Data Storage

Mobile Application Development in iOS

School of EECS

Washington State University

Instructor: Larry Holder

Data Storage

- Already seen: UserDefaults
- File I/O: Read, Write, Codable
- Database support: CoreData
- Data in the cloud: Firebase

File I/O

- FileManager.default
  - Singleton shared file manager for app
  - Numerous methods for manipulating files
- FileManagerDelegate
  - Constraint checking and error handling

File I/O Process

- Get URL to directory
- Append file name to URL
- Convert data to String
- Use String.write(to: URL, atomically: Bool, encoding: .utf8) to write
  - Atomically: write to auxiliary file first
- Use String(contentsOf: URL , encoding: .utf8) to read
- Both throw errors

File I/O: Write

```
func writeData(_ str: String, to fileName: String) -> Bool {
    if let directoryURL = FileManager.default.urls(for: .documentDirectory,
                                in: .userDomainMask).first {
        let fileURL = directoryURL.appendingPathComponent(fileName)
        do {
            try str.write(to: fileURL, atomically: true, encoding: .utf8)
            return true
        } catch {
            print("\(error)")
        }
    } else {
        print("Error accessing document directory.")
    }
    return false
}
```

File I/O: Read

```
func readData(from fileName: String) -> String? {
    if let directoryURL = FileManager.default.urls(for: .documentDirectory,
                                in: .userDomainMask).first {
        let fileURL = directoryURL.appendingPathComponent(fileName)
        do {
            let str = try String(contentsOf: fileURL, encoding: .utf8)
            return str
        } catch {
            print("\(error)")
        }
    } else {
        print("Error accessing document directory.")
    }
    return nil
}
```

File I/O: Codable

- Problem: Convert everything to a String ☹
- Solution: Codable types
  - Can be encoded/decoded to common formats, e.g., JSON
  - JSON data easily converted to/from String
- All basic types and containers are Codable
- Classes consisting of Codable properties are Codable

File I/O: Codable

```
class Player: Codable {
    var name: String
    var health: Int

    init(name: String, health: Int) {
        self.name = name
        self.health = health
    }
}
```

File I/O: Codable

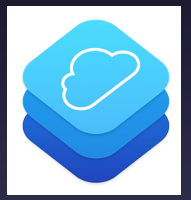
```
func writePlayers(_ players: [Player]) -> Bool {
    let jsonEncoder = JSONEncoder()
    do {
        let jsonData = try jsonEncoder.encode(players)
        if let jsonStr = String(data: jsonData, encoding: .utf8) {
            if writeData(jsonStr, to: playersFileName) {
                return true
            }
        }
    } catch {
        print("\(error)")
    }
    return false
}
```

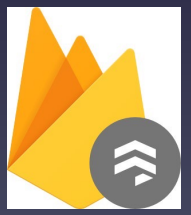
File I/O: Codable

```
func readPlayers() -> [Player] {
    if let str = readData(from: playersFileName) {
        if let jsonData = str.data(using: .utf8) {
            let jsonDecoder = JSONDecoder()
            do {
                let players = try jsonDecoder.decode([Player].self,
                                                     from: jsonData)
                return players
            } catch {
                print("\(error)")
            }
        }
    }
    return []
}
```

