# SwiftData:

SwiftData was announced at WWDC 2023 (June 2023). It is built on top of CoreData. SwiftData makes it easy to persist data using declarative code. It’s designed to integrate seamlessly with SwiftUI.

**Minimum Requirements**:

* Mac OS Sonoma
* Xcode 15
* iOS 17

**PROS:**

* Very simple than CoreData.
* We can perform CRUD operations, with very less number of lines.

**CONS:**

* It’s brand new, not completely mature yet.

# CoreData:

Core Data is an Object-Relational Mapping (ORM) framework. It's designed to manage the model layer objects in your application.

**PROS:**

* The graph part of the framework allows for an easy visualization of the entities.
* Core Data provides built-in support for undo and redo operations, making it easier to implement these features in your application.
* It provides a powerful mechanism for managing relationships between entities in the data model.
* It supports data migration, allowing to evolve the data model over time without losing existing data. Lightweight and Heavyweight(Manual) Migrations.
* Core Data can typically perform an automatic data migration, referred to as lightweight migration.

**Cons:**

* Core Data is less faster than Realm.

# Realm:

Realm is an object-oriented database and relationships are handled through object references rather than explicit foreign key constraints.

**PROS**:

* Performance is faster.
* Realm database files are cross-platform and can be shared among iOS and Android.
* Similar to Swift code, just need to create Class files to store data.

**Cons:**

* Consumes more memory and space as compared to SQLite.
* There are not many features to handle relationships between two different tables.
* No Support for Complex Queries, It provides excellent support for basic queries and filtering, complex queries involving multiple conditions or aggregations might be challenging.
* Realm databases might have a larger file size compared to other database solutions, especially for small datasets. This might be a consideration if your app requires efficient storage usage.
* Realm instances are not thread safe and cannot be shared across threads or dispatch queues. You must construct a new instance for each thread in which a Realm will be accessed.
* Realm write operations are synchronous and blocking, not asynchronous.

# SQLite:

SQLite is a relational database management system.

SQLite is a good choice for applications that need to store and manage large amounts of structured data.

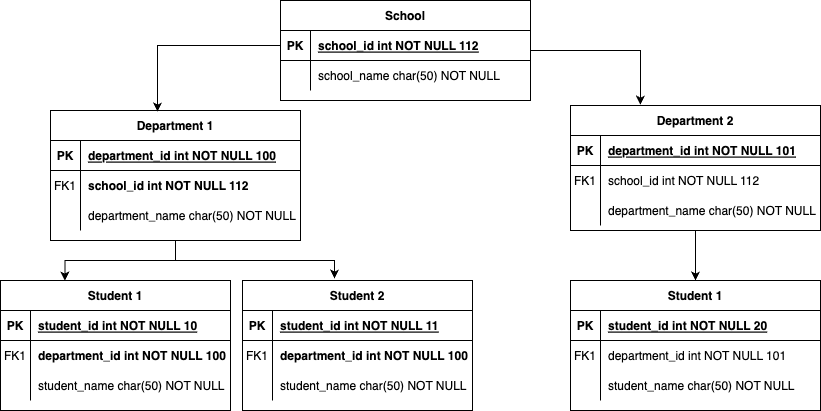
**PROS**:

* Stores data in tables with one or more columns that contain a specific type of data
* Can drop table and edit data without loading them in memory
* Simple and powerful queries can be used to add, delete or manipulate data
* SQLite is a lightweight, embedded database that is easy to integrate into iOS applications.
* SQLite is easy to set up and use, especially for small to medium-sized datasets.

**Cons:**

* Fetching data is a slower process
* You must have to provide relationships, manually
* While suitable for small to medium-sized datasets, SQLite may not scale well for very large datasets or high-traffic applications.
* **Manual Schema Management:** Unlike some other databases, SQLite requires manual management of database schema changes, which can be a consideration for evolving applications.

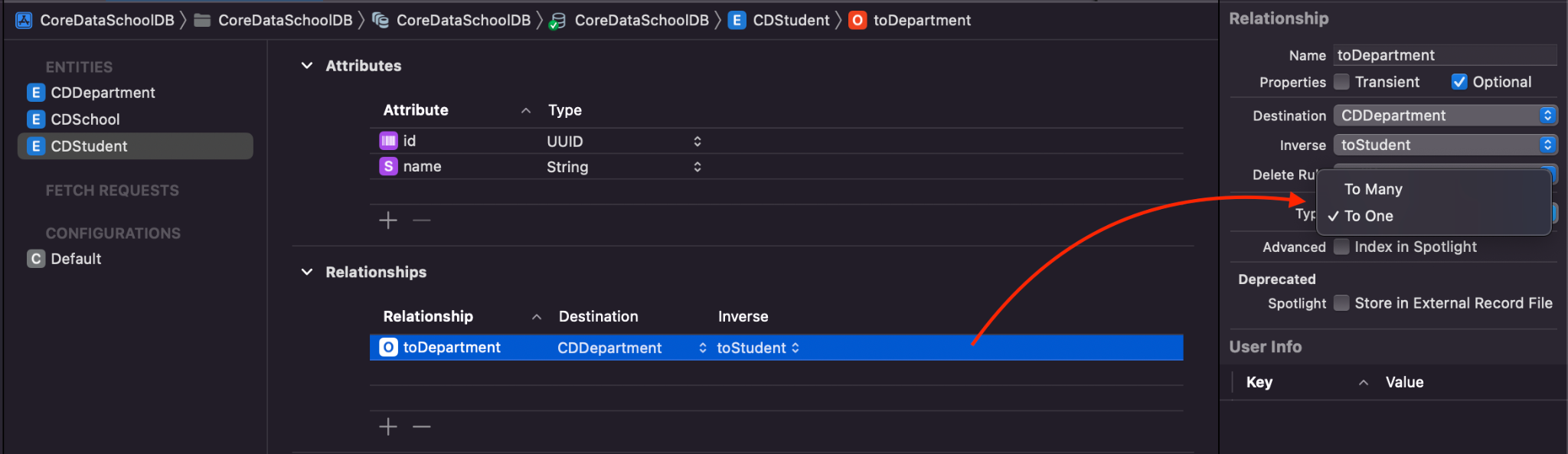
# Example:



## **CoreData:**

**Primary Key and Foreign Key:**

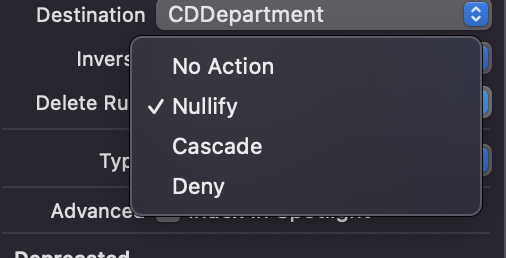
* There is no concept of primary keys or foreign keys in CoreData.
* If we want to establish a relationship between two entities. We have to define a relationship, CoreData takes care of how that relationship is stored.



* By selecting an entity we can provide one to one and one to many relationships as shown above.
* Here **toDepartment** will work as foreign key and id inside CDStudent entity work as Primary key.
* **toStudent** Inverse relationship is a way to establish a bidirectional connection between two entities CDStudent and CDDepartment.
* Inverse relationships help maintain consistency. When we add or remove objects from one end of the relationship, the inverse relationship ensures that the changes are reflected on the other end.

