MOV02 - Lab 14

Juan Pablo Arias Mora

Marzo 2021

1 Enunciado

> "Pienso que si haces algo y resulta ser una buena idea, entonces debes hacer otras cosas increíbles, no lo pienses mucho tiempo. Sólo descubre qué es lo que sigue." Steve Jobs

Hay una serie de tecnologías diferentes que se pueden utilizar cuando se necesitan almacenar datos en las aplicaciones. Core Data es la solución ideal en iOS. Con un rendimiento impresionante y un amplio conjunto de funciones, el **Apple Core Data Framework** administra toda la capa de modelo de su aplicación y maneja la persistencia en el disco de almacenamiento de su dispositivo.

La idea es refactorizar una aplicación para agregar persistencia y evitar la de perdida des datos cuando la aplicación se reinicie.

1.1 Versiones Permitidas

- MacOs 10.15.7 - Inglés/Español
- Xcode 12.4
- Simulador iPhone 12 Pro Max

1.2 Creación de aplicación base

Paso 1: Cree un nuevo proyecto en Xcode, utilizando el template App dentro de iOS, en la ventana de opciones para la configuración especifica de la nueva aplicación.

Product Name : todolist
Organization Identifier : com.cenfotec.mov02.14
Interface : SwiftUI
Life Cycle : SwiftUI App
Languge : Swift
Use Core Data: Sin Seleccionar
Include Test: Sin Seleccionar

1.3 Creación de aplicación base

1.4 Creación de aplicación base

Paso 1: Crearemos una estructura de **Groups** para nuestra aplicación, por tanto crearemos 4 nuevos **Groups** (ver figura 1) : **Data - Views - ViewModels - Managers**. Esto se realiza en el **Navigator Area**, sobre el **"Folder"** que lleva el nombre de nuestro proyecto. , la estructura final debe ser similar a la figura 2.

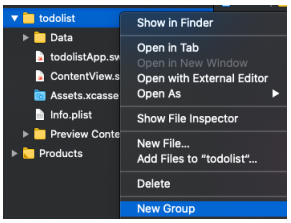


Figure 1: Como crear un Group en XCode

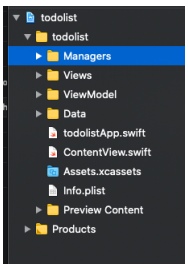


Figure 2: Estructura final de grupos

Paso 2: Como primer paso para trabajar con **Core Data**, crearemos un archivo nuevo de tipo **Data Model** (ver figura 3). Este es quien define la estructura de los objetos de la aplicación, incluidos sus tipos de objeto, propiedades y relaciones. El cual llevara por nombre **TaskDataModel** y lo colocaremos dentro del **Group Data**. La ventana final debe ser similar a la figura 4.

