Core Data

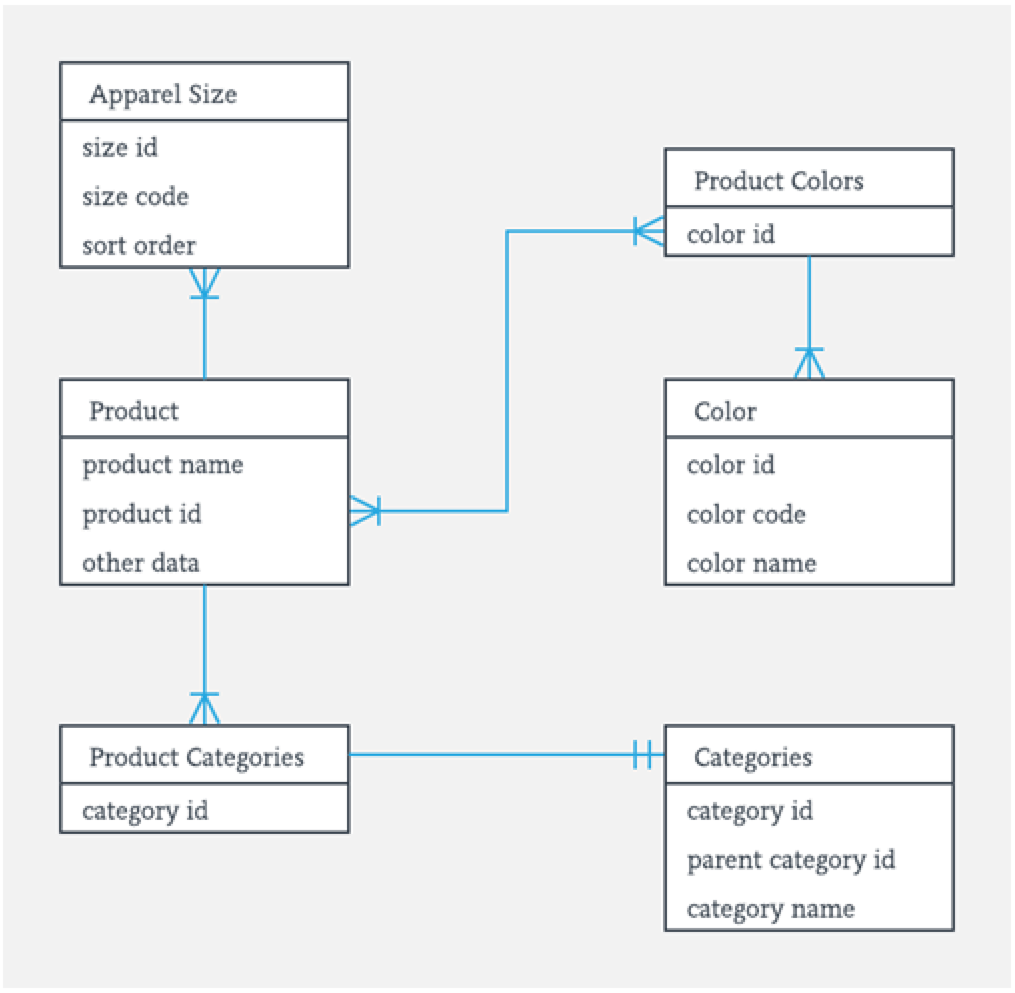
Jump to a Section

What is Core Data?

- There are situations when you may want to persist data to local device storage
- Local data persistence is provided by the **CoreData framework**
- The CoreData Framework persists data as an SQLite relational database
- The CoreData framework provides a library of functions to work with an SQLite database.

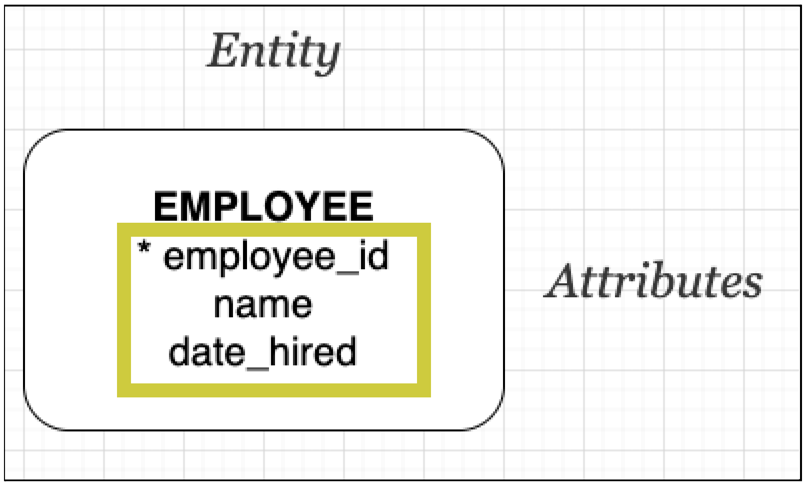
Review of Relational Databases

Databases can be modelled as a set of related entities:



Source: *https://www.guru99.com/er-diagram-tutorial-dbms.html*

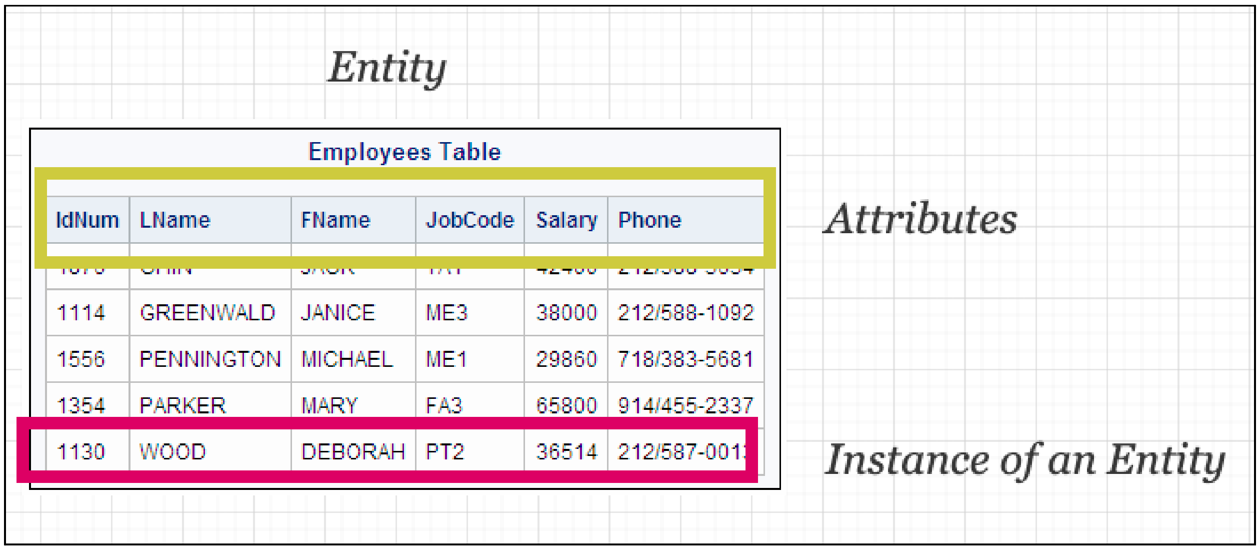
Each entity contains attributes that describe the object's properties:



