Lab 4 – Using Core Data

Overview:

The purpose of the lab for this week is twofold. First, this lab demonstrates how to implement Core Data within our projects. Secondly the lab shows a way to separate the Database and View Controller layers (we will replace Core Data with Firebase for the database in Lab 06).

This lab is a continuation of last week. The solution for Lab 3 can be found on Moodle, or you can use your own.

Note: Some of the Macs in the lab may be running Windows and need to be rebooted into the OS X environment. Restart the Mac and hold down the Option key while it is booting. You will need to select 'EFI Boot' and select OS X to launch the correct operating system.

The Task:

For this lab you will add a Core Data persistent database layer to the app from last week's lab. This will require no changes to the UI, only changes in the code. The summary of changes to be made are as follows:

- Create a Core Data Model
- Create a Database Listener
- Create a Database Protocol
- Create a Core Data Controller Class
- Change the Add Hero Controller to use these
- Change the All Heroes Controller to use these
- Change the Current Party Controller to use these

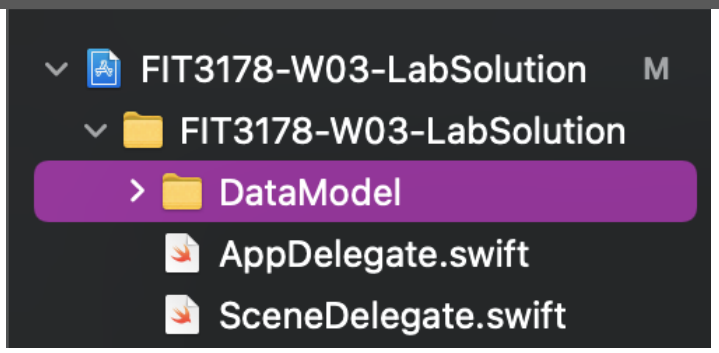
The steps are broken down to enable testing at multiple points. This represents our suggestion for how to implement such features—break them into small manageable chunks that can be written and tested independently.

Creating the Data Model:

To begin this week, open your solution to Week 3 (or download our starting point project from Moodle).

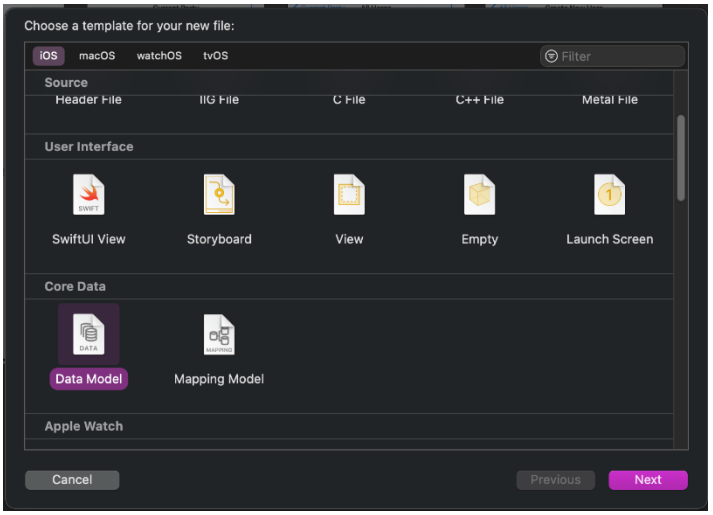
Begin by right-clicking on the project folder in the Navigation Explorer and selecting “New Group”.

Name this folder DataModel



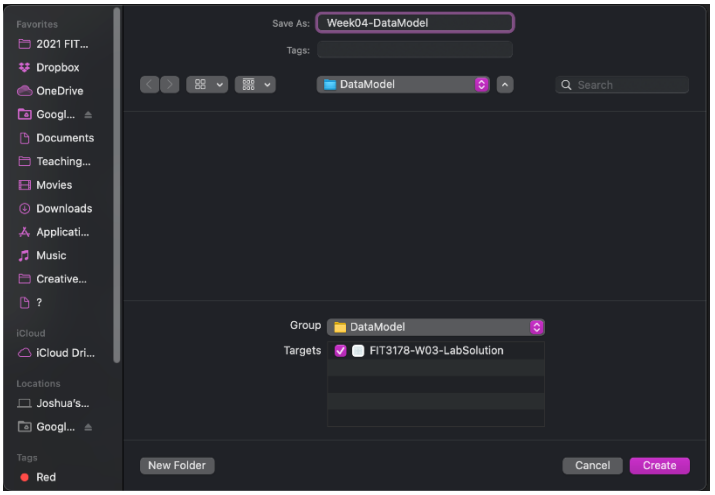
Next, right-click on this folder in the Navigator and select “New File...”

Scroll down until you see Core Data > Data Model. Select this and click Next.



Name this "Week04-DataModel”. Make sure the group is set as the "DataModel folder we created just before.