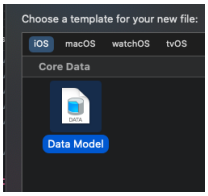


Figure 3: Data Model File

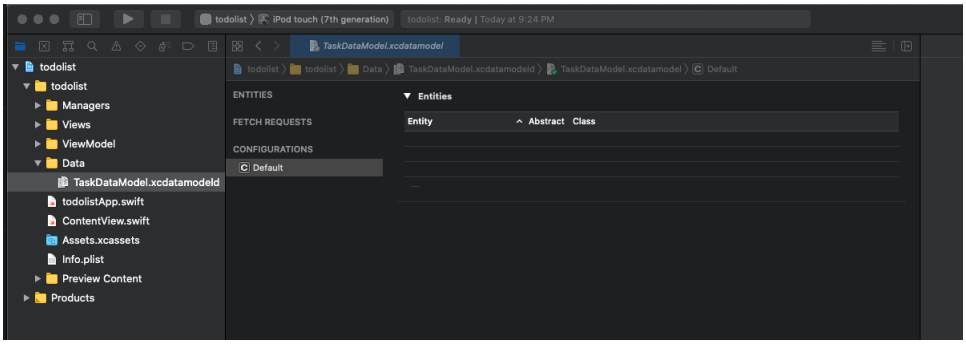


Figure 4: Data Model File en Editor

Paso 3: Debemos crear nuestra **Entity** inicial, la cual denominaremos **TaskEntity**, si bien es cierto algunas personas consideran redundante el uso de **Entity** dentro de los nombres de las mismas, pero para efectos de facilidad de seguimiento en laboratorio es mucho mejor. Para realizar esto debemos ir a la parte inferior del **Editor** y seleccionamos la opción **Add Entity**.(ver figura 5)

Paso 4: Una vez creada la entidad procedemos a cambiarle el nombre de **Entity** a **TaskEntity**, esto lo hacemos buscando la lista de **Entities** en la parte superior del **Editor**, doble **Click** sobre la misma y colocamos el nuevo nombre.(ver figura 6)

Figure 5: Boton Add Entity

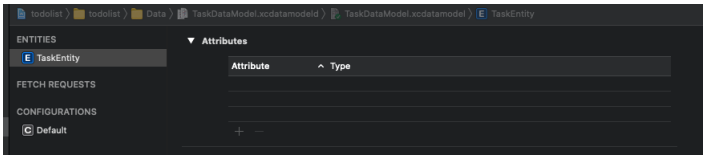


Figure 6: Lista de Entities

Paso 5: Procedemos a agregar los atributos de nuestra **Entity**, para esto manteniendo seleccionada **TaskEntity** presionamos el botón de **Add Attribute** en la parte inferior del **Editor**.

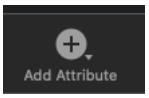


Figure 7: Boton Add Attribute

Paso 6: Agregamos el atributo **priority** de tipo **String**, **title** de tipo **String**, y **taskDescription** de tipo **String** (este ultimo debe ser así, ya que internamente los modelos tienen un atributo de nombre **description**). El resultado debe ser similar a la figura 8, con esto finalizamos la edicion de nuestro **TaskEntity**. Como nota importante todos en defecto son datos opcionales, pero en el **Attribute Editor**, podemos colocarlos como fijos, esto no afecta el desarrollo del laboratorio pero si debe existir la nota aclaratoria.

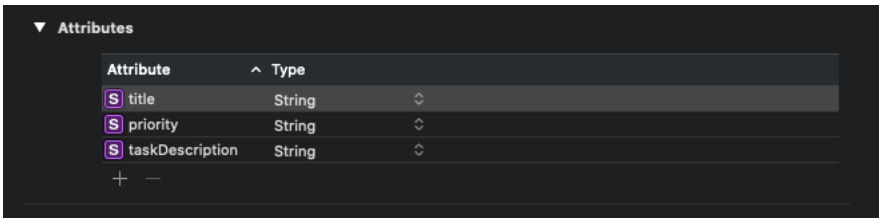


Figure 8: Lista de Attributes

Paso 7: Generaremos nuestro primer **View**, el cual llevara por nombre **AddTaskView** y pertenecerá al **Group View**.

Paso 8: Una vez creado reemplazaremos su código inicial por el siguiente:

```
struct AddTaskView: View {

    var body: some View {
        //NavigationView{
        VStack {
            Form{
                Text("Task Title")
            }
            Button(action: {}) {
                HStack {
                    Image(systemName: "plus.circle.fill")
                    Text("Add")

                }
            }
            .foregroundColor(Color.white)
            .padding()
            .background(Color.blue)
            .cornerRadius(15.0)

        }
        .toolbar {
            ToolbarItem(placement: .principal) {
                VStack {
                    Text("Add Task").font(.headline)
                }
            }
        }


    }

}
```

Paso 9: Utilizando el **Preview** el resultado del código anterior debe ser similar a la figura 9.