In a relational database:

- The entity represents your table
- The attributes represent the table columns
- An attribute marked with a * indicates the primary key column

An **instance of an entity** is a single occurrence of the entity:

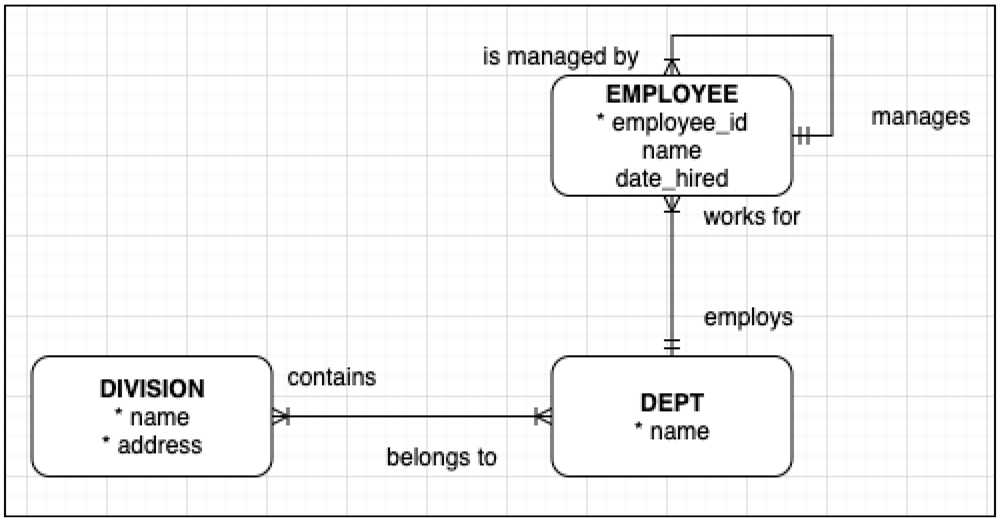

Entity
Attributes
Instance of an Entity

Employees Table

IdNum	LName	FName	JobCode	Salary	Phone
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
1114	GREENWALD	JANICE	ME3	38000	212/588-1092
1556	PENNINGTON	MICHAEL	ME1	29860	718/383-5681
1354	PARKER	MARY	FA3	65800	914/455-2337
1130	WOOD	DEBORAH	PT2	36514	212/587-001

Entities can have relationships with each other.

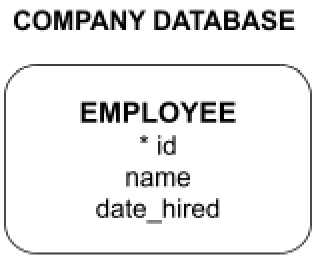
- These relationships are often modelled with an ER diagram



In IOS, you must use the Core Data framework to:

- Define your entities
- Specify the attributes of the entities
- Describe relationships between the entities
- Create occurrences of the entity (insert data)

Example - Employee Table



Implementing the database and tables in SQL:

```
CREATE DATABASE company_db;

-- Create the table
CREATE TABLE Employees (
      id int NOT NULL PRIMARY KEY,
      name varchar(255) NOT NULL,
      date_hired varchar(255),
);


-- Insert values into the table
INSERT INTO Employees (id, name, date_hired)
VALUES (1, 'Peter', '2003-05-01');
INSERT INTO Employees (id, name, date_hired)
VALUES (2, 'Abigail', '2007-11-23');
```

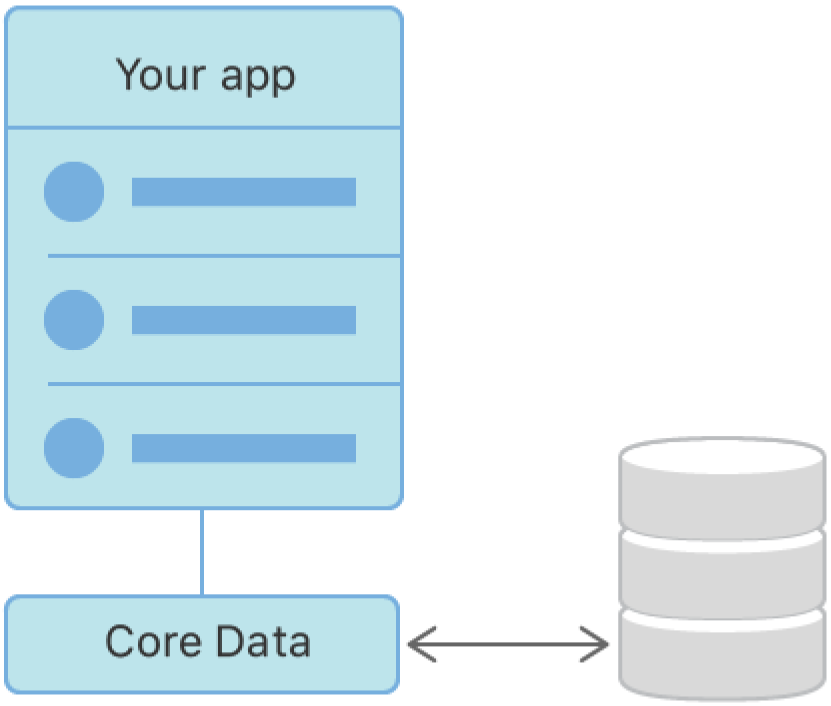
Implementing the database and tables in Core Data

1. Define entities and their attributes

Generate the classes for each entity (model classes)
Obtain a reference to the **persistent container**
Obtain a reference to the **managed context**
Use the managed context to interact with your entities

How does CoreData work?

Every app has access to a SQLite database.
Core Data acts as an interface between your app and the SQLite database.