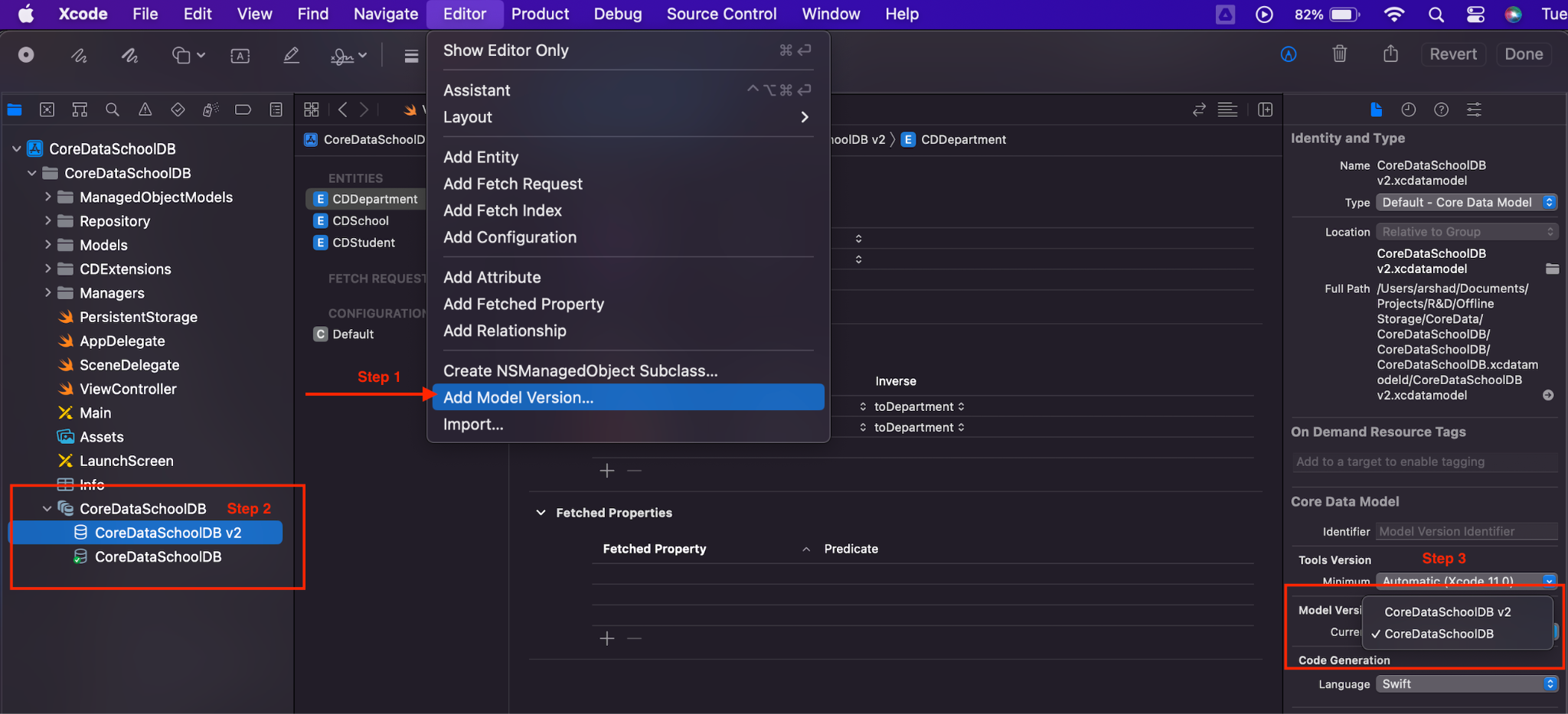
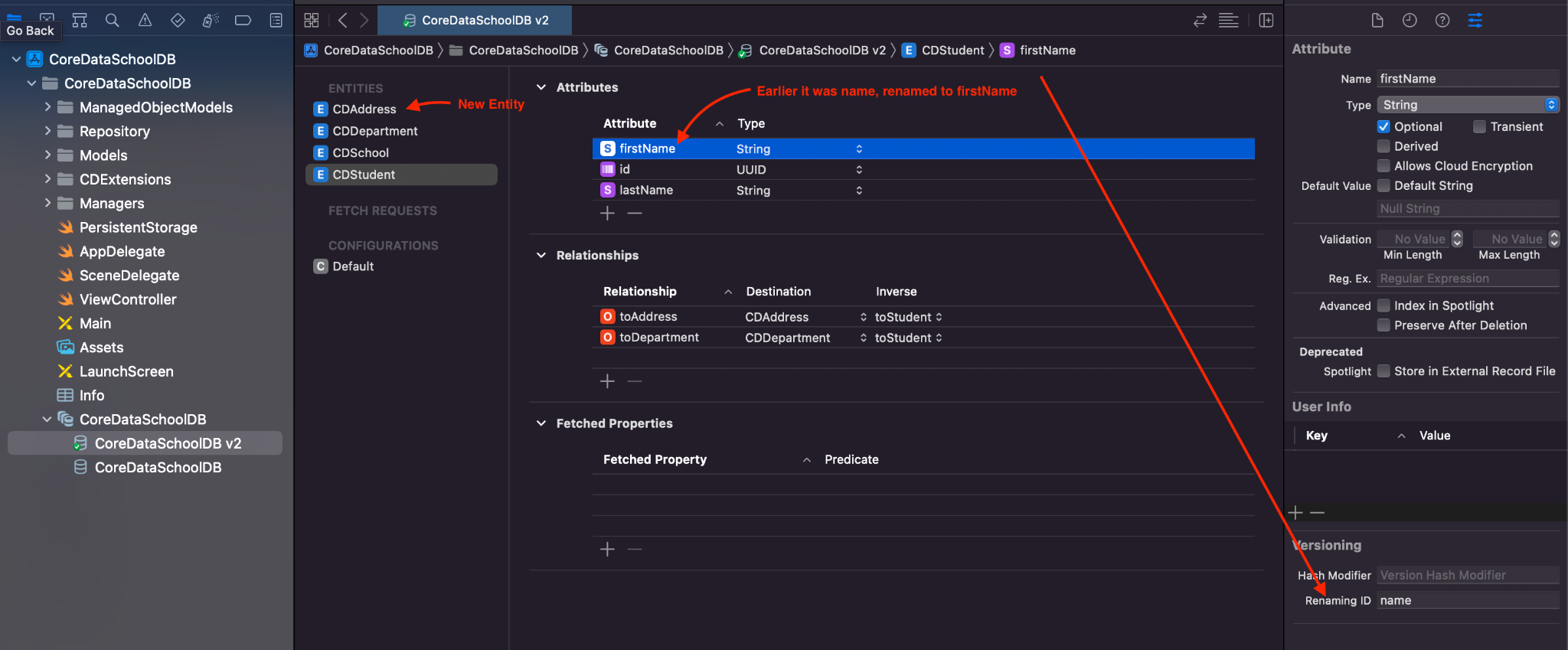
**Delete Rules:**

* It is very easy to manage delete rules in Xcode Editor.
* **No Action:** No action is taken when an object is deleted. It is set by default. Other entities that have relationships will not be notified of this event.
* **Nullify:** When an object is deleted, the relationship to that object in the related objects is set to null or nil.
* **Cascade**: When an object is deleted, all objects related to it are also deleted.
* **Deny**: Prevents the deletion of an object if there are any related objects. The delete operation fails.

