Extracted from:

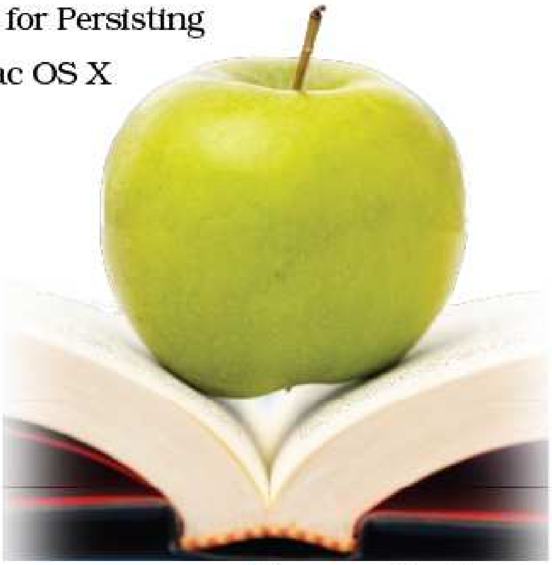
Core Data

Apple's API for Persisting Data on Mac OS X

The Pragmatic Programmers

Core Data

Apple's API for Persisting Data on Mac OS X



Marcus S. Zarra

Edited by Daniel H Steinberg

Printed in the United States of America.

ISBN-10: 1-934356-32-8

ISBN-13: 978-1-934356-32-6

Printed on acid-free paper.

B6.0 printing, July 29, 2009

Version: 2009-8-25

2.3 Advanced Readers

If you are already a bit familiar with Core Data and building a Core Data application, please feel free to move quickly through the rest of this chapter. In this chapter, we will walk through the construction of our project and how to build its data model. The end result will be a data model like the one shown in Figure 2.4, on page 23.

2.4 Creating Our Xcode Project

The first step is to create our Xcode project. With the recent versions of Xcode, quite a few project templates are available to us, and more than one of those is based on Core Data. If you are using Leopard, then you will want to use the Core Data Application template, and if you are using Snow Leopard, then you want to select the Cocoa Application template and ensure that the "Use Core Data for storage" checkbox is selected. Once we select which template, we will name the project Grokking Recipes, which will also be the name of our application.[1]

The basic Core Data template gives us an application that works somewhat like Address Book. In Address Book, the user has only one data file and generally accesses that data file via one window. Our recipes application will be designed around that same pattern. We will have exactly one data file that all the user's recipes will be stored in.

Once the project is created in Xcode, it is time to start building the Core Data aspects of our application.

2.5 Building the Data Model

Core Data applications are like database-driven applications, and in that light, we will start with building the data structures first and then move to the user interface. The three components we are going to be working with at this point are entities, attributes, and relationships.

In our design (see Section 2.2, *Our Application Design*, on the preceding page), we already described at least two of the data objects that we want to use and at least some of the attributes. Therefore, we will start with them. In our Xcode project, there is a group called Models, and

1. Although it can be changed later.

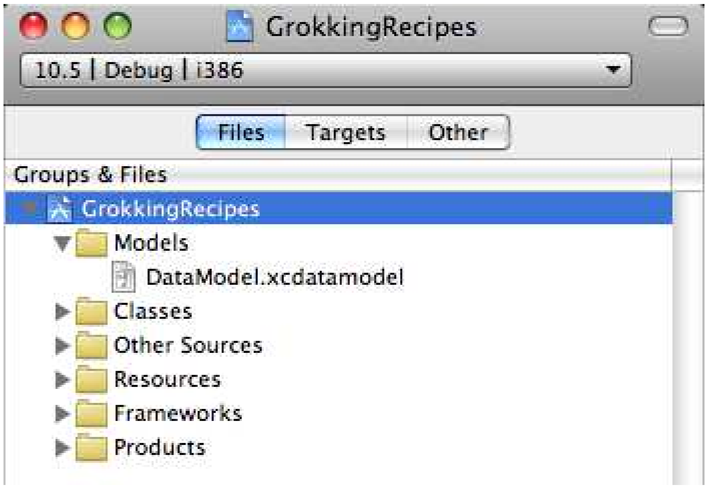


Figure 2.2: The data model in Xcode

within that group is a file called DataModel.xcdatamodel.[2] This file is a representation of the data structure that Core Data will use. This file has a lot of similarities to an entity-relationship diagram (ERD) except that Xcode will compile it directly into our final data file structure.