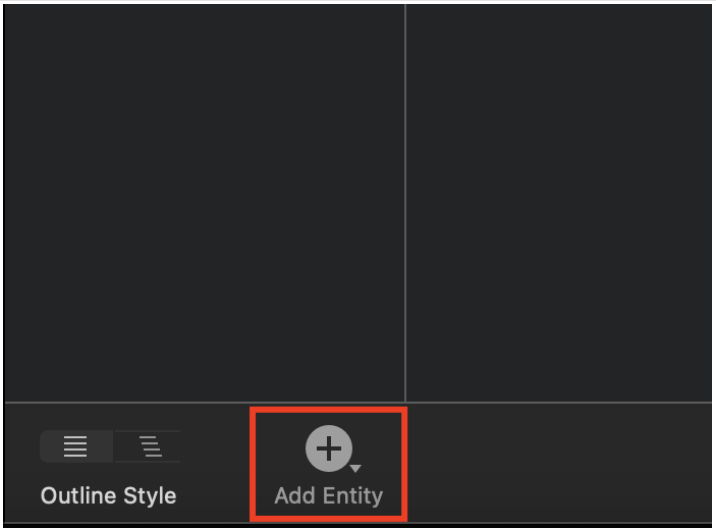
Click Create and this will create our Core Data Model



After it has been created open the data model if it has not already opened automatically.

Currently our model is empty. Let's create some entities that we will need.

Click the Add Entity button to create a new entity

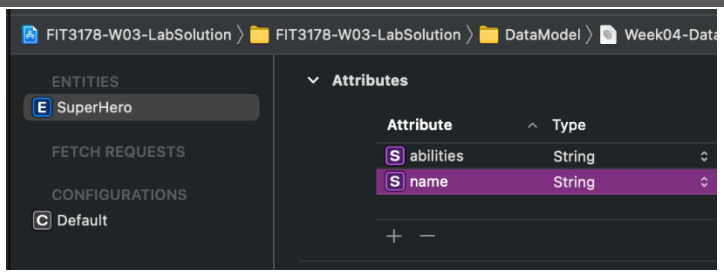


Name the new entity "SuperHero".

Under attributes click the "+" button to add some attributes.

Create a "name" Attribute of type String.

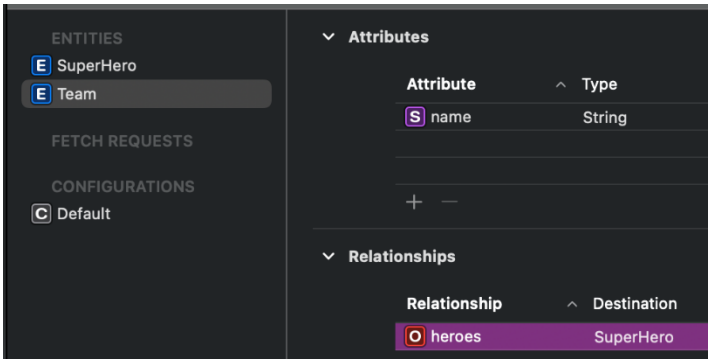
Create an "abilities" Attribute of type String.



Create a second entity “Team”.

Create a new attribute called "name" of type String

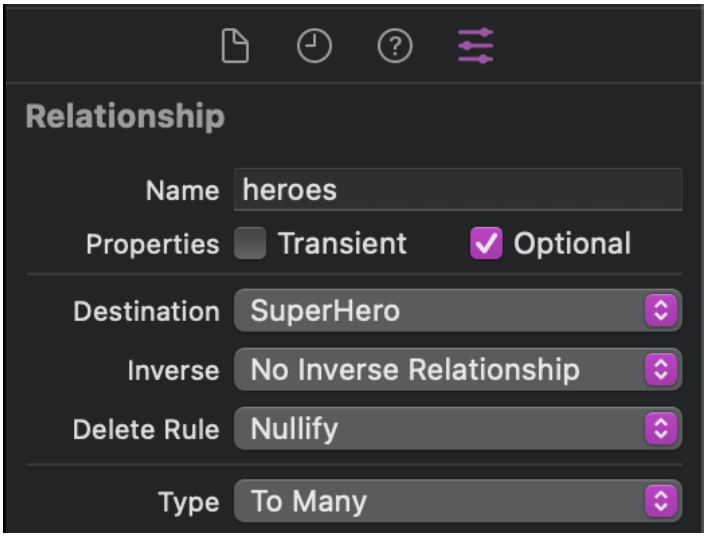
Create a new Relationship called "heroes". Set its Destination to be SuperHero



Select the "heroes" Relationship and go to the Data Model inspector.

Change the Type to be "To Many".

A team can hold many heroes after all!

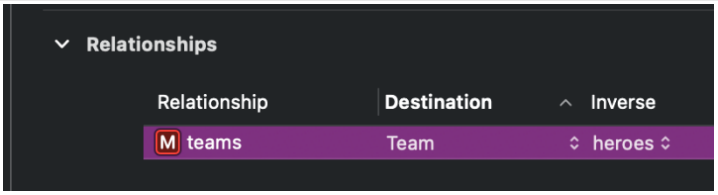


Select the SuperHero entity again.

Create a new relationship called "teams". Set its destination to "Team" and its inverse to "heroes".

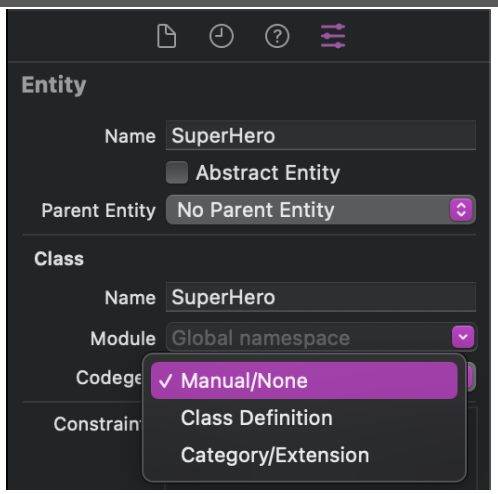
Change the Type to be "To Many".