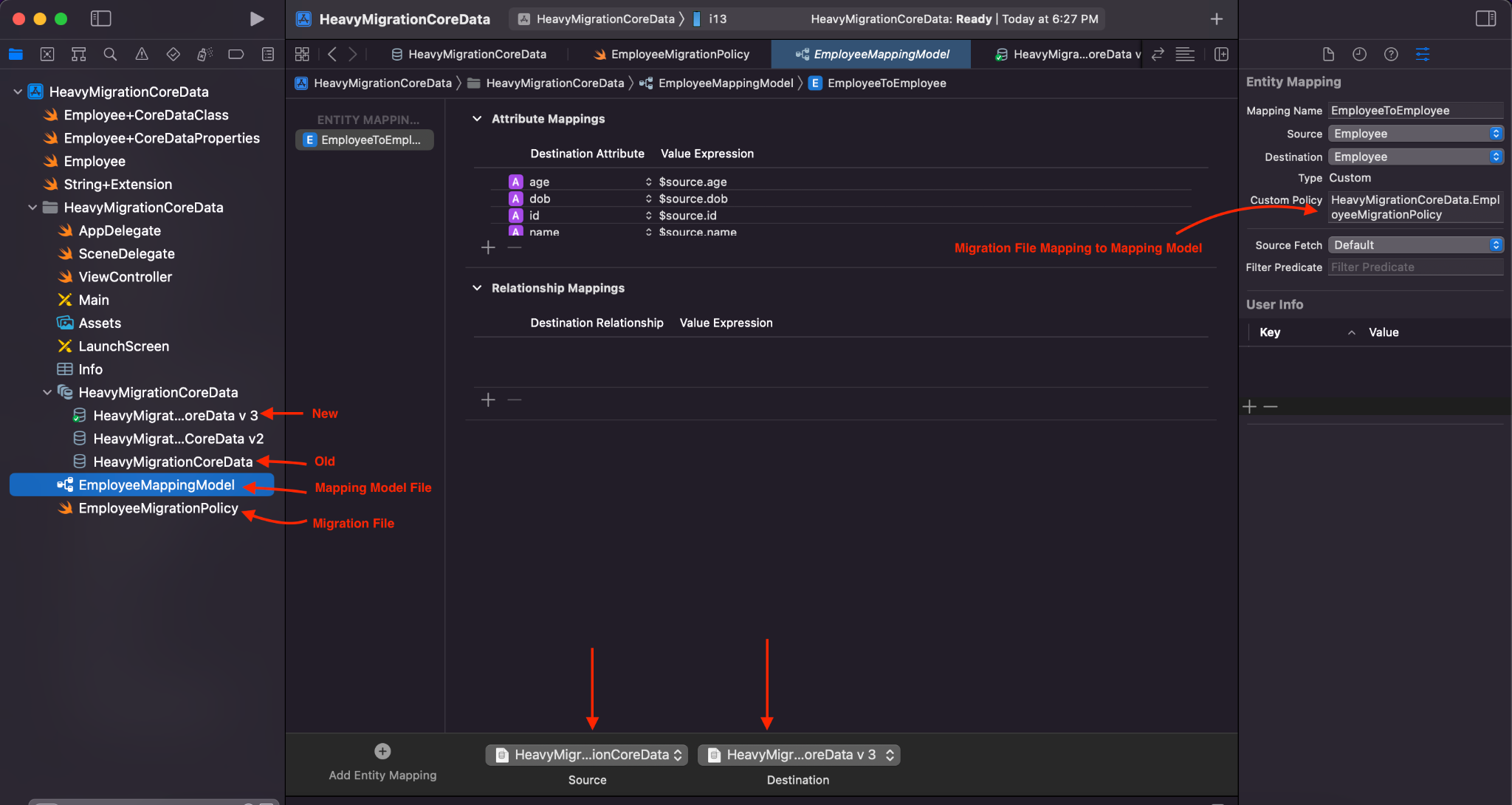
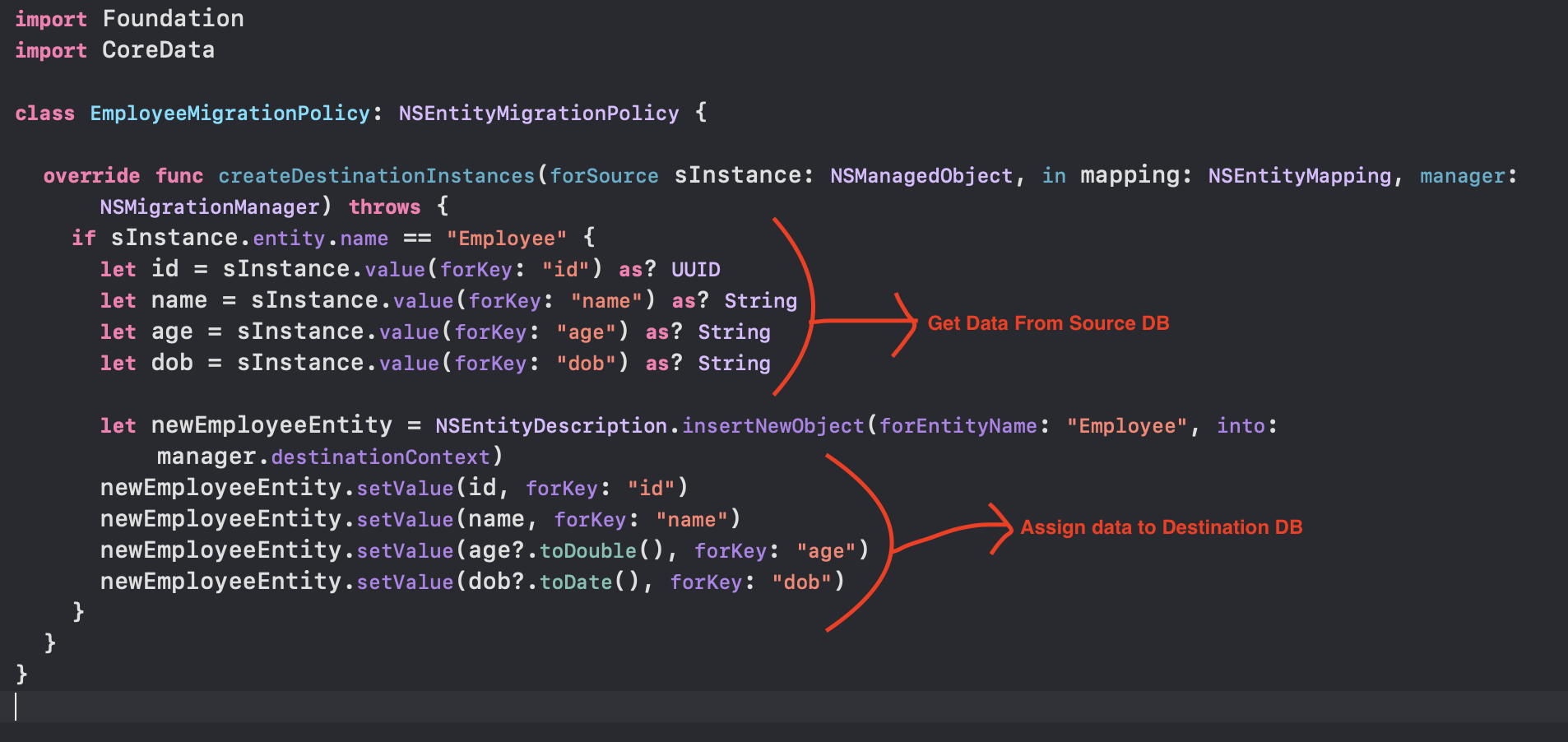


**Migration:**

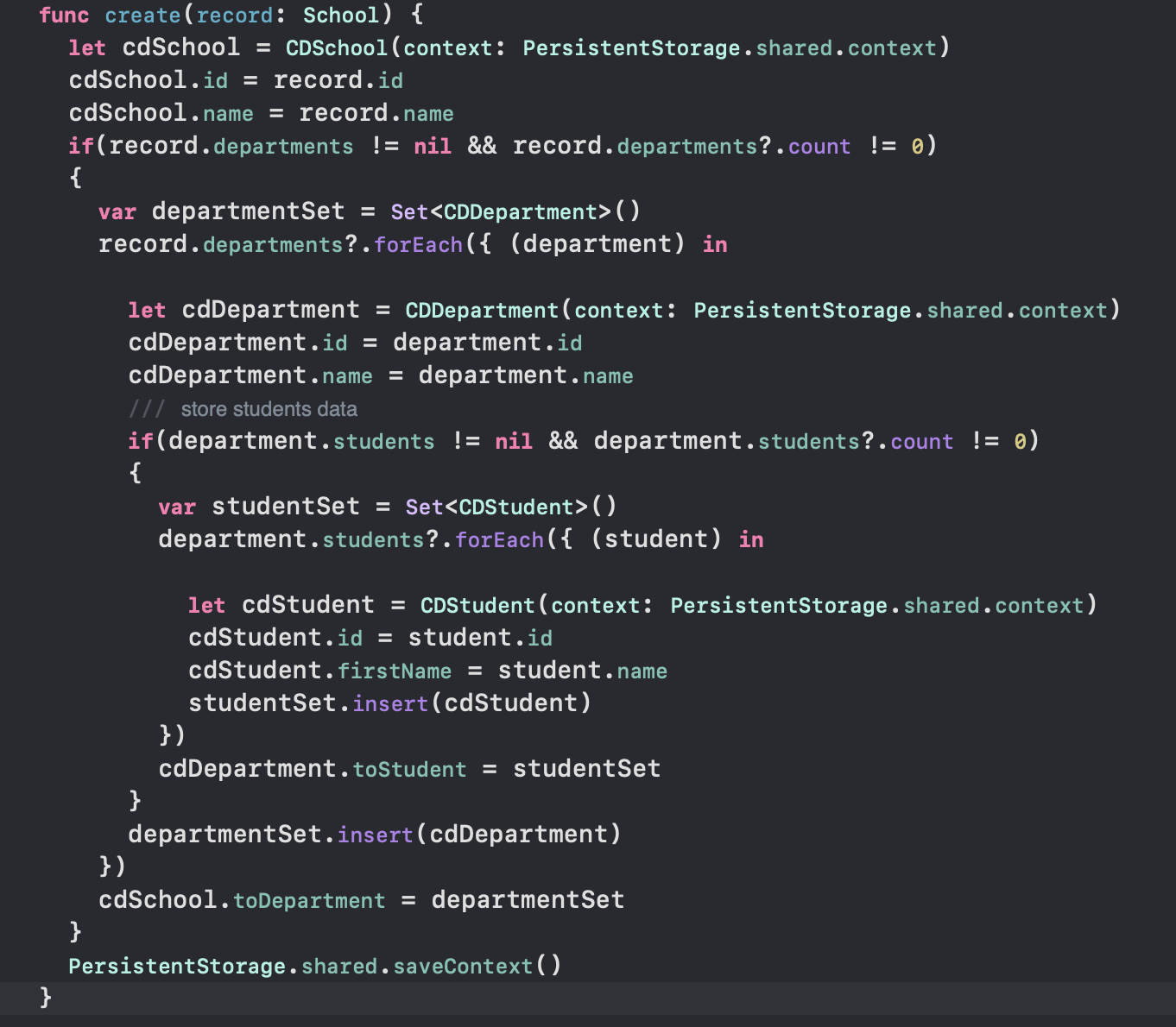
**Lightweight:** Below are the things we can achieve.