Source: Apple Developers

CoreData consists of two parts:

- The persistent container - directly interfaces with the database tables
- The managed object context - the interface your app interacts with

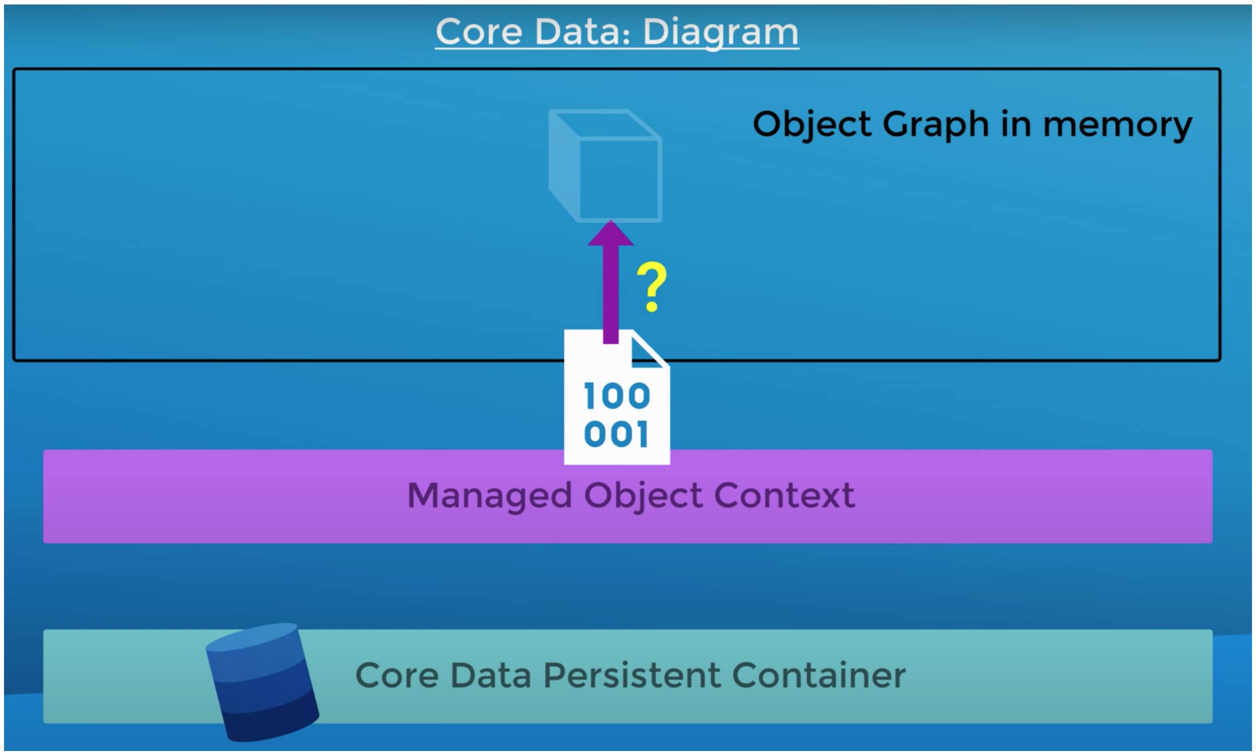
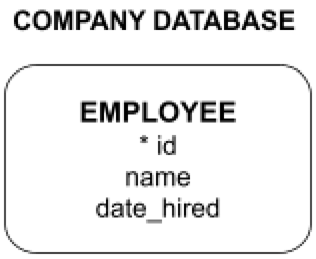


Diagram from Code with Chris

Coding with Core Data

In this example, we will create an Employee table and perform CRUD operations on it.

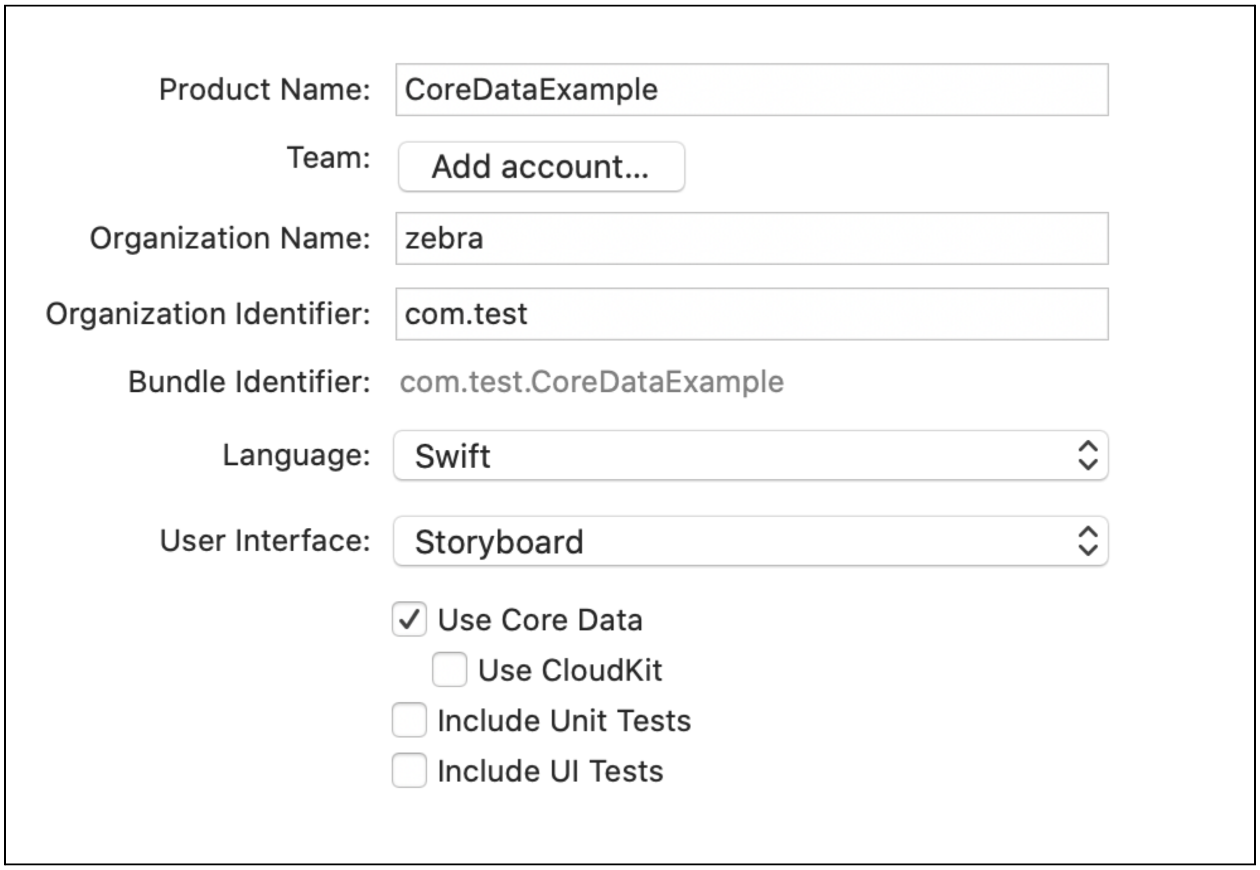


Learning Objectives:

