audreywelch / **ios-recipes**
forked from LambdaSchool/ios-recipes

No description, website, or topics provided. Edit

Manage topics

7 commits | 1 branch | 0 releases | 3 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

armadsen Merge pull request LambdaSchool#22 from WilliamBundy/master ... Latest commit 806a38a on Oct 19

Recipes	Update project to work in Xcode 9	4 months ago
README.md	Clarified a step in MainViewController	2 months ago

README.md

Recipes

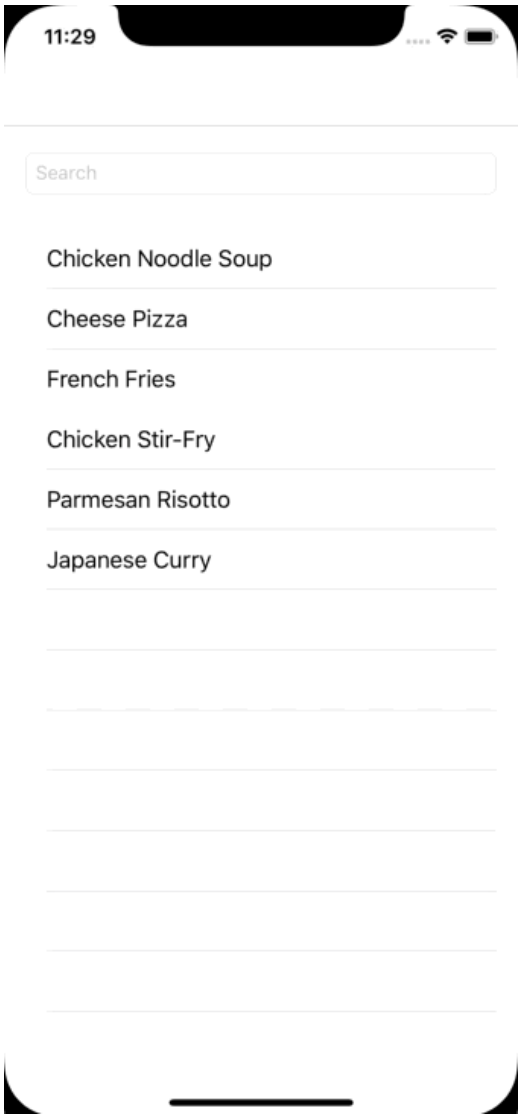
A student that completes this project shows that they can:

- understand and explain what a closure is and common scenarios for their use
- understand and use Swift's closure syntax
- use Swift's sort, filter, map, compactMap, and reduce functions
- understand and explain what concurrency is, and reasons why it is used
- use GCD APIs to dispatch work to another queue
- understand and explain the role of the main queue in iOS apps

Introduction

Recipes fetches a list of recipes from an API and allows the user to view instructions for them.

Please look at the screen recording below to know what the finished project should look like:



Instructions

Please fork and clone this repository. This repository has a starter project with the model and a network client already created for you.

Part 1 - Storyboard Layout

1. Using either the `UIViewController` scene provided in the Main.storyboard, or a new one if you already deleted it:
    - Embed it in a navigation controller. Set the navigation controller as the initial view controller.
    - Add a `UITextField` just below the navigation bar.
    - Add a container view right below the text field, and have it fill the rest of the view controller.
    - Delete the view controller attached to the container view. Add a `UITableViewController` scene, and embed it in the container view. Give the embed segue an identifier.
    - Create a Cocoa Touch subclass of `UIViewController` called `MainViewController`. Set this scene's class to `MainViewController`.
    - Add an outlet from the text field to the `MainViewController` class. Add an action from the text field as well, with the event as "Editing Did End". (It should be the default event)