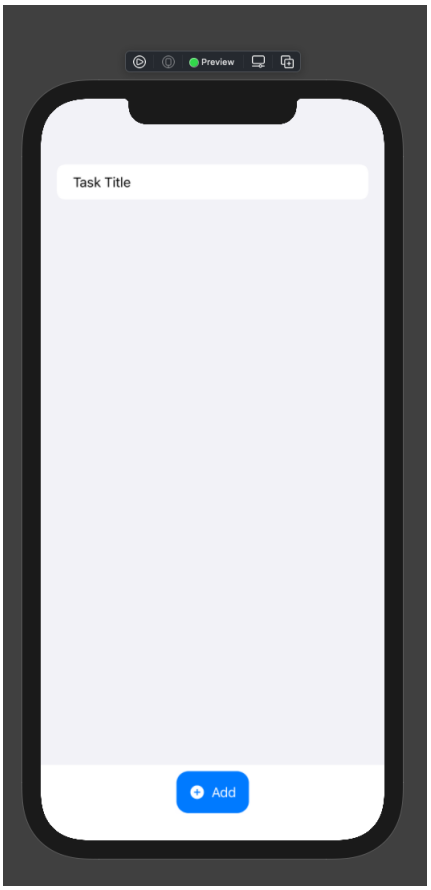


Figure 9: Base de Agregar Task

Paso 10: Agregamos tres variables con el **Property Wrapper State**, estas para manejar el ingreso de parámetros por parte del usuario, las mismas corresponden a los valores de los atributos previos definidos en nuestro **Entite**.

```
@State var taskTitle: String = ""
@State var taskText: String = ""
@State var priority: String = "Low"
```

Paso 11: Debemos ubicar dentro del código la linea que contiene Text("Task Title"), y justo despues de ella iniciaremos agregando el siguiente código:

```
TextField("I need to do ...", text: $taskTitle)
Text("Describe your task")
TextEditor(text: $taskText)
```

Paso 12: A este nivel ya contamos con dos entradas de datos por el usuario pero no tenemos la prioridad. La misma utilizaremos un objeto **Picker** el cual provee las capacidades para funcionar de selector, para esto utilizamos el siguiente **código**:

```
Picker("Priority", selection: $priority){
    ForEach(priorities, id: \.self){ text in
        Text(text)
    }
}
```

Paso 13: A este punto el **Editor** debe mostrar un error ya que no hemos definido la constante que lleva los tipos de prioridades validos. Así que justo después de la definición de los **@States** agregamos el siguiente código:

```
let priorities = ["Low", "Normal", "High"]
```