1. Modelling an entity and attributes
2. Inserting data
3. Updating data
4. Deleting data
5. Searching for data

Add Core Data to the Project

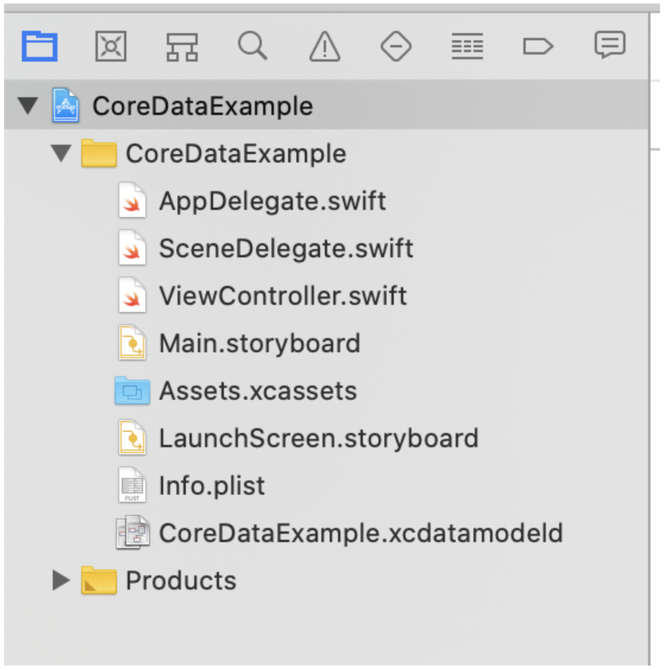
- When creating the project, ensure you check “Use Core Data”



Product Name:	CoreDataExample
Team:	Add account...
Organization Name:	zebra
Organization Identifier:	com.test
Bundle Identifier:	com.test.CoreDataExample
Language:	Swift
User Interface:	Storyboard

- [✓] Use Core Data
  - [ ] Use CloudKit
- [ ] Include Unit Tests
- [ ] Include UI Tests

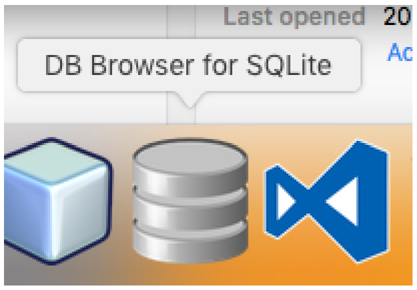
Expected Result



CoreDataExample
CoreDataExample
AppDelegate.swift
SceneDelegate.swift
ViewController.swift
Main.storyboard
Assets.xcassets
LaunchScreen.storyboard
Info.plist
CoreDataExample.xcdatamodeld
Products

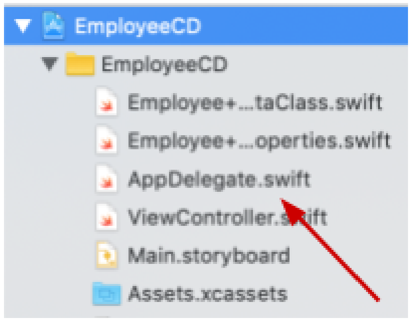
Locating the CoreData SQLite file

Software: https://sqlitebrowser.org/



1. In your app's AppDelegate.swift file, look for this function:

- Add the code in yellow to the function



```
func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        // Override point for customization after application launch.
        let path = FileManager
            .default
            .urls(for: .applicationSupportDirectory, in: .userDomainMask)
            .last?
            .absoluteString
            .replacingOccurrences(of: "file://", with: "")
            .removingPercentEncoding

        print("Core Data DB Path :: \(path ?? "Not found")")
        return true
    }
```

2. Run the app, look at terminal

The code from step 1 will cause your app to output the location of your app's writable document directory.

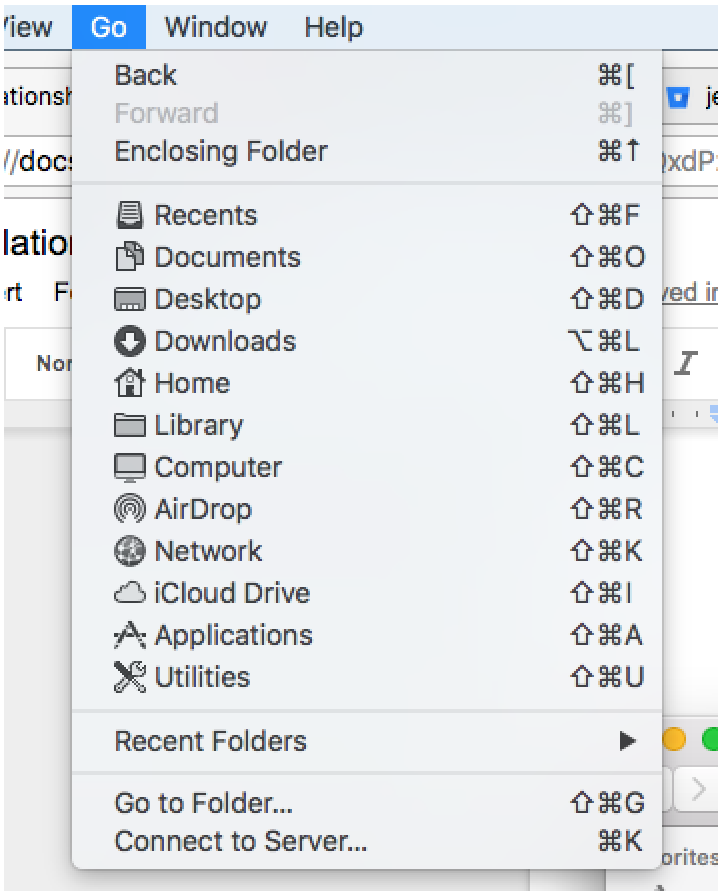
At the top of your terminal, you should see something like this:

```
Core Data DB Path ::
/Users/zebra/Library/Developer/CoreSimulator/Devices/E39E77E9-3964-4C57-8EF1
-5A374D5145BA/data/Containers/Data/Application/865CC101-055F-4CE9-B794-D2FB2
A55E996/Library/Application Support/
```

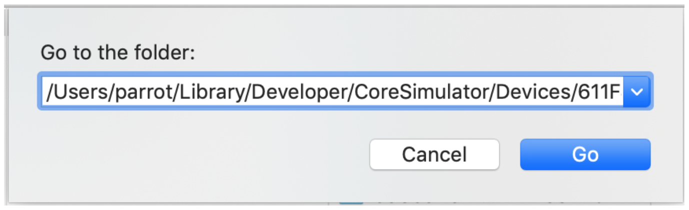
Make a note of the path highlighted in yellow above.

You will need to open this directory on your computer.