Database Support

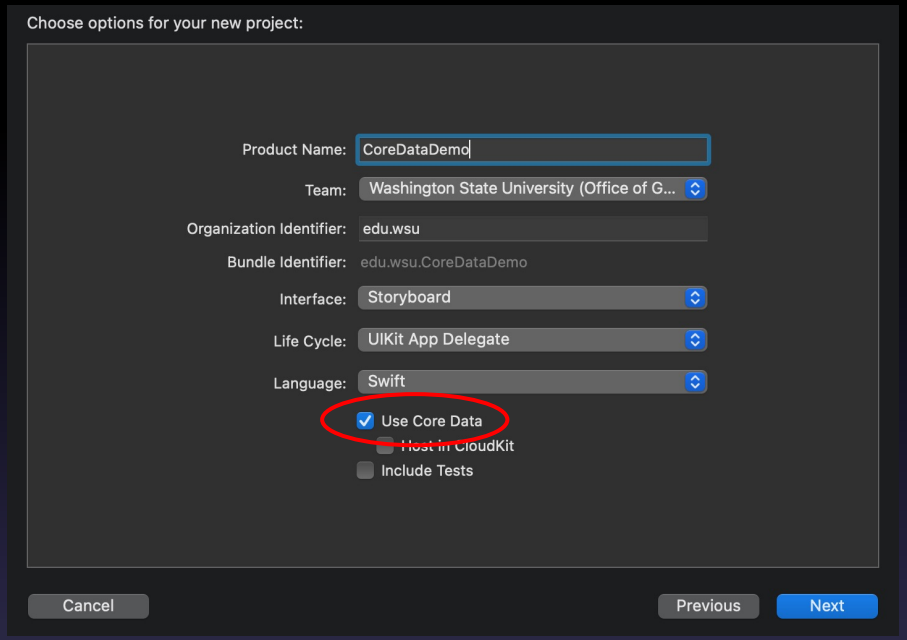
- Core Data
  - iOS-specific object store
- SQLite (www.sqlite.org)
  - Cross-platform table store (already available in iOS)
- Cloud
  - CloudKit
    - Works with Core Data
  - Firebase (firebase.google.com)
    - Cross-platform document store

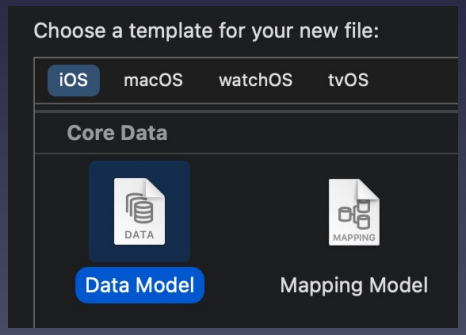




Core Data

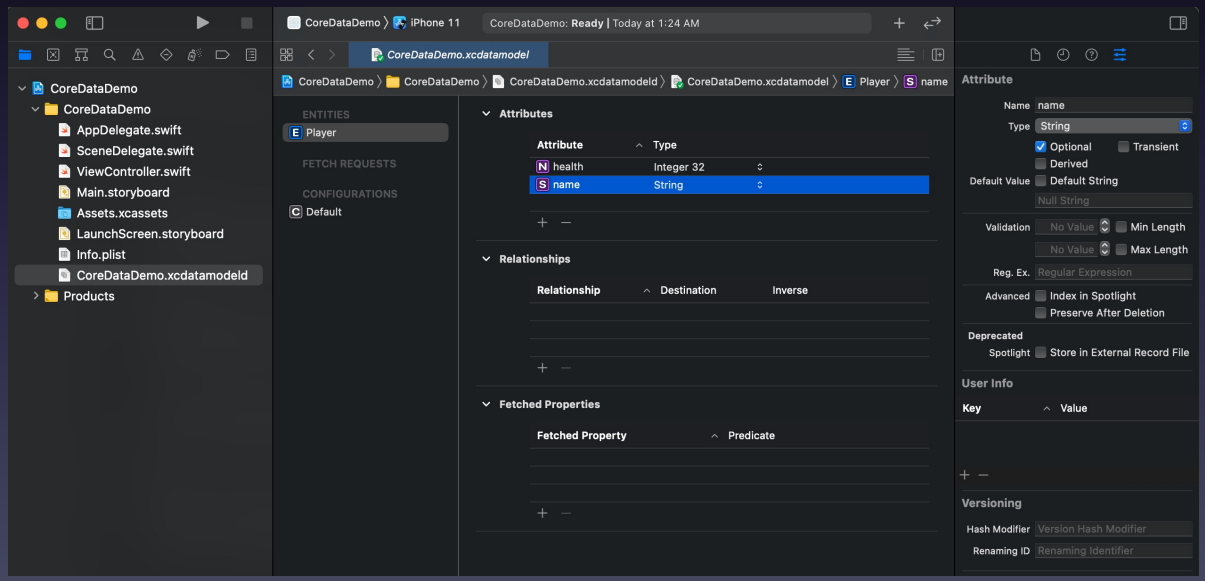
- Check “Use Core Data” for New Project
  - Includes empty data model
  - Includes boilerplate code to create database
- Or, add Core Data model to existing project