These two relationships are now “linked”. Changes made on one affect the other



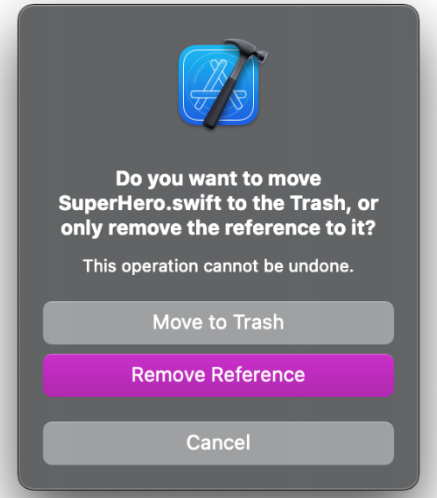
For both entities, go to the "Data Model Inspector" on the right and change Codegen to "Manual/None".

By default, the Core Data Model will try and automatically create our classes. This is not what we want.



Our Data Model is almost finished. Before we get to the final step delete our old "SuperHero.swift" class file (select move to Trash).

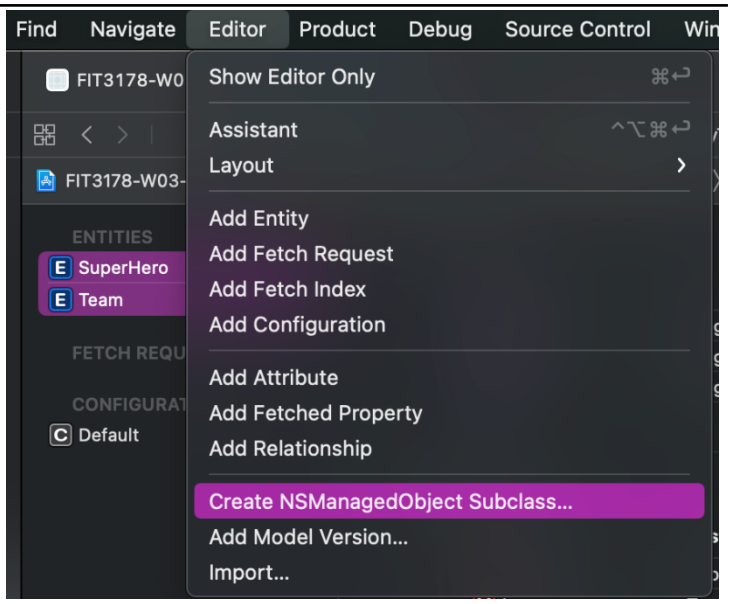
We will be generating a managed object SuperHero class as a replacement.



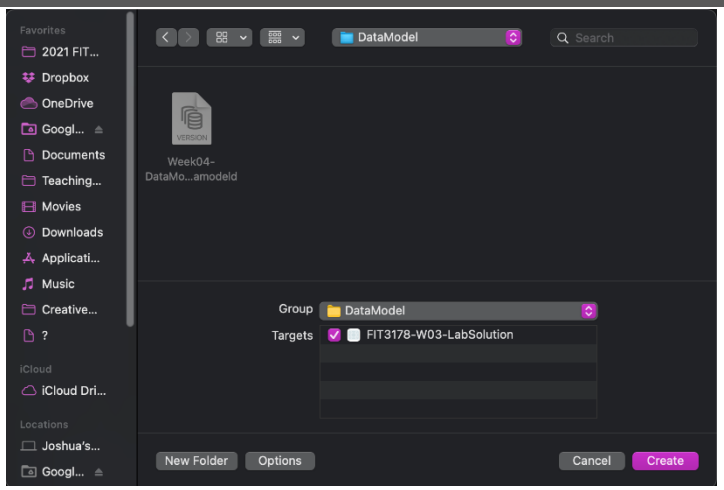
Open the Data Model and select both "SuperHero" and "Team" entities.

From the menu, select *Editor -> Create NSManagedObject Subclass*.

In the dialog that opens, ensure that both SuperHero and Team are selected and click Next.



Ensure the group is "DataModel" and click finish. This will create our class files for both SuperHero and Team.



With this our Data Model is complete and ready to be used.

Adding the MulticastDelegate class:

Create a new Group in our project called "Database".

Storing weak links to a list of delegates (used for the listeners below) is not trivial. We wrote the MulticastDelegate class for this purpose. This file, MulticastDelegate.swift, is available from Moodle.

This class has methods to add and remove a particular listener, and a method that invokes a closure (some code passed to the method) on all the listeners. It stores weak references to avoid reference cycles. You do not need to understand this class.

The file is licenced under the Apache free software licence. To quote the linked WIkipedia page, this means “It allows users to use the software for any purpose, to distribute it, to modify it, and to distribute modified versions of the software under the terms of the license, without concern for royalties.” You can include this file in your lab project, your second assignment, and for anything else you want.

Download the MulticastDelegate.swift and drag it into the Database folder. Xcode will show a dialog with some options for adding the file to the project. Make sure the “Copy items if needed” option is checked, and click FInish. The file will be added to the project.

Creating the Database Protocol:

For this step we will be creating the Database Protocol and a few associated classes and enumerators. These will be used to control what functionality a database will have, define the behaviour of its listeners, and define the types of listeners that a database can have.

