

# Define Your Data Model

In this lesson, you'll define and test a [data model](#) for the FoodTracker app. A data model represents the structure of information in an app.

## Learning Objectives

At the end of the lesson, you'll be able to:

- Create a data model
- Write failable initializers on a custom class
- Demonstrate a conceptual understanding of the difference between failable and nonfailable initializers
- Test a data model by writing and running unit tests

## Create a Data Model

Now you'll create a data model to store the information that the meal [scene](#) needs to display. To do so, you define a simple [class](#) with a name, a photo, and a rating.

### To create a new data model class

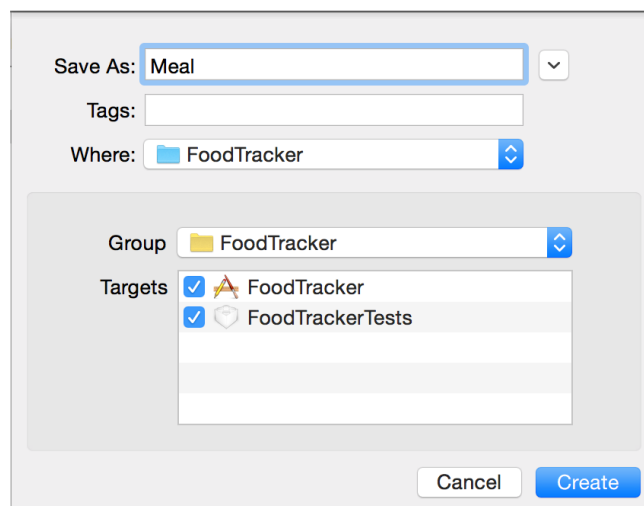
1. Choose File > New > File (or press Command-N).
2. On the left of the dialog that appears, select Source under iOS.
3. Select Swift File, and click Next.

You're using a different process to create this class than the `RatingControl` class you created earlier (iOS > Source > Cocoa Touch Class), because you're defining a [base class](#) for your data model, which means it doesn't need to inherit from any other classes.

4. In the Save As field, type Meal.
5. The save location defaults to your project directory.

The Group option defaults to your app name, FoodTracker.

In the Targets section, make sure your app and the tests for your app are both selected.



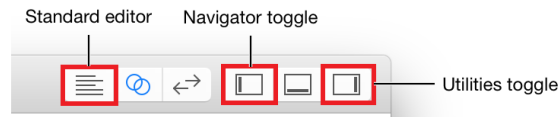
6. Click Create.

Xcode creates a file called `Meal.swift`.

In Swift, you can represent the name using a `String`, the photo using a `UIImage`, and the rating using an `Int`. Because a meal will always have a name and rating, but might not have a photo, you can make the `UIImage` an [optional](#).

## To define a data model for a meal

1. If the [assistant editor](#) is open, return to the standard editor by clicking the Standard button.



2. Open `Meal.swift`.
3. Change the import statement to import `UIKit` instead of `Foundation`:

```
import UIKit
```

By default, a Swift file imports the Foundation framework so you can work with Foundation data structures in your code. You'll be working with a class from the UIKit framework, so you need to include UIKit in your import statement. Importing UIKit also gets you access to Foundation, so you can remove the redundant import to Foundation.

4. Below the import statement, add the following code:

```
1 class Meal {
2     // MARK: Properties
3
4     var name: String
5     var photo: UIImage?
6     var rating: Int
7 }
```

This code defines the basic properties for the data you need to store. You're making these variables (`var`) instead of constants (`let`) because they'll need to change throughout the course of a `Meal` object's lifetime.

5. Below the properties, add this code to declare an initializer:

```
1 // MARK: Initialization
2
3 init(name: String, photo: UIImage?, rating: Int) {
4 }
```

Recall that an [initializer](#) is a method that prepares an instance of a class for use, which involves setting an initial value for each [property](#) and performing any other setup or initialization.