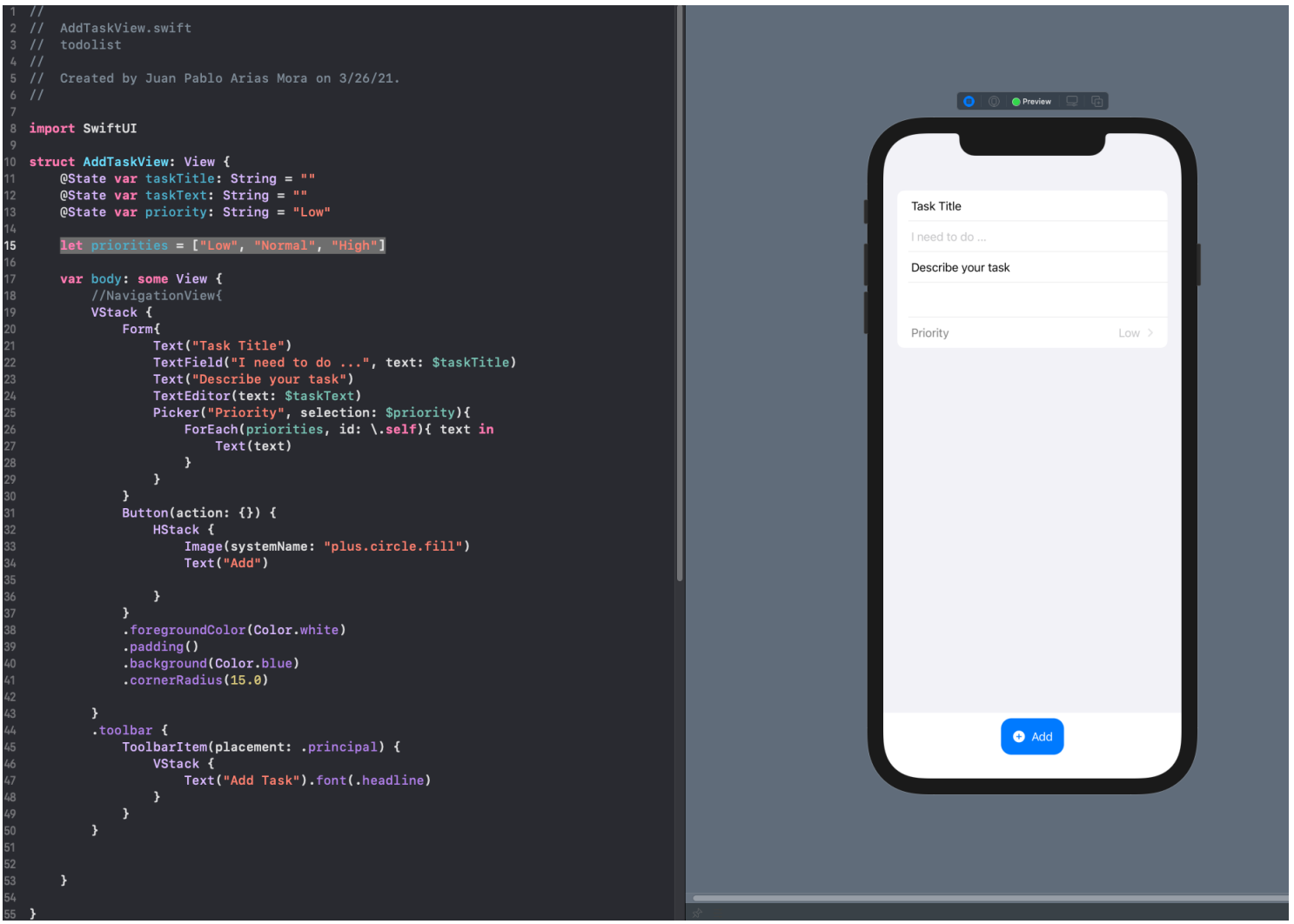


Figure 10: Segunda Etapa de Agregar Task

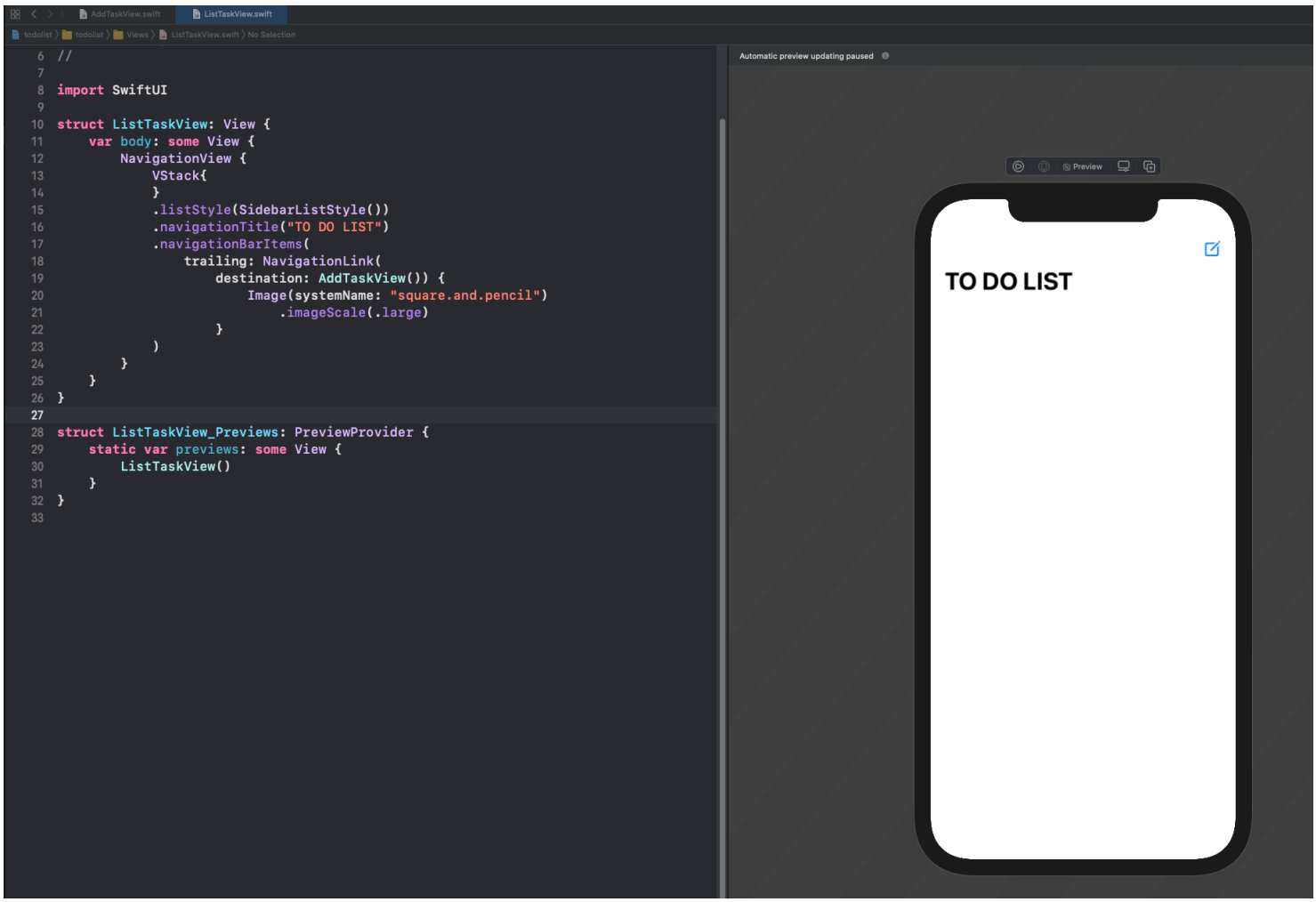
Paso 14: Ejecutamos nuevamente el **Preview**, y el mismo debe ser similar a la figura 10.

Paso 15: Generaremos nuestro segundo **View**, el cual llevara por nombre **ListTaskView** y pertenecerá al **Group Views**.

Paso 16: Una vez creado reemplazaremos su código inicial por el siguiente:

```
struct ListTaskView: View {
    var body: some View {
        NavigationView {
            VStack{
            }
            .listStyle(SidebarListStyle())
            .navigationTitle("TO DO LIST")
            .navigationBarItems(
                trailing: NavigationLink(
                    destination: AddTaskView()) {
                        Image(systemName: "square.and.pencil")
                            .imageScale(.large)
                    }
            )
        }
    }
}
```

Paso 17: Utilizando el **Preview** el resultado del código anterior debe ser similar a la figura 11.




```
//

import SwiftUI

struct ListTaskView: View {
    var body: some View {
        NavigationView {
            VStack{
            }
            .listStyle(SidebarListStyle())
            .navigationTitle("TO DO LIST")
            .navigationBarItems(
                trailing: NavigationLink(
                    destination: AddTaskView()) {
                        Image(systemName: "square.and.pencil")
                            .imageScale(.large)
                    }
            )
        }
    }
}

struct ListTaskView_Previews: PreviewProvider {
    static var previews: some View {
        ListTaskView()
    }
}
```

Figure 11: Base de View Principal

Paso 18: Ahora dentro del archivo **todolistApp.swift**, reemplazaremos la definicion de **ContentView**, por **ListTaskView**

Paso 19: Compilamos el código y ejecutamos la aplicación, el resultado, debe permitirnos navegar por medio del botón en la esquina superior derecha, a la ventana de creación de **Task**. Especial atención al comportamiento del **TextEditor**, y de la selección de **Priority**.

