Stanford CS193p

Developing Applications for iOS

Spring 2020

Lecture 12

Today

- Core Data
  - Object-Oriented Database
  - We've been doing a lot of functional programming
  - But we're going to switch now to some simple object-oriented programming
  - Using the Core Data infrastructure to store/retrieve data in a database
- SQL vs. OOP
  - Very mature technology exists to store large amounts of data efficiently.
  - The most popular of which is likely SQL.
  - But programming SQL is VERY different than what we're doing in Swift.
  - We're going to get the best of both worlds using the Core Data framework.
  - It does the actual storing using SQL.
  - But we interact with our data in an entirely object-oriented way.
  - We don't need to know a single SQL statement to do it.

Core Data

- Map

  The heart of Core Data is creating a map.
  It is a map between the objects/vars we want and "tables and rows" of a relational database.
  Xcode has a built-in editor for this map.
  It also lets us graphically create "relationships" (i.e. vars that point to other objects).

- Then what?

  Xcode will generate classes behind the scenes for the objects/vars we specified in the map.
  We can use extensions to add our own methods and computed vars to those classes.
  These objects then serve as the source of data for the elements of our UI.

Core Data

- Features

  Creating objects

  Changing the values of their vars (including establishing relationships between objects)

  Saving the objects

  Fetching objects based on specific criteria

  Lots of database-y features like optimistic locking, undo management, etc.

- SwiftUI Integration

  The objects we create in the database are `ObservableObject`s.

  And there is a very powerful property wrapper `@FetchRequest` which fetches objects for us.

  `FetchRequest` essentially represents a "standing query" that is constantly being fulfilled.

  This keeps our UI always in sync with what's happening in the database.

Core Data

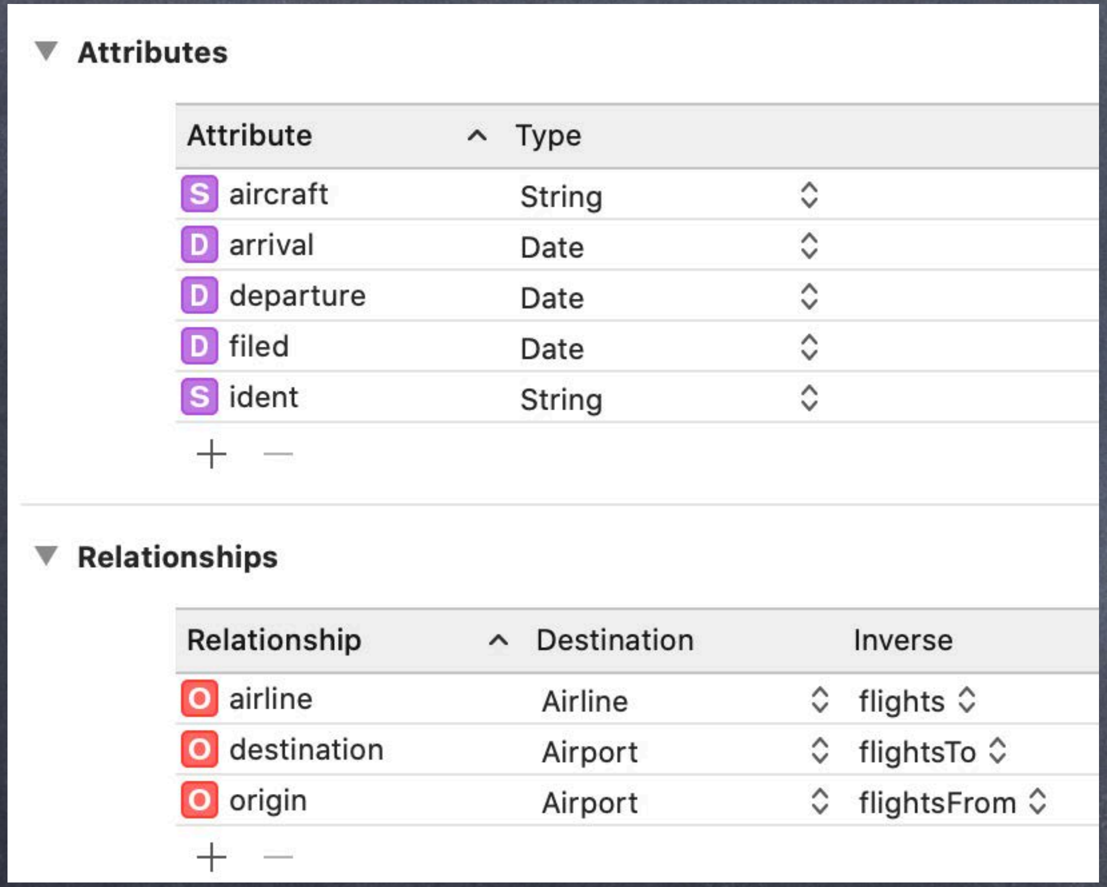
- The Setup

  Start by clicking the "Core Data" button when you create a New Project.
  This will add a blank "map" for you to your project.
  It adds a little bit of code to your `AppDelegate` to create the database store.
  You don't need to pay any attention to that code. It's invoked automatically.
  It will also add a couple of lines of code to your `SceneDelegate`.
  The first line gets a "window" onto the database that you use to access objects.
  This window is an `NSManagedObjectContext`.
  The second line passes that context into the `@Environment` of your SwiftUI `View`s.
  That's pretty much all the "setup" you need.

Core Data

- The Map

  We're going to demo creating a map, so not much needs to be said here.
  It looks something like this in the built-in editor in Xcode ...