3. Open Finder >Go > Go to Folder



3. Copy and paste the file path into the folder window:

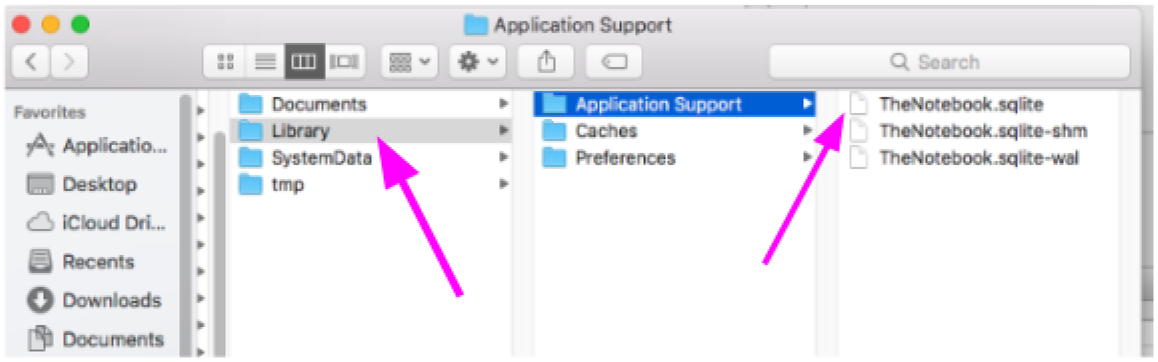


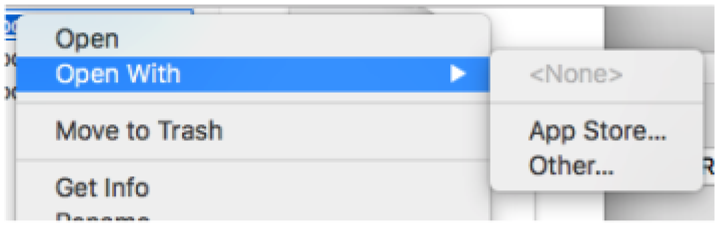
4. Open the *Library > Application Support* folder.

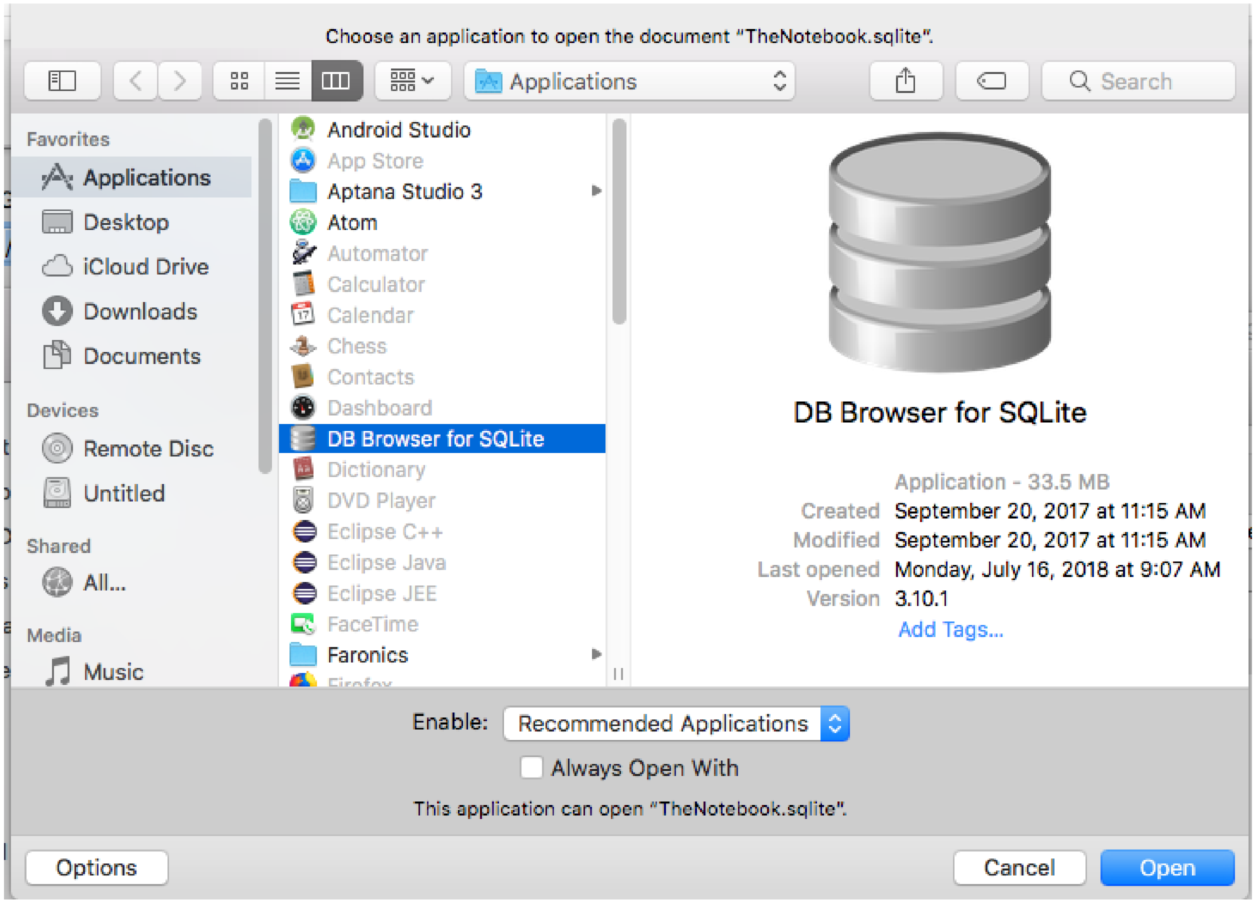
Inside the folder, you will find an .sqlite file.

5. Open the SQLite file using the DBBrowser:

- Right click on .sqlite file
- Choose Open With > Other
- Choose DB Browser for SQLite







Choose an application to open the document "TheNotebook.sqlite".

DB Browser for SQLite

Application - 33.5 MB
Created September 20, 2017 at 11:15 AM
Modified September 20, 2017 at 11:15 AM
Last opened Monday, July 16, 2018 at 9:07 AM
Version 3.10.1

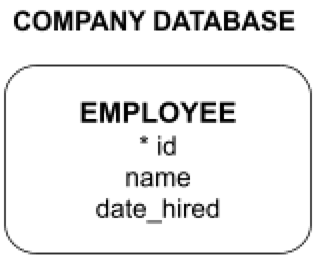
Enable: Recommended Applications
Always Open With

This application can open "TheNotebook.sqlite".