Adding an Entity to the Model

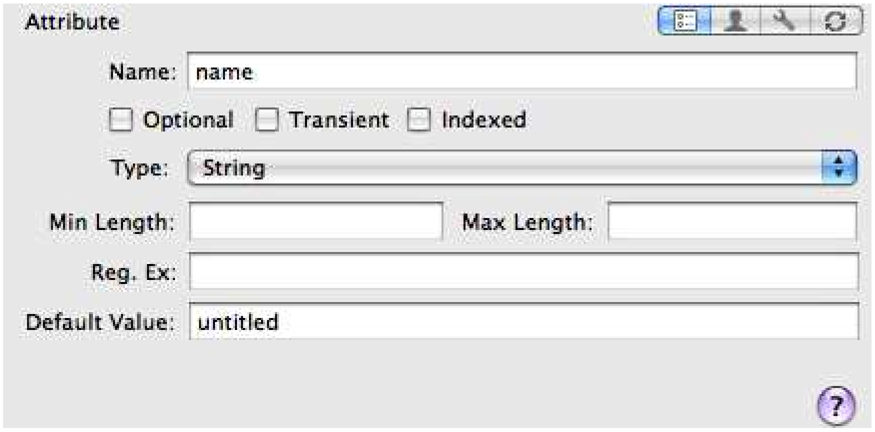
In Core Data, an entity has a lot of similarities to a table in a normal database design. Although this similarity is not exact, it is a good place to start.

To add our first entity to our data model, first open the .xcdatamodel file in the Models group, and then choose Design > Data Model > Add Entity from the menu bar (or use the + button in the entry area in the top left). This will add a blank entity to our data model. Next, double-click the name of the entity, and rename it to Recipe.

Adding an Attribute Property

Just as an entity has a lot of similarities to a table, a property has quite a few similarities to a column in that table. This similarity breaks down very quickly when we start adding relationships, but it helps in the

2. It is possible this file will be named ${PROJECT_NAME}_DataModel.xcdatamodel depending on the version of Xcode you are using.



Attribute

Name: name

☐ Optional ☐ Transient ☐ Indexed

Type: String

Min Length: Max Length:

Reg. Ex:

Default Value: untitled

Figure 2.3: Attribute details

beginning to think of it this way. Properties are further broken down into several subtypes; the two most relevant are attributes and relationships. Relationship properties describe the relationships between two entities, and attribute properties are the values of an entity.

To add our first attribute property to our recipe entity, select the entity, and choose Design > Data Model > Add Attribute from the menu bar; you can also use the top + button or the keyboard shortcut. Like the entity creation, this will create a new attribute property within our entity. Double-click the name of this property, and rename it to name. After renaming the attribute, select it to see its details, as shown in Figure 2.3.

These details allow us to control several aspects of the attribute including the default value, what type of attribute it is, and whether it is transient, indexed, optional, and so on. We will go through all of these settings in greater detail later, so for now set the type to String, set the default value to untitled, and make sure it is not optional.

Once the first attribute is finished, add the following attributes to the Recipe object:

- Set imagePath to an optional String without a default value.
- Set desc to an optional String without a default value.

- Set serves to an Integer 16 with a minimum value of 1 and a default value of 1. Be sure to flag it as nonoptional.
- Set type to an optional String with a default value of Meat.

Creating Our Second Entity

With the Recipe entity nearly complete, it is time to create our second entity. This second entity will store the ingredients that go into a recipe, and we will call it RecipeIngredient. Following the same steps, we can add these attributes:

- Set name to a nonoptional String with a default value of untitled.
- Set quantity to a nonoptional Integer 16 with a minimum value of 0 and a default value of 1.
- Set unitOfMeasure to a nonoptional String with a default value of untitled.

Adding a Relationship Property

Relationship properties are created in the same way as attribute properties, although the specifics naturally differ. Add a relationship to the Recipe entity by selecting Design > Data Model > Add Relationship from the menu bar. For this first relationship, name it ingredients, and flag it as optional.

