

---

# Table of Contents

引言	1.1
第一章：教程重点	1.2
第二章：学习Swift基础知识	1.3
第三章：构建基础的用户界面	1.4
第四章：将UI与代码连接	1.5

---

# My Awesome Book

本教程基于苹果官方教程翻译而来，由于本人水平有限，翻译如有不到位的地方，欢迎指出。当然，如若大家英文能力不错，请参照[苹果官方教程英文原版](#)~~

Calvin92 2016/07/12 Guangzhou

## 重点介绍

这个教程（入门教程—使用Swift开发iOS应用）对于构建运行于iPhone和iPad的应用来说是一个很好的入门教程。把这些课程当作一个指导性的入门教程来看，用它来创建你的第一个app—这个app包括工具（tools）、主要的概念和最好的体验，这些将会帮助你少走弯路。

每个章节都含有详细的教程，通过这些指导你需要去掌握这些概念性的信息。针对你们所有人创建的这些课程，将会帮助你们一步一步地创建一个简单而真实的iOS应用。

在你通过这些教程创建应用的过程中，你将会学习到关于iOS应用开发的概念，得到Swift更深层次的理解，熟悉苹果集成软件开发工具（Xcode）的很多特性。

## 课程信息

在这些课程中，你将会构建一个简单的“美食追踪”应用，我们管它叫“FoodTracker”。这个应用将会展示一个美食列表，包括美食名称、排名和图片。用户可以增加美食项，移除或者编辑一项已经存在的美食。为了能够增加一份美食，或者编辑一个已经存在的美食项，用户需要跳转到一个不同的界面——在这个界面，用户可以为美食项指定美食名称，排名和图片。

（效果如下图）

Carrier 9:41 AM

## Your Meals

+

	Caprese Salad	
	Chicken and Potatoes	
	Pasta with Meatballs	

你的第一课是一个playground。playground是Xcode文件的一种类型，它能让你跟你的代码交互，并能看到即时的输出结果。你将会下载这个playground文件，在Xcode中打开，使用它来熟悉Swift中的主要的概念。

剩余的课程是带你实现项目，教程的最后，你将会看到你项目最终的代码和界面的样子。在你学完一个课程之后，你可以下载这个工程，通过其检查你的成果。

如果你需要参考在整个教程中你已经学习过的概念，使用“词汇表（glossary）”来更新你的记忆。“词汇表”词条在整个课程中都有链接。

## 获取工具

为了能够使用在本教程中描述的最新的技术来开发iOS应用，你需要一台Mac电脑（OSX 10.10或更新的系统）来运行最新版本的Xcode。Xcode涵盖了你需要用来设计、开发和调试的所有特性，它还包含Xcode的拓展iOS SDK，这里面包括你用来特地开发iOS的工具，编译器，和框架。

通过App Store，在你的Mac上免费下载最新版本的Xcode。

下载最新版本的Xcode（步骤）：

1. 打开你的Mac上的App Store应用（默认情况下它在你的Dock中）
2. 在右上角的搜索框中，输入Xcode，然后按回车键。（Xcode应用将会在搜索结果的第一项展现出来）
3. 点击“Get”，然后点击“Install App”
4. 在弹出的确认框中输入你的Apple ID和密码，确认后Xcode将会下载在你的“/Application”目录中

### 重要提示

这个教程使用的是Xcode7.0和iOS SDK9.0。在你通过这个指南进行学习之前，请确保你使用的是这些版本的Xcode和iOS SDK。

让我们开始吧！

## 第二章：学习Swift基础知识

你的第一门课程将会以playground的形式呈现。如第一章所说，playground是一种文件类型，它能让你在其中修改你的Swift代码，并能立即看到代码的运行结果。Playgrounds对于Swift学习和尝试是很棒的一个功能，它能帮助你加速对Swift基础概念的了解。

### 学习目标

在本章的最后，你将掌握以下这些：

- 区分常量和变量
- 了解何时使用隐式和显示的类型声明
- 理解使用可选类型和可选类型绑定的优势
- 区分可选类型(optional)和隐式解包的可选类型(implicitly unwrapped optional)
- 理解条件语句和循环的目的
- 超过双重判断条件时使用switch语句来做条件判断
- 在条件语句中使用where来增加条件约束
- 区分函数、方法和构造器
- 区分类，结构体和枚举
- 理解继承和协议一致性(protocol conformance)的语法
- 确定隐式的类型，使用Xcode的快速帮助（quick help）快捷方式（Option-Click）查看更多的信息

### 基本类型

常量在一开始被声明之后就一直保持同样的值，而变量在声明之后可以被改变。常量意味着它是不可变的，而变量是可变的。如果你一开始就知道你的值在你的代码中就不会改变，那你应该声明它为常量而不是变量。使用let创建一个常量，var创建一个变量。

```
var myVariable = 42
myVariable = 50
let myConstant = 42
```

每个常量和变量在Swift中都有一个类型，但是你不必显式地把类型写出来。当你声明一个常量或者变量的时候，提供给这个常量或者变量一个值，让编译器去推断它的类型。在上面的例子中，编译器推断出myVariable是一个整型，因为它的初始值是一个整型值。这种特性叫做类型推断。一旦常量或者变量有了一个类型，那么这个类型就不能改变。

如果初始值提供不了足够的信息（或者没有初始值），那你需要通过在变量名之后为其指定类型，中间通过冒号分隔。

```
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

**动手操作**

在Xcode中，试试Option-Click(译者注：按住option键再点击，本教程中所有的这种写法的操作都类似)常量或者变量的名字，查看它的推断类型。使用上面代码中的常量去动手尝试。

值永远不会隐式地转换为另一种类型。如果你需要把一个值转换为另一种不同的类型，那你需要显式地创建一个你期望的类型的实例。在下面的例子中，你将转换一个Int类型的值为一个String类型的值。

```
let label = "The width is "
let width = 94
let widthLabel = label + String(width)
```

**动手操作**

尝试把上面代码最后一行的String去掉。你会发现什么错误？

还有一种更简单的方式把值转换成String类型：在圆括号中写值，并在圆括号中的开始处插入反斜杠“\”。这就是我们熟悉的字符串插值表达式。

```
let apples = 3
let oranges = 5
let appleSummary = "I have \(apples) apples."
let fruitSummary = "I have \(apples + oranges) pieces of fruit."
```

使用可选类型处理值有可能出现值缺失的情况。可选类型的值要么包含一个值，要么就是一个表明值缺失的nil（没有值）。在值的类型后面加上问号（?）以标记该值为可选类型。

```
let optionalInt: Int? = 9
```

为了从一个可选类型中得到隐含的值，你需要对它进行解包。你稍后将会学习解包可选类型，但是你可以采用最直接的方式——用强制解包操作符号（!）来进行解包。只有在你确定隐含的值不为nil的时候，你才能用强制解包操作符来进行解包。

```
let actualInt: Int = optionalInt!
```

可选类型在Swift中是普遍的。在很多情况下，当一个值也许需要又也许不需要呈现的时候，可选类型是非常有用的。

```
var myString = "7"
var possibleInt = Int(myString)
print(possibleInt)
```

在上面这些代码中，`possibleInt`的值为7，因为`myString`包含这个整型的值。但是如果你把`myString`改成一个不能转换为整型的值的时候，`possibleInt`将会变成`nil`。

```
myString = "banana"
possibleInt = Int(myString)
print(possibleInt)
```

数组是一种能跟踪有序集合中所有项的值的数据类型。使用中括号——“[ ]”创建数组，通过在中括号中写入索引来访问数组中的元素。数组是从索引0开始的。

```
var ratingList = ["Poor", "Fine", "Good", "Excellent"]
ratingList[1] = "OK"
ratingList
```

如果要创建一个空的数组，可以使用初始化语法。稍后，你将会学习到更多关于初始化（译者注：也叫构造器）方面的知识。