At the core will be the DatabaseProtocol. This protocol defines all the core behaviour of a database. It is crucial to note that this will be flexible enough to work for both an offline database such as Core Data AND online databases such as Firebase! The example shown in this lab is rudimentary. It should provide a good starting point for more complex applications.

Create a Swift file, name it "DatabaseProtocol" and save it in the Database folder.

Open the file and we can begin making the required enumerators and classes. Starting with the **DatabaseChange** enumerator

```
enum DatabaseChange {
    case add
    case remove
    case update
}
```

The **DatabaseChange** enumerator is used to define what type of change has been done to the database. There are several possible cases here that are very useful, these being add, remove, and update. For this week we will only be using update. Other cases can also be added for more complex implementations and functionality.

Underneath the **DatabaseChange** enum we need to create a second one called **ListenerType**

```
enum ListenerType {
    case team
    case heroes
    case all
}
```

The database we are building has multiple different sets of data that each require their own specific behaviour to handle. It can prove useful to specify the type of data each of our listeners will be dealing with. In the case of this app, we can have listeners that listen for team, hero or both. These will be used when the database has any changes (the changes from our previous enum!)

Now that we have the listener types defined, we need to define the listener itself.

```
protocol DatabaseListener: AnyObject {
    var listenerType: ListenerType {get set}
    func onTeamChange(change: DatabaseChange, teamHeroes: [SuperHero])
    func onAllHeroesChange(change: DatabaseChange, heroes: [SuperHero])
}
```

This protocol defines the delegate we will be using for receiving messages from the database. It has three things that any implementation must take care of.

- The implementation must always specify the listener's type
- An onTeamChange method for when a change to heroes in a team has occurred.
- An onAllHeroesChange method for when a change to any of the heroes has occurred.

Each of the onChange methods also returns a change type. Whilst not utilised for this week it enables us to slightly change the behaviour based on what kind of change has occurred to the database.

Another important note here is that the DatabaseListener is kept database agnostic. There are no specific calls or mentions of Core Data or any other technology here. This enables us to easily reuse this code if we do another implementation of a database (such as in week 6).

With these done, the last thing to define is the DatabaseProtocol itself. This protocol defines all the behaviour that a database must have. And these will be the public facing methods that can be accessed by other parts of the application. As before, the goal here is abstraction and re-usability as much as possible. Each specific database controller implementation may have additional functionality to support these methods, but these are the required ones.

```
protocol DatabaseProtocol: AnyObject {
    func cleanup()

    func addListener(listener: DatabaseListener)
    func removeListener(listener: DatabaseListener)

    func addSuperHero(name: String, abilities: String) -> SuperHero
    func deleteSuperHero(hero: SuperHero)
}
```

To make it easier to debug and build up the app progressively, we will start with only a couple methods required for adding/removing listeners and heroes.

Creating the Core Data Controller:

Create a new Cocoa Touch Class file, name it "CoreDataController", ensure it inherits from NSObject and DatabaseProtocol and save it within the Database folder.

Before coding any functionality for the class we need to import the CoreData framework. Previously we have been using components from the Foundation and UIKit frameworks, which are automatically added to the top of our class files. When using Core Data we need to add our own input statement.

```
import CoreData
```

Note: This should be above the class inside of the swift file. Not inside it

As with previous weeks the first step is to include the class properties.

```
var listeners = MulticastDelegate<DatabaseListener>()
var persistentContainer: NSPersistentContainer
```

The “listeners” property holds all listeners added to the database inside of the MulticastDelegate class that was added above. This creates a nice wrapper that we can safely hold multiple listeners in without having to worry about memory issues.

The persistentContainer property holds a reference to our persistent container and within it, our managed object context. Any time we need to create, delete, retrieve, or save our database we need to do so via the managed object context. This makes the Core Data Controller the ideal place to keep a reference to the persistent controller.

As the persistentContainer property is not optional and has not been assigned a value, we must create an initialiser to handle this. Unlike previous weeks where we create an initialiser with parameters, here we will override the default initialiser.

```
override init() {
    super.init()
}
```

Inside the initialiser add the following code to instantiate the Core Data stack before the call to super.init (all variables must have values before this call).

```
persistentContainer = NSPersistentContainer(name: "Week04-DataModel")
persistentContainer.loadPersistentStores() { (description, error ) in
    if let error = error {
        fatalError("Failed to load Core Data Stack with error: \(error)")
    }
}
```

The first line of code initializes the Persistent Container property using the data model named "Week04-Datamodel".

The second line loads the Core Data stack, and we provide a closure for error handling. In this case we are triggering a fatal error if the stack fails to load. Generally if this occurs it is because the name in the line above does not match what the xcdatamodel object is named in XCode.

cleanup method
This method will check to see if there are changes to be saved inside of the view context and then save, as necessary.

```
func cleanup() {
    if persistentContainer.viewContext.hasChanges {
        do {
            try persistentContainer.viewContext.save()
        } catch {
            fatalError("Failed to save changes to Core Data with error: \(error)")
        }
    }
}
```

Changes made to the managed object context must be explicitly saved by calling the save method on the managed object context. This method can throw an error, so must be done within a do-catch statement.

addSuperHero method
The addSuperHero method is responsible for adding new super heroes to Core Data. It takes in a name and abilities, generates a new SuperHero object then returns it. The SuperHero is a Core Data managed object stored within a specific managed object context.

```
func addSuperHero(name: String, abilities: String) -> SuperHero {
    let hero = NSEntityDescription.insertNewObject(forEntityName:
        "SuperHero", into: persistentContainer.viewContext) as! SuperHero
    hero.name = name
    hero.abilities = abilities

    return hero
}
```

Once a managed object has been created, all changes made to it are tracked. Note that any new object will not be saved to persistent memory until the save method has been called on its associated managed object context.

deleteSuperHero method
The deleteSuperHero method is a straightforward one. It takes in a SuperHero to be deleted and removes it from the main managed object context. As with other changes, the deletion will not be made permanent until the managed context is saved.

```
func deleteSuperHero(hero: SuperHero) {
    persistentContainer.viewContext.delete(hero)
}
```

fetchAllHeroes methods
The fetchAllHeroes method is used to query Core Data to retrieve all hero entities stored within persistent memory. It requires no input parameters and will return an array of SuperHero objects.