6. Fill out the basic implementation by setting the properties equal to the parameter values.

```
1 // Initialize stored properties.
2 self.name = name
3 self.photo = photo
4 self.rating = rating
```

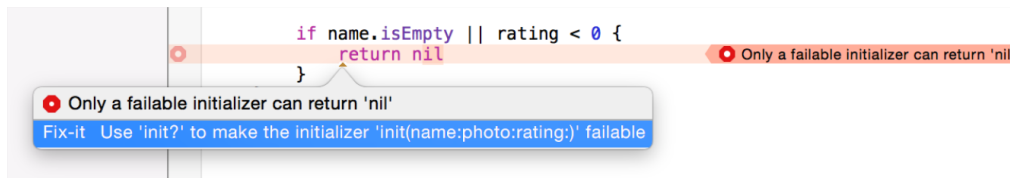
But what happens if you try to create a `Meal` with incorrect values, like an empty name or a negative rating? You'll need to return `nil` to indicate that the item couldn't be created, and has set to the default values. You need to add code to check for those cases and to return `nil` if they fail.

7. At the end of the initializer, add this `if` statement to check for invalid values and return `nil` if one of them is found.

```
1 // Initialization should fail if there is no name or if the rating is
   negative.
2 if name.isEmpty || rating < 0 {
3     return nil
4 }
```

Because the initializer now might return `nil`, you need to indicate this in the initializer signature.

- Click the error [fix-it](#) to add a question mark (?) to the end of the `init` keyword in the initializer.



```
init?(name: String, photo: UIImage?, rating: Int) {
```

An initializer written like this is known as a [failable initializer](#), which means that it's possible for the initializer to return `nil` after initialization.

At this point, your `init?(name:photo:rating:)` initializer should look something like the following:

```
1  // MARK: Initialization
2
3  init?(name: String, photo: UIImage?, rating: Int) {
4      // Initialize stored properties.
5      self.name = name
6      self.photo = photo
7      self.rating = rating
8
9      // Initialization should fail if there is no name or if the rating is negative.
10     if name.isEmpty || rating < 0 {
11         return nil
12     }
13 }
```

*Checkpoint:* Build your project by choosing **Product > Build** (or pressing **Command-B**). You're not using your new class for anything yet, but building it gives the compiler a chance to verify that you haven't made any typing mistakes, like forgetting the question marks (?). If you have, fix them by reading through the warnings or errors that the compiler provides, and then look back over the instructions in this lesson to make sure everything looks the way it's described here.

## Test Your Data

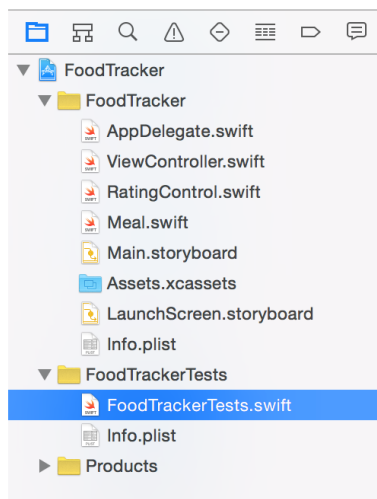
Although your data model code builds, you haven't fully incorporated it into your app yet. As a result, it's hard to tell whether you've implemented everything correctly, and if you might encounter edge cases that you haven't considered at runtime.

To address this uncertainty, you can write unit tests. [Unit tests](#) are used for testing small, self-contained pieces of code to make sure they behave correctly. The `Meal` class is a perfect candidate for unit testing.

Xcode has already created a unit test file as part of the Single View Application template.

### To look at the unit test file for FoodTracker

- Open the `FoodTrackerTests` folder in the project navigator by clicking the disclosure triangle next to it.



2. Open FoodTrackerTests.swift.

Take a moment to understand the code in the file so far.

```
1  import UIKit
2  import XCTest
3
4  class FoodTrackerTests: XCTestCase {
5
6      override func setUp() {
7          super.setUp()
8          // Put setup code here. This method is called before the invocation of each
test method in the class.
9      }
10
11     override func tearDown() {
12         // Put teardown code here. This method is called after the invocation of
each test method in the class.
13         super.tearDown()
14     }
15
16     func testExample() {
17         // This is an example of a functional test case.
18         XCTAssert(true, "Pass")
19     }
20
21     func testPerformanceExample() {
22         // This is an example of a performance test case.
23         self.measureBlock() {
24             // Put the code you want to measure the time of here.
25         }
26     }
27
28 }
```

The XCTest framework, which this file imports, is Xcode's testing framework. The unit tests themselves are defined in a class, FoodTrackerTests, which inherits from XCTestCase. The code comments explain the setUp() and tearDown() methods.

The main types of tests you can write are functional tests (to check that everything is producing the values you expect) and performance tests (to check that your code is performing as fast as you expect it to). Because you haven't written any performance-heavy code, you'll only want to write functional tests for now.