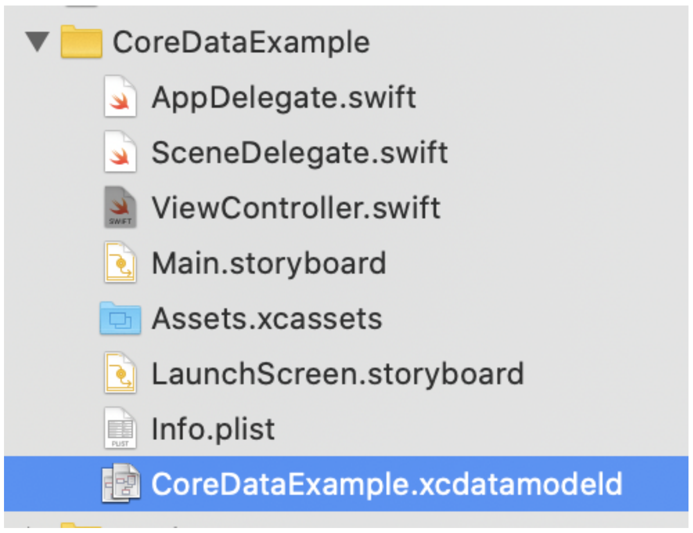
Use the CoreData GUI to create an entity

To create an entity and attributes, we use the CoreData modelling tool.

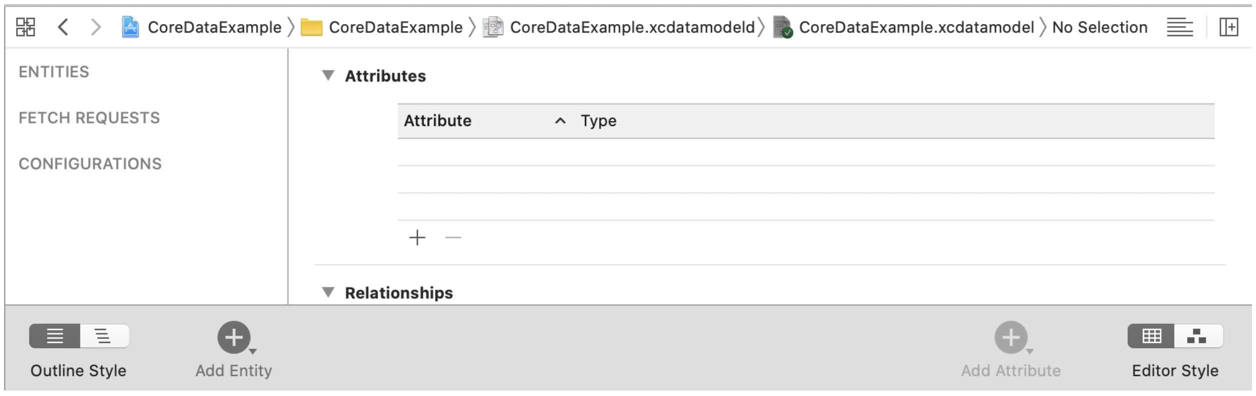
Suppose we want to create an Employee entity.