To query Core Data an NSFetchRequest is created. This is mostly handled for us within the pre-generated entity classes that were created for SuperHero and Team. Once a fetch request is created it must be passed to the managed object context to execute

```
func fetchAllHeroes() -> [SuperHero] {
    var heroes = [SuperHero]()

    let request: NSFetchRequest<SuperHero> = SuperHero.fetchRequest()

    do {
        try heroes = persistentContainer.viewContext.fetch(request)
    } catch {
        print("Fetch Request failed with error: \(error)")
    }

    return heroes
}
```

A fetch request can throw an error so it must be done within a do-catch statement. If it succeeds, we use it to populate our hero array and return it.

addListener method
The addListener method does two things. Firstly it adds the new database listener to the list of listeners. And secondly, it will provide the listener with initial immediate results depending on what type of listener it is.

```
func addListener(listener: DatabaseListener) {
    listeners.addDelegate(listener)

    if listener.listenerType == .heroes || listener.listenerType == .all {
        listener.onAllHeroesChange(change: .update, heroes:
            fetchAllHeroes())
    }
}
```

Checking what information to provide and then providing it requires us to check the listener type. In this case if the type is either heroes or all then the method will call the delegate method onAllHeroesChange and pass through all the heroes fetched from the database.

removeListener method
The removeListener method just passes the specified listener to the multicast delegate class which then removes it from the set of saved listeners.

```
func removeListener(listener: DatabaseListener) {
    listeners.removeDelegate(listener)
}
```

createDefaultHeroes method
The createDefaultHeroes method is one we have created for testing purposes. This method simply creates several superheroes that can be used for testing the application.

```
func createDefaultHeroes() {
    let _ = addSuperHero(name: "Bruce Wayne", abilities: "Money")
    let _ = addSuperHero(name: "Superman", abilities: "Super Powered
Alien")
    let _ = addSuperHero(name: "Wonder Woman", abilities: "Goddess")
    let _ = addSuperHero(name: "The Flash", abilities: "Speed")
    let _ = addSuperHero(name: "Green Lantern", abilities: "Power Ring")
    let _ = addSuperHero(name: "Cyborg", abilities: "Robot Beep Beep")
    let _ = addSuperHero(name: "Aquaman", abilities: "Atlantian")
    cleanup()
}
```

The "let _" may look a little strange but it is needed to stop a compiler warning for not using the value returned by calls to the addSuperHero method. The underscore indicates that we don't care about the returned value and don't use it again.

Finally, add the following code at the end of the init method.

```
if fetchAllHeroes().count == 0 {
    createDefaultHeroes()
}
```

Here we attempt to fetch all the heroes from the database. If this returns an empty array, we call the createDefaultHeroes method and populate the database with default values. This will only be done the first time the application runs.

With that, the first version of the CoreDataController is complete. The next step is to integrate it into the application by making a few changes across a number of classes.

Making Changes to AppDelegate and SceneDelegate:

The App Delegate class has been present in all previous weeks, but we haven't explained it. It is created when the application starts, continues to exist in the background and only disappears when the application is closed or killed. As such, it is the perfect place to store the reference to the core data controller.

Make the following changes (highlighted in yellow) to AppDelegate.swift:

```
var databaseController: DatabaseProtocol?

func application(_ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions:
        [UIApplication.LaunchOptionsKey: Any]?) -> Bool {

    databaseController = CoreDataController()
    return true
}
```

Our database will always be available while the app is running. (We have defined the databaseController property as conforming to DatabaseProtocol, rather than being a specific class. We will replace it with a different class in Lab 6.

Make the following changes to the SceneDelegate.swift file

```
func sceneWillResignActive(_ scene: UIScene) {
    let appDelegate = UIApplication.shared.delegate as? AppDelegate
    appDelegate?.databaseController?.cleanup()
}
```

When the scene moves to an inactive state it will attempt to call the cleanup method of the database controller from the AppDelegate. Note that this is also database agnostic and has no specific calls to a particular type of database Implementation. We won't need to change this when moving to Firebase in Week 6.

From here we can make a couple changes to the Create Hero View Controller to support saving to persistent storage.

Updating the CreateHeroViewController:

When you open the CreateHeroViewController you will notice an error has appeared. This is because we can no longer create SuperHero class instances the way we would a typical class. To fix this we will need to make a couple of changes.