Start any method that you want to run as a test with "test" in the title, and give it a specific title that'll be easy to identify later on. For example, a good test might check that a Meal gets initialized properly, and you could

name it `testMealInitialization`.

### To write a unit test for Meal object initialization

1. In `FoodTrackerTests.swift`, delete the template tests.

```
1  import UIKit
2  import XCTest
3
4  class FoodTrackerTests: XCTestCase {
5
6  }
```

You don't need to use any part of the template implementation for this lesson.

2. Before the last curly brace (`}`), add the following:

```
// MARK: FoodTracker Tests
```

This is a comment to help you (and anybody else who reads your code) navigate through your tests and identify what they correspond to.

3. Below the comment, add a new unit test:

```
1  // Tests to confirm that the Meal initializer returns when no name or a
    negative rating is provided.
2  func testMealInitialization() {
3  }
```

4. First, add a test case that should pass. Add the following comment and lines of code to the `testMealInitialization()` test:

```
1  // Success case.
2  let potentialItem = Meal(name: "Newest meal", photo: nil, rating: 5)
3  XCTAssertNotNil(potentialItem)
```

`XCTAssertNotNil` tests that the `Meal` object is not `nil` after initialization, which means the initializer successfully created a `Meal` object with the supplied arguments.

5. Now add a test case where the `Meal` object should fail initialization. Add the following comment and lines of code to the `testMealInitialization()` test:

```
1  // Failure cases.
2  let noName = Meal(name: "", photo: nil, rating: 0)
3  XCTAssertNil(noName, "Empty name is invalid")
```

`XCTAssertNil` asserts that an object is `nil`. In this case, that means the `noName` object is `nil`, which implies that it failed initialization. You expect this initialization to fail because the name is an empty string, which you explicitly test against in your initializer.

6. Now add a test case where the `Meal` object should fail initialization, but this time, try asserting that the initialization should succeed. Add the following lines of code to the `testMealInitialization()` test:

```
1  let badRating = Meal(name: "Really bad rating", photo: nil, rating: -1)
2  XCTAssertNotNil(badRating)
```

You expect this test case to fail because the rating is negative, which you explicitly test against in your initializer.

Your `testMealInitialization()` unit test should look like this:

```
1  // Tests to confirm that the Meal initializer returns when no name or a negative
    rating is provided.
2  func testMealInitialization() {
3      // Success case.
```

```

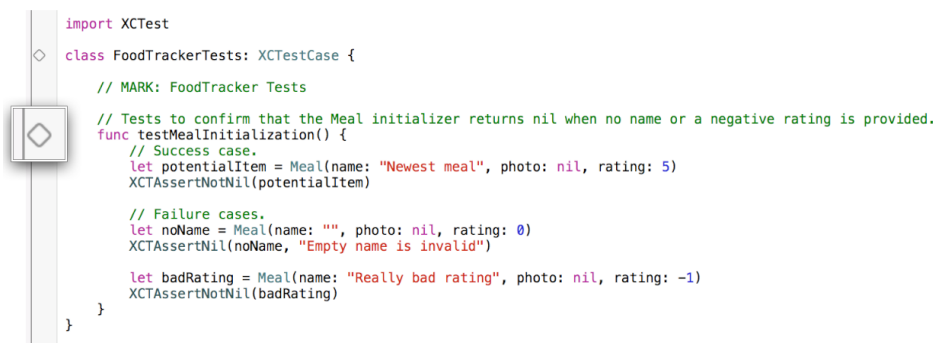
4     let potentialItem = Meal(name: "Newest meal", photo: nil, rating: 5)
5     XCTAssertNotNil(potentialItem)
6
7     // Failure cases.
8     let noName = Meal(name: "", photo: nil, rating: 0)
9     XCTAssertNil(noName, "Empty name is invalid")
10
11    let badRating = Meal(name: "Really bad rating", photo: nil, rating: -1)
12    XCTAssertNotNil(badRating)
13 }

```

You can run all your unit tests at the same time by pressing Command-U, or you can run an individual test. The last test case is expected to fail because you assert that the object is non-nil even though it's actually nil.