* Addition and Removal of an attribute and renaming an entity or property.
* Changing attribute from optional to non optional and non optional to optional and defining a default value.
* We can also manage changes to relationships and to the type of relationship.
* We can add a new relationship or delete an existing relationship.
* We can also rename a relationship by using a renaming identifier, just like an attribute.
* We can change a relationship from a to-one to a to-many, or a non-ordered to-many to an ordered (and vice versa).
* 
* Need to select the appropriate core Data DB from the Model **Version** on the right side.
* 

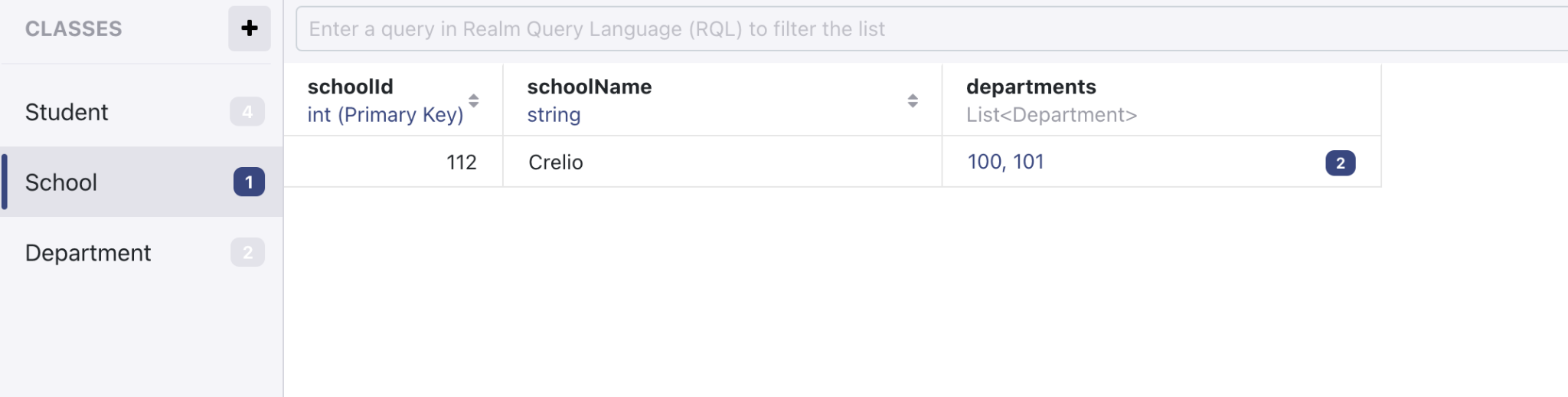
**Heavyweight(Manual):** When to use it.,

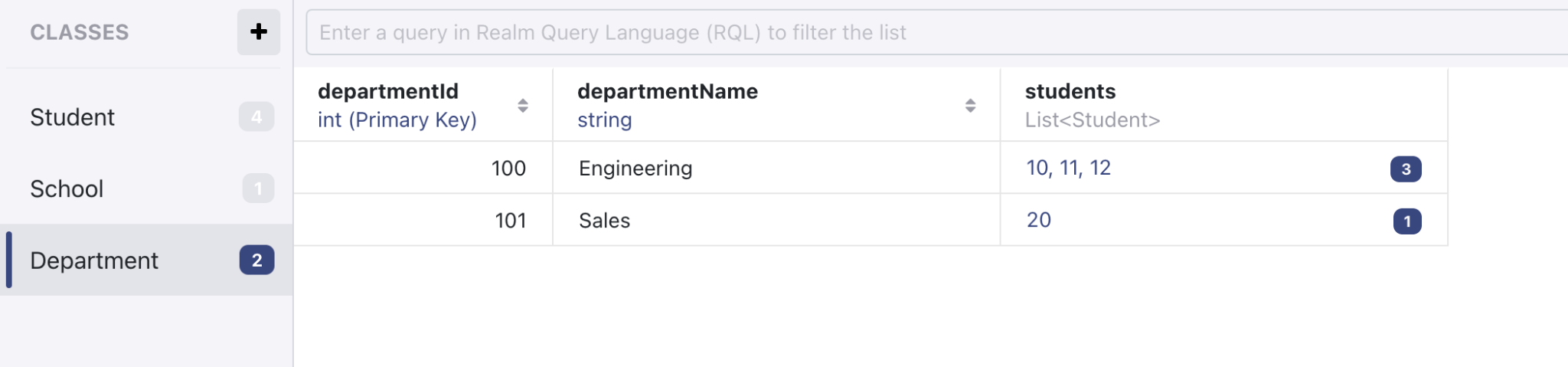
* Lack of database normalization.
* Wrong datatype for an attribute.
* Major changes to the project leading to a remodeling of the database design.
* Creating another core data model steps are the same as Light weight.
* We also need to create new file as New File -> CoreData -> **Mapping Model**
  + Need to select **Source** as old core data model and **Target** as new core data model.
  + Save file MappingModel as filename and also need to create a **Migration Policy** file.
  + 
  + **In the Employee table changing data types of age and dob from String to Double and Date.**
  + ****