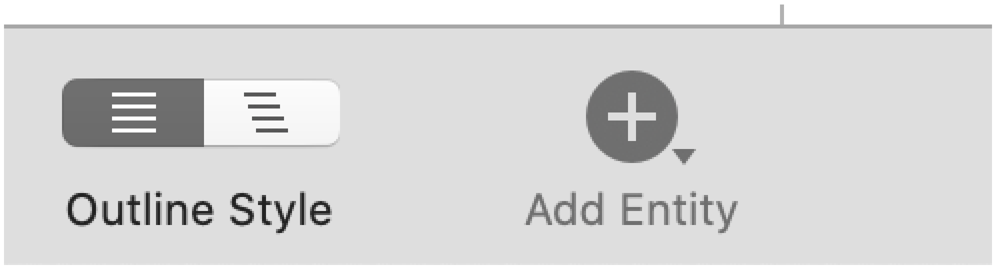


1/ Click on the *.xcdatamodelId file

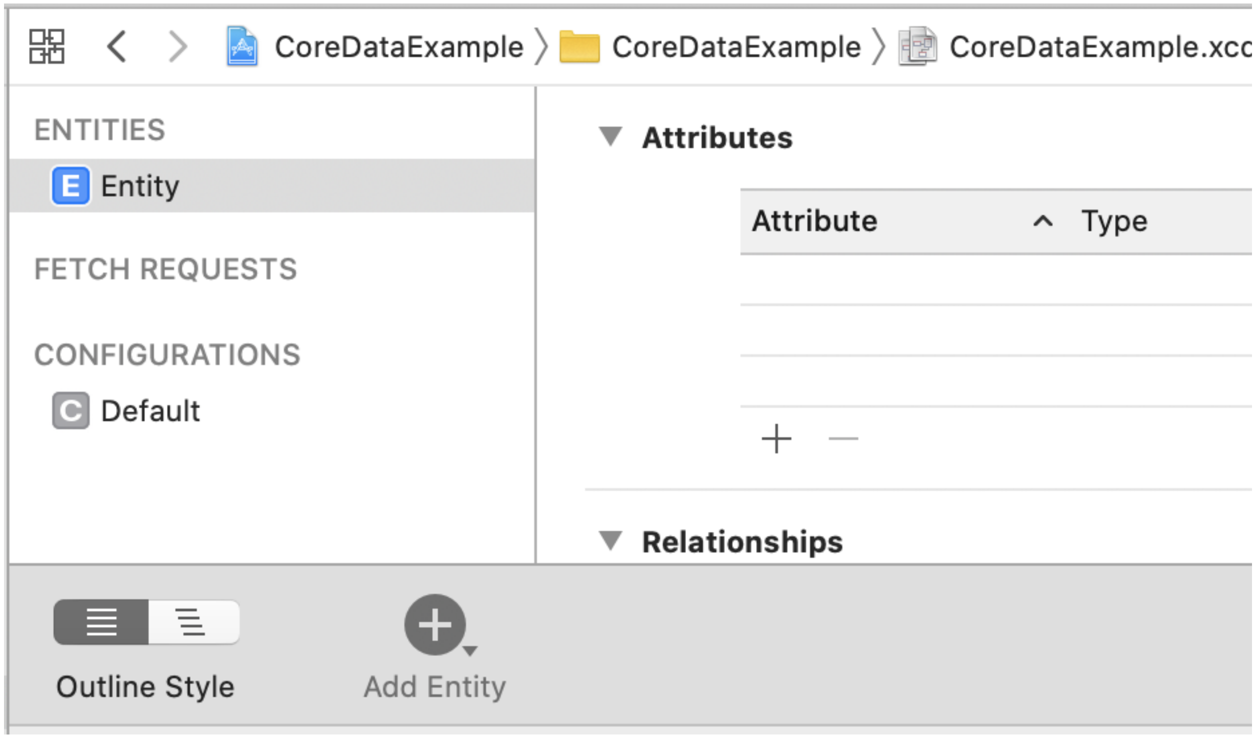


2/ In the interface, tap the +ADD ENTITY button

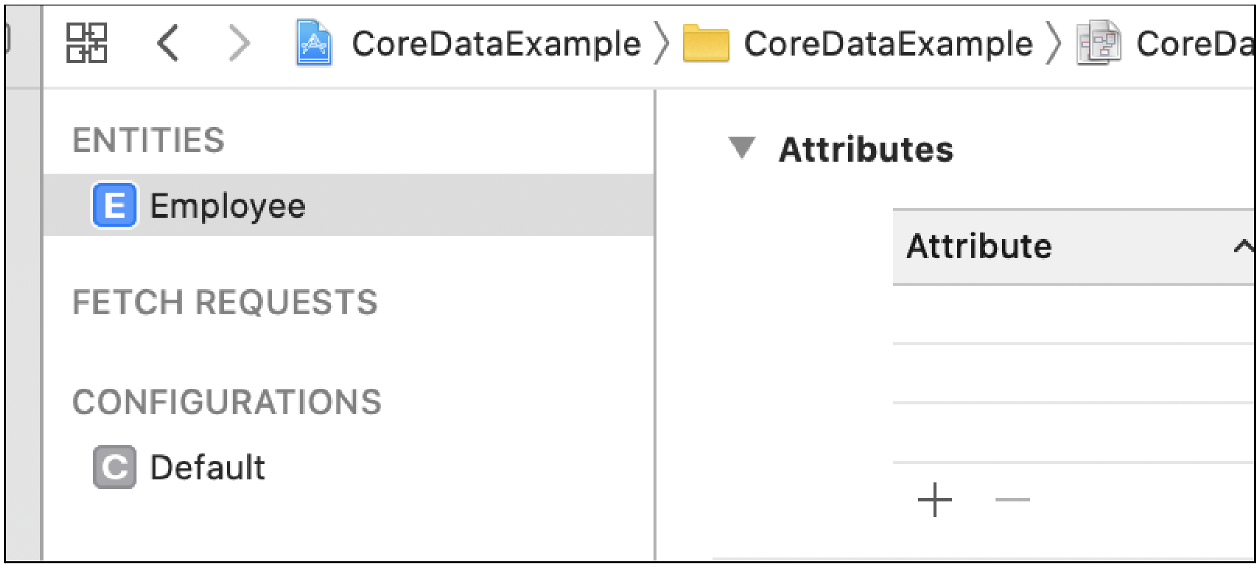




Expected result

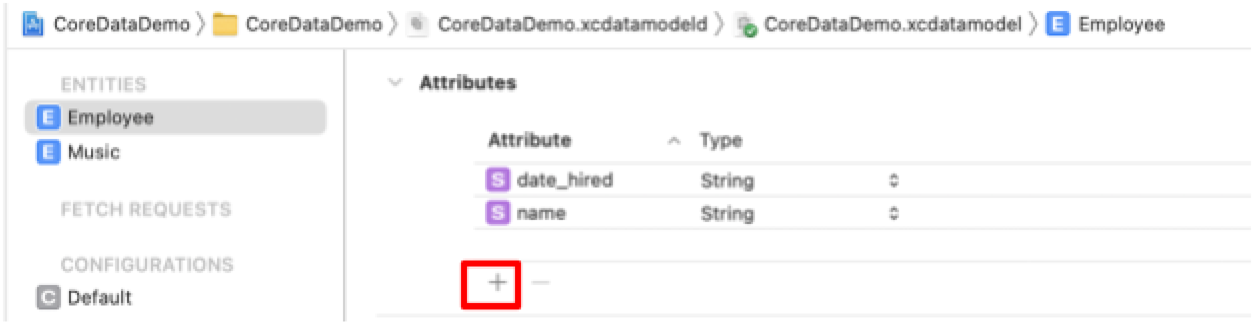


3/ Rename your Entity to “Employee”



Adding Attributes to your Employee Entity

Tap the + button to add a new attribute



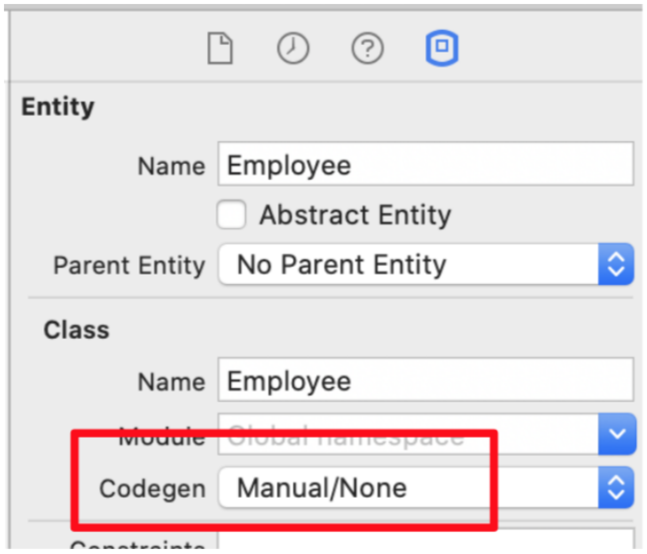
Attribute	Type
date_hired	String
name	String

Generating a Model Class for The Entity

- To work with your entity in code, we need to represent the entity as a *class*.
- XCode can automatically generate the class definition for your entity.