#### To run the testMealInitialization() unit test

1. In FoodTrackerTests.swift, find the testMealInitialization() unit test.
2. To the left of the test name, find a diamond shape.



```

import XCTest

class FoodTrackerTests: XCTestCase {

    // MARK: FoodTracker Tests


    // Tests to confirm that the Meal initializer returns nil when no name or a negative rating is provided.
    func testMealInitialization() {
        // Success case.
        let potentialItem = Meal(name: "Newest meal", photo: nil, rating: 5)
        XCTAssertNotNil(potentialItem)

        // Failure cases.
        let noName = Meal(name: "", photo: nil, rating: 0)
        XCTAssertNil(noName, "Empty name is invalid")

        let badRating = Meal(name: "Really bad rating", photo: nil, rating: -1)
        XCTAssertNotNil(badRating)
    }
}

```

3. Hover your mouse over the diamond to reveal a small Run button.



```

import XCTest

class FoodTrackerTests: XCTestCase {

    // MARK: FoodTracker Tests

    // Tests to confirm that the Meal initializer returns nil when no name or a negative rating is provided.
    func testMealInitialization() {
        // Success case.
        let potentialItem = Meal(name: "Newest meal", photo: nil, rating: 5)
        XCTAssertNotNil(potentialItem)

        // Failure cases.
        let noName = Meal(name: "", photo: nil, rating: 0)
        XCTAssertNil(noName, "Empty name is invalid")

        let badRating = Meal(name: "Really bad rating", photo: nil, rating: -1)
        XCTAssertNotNil(badRating)
    }
}

```

4. Click the Run button to run the unit test.

*Checkpoint:* Your app runs with the unit test you just wrote. The first two test cases should pass, and the last should fail.

```
import XCTest

class FoodTrackerTests: XCTestCase {
    // MARK: FoodTracker Tests

    // Tests to confirm that the Meal initializer returns nil when no name or a negative rating is provided.
    func testMealInitialization() {
        // Success case.
        let potentialItem = Meal(name: "Newest meal", photo: nil, rating: 5)
        XCTAssertNotNil(potentialItem)

        // Failure cases.
        let noName = Meal(name: "", photo: nil, rating: 0)
        XCTAssertNil(noName, "Empty name is invalid")

        let badRating = Meal(name: "Really bad rating", photo: nil, rating: -1)
        XCTAssertNotNil(badRating)
    }
}
```

As you see, unit testing helps catch errors in your code. If you actually expected the `Meal` object to be non-nil in the last test case, you would've caught this error during testing. (In this case, because you intentionally wrote a failing test case, you'll just go back and fix your test case.)

### To fix the test case

1. In `FoodTrackerTests.swift`, find the `testMealInitialization()` unit test.
2. Change the last line of code to this:

```
XCTAssertNil(badRating, "Negative ratings are invalid, be positive")
```

Your `testMealInitialization()` unit test should look like this:

```
1 // Tests to confirm that the Meal initializer returns when no name or a negative
  rating is provided.
2 func testMealInitialization() {
3     // Success case.
4     let potentialItem = Meal(name: "Newest meal", photo: nil, rating: 5)
5     XCTAssertNotNil(potentialItem)
6
7     // Failure cases.
8     let noName = Meal(name: "", photo: nil, rating: 0)
9     XCT
10
11     let badRating = Meal(name: "Really bad rating", photo: nil, rating: -1)
12     XCTAssertNil(badRating, "Negative ratings are invalid, be positive")
13 }
```

On This Page

*Checkpoint:* Your app runs with the unit test you just wrote. All test cases should pass.

```
import XCTest

class FoodTrackerTests: XCTestCase {
    // MARK: FoodTracker Tests

    // Tests to confirm that the Meal initializer returns nil when no name or a negative rating is provided.
    func testMealInitialization() {
        // Success case.
        let potentialItem = Meal(name: "Newest meal", photo: nil, rating: 5)
        XCTAssertNotNil(potentialItem)

        // Failure cases.
        let noName = Meal(name: "", photo: nil, rating: 0)
        XCTAssertNil(noName, "Empty name is invalid")

        let badRating = Meal(name: "Really bad rating", photo: nil, rating: -1)
        XCTAssertNil(badRating)
    }
}
```

Unit testing is an essential part of writing code because it helps you catch errors that you might otherwise overlook. As implied by their name, it's important to keep unit tests modular. Each test should check for a specific, basic type of behavior. If you write unit tests that are long or complicated, it'll be harder to track down

exactly what's going wrong.

NOTE

To see the completed sample project for this lesson, download the file and view it in Xcode.

 [Download File](#)