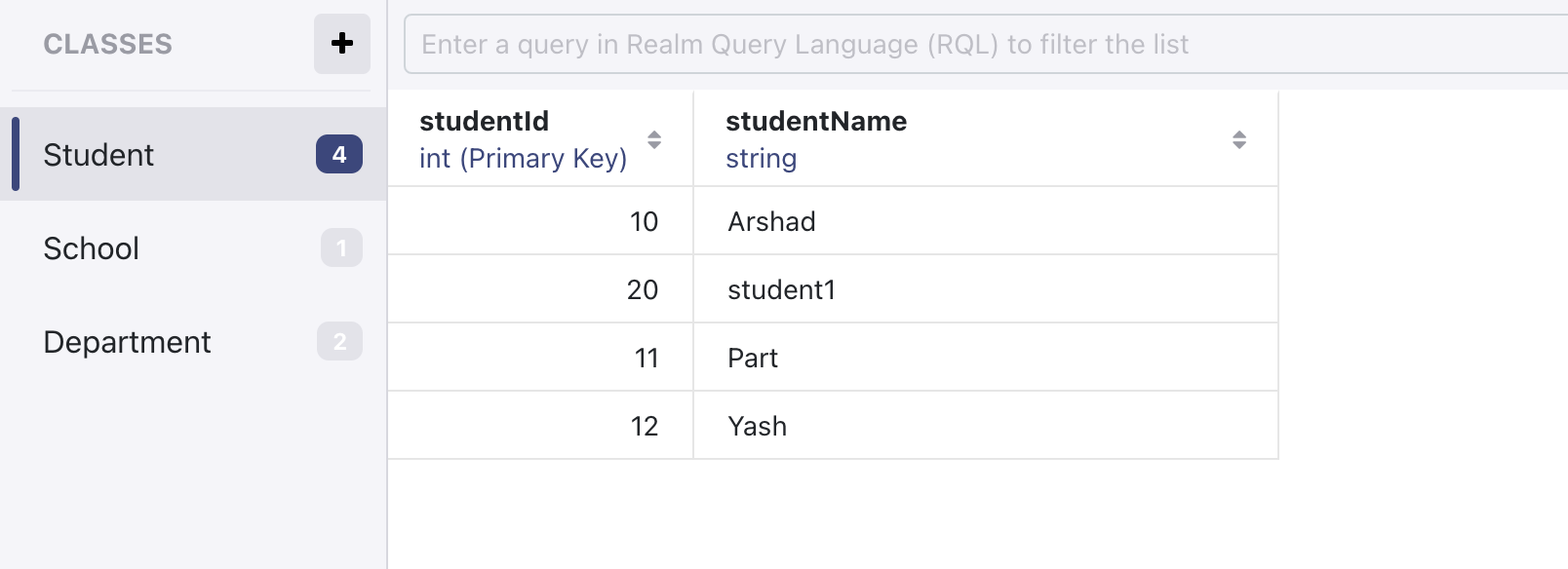
**Save Data:**

****

## **Realm**:



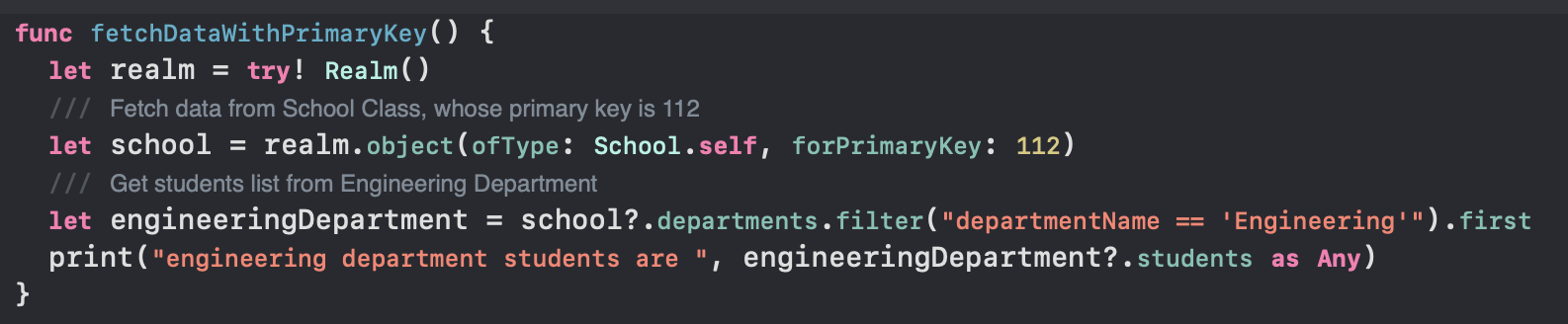




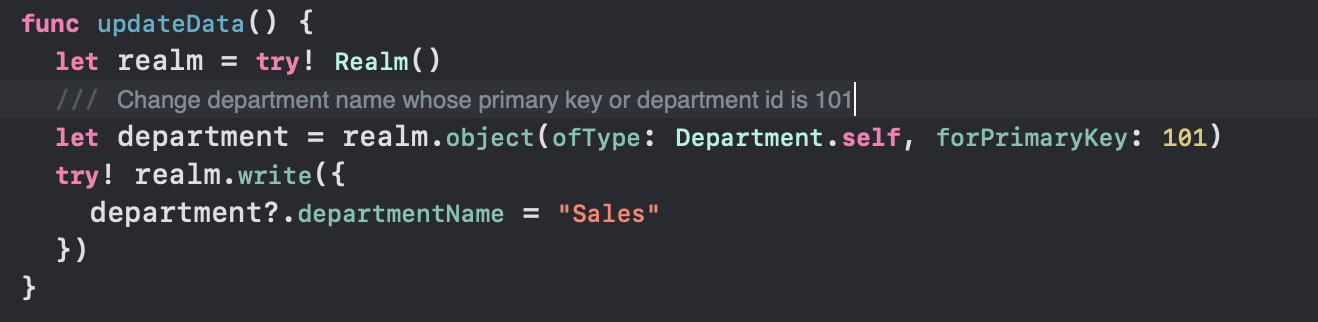
**Save Data to Realm:**



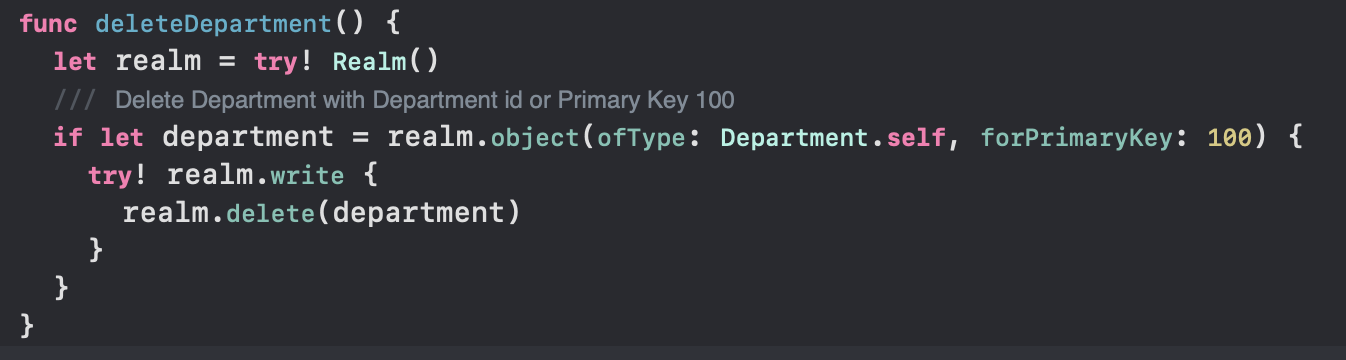
**Fetch Data using Primary Key:**



**Update Data:**

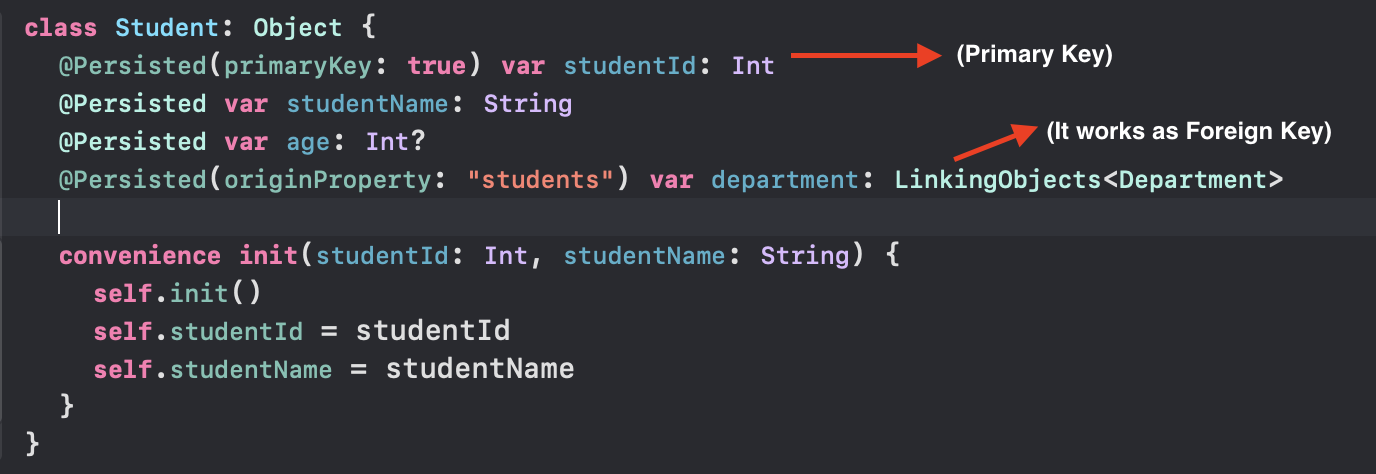
****

**Delete Department:**

****

**Primary Key and Foreign Key:**

* In Realm, primary keys and foreign keys work slightly differently compared to traditional relational databases.
* **Primary Key in Realm:**
* In Realm, each object is required to have a primary key, and the primary key must be of type **String**, **Int**, **Int8**, **Int16**, **Int32**, **Int64**, **ObjectId**, or **UUID**.
* **Foreign Key (Relationship) in Realm:**
* Realm does not use the traditional foreign key concept. Instead, relationships are established using **LinkingObjects** or **List** properties.
* In Realm, relationships are based on object references rather than explicit foreign keys, and there is no need to explicitly define foreign keys as in traditional databases.



