IS 543 Fall 2023

Mobile Platform Development

Project 2 Q&A

SwiftData Relationships

MVVM with SwiftData

Dr. Stephen W. Liddle
782 TNRB, office hours Tu 9:30-10:30am, Th 2-3pm
is543@byu.edu
801-422-8792 (rings on my cell)
Zoom: https://bit.ly/liddlezoom

Today

- Project 2 Q&A
- SwiftData

  Relationships:

  - One-to-many with existence dependency
  - Many-to-many without existence dependency (independent)

  How to adopt an MVVM pattern when using SwiftData

Tuesday's Devotional

- Elder Matthew L. Carpenter
  "We Believe in Being Honest"

Feeling Weary?

Wherefore, be not weary in well-doing, for ye are laying the foundation of a great work. And out of small things proceedeth that which is great.

— Doctrine and Covenants 64:33

Project 2 Q&A

- Any questions you'd like to discuss together as a class?
  (If we end early, I'll stick around and answer individual questions too)

SwiftData Relationships

- There are multiple types of relationships in a typical database

One-to-one

- One model, A, references another, B
- Could be optional (or not)
- Could be existence-dependent (or not)
  - If B is existence-dependent on A, if you delete A, you must delete B
  - Likewise, if A is existence-dependent on B, it works in reverse
- Could be symmetric (or not)
  - Symmetric means you can access B from A and A from B; they both have references to each other

One-to-many

- A references many B instances (A has an array of B instances)
- Same optionality, existence dependency, and symmetry issues as above

Many-to-many

- A references many B instances and B references many A instances
- Same optionality, existence dependency, and symmetry issues as above

SwiftData Relationships

- Typical examples

One-to-one

Student may have at most one Loan

(Optional, Loan has existence dependency on Student, symmetric)

One-to-many

Recipe may have many Ingredients

(Optional, Ingredient has existence dependency on Recipe, not symmetric)

Many-to-many

Movie may have many Actors (the Cast) and an Actor may appear in many Movies

(Optional, no existence dependency, symmetric)

SwiftData Relationships

- We use the `@Relationship` macro to declare relationships in SwiftData

```
macro Relationship(
    _ options: Schema.Relationship.Option...,                // Currently only .unique
    deleteRule: Schema.Relationship.DeleteRule = .nullify,   // Default is .nullify
    ...
    inverse: AnyKeyPath? = nil, ...
)
```

There are other parameters, but these are the only three we're interested in

You can indicate that the property must be unique by writing `@Relationship(.unique)`

There are four choices for `deleteRule`:

- `.nullify`: nullify the related model's reference to the deleted model (the default behavior)
- `.cascade`: delete any related models
- `.deny`: prevent deletion if the deleted model references a related model
- `.noAction`: the programmer assumes responsibility for managing references

The `inverse` parameter lets you specify a symmetric relationship

E.g. if `Movie` has an `actors` array, then `Actor` could specify

```
@Relationship(inverse: \Movie.actors) var movies = [Movie]()
```

SwiftData Relationships

- I will show you a working example of one-to-many and many-to-many
  If you haven't already, download the demo project from Learning Suite

MVVM with SwiftData

- Some people say SwiftData kills MVVM
  - From the previous examples we've used, you can see why that is
  - Get the `ModelContext` from the environment
  - Use `@Query` expressions for model access
  - The `ModelContext` acts kind of like the ViewModel because it has the CRUD capabilities
- But we actually still want MVVM
  - `@Query` is fine, but placing business logic in the View is just not a great idea

MVVM with SwiftData

- The key to the pattern is to pass the `ModelContext` to the View
  Then the View can build the ViewModel
  If other Views need access to the ViewModel, we can pass it to them
- We will also need to build queries slightly differently in the ViewModel
- And we will need to work harder to prepare our `#Preview`s too
  Because we need to build a `ModelContainer` and pass the corresponding `ModelContext` into the View being previewed

Let's go look at some code...

Coming Up...

- Just one more week of classes!
- More Project 2 Q&A, plus miscellaneous topics as we have time
  - There is much we haven't talked about yet
  - Let me know if there are topics you're especially interested in