Core Data Stack: Model

- Create Managed Object Model (Schema)
- Schema consists of entities, their attributes, and relationships



Core Data Stack: Persistent Container and Context

- Persistent container (NSPersistentContainer)
  - Data store (the "database")
  - Defined in AppDelegate.swift
  - Obtained from UIApplication.shared.delegate
- Managed object context (NSManagedObjectContext)
  - Tracks changes to data store until saved
  - Obtained from NSPersistentContainer.viewContext

Core Data Stack: Access

```
import CoreData

class TableViewController: UITableViewController {

    var players: [NSManagedObject] = []
    var managedObjectContext: NSManagedObjectContext!
    var appDelegate: AppDelegate!

    override func viewDidLoad() {
        super.viewDidLoad()
        appDelegate = UIApplication.shared.delegate as? AppDelegate
        managedObjectContext = appDelegate.persistentContainer.viewContext
    }
}
```

Core Data: Insert

- Methods
  - NSEntityDescription.insertNewObject(forEntityName: String, into: NSManagedObjectContext) -> NSManagedObject
  - NSManagedObject.setValue(value: Any?, forKey: String)
  - NSManagedObjectContext.save()

Core Data: Insert

```
func insertPlayer(name: String, health: Int) {
    let player = NSEntityDescription.insertNewObject(forEntityName:
                    "Player", into: self.managedObjectContext)
    player.setValue(name, forKey: "name")
    player.setValue(health, forKey: "health")
    appDelegate.saveContext() // In AppDelegate.swift
}
```

Core Data: Fetch

- Methods
  - Create fetch request
    - NSFetchRequest<NSManagedObject>(entityName: String) -> NSFetchRequest<NSManagedObject>
  - Call fetch with fetch request
    - NSManagedObjectContext.fetch(request: NSFetchRequest<NSManagedObject>) throws

Core Data: Fetch

```
func fetchPlayers() -> [NSManagedObject] {
    let fetchRequest = NSFetchRequest<NSManagedObject>(entityName: "Player")
    var players: [NSManagedObject] = []
    do {
        players = try self.managedObjectContext.fetch(fetchRequest)
    } catch {
        print("getPlayers error: \(error)")
    }
    return players
}

func printPlayer(_ player: NSManagedObject) {
    let name = player.value(forKey: "name") as? String
    let health = player.value(forKey: "health") as? Int
    print("Player: name = \(name!), health = \(health!)")
}
```

Core Data: Delete

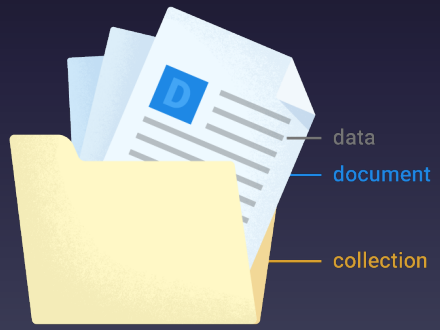
- Methods
  - NSManagedObjectContext.delete(object: NSManagedObject)
  - NSManagedObjectContext.save()

Core Data: Delete

```
func deletePlayer(_ player: NSManagedObject) {
    managedObjectContext.delete(player)
    appDelegate.saveContext()
}
```