```
// Creates an empty array.
let emptyArray = [String]()
```

你可能注意到上面的代码有一段注释。注释是在源代码文件中的一段文本，其在编译的时候不会被编译器编译，但是它能提供上下文或者关于个人代码片段的有用的信息。单行注释写在双斜杠“//”之后，而多行注释写在一对由斜杠和星号组成的集合之间（/...../）。在本教程中，你将会看到并且会写到这两种不同类型的注释。

隐式解包可选类型同样可以作为非可选值，这样就不需要每次访问它们的时候都去对其进行解包。这是因为隐式可选类型在值被初始化之后，我们可以认为它总是有值的，虽然这个值可以变化。隐式解包可选类型使用感叹号（!）来解包，而不是问号（?）。

```
var implicitlyUnwrappedOptionalInt: Int!
```

在你的代码中，你真的会需要创建隐式解包可选类型。很多时候，你会在在本教程的视图和源代码（你会在接下来的课程中学习关于这方面的东西）之间追踪outlets（译者注：这是一个把视图跟代码连接起来的桥梁）的时候看到它们的存在，在相关的API中，你也会看到它们的存在。

## 控制流

Swift中有两种类型的控制流语句：第一个是条件语句，比如if和switch——检查一个条件是否是真（true）——也就是说，在执行一个代码段前，先计算它的值是否是布尔值true；第二个是循环，比如for-in和while，多次执行相同的代码片段。

If语句对某一个条件进行检查，如果条件是真（true），就执行在语句体内的代码。你可以通过在if语句中增加else子句定义更复杂的行为。else子句可以和if语句合成为链式使用，也可以自己单独使用——在这种情况下，如果if的语句的结果不为真（true），那么就会执行else子句。

```
let number = 23
if number < 10 {
    print("The number is small")
} else if number > 100 {
    print("The number is pretty big")
} else {
    print("The number is between 10 and 100")
}
```

在if语句中使用可选绑定来检查可选类型是否有值。

```
var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

### 动手操作

将optionalName的值修改为nil，greeting会是什么值？增加一个else子句，在optionalName为nil的时候把greeting设置为不同的值。

如果可选值是nil，那么对应的条件就是false，相对应的括号中的代码就会被忽略。否则，可选值就被解包并赋给let之后的常量，使得被解包出来的值在代码块（if中）中能被使用。

你可以使用单个if语句绑定多个值。where子句可以添加到某一种情形中，以拓展条件语句。在这种情况下，if语句体只有在所有的值都成功绑定和条件都成立之后才被执行。

```
var optionalHello: String? = "Hello"
if let hello = optionalHello where hello.hasPrefix("H"), let name = optionalName {
    greeting = "\((hello)), \((name))"
}
```

Swift在Switch中非常强大。一个switch语句支持各种类型的数据和大量的比对操作——它并不局限于整型和简单地比较相等与否。在下面的例子中，switch语句对vegetable字符串的值在每一种case中进行比对，比对成功则执行相应的代码块。

```

let vegetable = "red pepper"
switch vegetable {
    case "celery":
        let vegetableComment = "Add some raisins and make ants on a log."
    case "cucumber", "watercress":
        let vegetableComment = "That would make a good tea sandwich."
    case let x where x.hasSuffix("pepper"):
        let vegetableComment = "Is it a spicy \(x)?"
    default:
        let vegetableComment = "Everything tastes good in soup."
}

```

### 动手操作

尝试移除default项，你会发现什么错误？

注意`let`是如何在一个匹配的模式中被赋值为常量的。就比如在`if`语句中，`where`子句能添加到某一个条件中拓展条件语句。然而，区别于`if`语句，`switch`中，有多个通过逗号分隔的条件只要匹配上任何一个，相对应的代码块就会执行。

执行完`switch`匹配的代码块之后，程序将会从`switch`语句中退出。程序不会继续执行下一个条件项，所以你不需要在每个条件中显式地退出`switch`语句体（也就是不需要在每个条件中写`break`）。

`Switch`语句必须是详尽的。你需要在每个`switch`中写上`default`项，除非你已经明确地从上下文中满足了所有的条件——比如当`switch`语句在遍历一个枚举的时候。这个要求确保了总会有一个`switch`条件被执行。

你可以通过使用范围（`range`）来获得索引。使用半开的范围操作符来创建一个索引的范围。

```

var firstForLoop = 0
for i in 0..<4 {
    firstForLoop += i
}
print(firstForLoop)

```

半开的范围操作符（`..）不包括最大的数字，所以这个范围是从0到3，总共四次循环迭代。使用封闭的范围操作符（...）创建包括最大和最小数字的范围（range）。`

```

var secondForLoop = 0
for _ in 0...4 {
    secondForLoop += 1
}
print(secondForLoop)

```

这个范围从0到4，总共执行五次循环。下划线（`_`）表示一个通配符，当你不需要知道循环中当前执行的迭代的时候，你可以使用它去忽略。

## 函数和方法

函数是一种可以重复使用，在其中定义了一些代码的代码片段，在整个程序中，函数可以在很多地方被引用。使用`func`去声明一个函数。函数声明可以包含0个或者多个参数，格式如`name:Type`。这些参数是额外的信息，当你去调用方法的时候，你必须把这些额外的信息传递进去。另外，函数有可选的返回类型，格式如：`->返回类型`，这表明这个函数返回的结果。函数的视线写在大括号(`{}`)中。

```
func greet(name: String, day: String) -> String {
    return "Hello \(name), today is \(day)."
}
```

通过在函数名字后面加上一个参数（你传进函数中用以满足参数的值）列表来调用函数。当你调用函数的时候，你传进函数中的第一个参数不需要写出它的参数名字，其他的参数值都需要写出对应的名字。

```
greet("Anna", day: "Tuesday")
greet("Bob", day: "Friday")
greet("Charlie", day: "a nice day")
```

方法(method)是在一个特定的类型中定义的函数。方法（method）会隐式地关联到它所定义的类型上，且只有在这个所关联的类型（或者这个类型的子类，这个概念稍后你会接触到）上才可以调用这个方法。在之前的`switch`语句的例子中，你看到了定义在`String`类型上的`hasSuffix()`方法，这里再看一遍：

```
let exampleString = "hello"
if exampleString.hasSuffix("lo") {
    print("ends in lo")
}
```

正如你所看到的，调用方法是通过点语法(.)发起调用的。当你调用方法的时候，你传进方法中的第一个参数值不需要写出它对应的名字，而其它每一个参数都带有名字。比如，在数组(Array)中的这个方法需要两个参数，而你只需要为第二个参数加上名字。

```
var array = ["apple", "banana", "dragonfruit"]
array.insert("cherry", atIndex: 2)
array //译者注：这里的意思应该是打印print(array)
```

## 类和构造器

在面向对象编程中，程序的行为主要是基于对象之间的交互的。对象是类（**class**）的实例，实例可以认为你是对象的蓝图。类使用属性的方式存储关于它们自身的信息，使用方法定义它们的行为。

通过在**class**关键字的后面加上类的名字来定义一个类。在类中，属性的写法和定义常量和变量相同，除了它是写在类的上下文中之外。同样地，方法的定义和函数的定义也是类似的。下面的例子声明了一个形状（**Shape**）类，其定义了一个名字为**numberOfSides**（边的数量）属性和一个**simpleDescription()**方法。

```
class Shape {
    var numberOfSides = 0
    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

创建一个类的实例，只需要在类的名字后面加上圆括号。使用点操作符（`.`）去访问实例中的属性和方法。下面的例子中，`shape`是**Shape**类的一个实例。