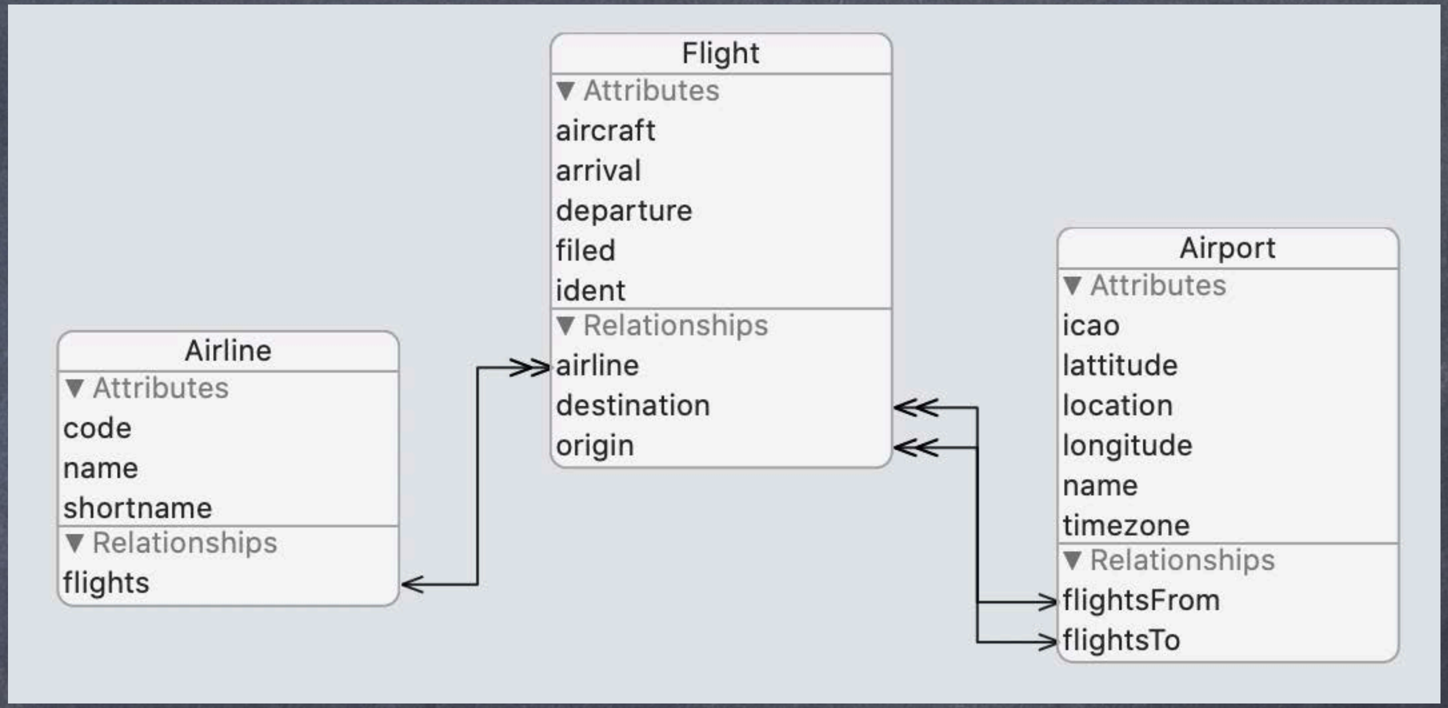


Attributes

Attribute	Type
S aircraft	String
D arrival	Date
D departure	Date
D filed	Date
S ident	String

Relationships

Relationship	Destination	Inverse
O airline	Airline	flights
O destination	Airport	flightsTo
O origin	Airport	flightsFrom



Core Data

- The Code

Again, we'll cover all this in the demo, but here are some highlights …

```
@Environnment(\.managedObjectContext) var context
let flight = Flight(context: context)
flight.aircraft = “B737” // etc.
let ksjc = Airport(context: context)
ksjc.icao = “KSJC” // etc.
flight.origin = ksjc // this would add flight to ksjc.flightsFrom too
try? context.save()
let request = NSFetchRequest<Flight>(entityName: “Flight”)
request.predicate =
    NSPredicate(format: “arrival < %@ and origin = %@“, Date(), ksjc)
request.sortDescriptors = [NSSortDescriptor(key: “ident”, ascending: true)]
let flights = try? context.fetch(request) // past KSJC flights sorted by ident
  // flights is nil if fetch failed, [] if no such flights, otherwise [Flight]
```

Core Data

- SwiftUI

  `@ObservedObject var flight: Flight`

  Now you could put a flight's `ident` on screen using `Text(flight.ident)`.

  These `ObservedObject`s don't seem to automatically `objectWillChange.send()`.

  ```
  @FetchRequest(entity:sortDescriptors:predicate:) var flights: FetchedResults<Flight>
  @FetchRequest(fetchRequest:) var airports: FetchedResults<Airport>
  ```

  `FetchedResults<Flight>` is a `Collection` (not quite an `Array`) of `Flight`/`Airport` objects.

  `flights` and `airports` will <u>continuously</u> update as the database changes.

  ```
  ForEach(flights) { flight in
      // UI for a flight built using flight
  }
  ```

  If a flight is saved to the database that matches the fetch of the `flights` `FetchRequest` above
  then this `ForEach` would immediately create a `View` for it because `flights` would be updated

  You can initialize a `FetchRequest` by doing `_flights = FetchRequest(…)` in an `init`.

Core Data

- Demo

  A demo's worth thousands of words!

  Change Enroute to fetch the FlightAware data into Core Data.

  And then have Enroute's UI get all of its information from Core Data.