Paso 20: Para corregir el crecimiento dinámico del **TextEditor** debemos agregarle la siguiente propiedad

```
.frame(minHeight: 200.0, maxHeight: 200)
```

Paso 21: Agregaremos la siguiente variable, de manera que nuestro **AddTaskView** tenga acceso a conocer si es hija de algún otro **View** y de esta manera poder ejecutar acciones al respecto.

```
@Environment(\.presentationMode) var presentationMode
```

Paso 22: Procederemos a agregar un método a nuestro **View** que maneje el guardar los datos provistos por el usuario, y hacemos su respectiva llamada en el **action** de nuestro **Botón Add**

```
func Save(){
    self.presentationMode.wrappedValue.dismiss()
}
```

Paso 23: Si bien es cierto la función anterior solamente nos permite completar el flujo completo de la navegación dentro de los **View**, podemos compilar y ejecutar, para ver el flujo del mismo.

Paso 24: Crearemos un nuevo archivo dentro del **Group Managers**, el cual lleva por nombre **CoreDataManager.swift**, el proposito del mismo es ser quien interactua con CoreData, por tanto ejecutara las funciones de **Save**, **Update**, **Delete**.

Paso 25: Una vez creado colocaremos el siguiente código:

```
import CoreData
class CoreDataManager{
    let persistentContainer: NSPersistentContainer

    init(){
        // Which file should the container use: In this case TaskDataModel
        persistentContainer = NSPersistentContainer(name: "TaskDataModel")
        persistentContainer.loadPersistentStores { (description, error) in
            if let error = error {
                print("Unable to load Core Data Store \(error)")

            }

        }
    }
}
```

Paso 26: La capa de abstracción otorgada por medio de **Apple** para el manejo de **Core Data**, se basa en el uso de **NSPersistentContainer**, quien simplifica la creación y gestión de **Core Data**. El código anterior solamente tiene como indicación utilizar nuestra definición de **Models** dentro del **TaskDataModel** como su fuente al momento de instanciar, es importante conocer que el método de almacenamiento por defecto es con el uso de **SQLite**, para efectos del laboratorio este es ideal, pero es importante conocer que se puede llegar a cambiar, y que la capa de abstracción de **Core Data** es quien permite esto.

Paso 27: Si bien es cierto a este punto no hemos colocado la función de guardar una **Task**, por tanto debemos agregar el siguiente código:

```
func saveTask(title: String, description:String, priority:String){
    let task = TaskEntity(context: persistentContainer.viewContext)
    task.title = title

    task.priority = priority
    task.taskDescription = description

    if let result = try? persistentContainer.viewContext.save(){
        print("Success to save task")
    }else{
        print("Failed to save task")
    }

}
```

Paso 28: Entendiendo el código anterior el mismo solo condiciona la función a realizar un manejo de **TaskEntity** de manera se ligue a la interacción con el **NSPersistentContainer**, asigne los valores a guardar de los atributos y luego intente generar el salvar usando la persistencia de datos.

Paso 29: Procedemos a crear un nuevo archivo dentro de los **ViewModels**, quien va ser el encargado de utilizar nuestro **CoreDataManager** dentro de los diferentes **Views**, el mismo lleva por nombre **CoreDataViewModel**.