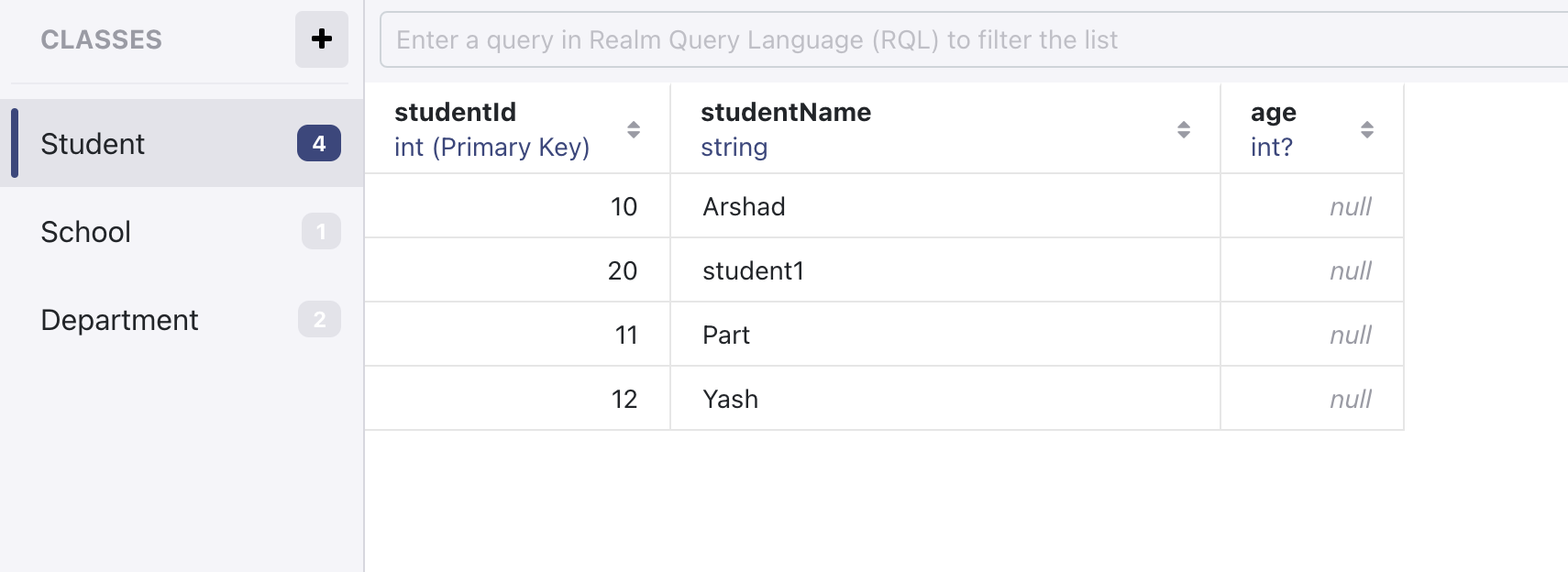
**Schema Migration:**

* By default Schema Version will be 0.
* If we need to add any new field or need to delete existing fields, we need to do schema migration.
* In the Student Table earlier age was not there. Now I added the “age” field as optional.
* Without migration if i try to perform any CRUD operation it will throw error as: **UserInfo={NSLocalizedDescription=Migration is required due to the following errors:**
* **- Property 'Student.age' has been added., Error Code=10}**

**Get Data From Student Table After Scheme Version:**



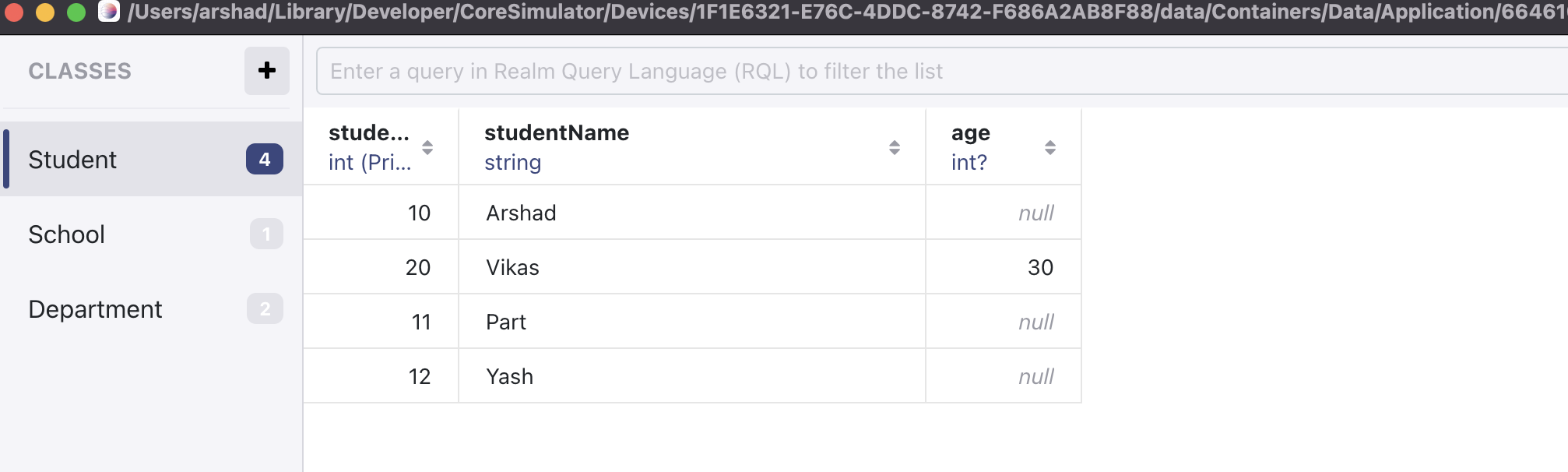
**RESULT:**



**Update Student Age After Scheme Version:**



**RESULT:**

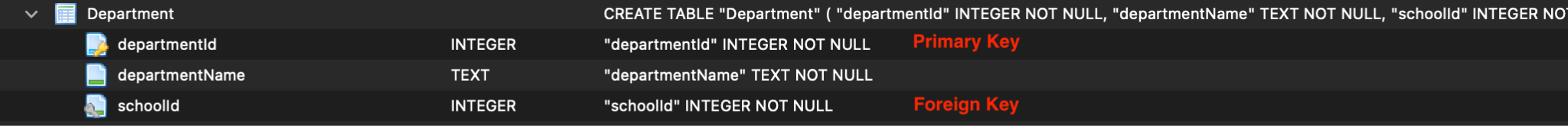


* After the migration is done, we need to provide the same schema version to perform CRUD operations.
* If i try to perform CRUD operation after migration without writing the schema migration code, it will throw error as: **UserInfo={NSLocalizedDescription=Provided schema version 0 is less than last set version 2., Error Code=1}**

## **FMDB:**

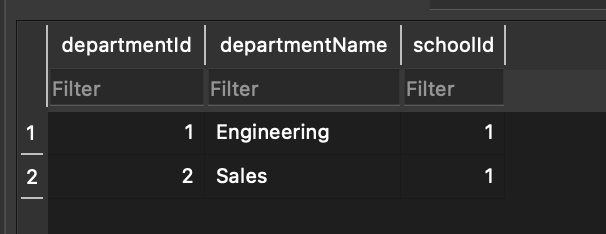
* **It is a wrapper on top of SQLITE.**
* Create a database and need to provide a local path.
* Before performing any CRUD operation, we need to open the database, after completing the operation we need to close the database.