First, add a weak databaseDontroller property as was done for the AppDelegate. Delete the superHeroDelegate property, it is no longer needed.

```
weak var databaseController: DatabaseProtocol?
```

Next, inside of the viewDidLoad method, add the following code to set the databaseController value. This is done by getting access to the AppDelegate and then storing a reference to the databaseController from there.

```
let appDelegate = UIApplication.shared.delegate as? AppDelegate
databaseController = appDelegate?.databaseController
```

Last, inside of the createHero method replace the code for creation of the Super Hero and callback to the delegate (orange code below), with a call to the databaseController's addSuperHero method, passing in the name and abilities.

```
let hero = SuperHero(name: name, abilities: abilities)
let _ = superHeroDelegate?.addSuperHero(newHero: hero)
let _ = databaseController?.addSuperHero(name: name, abilities: abilities)
```

These are all the changes required to CreateHeroViewController. However, before we can test the application we need to fix a couple errors in the AllHeroesTableViewController.

Updating the AllHeroesTableViewController:

As with the CreateHeroViewController, there are a couple of errors here due to the creation of temporary default heroes from last week. In addition, there are also a few changes required to ensure the database is used to populate the TableView.

To begin, remove the orange text. The class will no longer adopts this protocol.

```
class AllHeroesTableViewController: UITableViewController,
UISearchResultsUpdating, AddSuperHeroDelegate {
```

Since it doesn't adopt the protocol, you can delete the **addSuperHero** method.

Additionally, remove the **createDefaultHeroes** method. This was needed last week when there was no persistent data. As this application will be leveraging Core Data, the method is no longer required. Also remove the method call from **viewDidLoad.**

Lastly, remove the code from the **prepareSegue** method. This is no longer needed with the database.

With these removals done it is time to start adding the database methods and requirements. To begin, ensure that the class adopts the DatabaseListener protocol.

```
class AllHeroesTableViewController: UITableViewController,
UISearchResultsUpdating, DatabaseListener {
```

Create two new properties. One to hold a reference to the database and one to specify the listener type that this class will be.

```
var listenerType = ListenerType.heroes
weak var databaseController: DatabaseProtocol?
```

Inside of **viewDidLoad**, after the call to superviewDidLoad, add code to set the databaseController. You should use the same code from the previous step (CreateHeroViewController).

Next we need to override two existing methods of the view controller and use them to add and remove ourselves from the database listeners.

The first method is **viewWillAppear**. This method is called before the view appears on screen. In this method we need to add ourselves to the database listeners.

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    databaseController?.addListener(listener: self)
}
```

The second method is **viewWillDisappear**. Again, you can guess when it is called.

```
override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)
    databaseController?.removeListener(listener: self)
}
```

With these two methods the View Controller will automatically register itself to receive updates from the database when the view is about to appear on screen and deregister itself when it's about to disappear.

The next thing to do is code in the two required methods from the protocol: **onAllHeroesChange** and **onTeamChange**.

When **onAllHeroesChange** is called we need to update our full hero list then update our filtered list based on the search results. Fortunately this is nice and easy with two lines of code.

```
func onAllHeroesChange(change: DatabaseChange, heroes: [SuperHero]) {
    allHeroes = heroes
    updateSearchResults(for: navigationItem.searchController!)
}
```

The onTeamChange method is even easier. We need to implement the method, but it will do nothing. This class does not care about team updates.

```
func onTeamChange(change: DatabaseChange, teamHeroes: [SuperHero]) {
    // Do nothing
}
```

The final change is within the **editingStyleForRowAt** method. Previously we had a batch update that searched the allHeroes list for the selected hero, deleted it from the list and the filtered list and then from the table view. Several sequential steps. We can remove those for now and replace them with the following.

```
override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {
    if editingStyle == .delete && indexPath.section == SECTION_HERO {
        let hero = filteredHeroes[indexPath.row]
        databaseController?.deleteSuperHero(hero: hero)
    }
}
```

Testing the Application:

We can now test the application. Adding new heroes should now correctly save to the database and show on the AllHeroesTableViewController! Notes, changes to the Current Party are not saved into Core Data.

Warning: For the database changes to save the application must be closed in the simulator. Hitting stop within XCode halts the application immediately and never calls the SceneDelegates callbacks!

You may notice some very strange behaviour when trying to delete heroes from the AllHeroesTableView controller. They do not actually delete and can crash the application! This is because we have not yet updated the UI to reflect that changes were made to the database. Currently only when the view appears will there be an update from the database controller.

Updating the CoreDataController:

We will use a FetchedResultsController to monitor changes and tell all listeners when they occur.

First, ensure that the class implements the NSFetchedResultsControllerDelegate method.

```
class CoreDataController: NSObject, DatabaseProtocol,
NSFetchedResultsControllerDelegate {
```

Next, create a new property to hold a FetchedResultsController.

```
var allHeroesFetchedResultsController: NSFetchedResultsController<SuperHero>?
```

This controller will watch for changes to all heroes within the database. When a change occurs, the core data controller will be notified and can let its listeners know.

We can do this by replacing the implementation of the fetchAllHeroes method with the code below:

```
func fetchAllHeroes() -> [SuperHero] {
    if allHeroesFetchedResultsController == nil {
        // Do something
    }

    if let heroes = allHeroesFetchedResultsController?.fetchedObjects {
        return heroes
    }
    return [SuperHero]()
}
```

We first check if the fetched results controller is nil (i.e., not instantiated). Currently we do not have code here to initialise it, but this will be done below.

Assuming it is instantiated, we check if it contains fetched objects. If it does, we return the array.

To instantiate allHeroesFetchedResultsController, we need to create a fetch request. We must also specify a sort descriptor (required for a fetched results controller), ensuring the results have an order.