```
var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

这个**Shape**类缺少了一个重要的东西：构造器（**initializer**）。构造器是在类实例使用之前为实例做准备工作的一个方法，它包括为每一个属性设置一个初始值、执行任何其他的设置。使用**init**去声明一个构造器。下面的例子定义了一个新的类——**NamedShape**，它是一个带有**name**参数的构造器。

```
class NamedShape {
    var numberOfSides = 0
    var name: String

    init(name: String) {
        self.name = name
    }

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

需要注意**self**是怎么用以区分**name**属性和构造器中的**name**参数的。每个属性都需要指定一个值，这个值要么在属性声明中（比如**numberOfSides**），要么在构造器中（比图**name**）。

不要通过直接调用`init`去执行构造器；你应该通过在类名后面的圆括号中把合适的参数传递进去去调用。当你调用构造器的时候，你需要把所有的参数名字连同它的值传递进去。

```
let namedShape = NamedShape(name: "my named shape")
```

类可以从它的父类中继承父类的行为。一个类继承了另外一个类的行为，则称这个类为子类（**subclass**），而被继承的类为父类（或者超类，**superclass**）。子类在自身的类名后面加上父类的名字，中间通过冒号分隔的形式实现继承。一个类只能继承一个父类，即时父类还可以继承另一个父类，等等，由此形成类层级（**class hierarchy**）。

在子类中重写（**override**）父类的实现是通过`override`关键字标记的。为了防止意外重写了父类的实现，在没有`override`标记的时候，重写行为会被编译器当做`error`（错误）处理。编译器同样会检查在父类中没有存在却被标记为`override`的方法。

下面的例子定义了`Square`类和一个名字为`NamedShape`的子类。

```
class Square: NamedShape {
    var sideLength: Double

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }

    func area() -> Double {
        return sideLength * sideLength
    }

    override func simpleDescription() -> String {
        return "A square with sides of length \(sideLength)."
    }
}

let testSquare = Square(sideLength: 5.2, name: "my test square")
testSquare.area()
testSquare.simpleDescription()
```

需要注意`Square`类的构造器有三个不同的步骤：

1. 设置`Square`子类声明的属性值
2. 调用父类`NamedShape`的构造函数
3. 修改在父类`NamedShape`中定义的属性值。在这一步，任何额外的通过方法（**method**），获得方式（**get**）、设置方式（**set**）进行的设置工作都可以在这一步进行。

有时候，当提供给参数的值不在预想的范围内，或者当需要的数据缺失的时候，对象的构造函数需要支持构造失败的情况。这种有可能导致对象初始化失败的构造器就做可失败构造器。可失败构造器在初始化的时候有可能会返回nil。使用init?声明一个可失败构造器。

```
class Circle: NamedShape {
    var radius: Double

    init?(radius: Double, name: String) {
        self.radius = radius
        super.init(name: name)
        number_of_sides = 1
        if radius <= 0 {
            return nil
        }
    }

    override func simpleDescription() -> String {
        return "A circle with a radius of \(radius)."
    }
}

let successfulCircle = Circle(radius: 4.2, name: "successful circle")
let failedCircle = Circle(radius: -7, name: "failed circle")
```

构造器还可以用一些关键词去修饰。指定构造器（designated initializer）不需要任何的关键词，这种构造器在类中是一种主要的构造器。在类中的任何构造器最终都必须调用一个指定构造器。

和构造器紧邻的convenience关键词表明这是一个便利构造器。便利构造器是次要的构造器，通过其可以增加额外的行为或定制的服务。但是便利构造器最终必须调用一个指定构造器。

和构造器紧邻的required关键词表明：在父类中持有必须要在子类实现的构造器时，子类就必须通过override去实现构造器。

类型转换是一种检查实例类型的方式，在这个类层次结构中，通过把这个实例当做是一个不同的子类或者父类来检查实例的类型。

某一个类型的常量或者变量实际上对应一个看不见的子类的实例。当你确信这个类型是具体的类型时，你可以尝试用转换操作将其向下转换（downcast）成子类。

由于向下转换有可能导致失败，所以类型转换操作会有两种不同的形式。第一种是可选的形式——as?，这种形式将会返回一个可选的你尝试去转换的类型的值；第二种是强制的形式——as!，这种形式会做一个复合的操作：尝试去向下转换并且对结果进行强制解包。

当你不确定向下做类型转换能成功的时候，你可以使用可选类型绑定操作符（optional type cast operator）——as?。这种形式的操作总会返回一个可选值，如果不能向下转换，这个值将会是nil。这可以让你检查向下类型转换是否成功。

只有当你确定你的向下转换操作会成功的时候，你才可以使用强制类型转换操作符（**forced type cast operator**）——`as!`。当你尝试转换了一个错误的类型时，这种形式的操作会触发一个运行时错误。

下面的例子向你展示了可选类型转换操作（`as?`）的使用：这个例子检查了在`shapes`数组中的某一项`shape`是否是一个正方形（`square`）或者三角形（`triangle`）。每一次找出`Shape`时，会对相对应的变量`squares`和`triangles`加一，最后输出结果值。

```
class Triangle: NamedShape {
    init(sideLength: Double, name: String) {
        super.init(name: name)
        numberOfSides = 3
    }
}

let shapesArray = [Triangle(sideLength: 1.5, name: "triangle1"), Triangle(sideLength: 4.2, name: "triangle2"), Square(sideLength: 3.2, name: "square1"), Square(sideLength: 2.7, name: "square2")]
var squares = 0
var triangles = 0
for shape in shapesArray {
    if let square = shape as? Square {
        squares++
    } else if let triangle = shape as? Triangle {
        triangles++
    }
}
print("\(squares) squares and \(triangles) triangles.")
```

### 动手操作

尝试将`as?`替换成`as!`。这时候你会捕获到什么错误？

## 枚举和结构体

在Swift中，类不是定义数据类型唯一的方式。枚举（**enumerations**）和结构体（**structures**）和类有相似的功能，但是它们在不同的上下文中却大有用处。

枚举可以为一组相关的值定义一个公共的类型，它能让你在你的代码中使用一种类型安全的方式去处理这些值。枚举可以拥有与它们相关的方法。

使用`enum`创建一个枚举。

```

enum Rank: Int {
    case Ace = 1
    case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
    case Jack, Queen, King
    func simpleDescription() -> String {
        switch self {
        case .Ace:
            return "ace"
        case .Jack:
            return "jack"
        case .Queen:
            return "queen"
        case .King:
            return "king"
        default:
            return String(self.rawValue)
        }
    }
}
let ace = Rank.Ace
let aceRawValue = ace.rawValue

```

在上面的例子中，枚举的原始值的类型是整型（Int），所以你只需要指定第一个原始值。剩余的原始值按照顺序赋值。你同样可以使用字符串或者浮点数作为枚举的原始值。使用 rawValue 属性可以访问到枚举成员的原始值。

使用 init?(rawValue:) 构造函数创建一个从原始值开始的枚举的实例。

```

if let convertedRank = Rank(rawValue: 3) {
    let threeDescription = convertedRank.simpleDescription()
}

```

枚举的成员值是实际值，而不是通过另外一种方式写的原始值。事实上，在原始值没有实际意义的情况下，你不必提供原始值。

```

enum Suit {
    case Spades, Hearts, Diamonds, Clubs
    func simpleDescription() -> String {
        switch self {
        case .Spades:
            return "spades"
        case .Hearts:
            return "hearts"
        case .Diamonds:
            return "diamonds"
        case .Clubs:
            return "clubs"
        }
    }
}
let hearts = Suit.Hearts
let heartsDescription = hearts.simpleDescription()

```

注意上面的例子中，枚举中的成员 `Hearts` 中两种不同的表示方式：当 `hearts` 常量被赋值时，枚举成员 `Suit.Hearts` 是它的完整写法，因为常量没有隐式的类型指定。在 `switch` 中，枚举成员 `.Hearts` 是其简写的形式，因为 `self` 的值已经明确地确定为 `suit`。任何时候，在你知道值的类型之后，你可以使用简写的形式。

结构体和类一样支持很多相同的行为，包括方法和构造器。类和结构体之间最重要的一个区别是：结构体在代码中被传递时总是以复制的形式传递的，但是类是以引用传递的。结构体在定义轻量级的数据类型上是很完美的，它不需要有很多类似于继承和类型转换的功能。

使用 `struct` 来创建一个结构体。