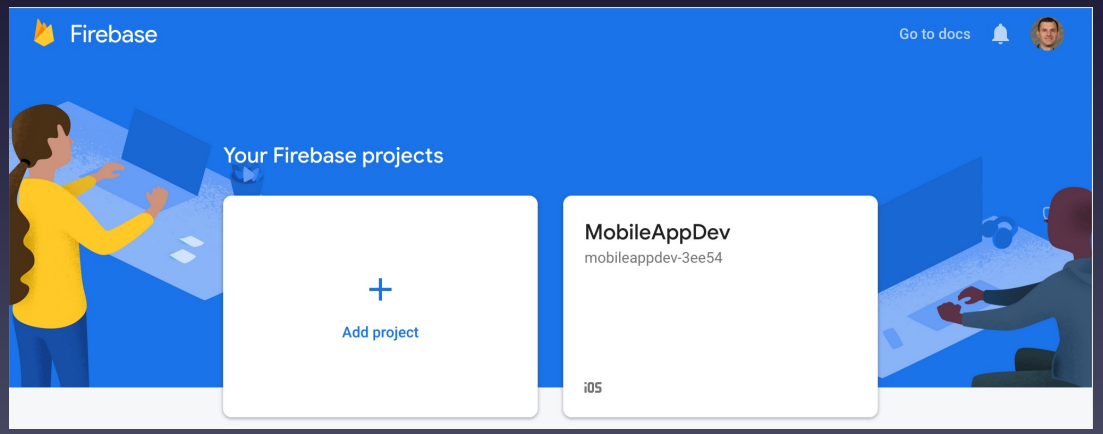
Firebase

- firebase.google.com
- Cloud Firestore
  - Cross-platform document store (NoSQL)
  - Stores Collections of Documents
  - Documents contain key/value pairs



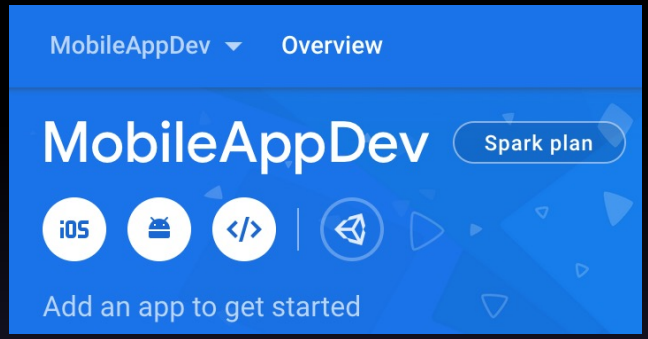
Firebase Setup

- Create Google account
- Goto firebase.google.com
- "GO TO CONSOLE" and sign in
- Add project



Firebase Setup

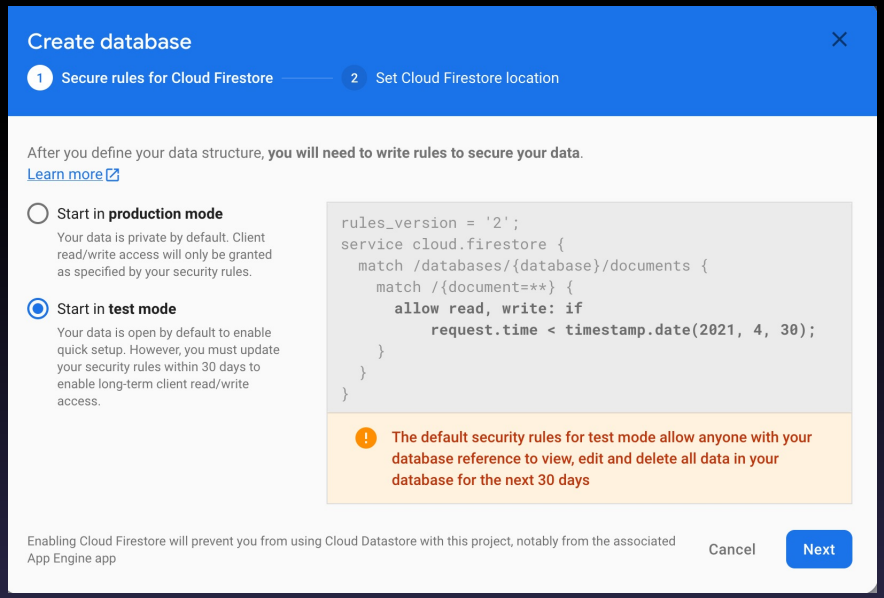
- Add Firebase to app
  - Register app
  - Download config property list file and add to app
  - Add Firebase SDK to app
    - Cocoapods: pod 'Firebase/Analytics'
  - Add initialization code