```
let request: NSFetchRequest<SuperHero> = SuperHero.fetchRequest()
let nameSortDescriptor = NSSortDescriptor(key: "name", ascending: true)
request.sortDescriptors = [nameSortDescriptor]
```

Next we initialise the fetched results controller. This requires a number of arguments. We can ignore the last two but we do need to provide the fetch request and the managed object context we want to perform the fetch on. Then the database controller is set to be its delegate.

```
// Initialise Fetched Results Controller
allHeroesFetchedResultsController =
    NSFetchedResultsController<SuperHero>(fetchRequest: request,
    managedObjectContext: persistentContainer.viewContext,
    sectionNameKeyPath: nil, cacheName: nil)

// Set this class to be the results delegate
allHeroesFetchedResultsController?.delegate = self
```

The last step is to perform the fetch request (which will begin the listening process).

```
do {
    try allHeroesFetchedResultsController?.performFetch()
} catch {
    print("Fetch Request Failed: \(error)")
}
```

This concludes the changes to fetchAllHeroes.

There is one last step to complete before testing the application again. As part of the NSFetchedResultsControllerDelegate we must implement another method. This being the **controllerDidChangeContent** method. This will be called whenever the FetchedResultsController detects a change to the result of its fetch.

```
// MARK: - Fetched Results Controller Protocol methods

func controllerDidChangeContent(_ controller:
    NSFetchedResultsController<NSFetchRequestResult>) {

    if controller == allHeroesFetchedResultsController {
        listeners.invoke() { listener in
            if listener.listenerType == .heroes
                || listener.listenerType == .all {

                listener.onAllHeroesChange(change: .update,
                    heroes: fetchAllHeroes())
            }
        }
    }
}
```

We first check to see if the controller is our allHeroesFetchedResultsController. (Once teams are implemented there will be two separate FetchedResultsControllers that trigger calls to this method when changes are detected.)

If it is the correct method, we call the MulticastDelegate's invoke method and provide it with a closure that will be called for each listener. For each listener, it checks if it is listening for changes to heroes. If it is, it calls the onAllHeroesChange method, passing it the updated list of heroes.

With this done we can test the application again. Deleting heroes from the All Heroes screen should now function correctly!

The last step is to add the necessary code to enable saving the Current Party to persistent storage.

Teams — Updating the Database Protocol:

Currently the Database Protocol only specifies methods for adding, deleting and saving Heroes. Before any changes are made to the Core Data controller, the protocol must first be updated to add the following property and methods for Teams.

```
protocol DatabaseProtocol: AnyObject {
    var defaultTeam: Team {get}

    func addTeam(teamName: String) -> Team
    func deleteTeam(team: Team)
    func addHeroToTeam(hero: SuperHero, team: Team) -> Bool
    func removeHeroFromTeam(hero: SuperHero, team: Team)
}
```

The property is to store the default team. For this implementation we assume there is only the single team of Super Heroes. This could be changed to support multiple teams, but for now we need the default team property.

The four methods give us the functionality for adding and deleting teams, as well as the ability to add/remove heroes from them.

Teams — Updating the CoreDataController:

At this point the CoreDataController will show an error message again. Due to the changes made in the DatabaseProtocol a number of methods need to be implemented. Before that however, a few new properties are needed.

```
let DEFAULT_TEAM_NAME = "Default Team"
var teamHeroesFetchedResultsController: NSFetchedResultsController<SuperHero>?
```

Next, the default team property must be implemented. As a lazy property, it is not initialized when the rest of the class is initialized. Instead it is initialized the first time that its value is requested.

```
// MARK: - Lazy Initilisation of Default Team
lazy var defaultTeam: Team = {
    var teams = [Team]()

    let request: NSFetchRequest<Team> = Team.fetchRequest()
    let predicate = NSPredicate(format: "name = %@", DEFAULT_TEAM_NAME)
    request.predicate = predicate

    do {
        try teams = persistentContainer.viewContext.fetch(request)
    } catch {
        print("Fetch Request Failed: \(error)")
    }

    if let firstTeam = teams.first {
        return firstTeam
    }
    return addTeam(teamName: DEFAULT_TEAM_NAME)
}()
```

A fetch request is used here to find all instances of teams with the name "Default Team". If none are found, we create one. This will be done on the first run of the application. After this point there should always be a Default Team.

Next up are the required methods from DatabaseProtocol.

addTeam method
This method will add a new team to the database given a team name and then return it. It is very similar to how the addSuperHero method works.

```
func addTeam(teamName: String) -> Team {
    let team = NSEntityDescription.insertNewObject(forEntityName:
        "Team", into: persistentContainer.viewContext) as! Team
    team.name = teamName

    return team
}
```

deleteTeam method
This method deletes a given team from the managed object context. Again almost identical to how the deleteSuperHero method works.

```
func deleteTeam(team: Team) {
    persistentContainer.viewContext.delete(team)
}
```

addHeroToTeam method
This method attempts to add a hero to a given team and will return a boolean to indicate whether or not it was successful. It can fail if the team already has 6 or more heroes or if the team already contains the hero.

```
func addHeroToTeam(hero: SuperHero, team: Team) -> Bool {
    guard let heroes = team.heroes, heroes.contains(hero) == false,
        heroes.count < 6 else {

        return false
    }

    team.addToHeroes(hero)
    return true
}
```

removeHeroFromTeam
This method removes a hero from the team.

