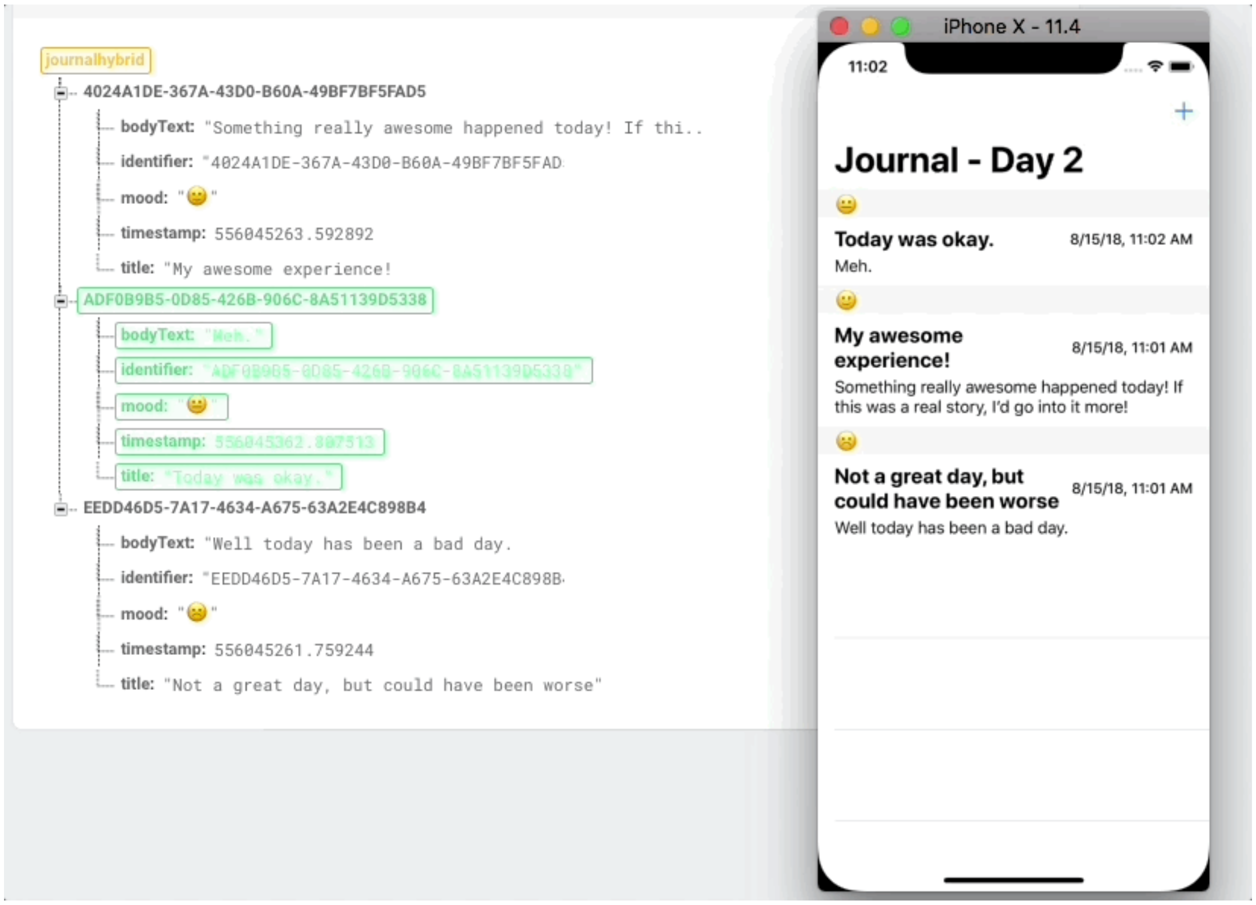
# Journal (Core Data) Day 3

## Introduction

Today's project will continue to add more functionality to the Journal project. You will add syncing between Core Data and a server. In this case, Firebase will be used as the server. Since we have already set up an `NSFetchedResultsController` to update the table view when the persistent store's managed objects change, you will only need to do work on the model and model controller layers. This is a good example of adding functionality without having to tear up your entire application.

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



```
journalhybrid
- 4024A1DE-367A-43D0-B60A-49BF7BF5FAD5
    bodyText: "Something really awesome happened today! If thi..
    identifier: "4024A1DE-367A-43D0-B60A-49BF7BF5FAD
    mood: "😐"
    timestamp: 556045263.592892
    title: "My awesome experience!
- ADF0B9B5-0D85-426B-906C-8A51139D5338
    bodyText: "Meh."
    identifier: "ADF0B9B5-0D85-426B-906C-8A51139D5338"
    mood: "😐"
    timestamp: 556045362.807313
    title: "Today was okay."
- EEDD46D5-7A17-4634-A675-63A2E4C898B4
    bodyText: "Well today has been a bad day.
    identifier: "EEDD46D5-7A17-4634-A675-63A2E4C898B
    mood: "☹"
    timestamp: 556045261.759244
    title: "Not a great day, but could have been worse"
```

Journal - Day 2

😐

**Today was okay.** 8/15/18, 11:02 AM
Meh.

😐

**My awesome experience!** 8/15/18, 11:01 AM
Something really awesome happened today! If this was a real story, I'd go into it more!