```

struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \(suit.simpleDescription())"
    }
}
let threeOfSpades = Card(rank: .Three, suit: .Spades)
let threeOfSpadesDescription = threeOfSpades.simpleDescription()

```

## 协议

协议是定义了一系列方法、属性和其它特定任务或功能的蓝图。协议事实上不提供任何这些需求的实现，它只是描述了实现的轮廓。协议定义之后可以被一个类、结构体或枚举采用以提供那些需求的具体实现。任何满足了协议的需求的行为，我们称之为遵循（conform）了那个协议。

使用protocol定义一个协议。

```
protocol ExampleProtocol {
    var simpleDescription: String { get }
    func adjust()
}
```

### 注意

在simpleDescription属性之后的{ get }表示这是一个只读的属性，意味着这个属性的值只能获取，不能改变。

协议可以要求遵循它的类型持有指定的实例属性，实例方法，操作符和下标脚本。协议要求遵循它的类型必须实现其中的实例方法和类型方法。这些方法和普通的实例和类型方法一样，定义在协议中，但是没有圆括号或方法体。

类、结构体和枚举通过在其名字后面写上一个冒号，冒号后列出协议的名字的写法来实现遵循协议。一个类型可以遵循任何数量的协议，协议中间通过逗号分隔。如果类有父类，那么父类必须出现在列表的最前面，后面才跟着协议。你需要通过实现所有的协议的需求来实现协议的遵循。

下面的例子中，SimpleClass采用了ExampleProtocol协议，通过实现simpleDescription属性和adjust()方法以实现遵循该协议。

```
class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += " Now 100% adjusted."
    }
}
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription
```

协议是高级类型，这意味着它们可以当做其它命名过的类型使用。比如，你可以创建一个ExampleProtocol数组，然后在数组的每一个实例中调用adjust()方法（因为在数组中的任何实例会保证实现了adjust()方法——这是协议的要求）。

```

class SimpleClass2: ExampleProtocol {
    var simpleDescription: String = "Another very simple class."
    func adjust() {
        simpleDescription += " Adjusted."
    }
}

var protocolArray: [ExampleProtocol] = [SimpleClass(), SimpleClass(), SimpleClass2()]
for instance in protocolArray {
    instance.adjust()
}
protocolArray

```

## Swift和Cocoa Touch框架

Cocoa Touch是用来开发苹果iOS应用的一个框架的集合，Swift是按照与Cocoa Touch框架无缝衔接来设计的。在你往后的课程的学习中，你会对Swift如何与Cocoa Touch框架进行交互有一个基本的理解。

至此，你已经学习了Swift标准库中所有的数据类型。Swift标准库(Swift standard library)是为Swift设计的一个数据类型和功能的集合，并将这些融合成为一门语言。在标准库中，像String和Array这样的类型就是数据类型的一些例子。

```

let sampleString: String = "hello"
let sampleArray: Array = [1, 2, 3.1415, 23, 42]

```

### 动手操作

通过Option-click在Xcode中的数据类型，查阅Swift标准库中的类型的具体实现。在上面的playground代码中，在String和Array上面Option-click，查看相应地结果。

当你在开发iOS应用的时候，你不会仅仅使用到Swift标准库。在iOS应用开发中，另一个你会频繁用到的框架是UIKit。UIKit包含了针对应用的UI层级（用户界面，user interface）设计的有用的类。

要使用UIKit，只需要在Swift文件或playground文件中简单地把它当做模块导入即可。

```
import UIKit
```

导入了UIKit之后，你可以使用Swift语法来使用UIKit中的类型以及它们相关的方法、属性等等。

```

let redSquare = UIView(frame: CGRect(x: 0, y: 0, width: 44, height: 44))
redSquare.backgroundColor = UIColor.redColor()