Where a relationship is different from an attribute, however, is in the properties. Instead of defining an object type, default values, and so on, we are instead defining a destination entity, an inverse relationship, and whether this relationship is “to-many.” For this relationship, we will start by naming it ingredients, and then we set the destination entity to RecipeIngredient, but we are not going to set the inverse relationship yet. We are also going to flag it as to-many, since a recipe can definitely have more than one ingredient.

The last option, the delete rule, instructs Core Data on how to handle the relationship when this, the Recipe entity, is deleted. In this relationship, we will delete the RecipeIngredient object to avoid any disconnected objects. Therefore, we will select the cascade option, which will remove any associated RecipeIngredient objects when the Recipe entity is deleted.

Joe Asks...

What Is One-to-Many?

One-to-many is a database term that describes the relationship between two tables in the database. Normally, there are three kinds of relationships: one-to-one, one-to-many, and many-to-many. A *one-to-one* relationship means that for each record in the first table there can be no more than one record in the second table. In a *one-to-many* relationship, for each record in the first table, there can be more than one record in the second table. The last relationship type, *many-to-many*, means that for any record in the first table, there can be any number of records in the second table, and, likewise, for each record in the second table, there can be any number of records in the first table.

Completing the Relationship

One rule that is often repeated by the developers of Core Data is that every relationship in your database should have an inverse. Although this may not make logical sense for the data, it is important for data integrity within Core Data. What this means from our programming perspective is that we need to be able to reference each object in the relationship from either side. Apple recommends this inverse relationship for many reasons, which will be discussed in greater detail throughout this book.

To set up the inverse relationship, we select the RecipeIngredient entity and add a Relationship property to it just like we did in the Recipe entity earlier. This new Relationship property is named recipe with a destination of the Recipe entity. Next, we set the inverse relationship to be ingredients, which was the name of the relationship we set in the Recipe entity. As soon as we set the inverse relationship on the RecipeIngredient, the graphical view of the relationships will change. Instead of two lines connecting the objects, they are replaced with one line, making the graphical view quite useful for debugging relationship settings. In our current design, an ingredient can have only one recipe; therefore, we leave the to-many option unselected. Lastly, we set the Delete Rule setting to Nullify. This setting will not delete the Recipe entity when a RecipeIngredient object is deleted. Instead, it will just break the connec-

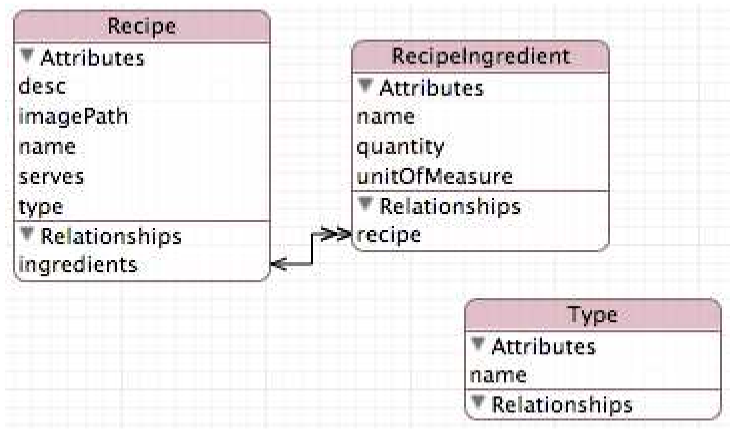


Figure 2.4: The managed object model (MOM)

tion between the two.

Adding the Last Entity

We have one more entity to add in this release of our recipe application. We will be categorizing the recipes that are added. For example, we will be separating desserts from appetizers, and so on. To keep these categories consistent, we store the actual category names in their own object. Therefore, add one more entity to our model called Type. This entity has only one attribute property, called name, which is a nonoptional string with no default value. Lastly, this entity has no relationships because it will be used only as a lookup to populate the type NSComboBox discussed in Section 2.7, *Adding the Recipe Details*, on page 29.

And with that last entity, that concludes the construction of the data model for our application. The final result should look similar to Figure 2.4.

Build the Data Objects

In other languages, or even in Cocoa applications that do not use Core Data, the next step would normally be to build the data objects that are associated with the "tables" in the "database." Fortunately, we are working with Core Data, and there are no data objects to construct. As part of Core Data, defining the data model also defines the base