☹

**Not a great day, but could have been worse** 8/15/18, 11:01 AM
Well today has been a bad day.

## Instructions

Use the Journal project you made yesterday. Create a new branch called `day3`. When you finish today's instructions and go to make a pull request, be sure to select the original repository's `day3` branch as the base branch, and your own `day3` branch as the compare branch.

### Part 1 - PUTting and deleting Entries

First, you'll set up the ability to PUT entries to Firebase. Since the `Entry` entity already has an `identifier` attribute, there is no need to make a new model version.

1. Create a new Firebase project for this application. Choose to use the "Realtime Database" and set it to testing mode so no authentication is required.

#### Entry+Encodable

1. Create a new Swift file called "Entry+Encodable.swift".
2. Create an extension on `Entry`, and adopt the `Encodable` protocol. Since you're using "Class Definition" codegen in your data model, you don't have access to the `Entry` class directly. The effect this has is that when you adopt `Encodable`, its required method in can't be synthesized for you.
3. First create a `CodingKeys` enum. It should have string raw values, and adopt the `CodingKey` protocol. Add a case for the five attributes in the `Entry` entity.
4. Now that you have the coding keys, you can implement the required `public func encode(to encoder: Encoder) throws` method yourself.

  - In the `encode(to encoder: ...)` method, create a variable called `container`. Set use the `encoder` parameter's `container(keyedBy: ...)` method, and pass in `CodingKeys.self` to it.
  - Use the container's `encode(value: ..., forKey: ...)` method to encode each of the five attributes of the `Entry` individually.

You may be wondering why you're adopting `Encodable` and not `Codable`. `Codable` is just a combination of the `Encodable` and `Decocable` protocols. This will be explained later in the instructions.

#### EntryController

5. In the `EntryController`, add a `baseURL: URL` constant that is the URL from the new Firebase database you created for this app.
6. Create a function called `put`, that takes in an entry and has an escaping completion closure. The closure should return an optional error. Give this completion closure a default value of an empty closure. (e.g. `{ _ in }`). This will allow you to use the completion closure if you want to do something when `completion` is called or just not worry about doing anything after knowing the data task has completed. This method should:
   - Take the `baseURL` and append the identifier of the entry parameter to it. Add the `"json"` extension to the URL as well.
   - Create a `URLRequest` object. Set its HTTP method to PUT.
   - Using `JSONEncoder`, encode the entry into JSON data. Set the URL request's `httpBody` to this data.
   - Perform a `URLSessionDataTask` with the request, and handle any errors. Make sure to call completion and resume the data task.
7. Call the `put` method in the `createEntry` and `update(entry: ...)` methods.
8. Create a `deleteEntryFromServer` method. It should take in an entry, and a completion closure that returns an optional error. Again, give the closure a default value of an empty closure. This method should:
   - Create a URL from the `baseURL` and append the entry parameter's identifier to it. Also append the "json" extension to the URL as well. This URL should be formatted the same as the URL you would use to PUT an entry to Firebase.
   - Create a `URLRequest` object, and set its HTTP method to DELETE.
   - Perform a `URLSessionDataTask` with the request and handle any errors. Call completion and don't forget to resume the data task.
9. Call the `deleteEntryFromServer` method in your `delete(entry: ...)` method.

Test the app at this point. You should be able to both create and update entries and they will be sent to Firebase as well as to the `NSPersistentStore` on the device. You should also be able to delete entries from Firebase also.

### Part 2 - Syncing Databases

Something to keep in mind when trying to sync multiple databases like we are in this case is that you need to make sure you don't duplicate data. For example, say you have an entry saved in your persistent store on the device, and on Firebase. If you were to go about fetching the entries from Firebase and decoding them into `Entry` objects like you've done previously before today, you would end up with a duplicate of the entry in your persistent store. This would occur every single time that you fetch the entry from Firebase.

The way to prevent this is to create an intermediate data type between the JSON and the `Entry` class that will serve as a temporary representation of an `Entry` without being added to a managed object context.

#### EntryRepresentation

1. Create a new Swift file called "EntryRepresentation". In the file, create a struct called `EntryRepresentation`.
2. Adopt the `Decodable` protocol.