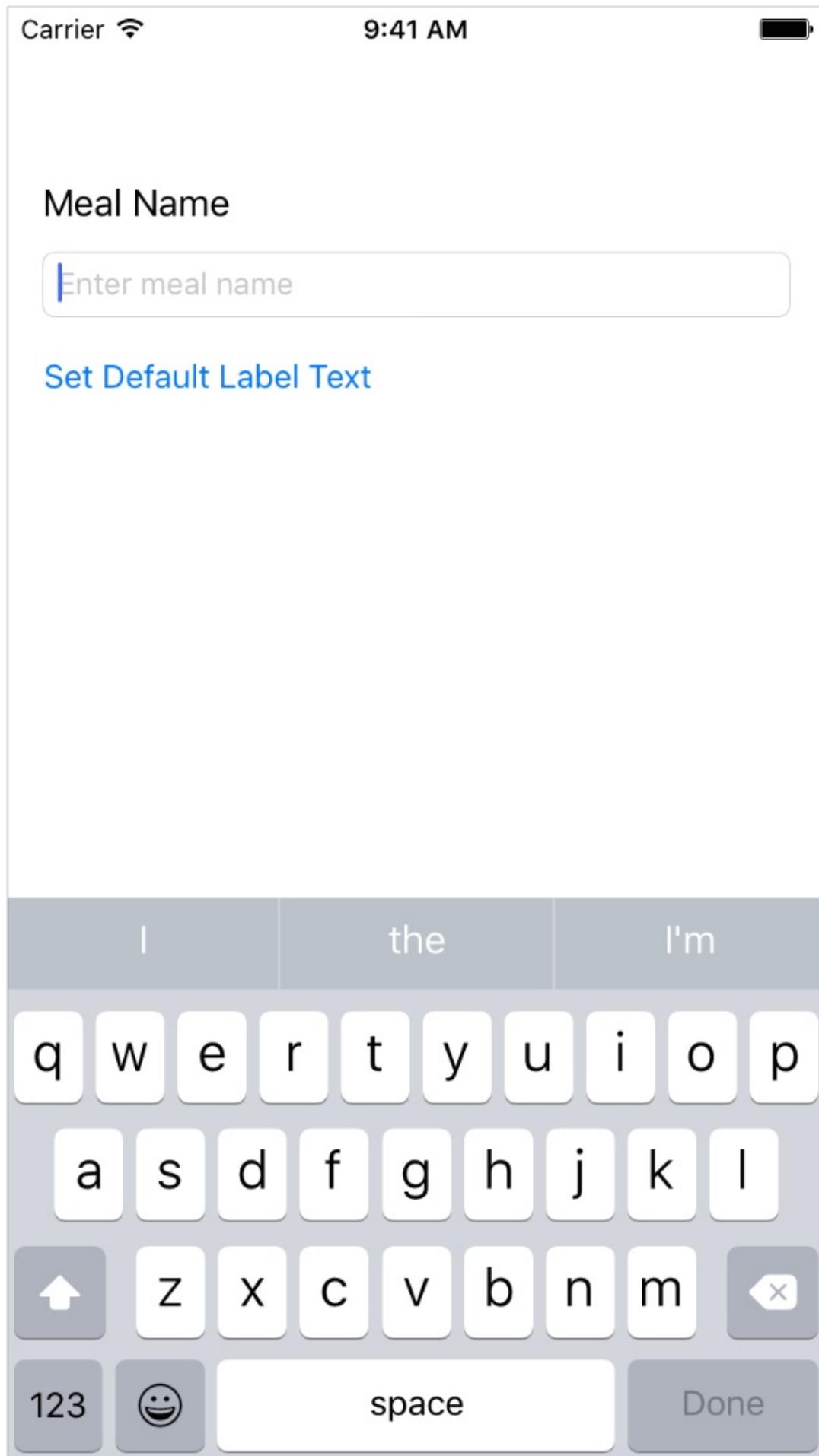
```

在教程中很多介绍到的类都是来自于UIKit中的，所以你会经常看到这个导入UIKit的语句。

在经过大量的Swift知识的介绍之后，相信你已经准备好在下一门课程中创建一个功能完整的应用了。虽然这一章节结束之后，你将不会再需要使用到playground了，但是请记住在应用开发中playground是一个很强大的工具，它能帮助你调试，查看上下文代码，提高原型设计效率等。

## 第三章：构建基础的用户界面

这章会带你熟悉用来写应用的工具——Xcode。你将会熟悉Xcode中的项目结构，学会如何在导航之间跳转以及学会使用基本的项目组件。这一整章中，你将会开始为FoodTracker应用创建一个简单的用户接界面（UI,user interface），并且让它在模拟器中显示出来。当你完成这些之后，你的应用（app）将会像下面这样：



## 学习目标

在本章的最后，你将会学习到以下知识：

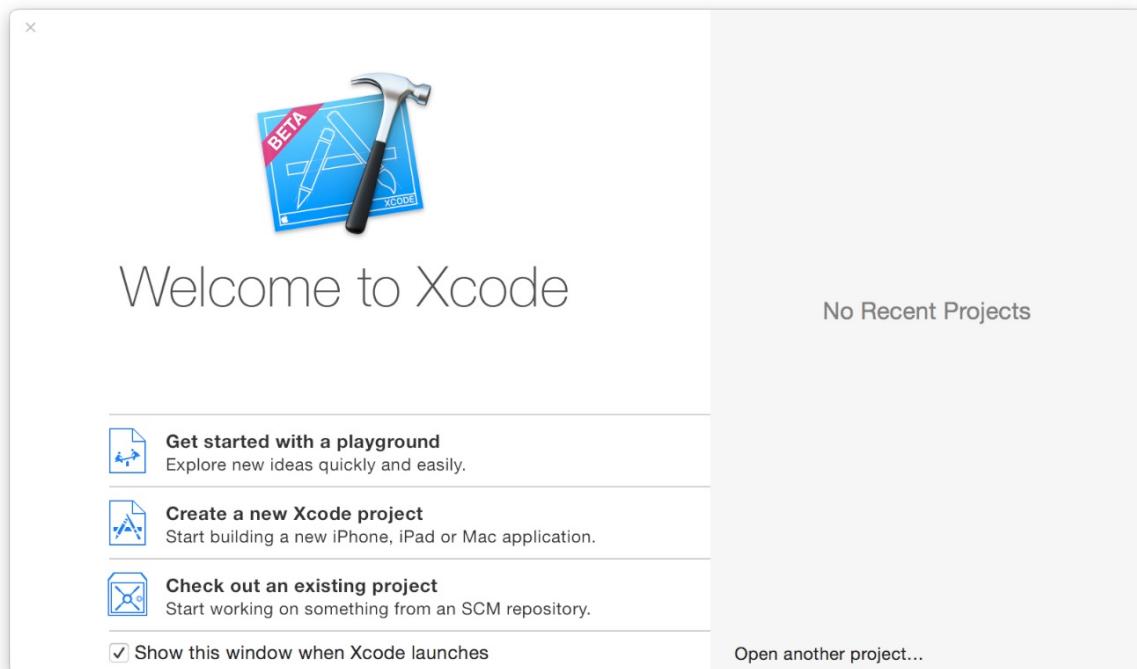
- 在Xcode中创建一个工程
- 认识在创建Xcode项目时，主文件中的样板方法
- 在项目中打开和切换文件
- 在模拟器中运行应用
- 在Storyboard中增加、移除和调整用户界面
- 使用属性检查（Attributes inspector）在Storyboard中编辑用户界面元素的属性
- 使用轮廓视图（outline view）浏览和重新排列storyboard中的用户界面
- 使用预览助手编辑器（Preview assistant editor）预览storyboard中的用户界面
- 使用自动布局（Auto Layout）布局能够自动适应用户设备尺寸大小的用户界面

## 创建一个新工程

Xcode内置了几个应用模版，如游戏、基于触摸的导航应用和基于表格视图的应用，用来辅助开发普遍类型的iOS应用。这些模版大多数都有预先配置好的用户界面和源代码文件。在本章中，你将会从最基本的模版——单页应用开始。