```
func removeHeroFromTeam(hero: SuperHero, team: Team) {
    team.removeFromHeroes(hero)
}
```

One last method is required to support the functionality of fetching teams from the database. This is not part of the DatabaseProtocol but is used internally by the CoreDataController to define how to get the team results.

Create a new method called fetchTeamHeroes that takes no parameters and returns an array of super heroes.

```
func fetchTeamHeroes() -> [SuperHero] {

}
```

This method will be fairly similar to the fetchAllHeroes method with a few main differences. Firstly it returns an array of super heroes which are part of a specified team. For this application we have hard coded it to be the default team. All of this will be done through a fetched results controller as well

```
if teamHeroesFetchedResultsController == nil {
    let fetchRequest: NSFetchRequest<SuperHero> = SuperHero.fetchRequest()
    let nameSortDescriptor = NSSortDescriptor(key: "name", ascending: true)
    let predicate = NSPredicate(format: "ANY teams.name == %@",
DEFAULT_TEAM_NAME)
    fetchRequest.sortDescriptors = [nameSortDescriptor]
    fetchRequest.predicate = predicate

    teamHeroesFetchedResultsController =
        NSFetchedResultsController<SuperHero>(fetchRequest: fetchRequest,
        managedObjectContext: persistentContainer.viewContext,
        sectionNameKeyPath: nil, cacheName: nil)

    teamHeroesFetchedResultsController?.delegate = self

    do {
        try teamHeroesFetchedResultsController?.performFetch()
    } catch {
        print("Fetch Request Failed: \(error)")
    }
}

var heroes = [SuperHero]()
if teamHeroesFetchedResultsController?.fetchedObjects != nil {
    heroes = (teamHeroesFetchedResultsController?.fetchedObjects)!
}

return heroes
```

Now the team fetched results controller is enabled. Two final changes to CoreDataController are required to integrate the functionality into the application.

First, add the following to the **addListener** method.

```
if listener.listenerType == .team || listener.listenerType == .all {
    listener.onTeamChange(change: .update, teamHeroes: fetchTeamHeroes())
}
```

This ensures the listeners get a team of heroes when added to the Multicast Delegate.

Lastly add the following to the controllerDidChangeContent method

```
else if controller == teamHeroesFetchedResultsController {
    listeners.invoke { (listener) in
        if listener.listenerType == .team || listener.listenerType == .all {
            listener.onTeamChange(change: .update,
                teamHeroes: fetchTeamHeroes())
        }
    }
}
```

With this, the fetchedResultsController for Team objects is fully functional.

Teams — Changes to CurrentPartyTableViewController:

Open the Current Party Table View Controller. A number of changes will need to be made here in order to connect with the database. To begin, ensure that the class implements the DatabaseListener protocol.

```
class CurrentPartyTableViewController: UITableViewController,
AddSuperHeroDelegate, DatabaseListener {
```

As with the AllHeroesTableViewController, the class doesn't need delegation to be informed of changes, so you can remove adoption of the AddSuperHeroDelegate. You can also delete the **addSuperHero** method, and remove the code in the **prepareForSegue** method.

Add the required listenerType property and set it to ".team".
Create a weak property for the databaseController

```
var listenerType: ListenerType = .team
weak var databaseController: DatabaseProtocol?
```

Inside of **viewDidLoad** add the code to set the databaseController value (as done for the AllHeroesViewController).

Set up the **viewWillAppear** and **viewWillDisappear** methods like in the AllHeroesTableViewController to add and remove this class from the database listeners.

Implement the two required methods of DatabaseListener, **onAllHeroesChange** and **onTeamChange**. This time onAllHeroesChange should be empty as we are not listening for these events. Inside of the onTeamChange method set the current party to be the updated party and reload the table view.

```
func onTeamChange(change: DatabaseChange, teamHeroes: [SuperHero]) {
    currentParty = teamHeroes
    tableView.reloadData()
}
```

As before, remove the current deletion code from the editingStyleForRowAt method and replace it with the following.

```
if editingStyle == .delete && indexPath.section == SECTION_HERO {
    self.databaseController?.removeHeroFromTeam(hero:
        currentParty[indexPath.row], team: databaseController!.defaultTeam)
}
```

Finally remove the code from the addSuperHero method and replace it with the following

```
func addSuperHero(_ newHero: SuperHero) -> Bool {
    return databaseController?.addHeroToTeam(hero: newHero,
        team: databaseController!.defaultTeam) ?? false
}
```

Teams — Changes to AllHeroesTableViewController:

Since we are not using delegation to add heroes to the current party, we can update the **didSelectRowAtIndexPath** method and replace the code with the following:

```
let hero = filteredHeroes[indexPath.row]
let heroAdded = databaseController?.addHeroToTeam(hero: hero, team:
        databaseController!.defaultTeam) ?? false
if heroAdded {
    navigationController?.popViewController(animated: false)
    return
}
displayMessage(title: "Party Full", message: "Unable to add more members to party")
tableView.deselectRow(at: indexPath, animated: true)
```

The **superHeroDelegate** property is no longer needed and can be deleted.

With this we have finished the changes to existing code in the application.

Due to the decoupling of the database and view controllers, the changes to existing code were minimal. With a change to a different database the only code that would need to be tweaked are the AppDelegate class, the Database Controller, plus the onTeamChange and onAllHeroesChange methods! We will examine this in Lab 6 with Firebase.

Teams — Testing the Application:

Try adding a couple heroes to the party and restart the application. You will notice that the data is now persistent!