3. Add a property in this struct for each attribute in the `Entry` model. Their names should match exactly or else the JSON from Firebase will not decode into this struct properly.
4. Adopt the Equatable protocol.
5. Outside of the `EntryRepresentation` struct, implement the `==` method. The left hand side ( `lhs` ) should be of type `EntryRepresentation` and the right hand side ( `rhs` ) should be an `Entry` .
6. Implement the `==` method, this time with `Entry` as the left hand side and `EntryRepresentation` as the right hand side. You can simply return `rhs == lhs` , since you've implemented the logic to compare the two objects in the first `==` implementation.
7. Implement the `!=` method with `EntryRepresentation` as the left hand side, and `Entry` as the right hand side. This should return `!(rhs == lhs)`
8. Implement the `!=` again but swapping the left hand side's type to `Entry` and `EntryRepresentation` as the right hand side. Simply return `rhs != lhs` .
9. In the "Entry+Convenience.swift" file, add a new convenience initializer. This initializer should be failable. It should take in an `EntryRepresentation` parameter. This should simply pass the values from the entry representation to the convenience initializer you made earlier in the project.

**EntryController**

The goal when fetching the entries from Firebase is to go through each fetched entry and check a couple things:

- **Is there a corresponding entry in the device's persistent store?**
  - No, so create a new `Entry` object. (This would happen if someone else created an entry on their device and you don't have it on your device yet)
  - Yes. Are its values different from the entry fetched from Firebase? If so, then update the entry in the persistent store with the new values from the entry from Firebase.

You'll use the `EntryRepresentation` to do this. It will let you decode the JSON as `EntryRepresentation` s, perform these checks and either create an actual `Entry` if one doesn't exist on the device or update an existing one with its decoded values.

Back in the `EntryController` , you will make a couple methods that will help when fetching the entries from Firebase.

1. Create a new "Update" function called `update` . It should take in an `Entry` whose values should be updated, and an `EntryRepresentation` to take the values from. This should simply set each of the `Entry` 's values to the `EntryRepresentation` 's corresponding values. **DO NOT** call `saveToPersistentStore` in this method. It will be explained why later on.
2. Create a method called `fetchSingleEntryFromPersistentStore` . This method should take in a string that represents an entry's identifier, and return an optional `Entry` . This method should:
   - Create a fetch request from `Entry` object.
   - Give the fetch request an `NSPredicate` . This predicate should see if the `identifier` attribute in the `Entry` is equal to the identifier parameter of this function. Refer to the hint below if you need help with the predicate.
   - ▶ Predicate Hint:
   - Perform the fetch request on your core data stack's `mainContext` and return the first `Entry` from the array you get back. In theory, there should only be one entry fetched anyway, because the predicate uses the entry's identifier. Handle the potential error from performing the fetch request.
3. Create a function called `fetchEntriesFromServer` . It should have a completion closure that returns an optional error and its default value should be an empty closure. This method should:

- Take the `baseURL` and add the "json" extension to it.
- Perform a GET `URLSessionDataTask` with the url you just set up.
- In the completion of the data task, check for errors
- Unwrap the data returned in the closure.
- Create a variable of type `[EntryRepresentation]`. Set its initial value to an empty array.
- Decode the data into `[String: EntryRepresentation].self`. Set the value of the array you just made in the previous step to the entry representations in this decoded dictionary. **HINT:** loop through the dictionary to return an array of just the entry representations without the identifier keys.
- Loop through the array of entry representations. Inside the loop, create a constant called `entry`. For its value, give it the result of the `fetchSingleEntryFromPersistentStore` method. Pass in the entry representation's identifier. This will allow us to compare the entry representation and see if there is a corresponding entry in the persistent store already.
- Check to see if the entry returned from the persistent store exists. If it does, check whether the entry and the entry representation have the same values. If they do, then you don't need to do anything because the entry on the server and the entry in the persistent store are synchronized.
- If the entry exists, but the entry and the entry representation's values are not the same, then call the new `update(entry: ...)` method that takes in an entry and an entry representation. This will then synchronize the entry from the persistent store to the updated values from the server's version of the entry.
- If there was no entry returned from the persistent store, that means the server has an entry that the device does not. In that case, initialize a new `Entry` using the convenience initializer that takes in an `EntryRepresentation`.
- Outside of the loop, call `saveToPersistentStore()` to persist the changes and effectively synchronize the data in the device's persistent store with the data on the server. Since you are using an `NSFetchedResultsController`, as soon as you save the managed object context, the fetched results controller will observe those changes and automatically update the table view with the updated entries.
- Call completion and pass in `nil` for the error.
- Don't forget to resume the data task.

4. Write an initializer for the `EntryController`. It shouldn't take in any values. Inside of the initializer, call the `fetchEntriesFromServer` method. As soon as the app runs and initializes this model controller, it should fetch the entries from Firebase and update the persistent store.

The app should be working at this point. Test it by going to the Firebase Database in a browser and changing some values in the entries saved there. The easiest thing to change is the mood. This will allow you to easily see if the table view will update according to the new changes. It may take a few seconds after the app launches, but you should see the cell(s) move to different sections if you changed the mood of some entries in Firebase.

## Go Further

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.