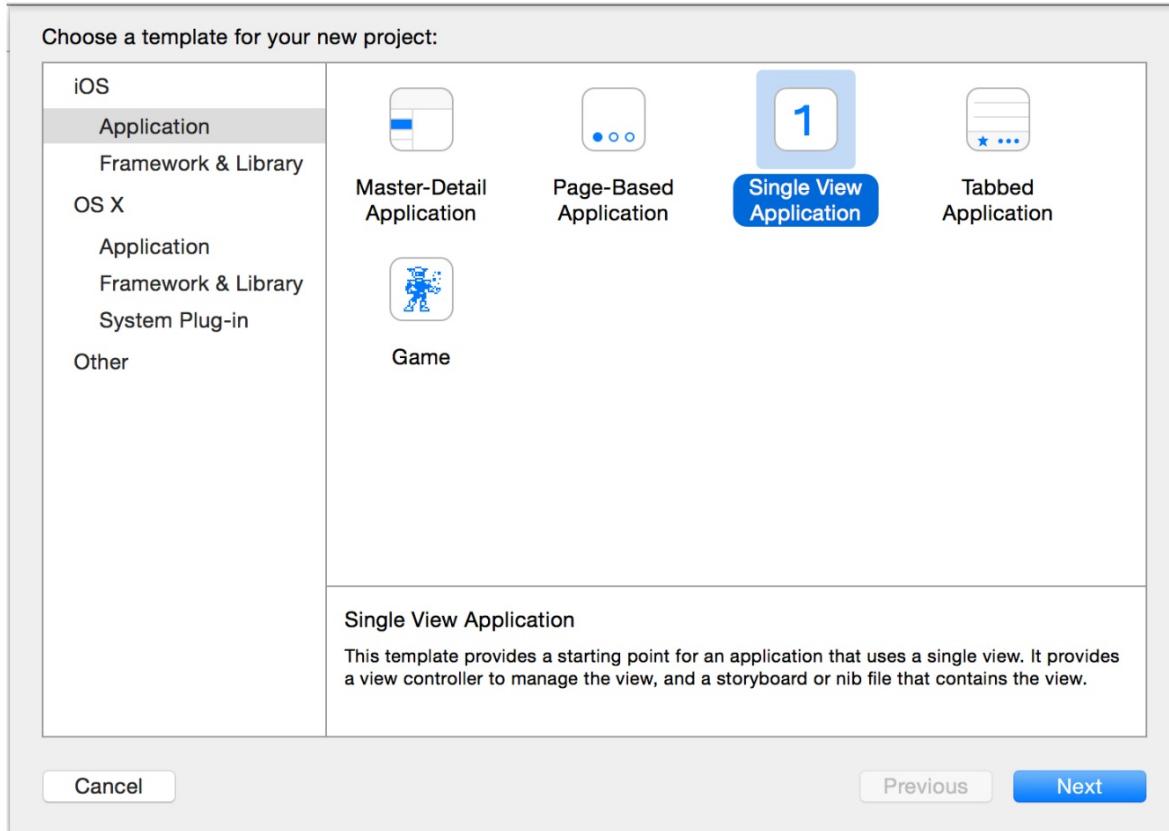
创建一个新工程的步骤：

1. 在/Application目录中打开Xcode（这将会出现Xcode的欢迎界面）



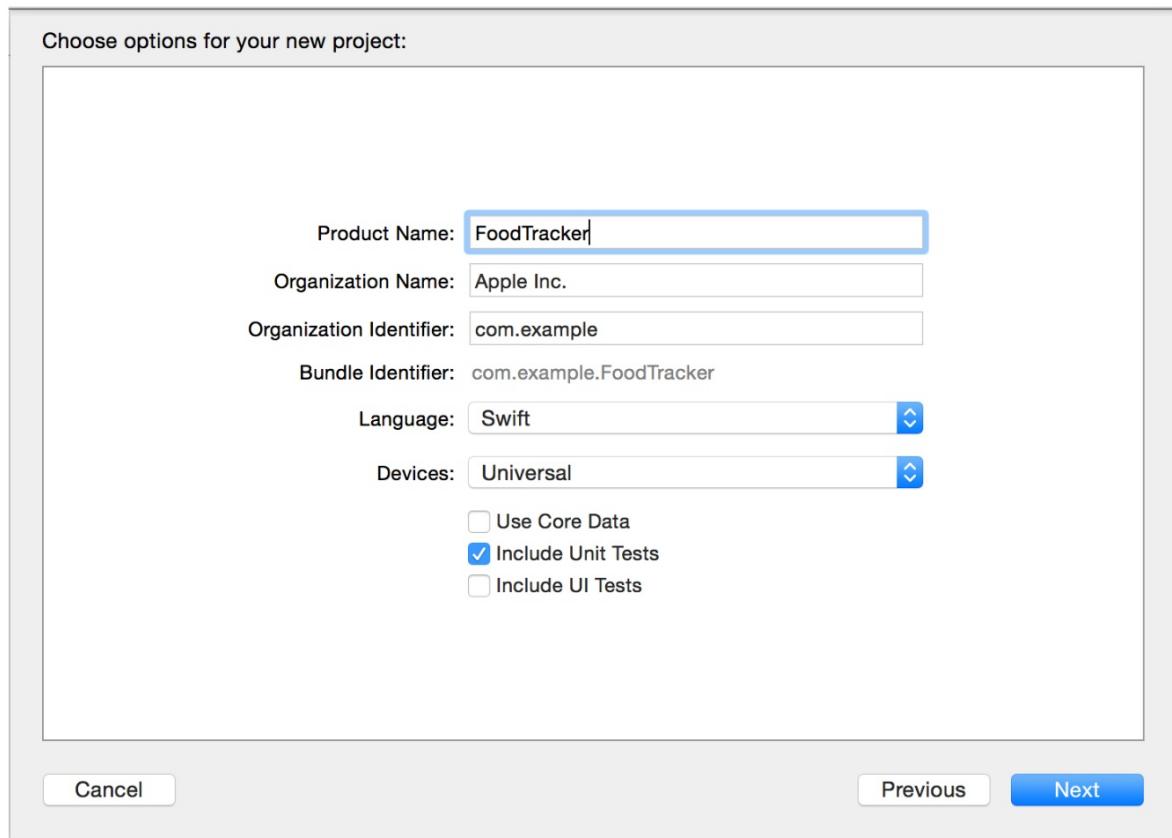
如果你的界面不是欢迎界面的样子，不用担心——你的Xcode可能在之前创建或打开过项目。你只需要在下一步中使用菜单选项即可创建工作。

2. 在欢迎界面中，点击“Create a new Xcode project”（或者选择 File > New > Project）。Xcode将会打开一个新的窗口，并且显示一个对话框。在这个对话框中，选择一个模版。
3. 在对话框左边的iOS区域中，选择Application。
4. 在对话框的主区域里，点击Single View Application，然后点击Next。



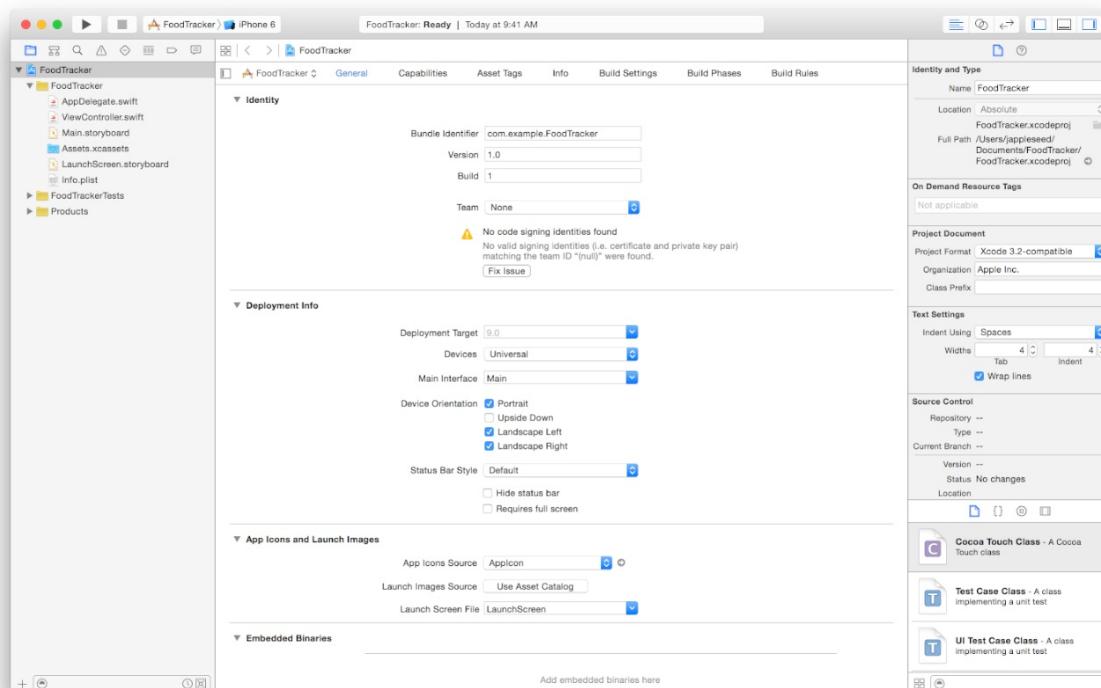
5. 在显示的对话框中，使用下面的值命名你的应用，并为你的应用选择其它合适的选项。
6. Product Name: FoodTracker Xcode将会使用你输入的这个名字命名你的工程和应用。
7. Organization Name: 你的组织的名字或者你自己的名字。你可以把这项留空。
8. Organization Identifier: 你的组织的标识，如果你有的话，就如实填写；如果没有，可以使用com.example。
9. Bundle Identifier：这个值是基于你的Product Name和Organization identifier自动生成的。
10. Language : Swift
11. Devices : Universal 一个通用（Universal）的应用可以同时在iPhone和iPad上运行。
12. Use Core Data : 不选。
13. Include Unit Tests : 选上。