2. In the embedded `UITableViewController`:
    - Set the cell's style to "Basic" if it isn't already.
    - Give the cell an identifier.
    - Create a Cocoa Touch subclass of `UITableViewController` called `RecipesTableViewController`. Set this scene's class to `RecipesTableViewController`.
    - Add a `UIViewController` scene. This will serve as the table view's detail view controller.
    - Create a "Show" segue from the cell to the detail view controller. Give the segue an identifier.
3. In the detail view controller scene:
    - Add a label to the top of the scene. This will display the name of the recipe. Set its text to be fairly large.
    - Add a text **view** under the label, and have it fill the rest of the view controller. This will display the instructions for the recipe. Text views work fantastic for displaying large strings. They have built in support for selecting text and scrolling if the text is too large to be displayed all at once without you having to do any extra work.
    - With the text view selected, uncheck the `Editable` box in Attributes Inspector. With it checked, the user would be able to edit the text view's text, so in order to make it more like a label we make the text view uneditable, but they're still able to scroll and select text.
    - Create a Cocoa Touch subclass of `UIViewController` called `RecipeDetailViewController`. Set this scene's class to `RecipeDetailViewController`.
    - Add outlets from the label and text view to the `RecipeDetailViewController` class

Part 2 - View Controller Implementation

We're going to start the view controller implementation backwards from the actual flow of the application, starting with the `RecipeDetailViewController`.

RecipeDetailViewController

1. In the `RecipeDetailViewController`, add a variable `recipe: Recipe?`.
2. Create a function called `updateViews()`. This should take the values of the `recipe` and place them in the corresponding outlets. The recipe's `name` should go in the label, and the `instructions` in the text view. Check that the view is loaded when unwrapping the `recipe` object by using the view controller's `isViewLoaded` property.
3. Call `updateViews()` in the `viewDidLoad()`, and in the `didSet` of the `recipe` variable.

RecipesTableViewController

Since this table view controller is embedded in the `MainViewController`, it will be relatively simple as its only job is to fill out the table view.

1. Add a variable `recipes: [Recipe] = []`. Add a `didSet` property observer that reloads the table view.
2. Fill out the `numberOfRowsInSection`, and `cellForRowAt` methods. The cell should display the name of its corresponding `Recipe` object.
3. In the `prepare(for segue: ...)`, check the segue's identifier. If the segue is going to the `RecipeDetailViewController` and triggered by tapping a cell on this table view controller, pass the `Recipe` that corresponds with the cell that was tapped.

MainViewController

1. In the `MainViewController`, create a constant called `networkClient`. Set its value to a new instance of `RecipesNetworkClient`.

2. Create a variable `allRecipes: [Recipe] = []`.
3. In the `viewDidLoad`, call the `networkClient`'s `fetchRecipes` method. In its completion closure, if there is an error, NSLog it, and return from the function. If there is no error, set the value of `allRecipes` to recipes returned in this completion closure.
4. Create a variable `recipesTableViewController: RecipesTableViewController?`. Later, we will make it hold a reference to the embedded table view controller.
5. In the `prepare(for segue: ...)`, check for the embed segue's identifier. If it is, set the `recipesTableViewController` variable to the segue's `destination`. You will need to cast the view controller as the correct subclass.
6. Create a variable `filteredRecipes: [Recipe] = []`.
7. Create a function called `filterRecipes()`. This will take the text from the text field and filter the recipes with it. In the function:
    - Unwrap the search term and make sure it isn't an empty string. If search term is empty or nil, set the value of `filteredRecipes` to `allRecipes`. If there is no search term, that means you should display all of the recipes.
    - If there is a non-empty search term in the text field, using the `filter` higher-order function to filter the `allRecipes` array. It should filter by checking if the recipe's `name` or `instructions` contains the search term. Set the value of the `filteredRecipes` to the result of the `filter` method.
8. In the action of the text field, call `resignFirstResponder()` on the text field, then call `filterRecipes()`.
9. Add a `didSet` property observer to the `filteredRecipes` variable. It should set the `recipeTableViewController`'s `recipes` to the `filteredRecipes`.
10. Add a `didSet` property observer to the `recipesTableViewController` variable. Just like the `didSet` in the `filteredRecipes`, it should set the `recipeTableViewController`'s `recipes` to the `filteredRecipes`. Between these two property observers, it will ensure that the table view controller has the most current array of recipes, whether filtered or not.
11. Add a `didSet` property observer to the `allRecipes` variable. Call `filterRecipes()` in it.

Go Further

- Add local persistence so that the recipes only have to be fetched from the API once.
- Add the ability to update the recipes.