**Create a Department Table as below.**

****

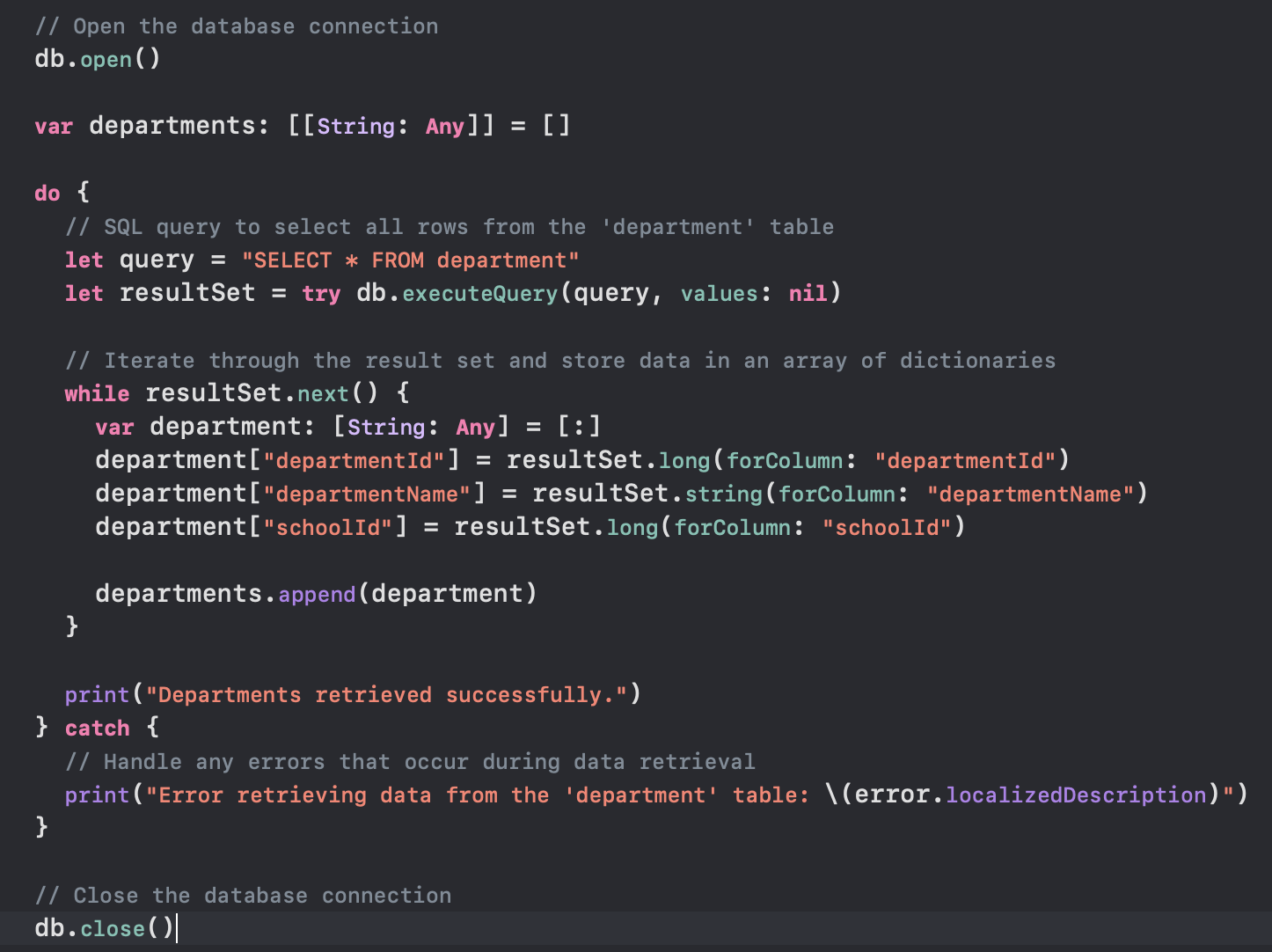


**Save:**

****

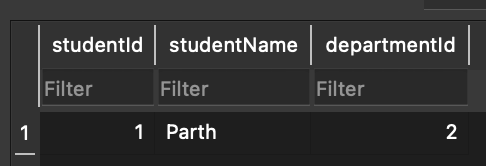


**Fetch:**

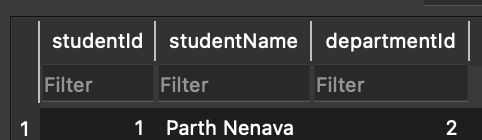


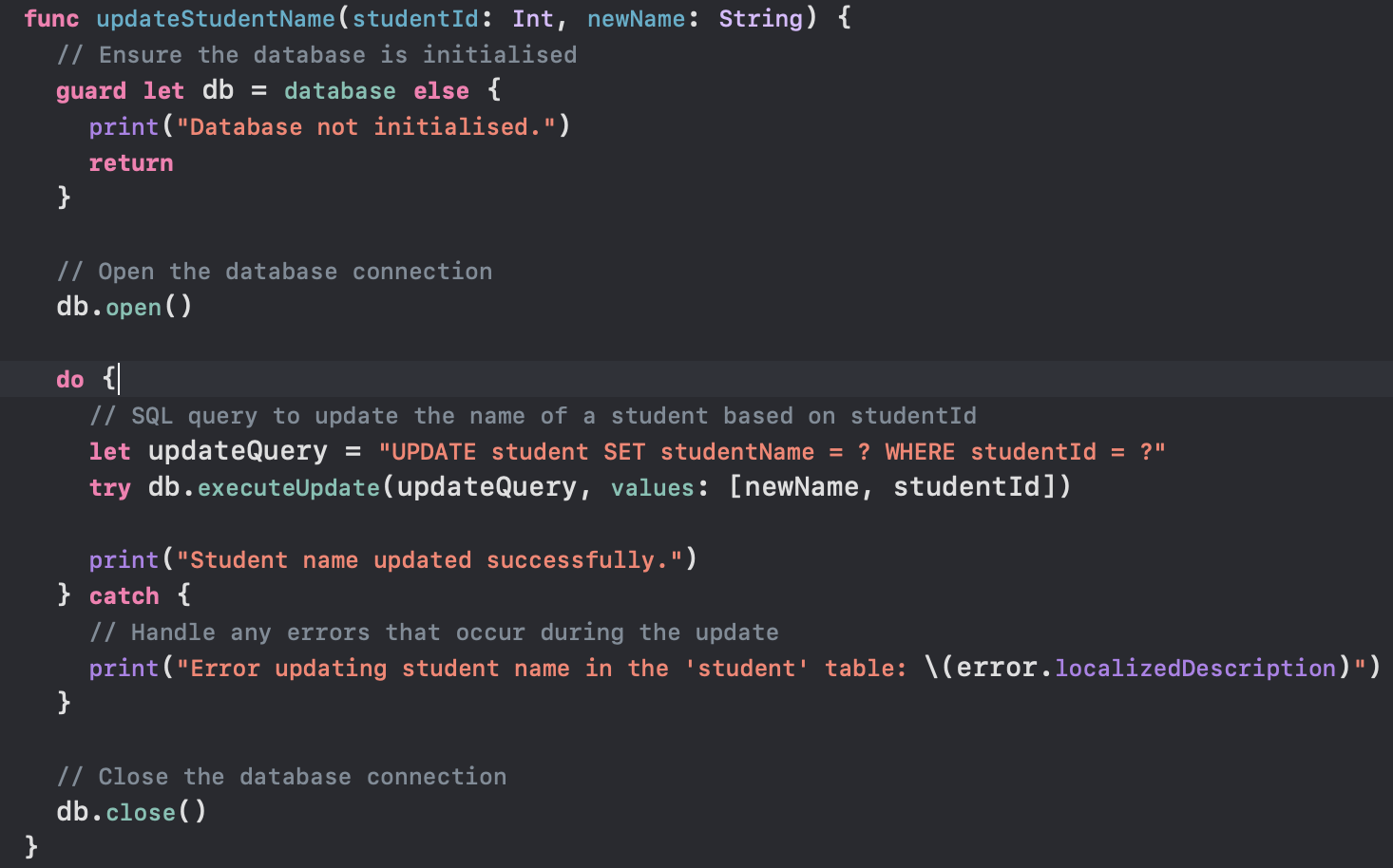
**Update:**

Before Update:

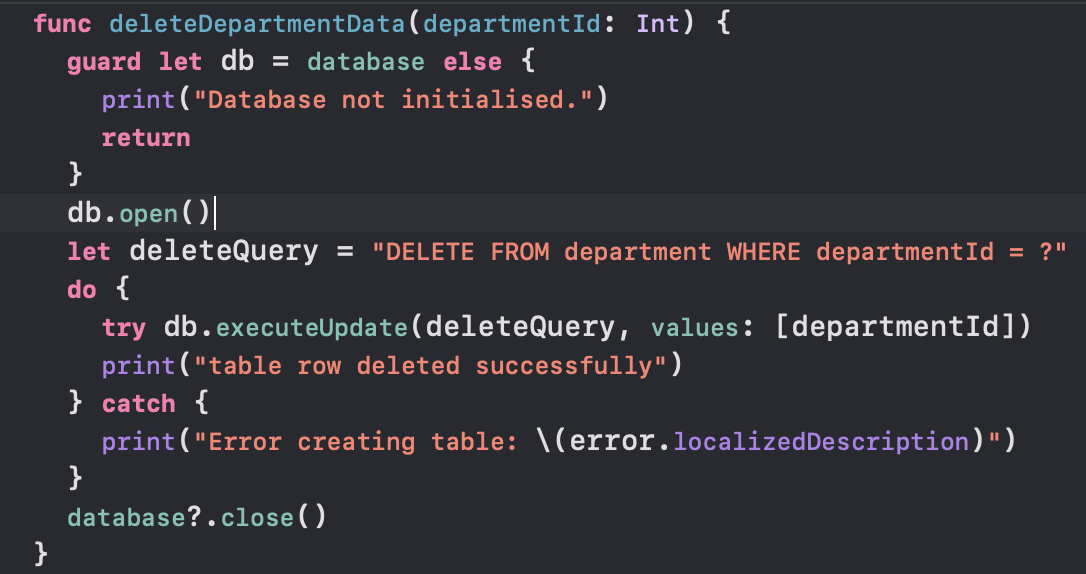


After Update:

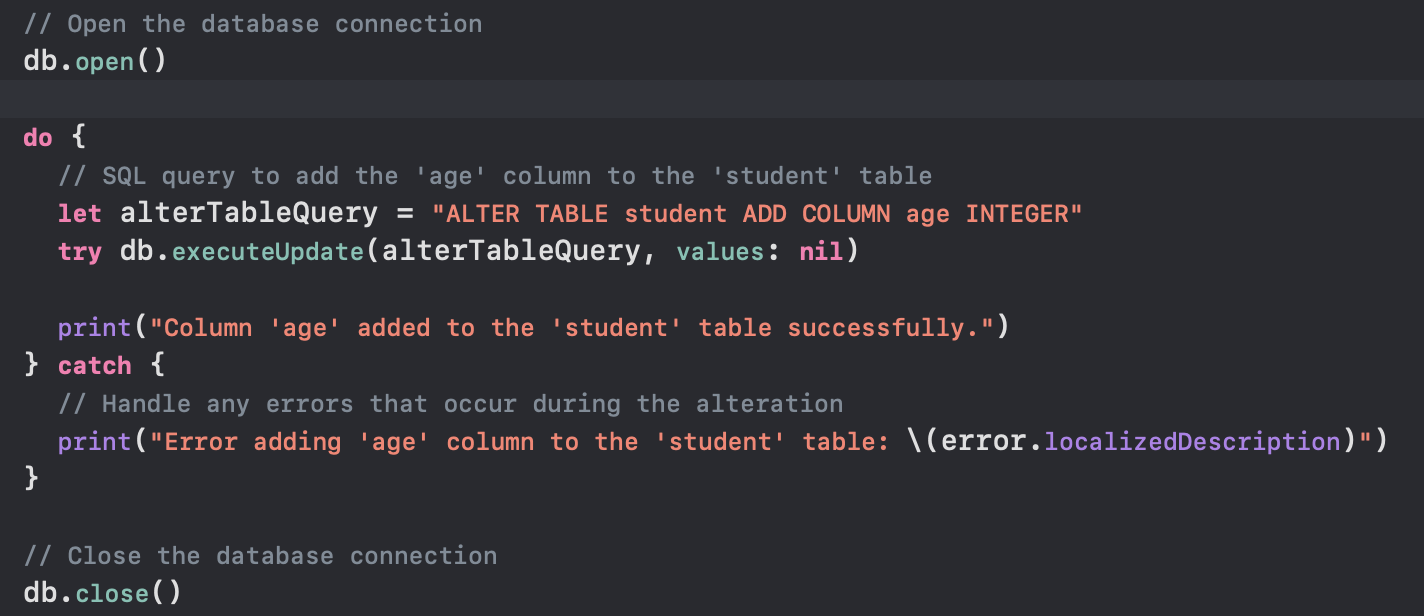




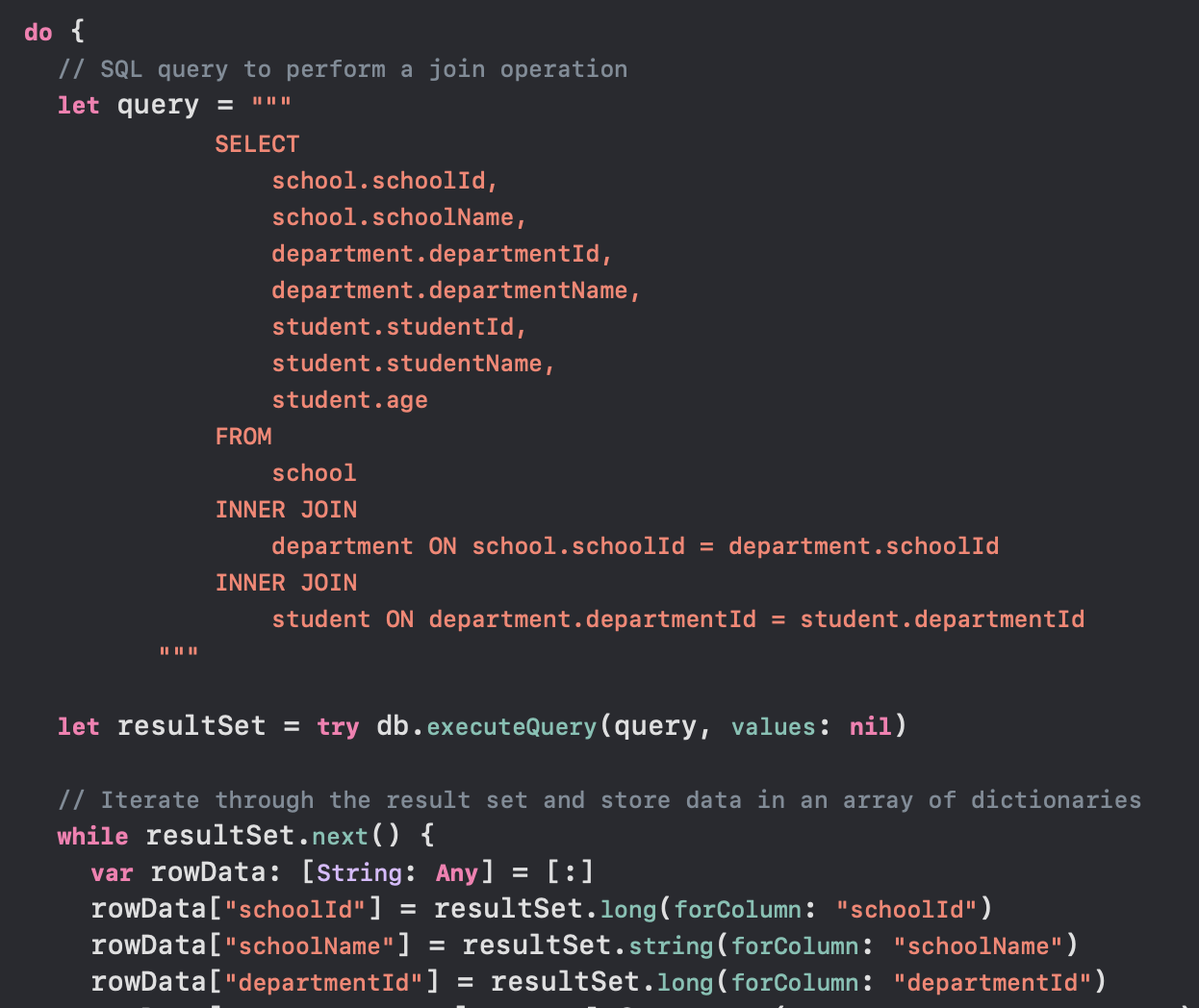
**Delete:**



**Alter:**

****

**Join:**



**Note:**

* Will update the document with code snippets for both Realm and SQLite.