Fetching, sorting, and filtering data

Previously, you learned how to create a SwiftData model context and add objects to it. In this article, you'll learn how Article Accelerator fetches objects from a persistent store. And you'll explore how to sort and filter the objects to list only the objects that you want.

15mins

Estimated Time

> **Project Files**
>
> Be sure to download the project files to follow along with this article.

Fetching data using SwiftData

Article Accelerator displays two sections of articles in `MyStuffView`, one for bookmarked articles and another for completed articles. The My Stuff view gets the bookmarked and completed collections from the detail view. In `DetailView`, the app fetches the entire collection of article state objects before it sorts the articles into those that are bookmarked and those that are completed:

```
1  @Query private var articleStates: [ArticleState]
```

The query macro in the code above directs SwiftData to fetch the entire array of article state objects and to manage them so that the view automatically stays in sync with the underlying data. If properties in the underlying article state data change or if you add or remove an article state, the enclosing view updates accordingly. You don't need to manually refresh fetched objects or track which UI elements require updates when their state changes.

> **Note**
>
> The `Query()` macro uses the same context to perform fetches as was used to insert the model objects. You can access the context using an environment property, `@Environment(\.modelContext)`.

Sorting articles

`TrainingPlanView` displays a list of articles that the user prioritizes in their training plan. Article Accelerator uses SwiftData to manage and persist training plan items.

When a user drags articles into their training plan, the app stores the identifiers associated with those articles. Users can reorder and remove articles from their training plan. Each training plan item has an index that the app uses to manage and persist its position in the plan:

```
@Model
final class TrainingPlanItem {
    let articleID: String
    var index: Int
}
```

When the training plan view fetches the collection of training plan items, the app makes sure to maintain the user's preferred order:

```
 @Query(sort: \TrainingPlanItem.index)
     private var trainingPlanItems: [TrainingPlanItem]
```

The above query directs SwiftData to fetch all the `TrainingPlanItems` and sort them from lowest to highest according to their `index` properties. The query returns the fetched and sorted items in the `trainingPlanItems` array.

Conversely, if you want the items sorted in descending order, you could use the following query:

```
 @Query(sort: \TrainingPlanItem.index, order: .reverse)
    private var trainingPlanItems: [TrainingPlanItem]
```

Examine `TrainingPlanView`, and find the SwiftData query that accesses the set of training plan items. Observe how the `removeTrainingPlanItems(at:)`, `insertTrainingPlanItem(with:at:)`, and `moveTrainingPlanItems(fromOffsets:toOffset:)` methods work with the list to implement the training plan feature in Article Accelerator.

Using sort descriptors

If you want to sort by multiple model properties, define your own sort descriptors and include them in your SwiftData queries:

```
@Query(sort: [SortDescriptor(\Student.age, order: .reverse), SortDescriptor(\Student.name)])
   var students: [Student]
```

This query fetches all the students in a collection and sorts them by age from oldest to youngest. The

This query fetches all the students in a collection and sorts them by age from oldest to youngest. The query sorts groups of students of the same age alphabetically by name.

Filtering articles

The examples above return every model object of a certain type that a query requests. But what if your app needs only objects that satisfy some criteria?

If you want a query that fetches only the article states that are bookmarked and completed, you could use a Swift predicate to filter the fetch:

```
1      @Query(filter: #Predicate<ArticleState> { state in
2          state.isBookmark && state.isComplete
3      }) var completedAndBookmarkedArticles: [ArticleState]
```

The predicate in a query fetches article states that match one or more criteria. The predicate syntax specifies the criteria as a Boolean expression. Think of the predicate closure as an expression that's evaluated for every item in the persistent store. The query returns an item in the collection only if the expression evaluates to `true` for that item. In this example, the filter closure takes an article state and returns `true` only if the state is both bookmarked and completed.

Using dynamic queries

Sometimes the filtering criteria you want depends on a property of the view in which you're performing the query.

Imagine you have a view that displays students older than a certain age, which the user chooses from an age picker included in that view. You'd like the view to update the list of students whenever the user interacts with the age picker to always reflect students who are older than the age that the user chooses. You need a dynamic query that updates its predicate and fetches the data again each time the user changes the age property.

Article Accelerator uses a similar dynamic query. `ArticleView` needs the article state for the article it's displaying in order to list the user-added highlights and bookmarked and completed statuses for that article. It doesn't need all the article states, so the article view uses a predicate to fetch only the article state whose ID matches the article's ID. It's critical that the query updates every time the app displays an article so that the app always shows the user data appropriate for that article.

Here's the code for the dynamic query that fetches the article state matching the current article's ID:

```
1   struct ArticleView: View {
2
3       @Environment(\.modelContext) private var modelContext
4       @Query private var articleStates: [ArticleState]
5
6       let article: Article
7
8       init(article: Article) {
9           self.article = article
10          _articleStates = Query(filter: #Predicate { $0.articleID == article.id } )
11      }
```

```
    var body: some View {
        ... // SwiftUI view that depends on query data.
    }
}
```

To use dynamic queries in your app, initialize a SwiftData query that fetches all the objects of the type that you need. Add an initializer that accepts a property on which the view and the fetch depend. In the initializer, revise the query to fetch only the items that match the predicate rules that your app requires.

In the Article Accelerator code above, the view needs only the article state whose ID matches the article's ID. The predicate ensures only matching articles are returned.

Experiment

Refactor the My Stuff view in Article Accelerator to filter query results for fetching bookmarked and completed articles.

Considering user data

Treat the content your users add to your app with reverence. Ensure that your app beautifully displays and reliably persists their data from one app session to the next.

What's next?

In this article, you explored how Article Accelerator queries the persistent store to access user data and how it filters and sorts data to list only the objects you want, in the order that you want them. And you developed an appreciation for the simplicity of using SwiftData to keep your data—and the views that depend on that data—in sync throughout your app.

In the next series of articles, you'll explore a variety of features and frameworks that make Article Accelerator distinctively designed for and efficient to use in macOS.

Next

Integrating AppKit

The AppKit framework provides a powerful and extensive API for building graphical interfaces for macOS apps. Using AppKit, you'll create amazing apps that tap into the full power of macOS.

Read article