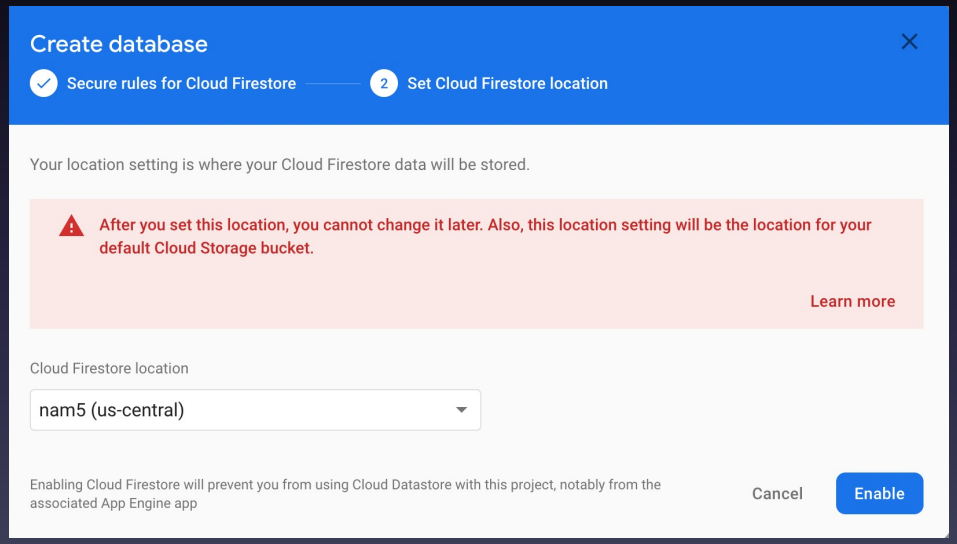


Firebase Setup

```
// AppDelegate.swift

import UIKit
import Firebase

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    func application(_ application: UIApplication, didFinishLaunchingWithOptions
        launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        // Override point for customization after application launch.
        FirebaseApp.configure()
        return true
    }

    // ...
}
```

Cloud Firestore: Create Database





Firestore Setup

- Pod 'Firebase/Firestore'
- pod install (takes a while)

```
import Firebase

let collection = Firestore.firestore().collection("players")
```

Player Class

```
class Player {
    var name: String
    var health: Int
    var id: String?

    init(name: String, health: Int) {
        self.name = name
        self.health = health
    }

    init(dict: [String: Any]) {
        self.name = dict["name"] as! String
        self.health = dict["health"] as! Int
    }

    func toDict() -> [String: Any] {
        return ["name": name, "health": health]
    }
}
```