Paso 30: Una vez creado colocamos el siguiente código:

```
class CoreDataViewModel: ObservableObject{
    @Published var coreDM = CoreDataManager()

    func saveTask(title: String, description:String, priority:String){
```

```
        //Validations
        coreDM.saveTask(title: title, description: description, priority: priority)
    }
}
```

Paso 31: Nuevamente en el archivo **AddTaskView**, agregamos la siguiente variable, para permitir el uso de nuestro recién creado **ViewModel**

```
@ObservedObject var coreDataVM = CoreDataViewModel()
```

Paso 32: Y dentro de la función **Save** de **AddTaskView** antes de realizar el **dismiss**, agregamos el siguiente codigo:

```
self.coreDataVM.saveTask(title: taskTitle, description: taskText, priority: priority)
```

Paso 33: Compilamos y ejecutamos, luego vamos a agregar una **Task**, y una vez presionamos el botón de **Add**, deberemos ver un mensaje en **Debug Console**, que nos indica **Success to save task**, en caso contrario debemos revisar los códigos que agregamos anteriormente. Como dato importante a este punto no hay validación de datos, por tanto se pueden agregar elementos vacíos a **SQLite**. Pero para propósitos del laboratorio es lo esperado.

Paso 34: Si bien es cierto los datos se están almacenando dentro de **Core Data**, pero no tenemos de momento forma de visualizarlo. Para esto crearemos una función dentro de **CoreDataManager**, la cual permita ver la lista de **Task** almacenados, para luego presentarlos en nuestros **View**. Dentro del archivo **CoreDataManager** agregamos la siguiente función:

```
func getAllTask() -> [TaskEntity] {
    let fetchRequest: NSFetchRequest<TaskEntity> = TaskEntity.fetchRequest()

    if let result = try? persistentContainer.viewContext.fetch(fetchRequest){
        print("Success to retrieve all tasks")
        return result
    }
    print("Failed to retrieve all tasks")
    return []
}
```

Paso 35: Analizando la funcion esta hace un llamado a **NSFetchRequest** quien recopila los criterios necesarios para seleccionar y clasificar un grupo de objetos que se encuentran en un almacenamiento persistente (**Core Data**). El mismo es lo suficientemente genérico por tanto debemos colocar de manera explicita que se tratan de **TaskEntity**.

Paso 36: Dentro de nuestro **CoreDataViewModel** declaramos la siguiente variable:

```
@Published var tasks: [TaskEntity] = [TaskEntity]()
```

La cual nos permitirá invalidad con sus cambios los diferentes **Views**.

Paso 37: Y la siguiente función:

```
func getAllTask(){
    tasks = self.coreDM.getAllTask()
}
```

Paso 38: Cambiamos de momento al archivo **ListTaskView** y agregamos una función que nos permita suplantar los valores de prioridades en nuestro **Task**, por un icono correspondiente:

```
func priorityImages(priority:String)->String{
    switch priority {
    case "Low":
        return "3.square"
    case "Normal":
        return "2.square"
    default:
        return "1.square"
    }
}
```

Paso 39: Continuando sobre **ListTaskView** y dentro del **VStack** colocamos el siguiente código:

```
List(coreDataVM.tasks, id: \.self) {
    task in
        Label(
            task.title ?? "No Name",
            systemImage: priorityImages(
                priority: task.priority ?? "")
        )
}
```

El cual nos permite simplemente la visualización de cada uno de los valores almacenados dentro de **Core Data**.

Paso 40: A este punto deberíamos tener un error ya que no hemos declarado nuestro asocie con **CoreDataViewModel**

```
@ObservedObject var coreDataVM = CoreDataViewModel()
```

Paso 41: Para finalizar debemos de utilizar la propiedad de **onAppear** de nuestro **NavigationView**, para hacer el llamado a la actualización de valores de la lista de **task**

```
.onAppear(perform: {
    self.coreDataVM.getAllTask()
})
```

Paso 42: Compilamos y ejecutamos, y a este punto deberíamos se capaces de visualizar en nuestra aplicación las tareas anteriormente creadas. Podemos realizar nuevamente un flujo completo de creación de **Task** y la nueva **Task** debería presentarse en pantalla también, por efecto del **onAppear**.