1/ Select the Entity. In the Inspector, go to Class > Code Gen

- Select Manual/None


Entity

Name: Employee

Abstract Entity: ☐

Parent Entity: No Parent Entity

Class

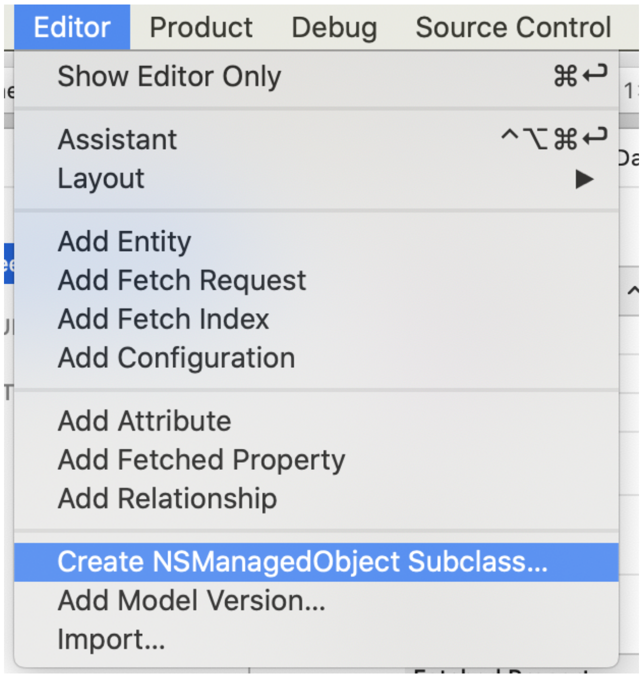
Name: Employee

Module: Global namespace

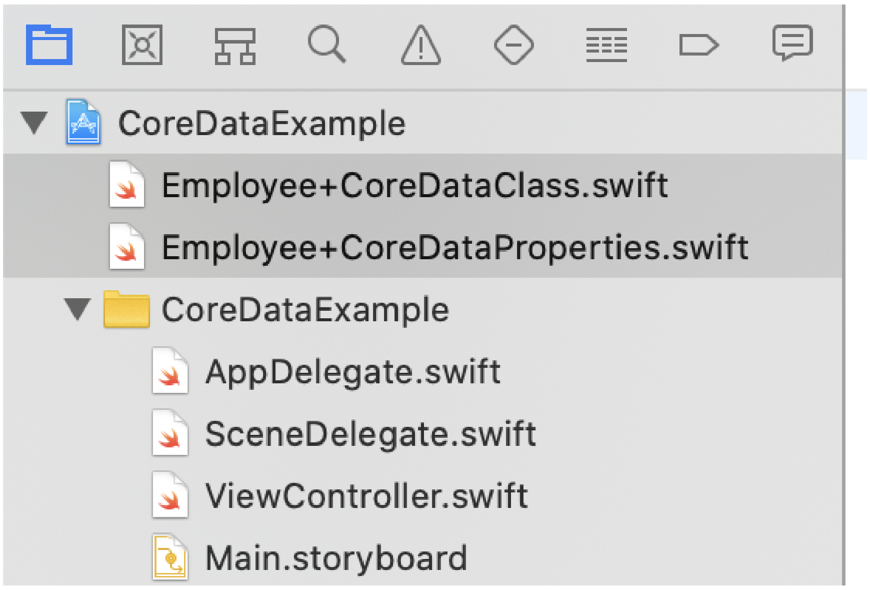
Codegen: Manual/None

2/ Select Editor > Create NSManagedObject Subclass

- This will generate a Swift class that represents your Entity



After completing the wizard, you should get two new files:



Accessing the context variables

To perform CRUD operations, we need a reference to Core Data's managed context variable.

On the screens that need to access CoreData, add a context variable, as follows:

ViewController.swift

```
class ViewController: UIViewController {

   // context variable
   let context = (UIApplication.shared.delegate as!
AppDelegate).persistentContainer.viewContext

   // MARK: Outlets
   override func viewDidLoad() {
      super.viewDidLoad()
      // Do any additional setup after loading the view.
   }
}
```

CRUD operations are performed through CoreData's managed context object.

Insert data:

1/ Using Swift, create an **instance** of the data you want to insert

2/ Set the data's properties to se

2/ Call save() on the context variable

- Saves can fail - so this function call must be marked with try wrapped in a do-catch

```
func addEmployee(name:String, dateHired:String) {
        // 1. Create an employee object
        let e1 = Employee(context: self.context)

        // 2. Set the properties of that object
        e1.date_hired = dateHired
        e1.name = name

        // 3. Use the context variable to save the Employee to the database table
        do {
           try self.context.save()
           print("Employee saved!")
        } catch {
           print("Saved failed.")
        }
    }
```

Searching fo Data Fetch data

1/ Create a fetch request object using the entity's autogenerated fetchRequest() function
2/ Call the content variable's fetch() function:

- The fetch() function is the equivalent of running a SELECT * FROM EMPLOYEE
- The fetch() function returns an *array* of Employee objects
- The fetch() can fail, so the call must be wrapped in a do-catch

```
@IBAction func fetchAll(_ sender: Any) {

     // 1. Create a fetch request object using Employee's fetchRequest() function
    let request:NSFetchRequest<Employee> = Employee.fetchRequest()

    // 2. Initiate the fetch request
    do {
        let results:[Employee] = try self.context.fetch(request)

        print("Fetch succeeded")
        print("Number of results: \(results.count)")

        for employee in results {
            print("Name: \(employee.name ?? "N/A")")
            print("Date Hired: \(employee.date_hired ?? "N/A")")
            print("------")
        }

    } catch {
        print("Error while fetching")
    }
}
```

2/ If fetch() was successful, then do something with the results

Filtering the Search Results

Search queries can be filtered by adding a *predicate* to the search request

* By default, the context.fetchRequest() function will always perform a SELECT * FROM _____

If you want to filter the results (eg: SELECT * FROM____ WHERE ____), then you need to use something called an NSPredicate

```
@IBAction func fetchAll(_ sender: Any) {
      // 1. Create a fetch request object using Employee's fetchRequest() function
      let request:NSFetchRequest<Employee> = Employee.fetchRequest()

      // predicate (filters that you add to the search query to narrow down the results)
     // this predicate updates the query to be:
     //           SELECT * FROM employees where name == 'Carl'
      request.predicate = NSPredicate(format:"name ==  %@", "Carl")


      // 2. Initiate the fetch request
      do {
         let results:[Employee] =  try self.context.fetch(request)

         print("Fetch succeeded")
         print("Number of results: \(results.count)")

         for employee in results {
            print("Name: \(employee.name ?? "N/A")")
            print("Date Hired: \(employee.date_hired ?? "N/A")")
            print("------")
         }

      } catch {
         print("Error while fetching")
      }
   }
```

Update Data

1/ Retrieve

2/ Use the data's class properties to update the data's values

3/ Call .save() on the context variable to persist the change

- Persisting the change can fail, so need to mark this call with a try and wrap it in do-catch

```
@IBAction func updatePressed(_ sender: Any) {
    // 1. Search for the employee you want to update (Abigail)
    let request:NSFetchRequest<Employee> = Employee.fetchRequest()
    request.predicate = NSPredicate(format:"name ==  %@", "Abigail")
    do {
        let results:[Employee] = try self.context.fetch(request)

        // retrieve the employee from the array
        let abigail = results.first!
        print(abigail.name)

        // 2. Update their data
        abigail.name = "Alex"
        abigail.date_hired = "2035-05-23"

        // 3. Save the results
        try self.context.save()

    } catch {
        print("Error while finding a matching employee OR during the update")
    }


}
```

Delete data

1/ Get a copy of the data you want to delete

2/ Call .delete() on the context variable. Pass in a reference to the data you want to delete (From step #1)

- Does not require a try,do-catch

3/ Call .save() on the context variable. to persist your change

- Persisting the change can fail, so need to mark this call with a try and wrap it in do-catch

```
@IBAction func deletePressed(_ sender: Any) {
        // 1. Search for the employee you want to delete
        let request:NSFetchRequest<Employee> = Employee.fetchRequest()
        request.predicate = NSPredicate(format:"name ==  %@", "Alex")
        do {
            let results:[Employee] =
                try self.context.fetch(request)

            let alex = results.first!
            print(alex.name)

            // 2. Delete them
            self.context.delete(alex)

//          // 3. Save the results
            try self.context.save()

            print("Delete success")

        } catch {
            print("Error while finding a matching employee OR deleting")
        }
    }
```