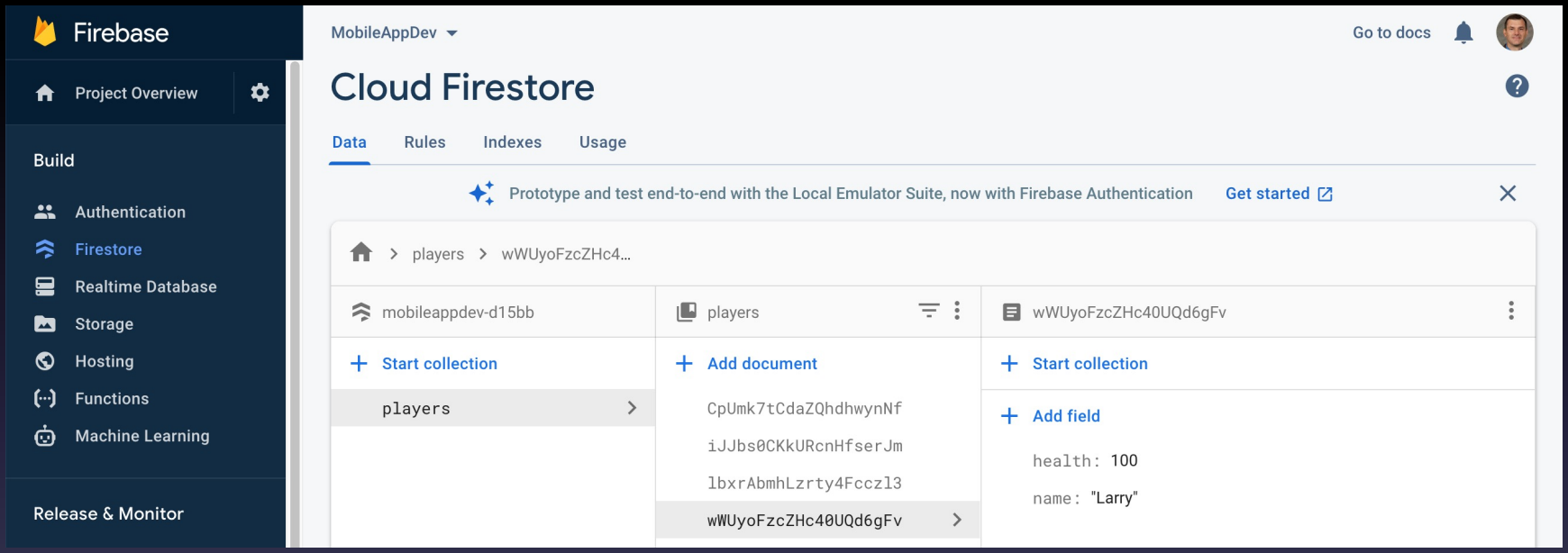
Firestore: Insert

- Collection.addDocument(data: [String: Any],

  completion: ((Error?) -> Void)?) -> DocumentReference

```
func insertPlayer(_ player: Player) {
    var ref: DocumentReference?
    ref = collection.addDocument(data: player.toDict()) { error in
        if let err = error {
            print("Error adding document: \(err)")
        } else {
            print("Document added with ID: \(ref!.documentID)")
            player.id = ref!.documentID
        }
    }
}
```

Firestore Database



Firestore: Fetch

- Collection.getDocuments(completion:

  ((QuerySnapshot?, Error?) -> Void)?)

```
func fetchPlayers() {
    // Following returns immediately
    collection.getDocuments() { (querySnapshot, error) in
        if let err = error {
            print("Error getting documents: \(err)")
        } else {
            self.players = []
            for document in querySnapshot!.documents {
                print("\(document.documentID) => \(document.data())")
                let player = Player(dict: document.data())
                player.id = document.documentID
                self.players.append(player)
            }
            self.tableView.reloadData()
        }
    }
}
```

Firestore: Delete

- Collection. Document(documentID).delete(completion:

  ((Error?) -> Void)?)

```
func deletePlayer(_ player: Player) {
    collection.document(player.id!).delete() { error in
        if let err = error {
            print("Error removing document: \(err)")
        } else {
            print("Document successfully removed")
        }
    }
}
```

Firestore: Listener

- Listeners react to changes to the data store
- Various listeners available
  - Documents
  - Collections
- E.g., receive snapshot of collection when anything changes:
  - Collection.addSnapshotListener(completion: ((QuerySnapshot?, Error?) -> Void)?)

Firestore: Listener

```
func addListener() {
    collection.addSnapshotListener { querySnapshot, error in
        if let err = error {
            print("Error retrieving collection: \(err)")
        } else {
            self.players = []
            for document in querySnapshot!.documents {
                let documentID = document.documentID
                let data = document.data()
                print("listener: \(documentID) => \(data)")
                let player = Player(dict: data)
                player.id = documentID
                self.players.append(player)
            }
            self.tableView.reloadData()
        }
    }
}
```

Resources

- File I/O
  - developer.apple.com/documentation/foundation/filemanager
- Codable types
  - developer.apple.com/documentation/swift/codable
- Core Data
  - developer.apple.com/documentation/coredata
- Firebase: firebase.google.com
- Firestore: firebase.google.com/docs/firestore