14. Include UI Test : 不选。



15. 点击 Next

16. 在显示出来的对话框中，选择你要保存这个工程的位置，然后点击Create。Xcode将会在工作区窗口（workspace window）中打开你的新工程。

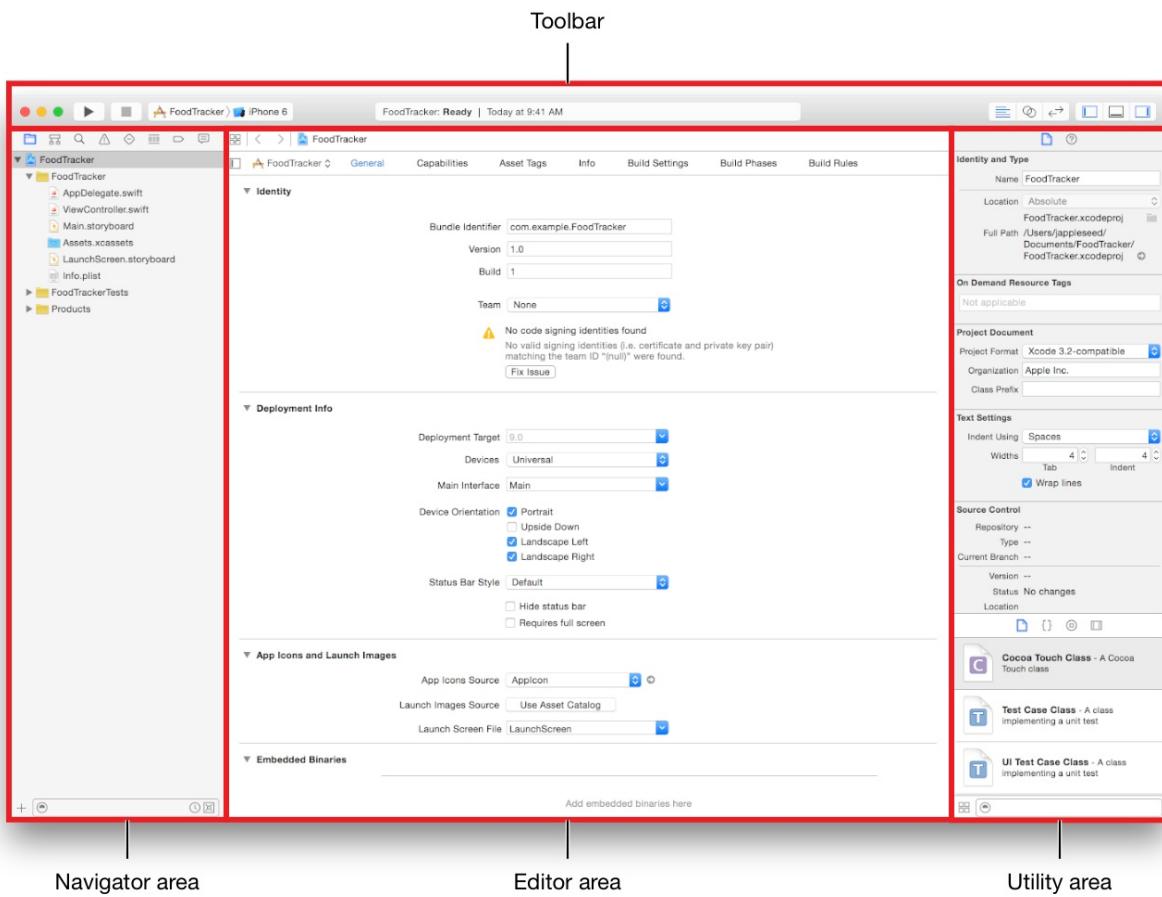


在工作区窗口中，你有可能会看到这样的一条警告消息：“No code signing identities found.”。这条警告的意思是，你还没有为iOS开发安装Xcode，但是不用担心，你可以在无视它的情况下完成这些课程。

## 熟悉Xcode

Xcode涵盖了所有你想创建一个应用所需要的东西。它不仅帮你组织好创建一个应用所需要的代码，还为代码和视图元素提供了编辑器，允许你打包你的应用，同时还包含了一个强大的集成调试器。

要熟悉Xcode工作区中的主要的区域，你需要花费一点时间。你将会在整个教程中使用到下图窗口中标记的区域。不要这些东西打击，因为当你需要使用到的时候，每一个区域都会做详细的描述的。



## 运行模拟器

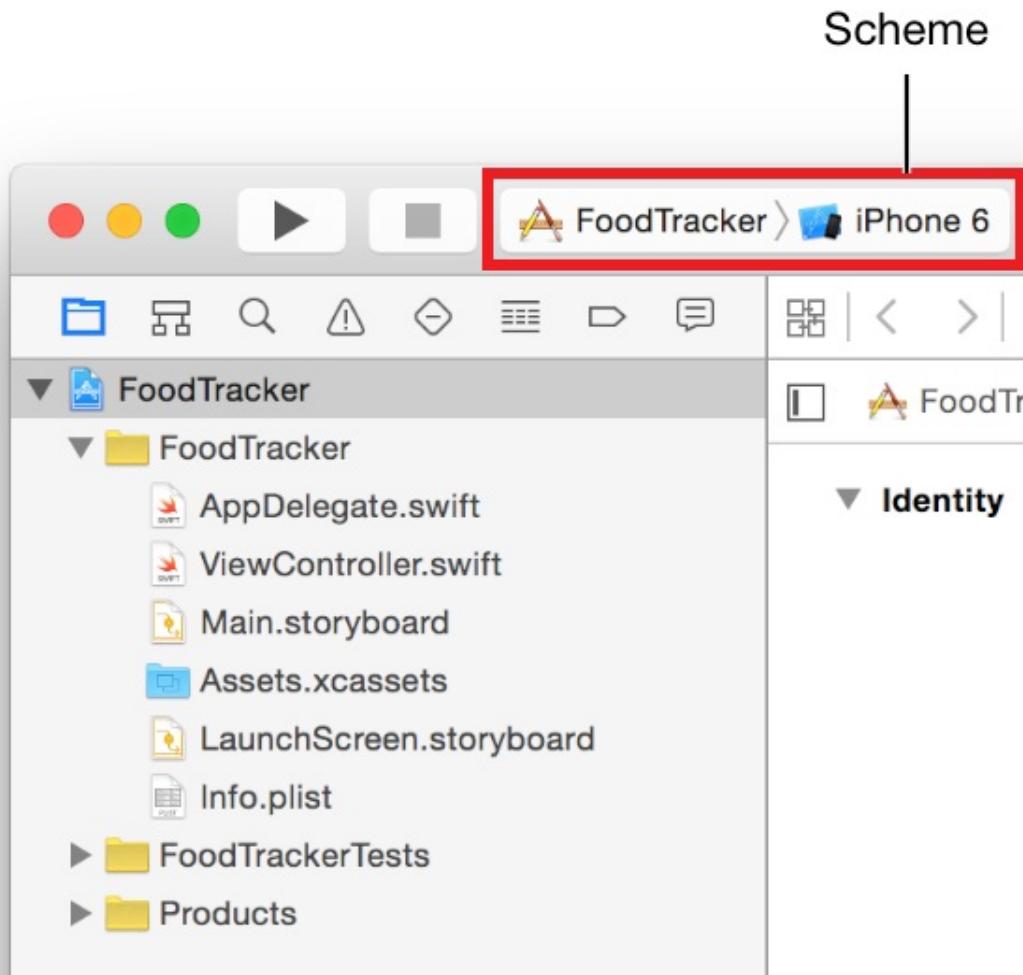
因为你是基于Xcode的模版建立项目的，所以基本的应用环境会自动地为你设置。即便你还没有写任何的代码，你也可以在没有额外的配置的情况下编译和运行你的单页应用模版。

使用Xcode中的模拟器（Simulator）可以运行你的app。模拟器能够让你知道当应用在实际设备中运行时，整个应用的外观以及行为。

模拟器能模拟不同的硬件设备类型——不同屏幕大小的iPad和iPhone等。所以你可以在你开发的平台的设备中模拟你的应用。在本教程中，使用iPhone6作为模拟器。

运行你的app的步骤：

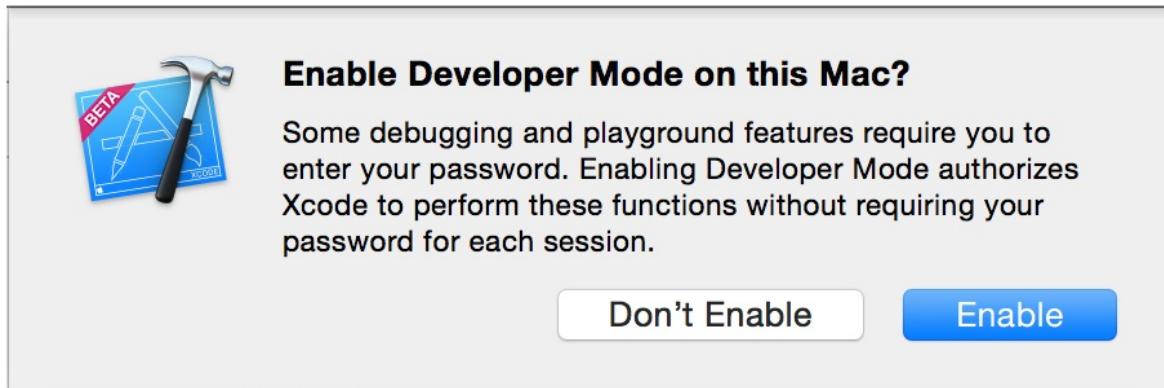