Paso 43: Vamos a proceder a agregar a nuestro **CoreDataManager**, la funcionalidad de borrado utilizando el siguiente código.

```
func deleteTask(task: TaskEntity){
    persistentContainer.viewContext.delete(task)

    if let result = try? persistentContainer.viewContext.save(){
        print("Success to delete task")
    }else{
        persistentContainer.viewContext.rollback()
        print("Failed to delete task")
    }

}
```

Dentro de este nuevo código es importante destacar que el **delete** solamente marca nuestra **Entity** para ser borrada, pero hasta la acción **Save** es cuando se ejecuta, con lo que permite realizar la funcionalidad **rollback** en caso de fallo y borrar el marcado de borrado. Para efectos de laboratorio es solo para conocimiento general ya que no existe condición critica en el ejemplo que nos lleve a realizar un **rollback**.

Paso 44: Dentro del **CoreDataViewModel** debemos agregar la conexión a esta nueva función:

```
func deleteTask(task:TaskEntity){
    //Validations
    coreDM.deleteTask(task: task)
}
```

Paso 45: Dentro del **ListTaskView** realizaremos un cambio sobre el **List** que utilizamos dentro del **VStack**, el cambio es para poder aprovechar de una propiedad dentro de los **ForEach** que nos permite ejecutar borrados.

El código actual es:

```
List(coreDataVM.tasks, id: \.self) {
```

Paso 46: Debemos cambiarlo por:

```
List{ ForEach(coreDataVM.tasks, id: \.self) {
   ...
}
```

La llave inferior debe cerrar la que esta cerca de **List**

Paso 47: Compilamos y ejecutamos, el flujo de la aplicación no debe tener cambios.

Paso 48: Procedemos a agregar a nivel del **ForEach**, la propiedad **onDelete**, utilizando el siguiente código:

```
.onDelete(perform: { indexSet in
    indexSet.forEach { index in
        let task = coreDataVM.tasks[index]
        coreDataVM.deleteTask(task: task)
        coreDataVM.getAllTask()
    }
})
```

Podemos considerar los closures como necesarios para tener el efecto, aunque en realidad lo que nos permite es agregar la visualización del borrado, así como una función en caso de que se realice.

Paso 49: Compilamos y ejecutamos a este nivel debemos ser capaces de eliminar las **Task** con solo deslizar el objeto hacia la izquierda.

Paso 50: En este momento procederemos a crear un nuevo **View**, el cual lleva por nombre **TaskDetailsView** y contendrá la información de un determinado **Task**.

Paso 51: Una vez creado procedemos a copiar el siguiente código:

```
struct TaskDetailsView: View {

    let task: TaskEntity
    var body: some View {
        VStack {
            Form{
                Text("Task Title").bold()
                Text(task.title ?? "")
                Text("Description").bold()
                Text(task.taskDescription ?? "")
                .frame(width: 100, height: 300, alignment: .center)
            }
            .toolbar {
                ToolbarItem(placement: .principal) {
                    VStack {
                        Text("Task Details").font(.headline)
                    }
                }
            }
        }
    }
}

struct TaskDetailsView_Previews: PreviewProvider {
    static var previews: some View {
        let Task = TaskEntity()
        TaskDetailsView(task: Task)
    }
}
```

Paso 52: Para finalizar la implementación, volvemos al **ListTaskView** dentro del cual identificamos el **Label**.

El código actual es:

```
task in
    Label(
```

Paso 53: Debemos cambiarlo por:

```
task in
    NavigationLink(
        destination: TaskDetailsView(task: task),
        label: {
                Label(
        ...
        }
    )
```

Paso 54: Compilamos y ejecutamos, y por cada **Task** almacenado deberíamos ser capaces de entrar y consultar los detalles, con solo seleccionar su **Title** en pantalla.

Paso 55: Como ejercicio adicional cree un nuevo proyecto en **XCode**, pero esta vez seleccione la opción de **Use Core Data**. Observe en que se asemejan los archivos creados, principalmente **Persistence.swift** con **CoreDataManager**.