Sample Code

Handling Different Data Types in Core Data

Create, store, and present records for a variety of data types.

Download

iOS 13.2+ | iPadOS 13.2+ | Mac Catalyst 13.0+ | Xcode 11.4+

Overview

Many apps need to persist and present different kinds of information. Core Data provides different attributes, including those common for all databases, such as `Date` or `Decimal` type, and non-standard attributes handled with `Transformable` type. It also provides `Transient` and `Derived` attributes so apps can derive one piece of data from the other.

This sample shows how to handle all these data types, by creating and presenting a set of book records. After you launch it, this sample automatically generates the records if they don't already exist, and shows them in a list. The records are grouped by their published month and can be searched with their canonical form of title.

Derive a Non-Persistent Value Using a Transient Attribute

A `Transient` attribute is derived from one or multiple stored attributes in the same entity. As its name implies, a `Transient` attribute isn't persisted to the store, so an app can use it to provide a new attribute based on the stored ones, without consuming extra storage space.

To make an attribute `Transient`, select the Core Data model in Xcode Project Navigator, navigate to the Core Data entity, select the attribute in the attributes list, and check the `Transient` box in the Data Model Inspector.

In this sample, `publishMonthID` is a `Transient` attribute derived from `publishDate`. To implement the derivation, this sample provides a custom accessor for `publishDate` and `publishMonthID`. The `setter` method of `publishDate` nullifies `primitivePublishMonthID`, which allows the `getter` method of `publishMonthID` to recalculate the value based on the current `publishDate`.

```
@objc public var publishDate: Date? {
    get {
        willAccessValue(forKey: Name.publishDate)
        defer { didAccessValue(forKey: Name.publishDate) }
        return primitivePublishDate
    }
    set {
        willChangeValue(forKey: Name.publishDate)
        defer { didChangeValue(forKey: Name.publishDate) }
        primitivePublishDate = newValue
        primitivePublishMonthID = nil
    }
}
```

The `getter` method of `publishMonthID` recalculates the value if `primitivePublishMonthID` is nil.

```
@objc public var publishMonthID: String? {
    willAccessValue(forKey: Name.publishMonthID)
    defer { didAccessValue(forKey: Name.publishMonthID) }

    guard primitivePublishMonthID == nil, let date = primitivePublishDate else {
        return primitivePublishMonthID
    }
    let calendar = Calendar(identifier: .gregorian)
    let components = calendar.dateComponents([.year, .month], from: date)
    if let year = components.year, let month = components.month {
        primitivePublishMonthID = "\(year * 1000 + month)"
    }
    return primitivePublishMonthID
}
```

With these two methods, `publishMonthID` is associated with `publishDate` and always stays current.

In the case where publishMonthID is key-value observed, the following code ensures that the observations are triggered when publishDate changes.

```
class func keyPathsForValuesAffectingPublishMonthID() -> Set<String> {
    return [Name.publishDate]
}
```

Derive One Value From Another Using a Derived Attribute

This sample uses a Derived attribute, canonicalTitle, to support searching the canonical form of book titles. canonicalTitle is configured as the canonical form of title by setting the following expression as the value of the Derivation field shown Xcode's Data Model Inspector.

```
canonical:(title)
```

Derived attributes are used in cases where performance is more critical than storage space. In this sample, the app gets the same result by setting up a predicate with CONTAINS[cd] (where cd means case- and diacritic-insensitive) to search title directly. By searching canonical Title which is persisted, the app performs more quickly because it doesn't need to do diacritic-insensitive comparison for every book title.

Derived attributes are only updated when the user saves the managed context. Concretely, canonicalTitle won't change if the sample app changes the title attribute without saving it.

Configure and Implement a Non-Standard Data Type

Transformable attributes store objects with a non-standard type, or a type that isn't in the attribute type list in Xcode's Data Model Inspector. To implement a Transformable attribute, configure it by setting its type to Transformable and specifying the transformer and custom class name in Data Model Inspector, then register a transformer with code before an app loads its Core Data stack.

```
// Register the transformer at the very beginning.
// .colorToDataTransformer is a name defined with an NSValueTransformerName extensio
ValueTransformer.setValueTransformer(ColorToDataTransformer(), forName: .colorToData
```

Core Data requires the transformer be NSSecureUnarchiveFromData or its subclass, and that its transformedValue(_:) method converts a Data object to an instance of the custom class specified in Data Model Inspector and that reverseTransformedValue(_:) does the opposite – converts an instance of the custom class to a Data object.

Store and Present a Date Type

In a Core Data store, a `Date` attribute is a `double` value that represents a number of seconds since 1970. Using a variety of calendars, time zones, and locales, an app can convert a `Date` value to different date strings, or convert a date string to different `Date` values. When parsing a date string, configure the `DateFormatter` with the right calendar, time zone, and locale. Typically, if the string is generated by the current user, the user-perceived calendar, time zone, and locale will be the current system ones, so an app can use a default `DateFormatter` instance. In other cases, configure `DateFormatter` in the same way the app generated the string.

Store and Present a Decimal Type

This sample uses a `Decimal` attribute to represent the book price, which is then mapped to a variable of `NSDecimalNumber` type. `NSDecimalNumber` has a convenient method to process a currency value.

```
newBook.price = NSDecimalNumber(mantissa: value, exponent: -2, isNegative: false)
```

`NSDecimalNumber` also provides a convenient way to present a value with locale in mind.

```
cell.price.text = book.price?.description(withLocale: Locale.current)
```

See Also

Essentials

- Creating a Core Data model
  Define your app's object structure with a data model file.
- Setting up a Core Data stack
  Set up the classes that manage and persist your app's objects.
- Core Data stack
  Manage and persist your app's model layer.
- Linking Data Between Two Core Data Stores
  Organize data in two different stores and implement a link between them.