Clone Project Start

You'll be starting with the previously-built Journal app. This repo includes the un-optimized version of the app that you've worked on before. In order to ensure that you're at a consistent starting point, you should use this version rather than the app you built yourself. Do the following:

1. Fork this repository to your own GitHub account.
2. Clone your fork to your local machine.
3. Open Journal.xcodeproj

Part 1 - Profiling

Before you begin trying to improve the app's performance, you need to understand exactly where it performs poorly, and investigate the cause of the slowness.

1. Begin by deleting the existing app from your simulator to start with a fresh, empty Core Data store.
2. Run the app. Observe the CPU usage gauge in the Debug Navigator. Note that CPU usage is very high, and the UI remains empty.
3. Pause in the debugger. You may notice by looking at the call stack that the app is in the middle of the sync process, which continues for a very long time (at least many minutes, if not longer).
4. Add a print statement before syncing starts and after it finishes so that you can measure total time spent syncing. This will be useful as you work on improvements to this process.

Now that you've seen that the initial sync from the server is slow, you should use Instruments to profile and find out what is taking so much time.

1. Quit the app, and delete it from the simulator again to start fresh.
2. From the Product menu, select Profile (or press command-I). This will rebuild the app and start Instruments.
3. In Instruments, choose the Time Profiler instrument.
4. In the Instruments window that appears, click the Record button in the top left corner to launch the app and begin profilng.
5. Let the app run for a while to gather data. When you think you've gotten enough data, click the stop button (you needn't wait for a full sync to finish).
6. Use Instruments Time Profiler pane to understand which function(s)/code paths are taking the most time. Which thread(s) are taking the most time? fetchSingleEntryFromPersistentStore function

Part 2 - Make Background Sync Performance Improvements

Having discovered the slow code in the app, think about how you might fix it? Are there inefficiencies in the method the app is using to fetch data, update or create corresponding objects in Core Data, and update the UI? How might you refactor the code to eliminate those inefficiencies?

You need to figure most of this out yourself, so we won't give too much away. But here are some things to think about:

- Is the actual network request slow? I don't think so.
- What about parsing/decoding the JSON data into `EntryRepresentations`? I don't think so.
- Is the code that reconciles the received data with Core Data slow? If so, which specific part(s) of that code are taking the most time?
- For each `EntryRepresentation` we do a fetch request to find an existing corresponding Core Data `Entry` object if there is one. Is this efficient? This function is looping through each Entry Representation in the Entries array and fetching it before checking if it matches and updating
- How could you reduce the number of fetch requests you're required to do?

Click below to get more hints (don't use unless you really need them!)

▼ Hint 1

Fetch request predicates can use the `IN` operator to check for a value in an array. e.g. `NSPredicate("identifier IN %@", arrayOfIdentifiers)`.

▼ Hint 2

Looking up values in a dictionary given a key is quite fast. Even with a large dictionary, `let value = dictionary[key]` will be efficient. Can you use this in your code to improve performance?

Work on this for about 1.5-2 hours, making sure you ask PMs or instructors if you get stuck. You should make improvements then profile as a continuous, iterative project (see Part 2).

Part 2 - Profile Again!

Tuning the performance of your code is usually an iterative process. You profile, make changes, then profile again to see if your changes worked, and where remaining bottlenecks are, before making more changes and continuing the process.

When you're confident that your solution to the background fetch/sync process is good, profile one last time. Do you see remaining slowness? Is the UI fast to update after a successful sync? If you still see slowness, profile again! You may notice that despite major improvements to the process of fetching data and reconciling it with Core Data, the subsequent update to the table view is slow, and the UI stays locked up. Repeat the profiling process until you understand something about the slowness.

Part 3 - Improve UI Update

When you make changes to a background managed object context, then save that context, you need to merge those changes into the main context. So far, we've made the main context do this automatically by setting its `automaticallyMergesChangesFromParent` to true. With this property set, when you save the background sync context, the main context automatically merges any changes in, as well as triggering the fetched results controller to update its data. This process can be very slow for large data sets.

You know that after a sync, there may be a very large change dataset that the managed object context and/or fetched results controller need to process. Profiling should reveal the exact mechanism by which the update proceeds and where its slow? Is there an alternate way for you to do this process?

Click below to see hints (don't use until you need them!)

▼ Hint 1
You can tell a managed object context to completely reset itself and re-read data from its parent (e.g. the persistent store (coordinator)) by calling `reset()`.
▼ Hint 2
Could you tear down the fetched results controller and create a new one with fresh data?
As always, if you need help, ask your PM or an instructor.

Go Farther

If you finish with time to spare or would like to push yourself, try the following bonus goals:

- Add UI to show the progress of the sync progress.
- Use the Allocations instrument to profile the app's memory usage. Is there room for improvement? Where?
- Use Firebase's [support] for range queries to fetch data a bit at a time and sync incrementally.