1. 在Xcode工具栏的弹出式Scheme菜单中，选择iPhone6. 这个弹出式的Scheme菜单能让你选择各种模拟器或设备，以运行你的app。确保你选择的是iPhone6模拟器，而不是iOS Device（设备）。



2. 点击位于Xcode工具栏左上角的 Run 按钮。



除此之外，你也可以选择 Product > Run (或者使用键盘Command-R)来运行。如果你是第一次运行app,Xcode会询问你是否愿意在你的Mac上启用开发者模式。开发者模式(developer mode)允许Xcode在访问某一个特性的时候，不需要每次都输入你的密码。你可以决定是否启用开发者模式，在提示框中做出你的选择。

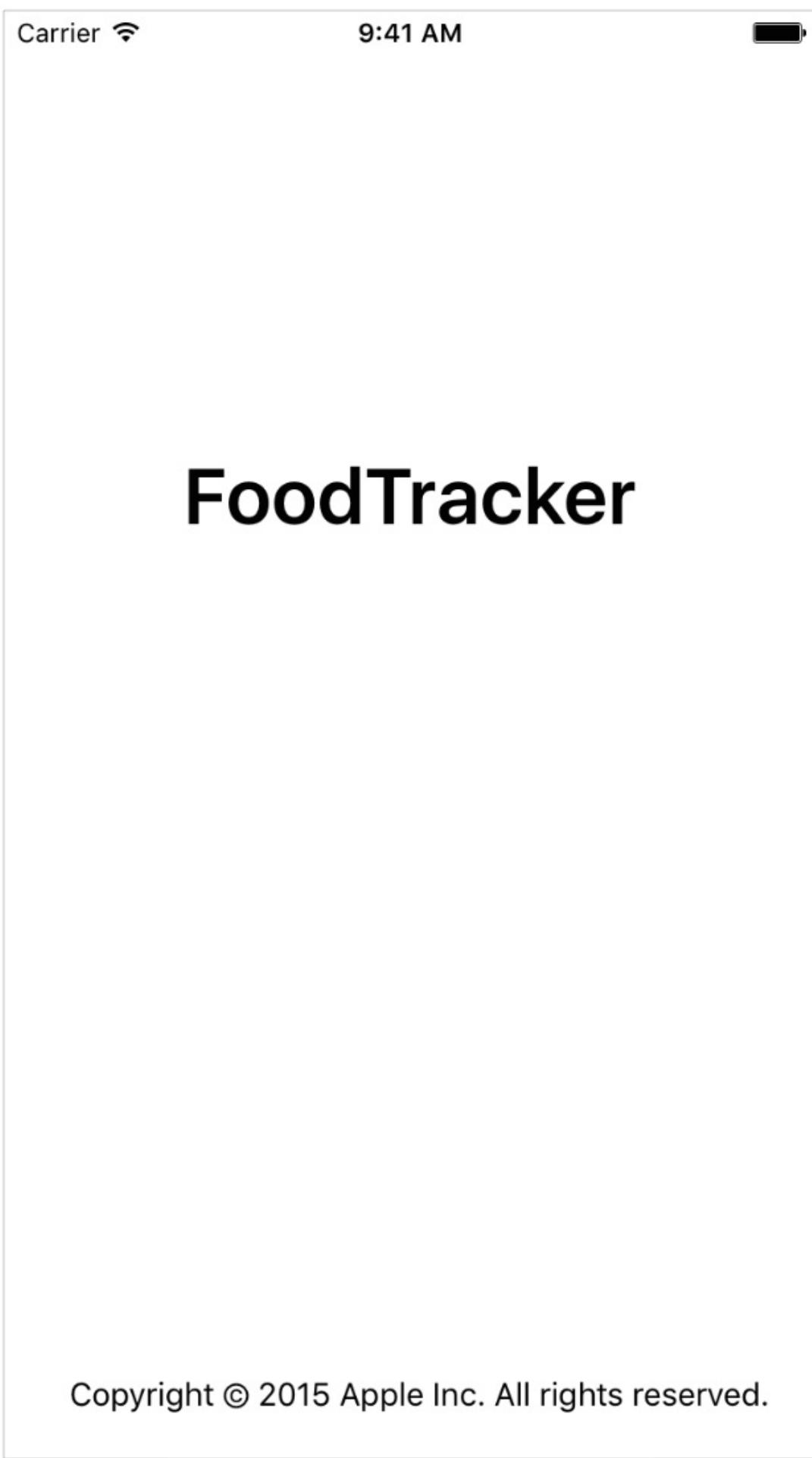


如果你选择了不启用开发者模式，你可能会在之后被询问密码。本教程中假定开发者模式是开启的。

3. 在Xcode工具栏中查看编译完成的过程。Xcode会在工具栏中间的活动视图（Activity viewer）中显示编译过程的信息。

Xcode编译项目完成后，会自动启动模拟器。第一次启动的时候，可能需要花费一点时间。

模拟器在你指定的iPhone模式下打开。在模拟的iPhone窗口，模拟器将会启动你的app。在app完成启动之前，你将会在启动窗口中短暂地看到你的app的名字，FoodTracker。



然后，你应该会看到像下面这样的界面。



现在，这个单页应用模版还没有做什么工作——它只是显示一个白屏。其它模版会有更复杂的行为。在扩展你的app之前，理解一个模版的使用是非常重要的。在没有做任何修改之前运行你的app，是开始开发应用时的一个很聪明的做法。

通过选择 Simulator > Quit Simulator (或者按下键盘Command-Q)退出模拟器。

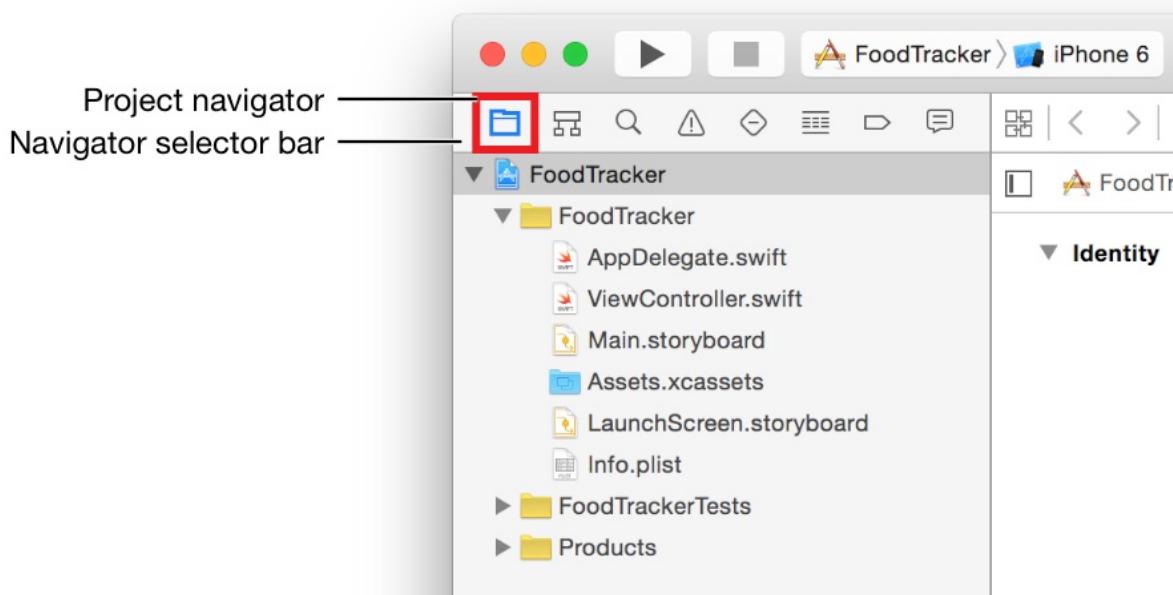
## 查看源代码

这个单页应用模版附带了一些源代码文件来建立应用环境。首先，看一看AppDelegate.swift文件。

### 查看AppDelegate.swift文件源码

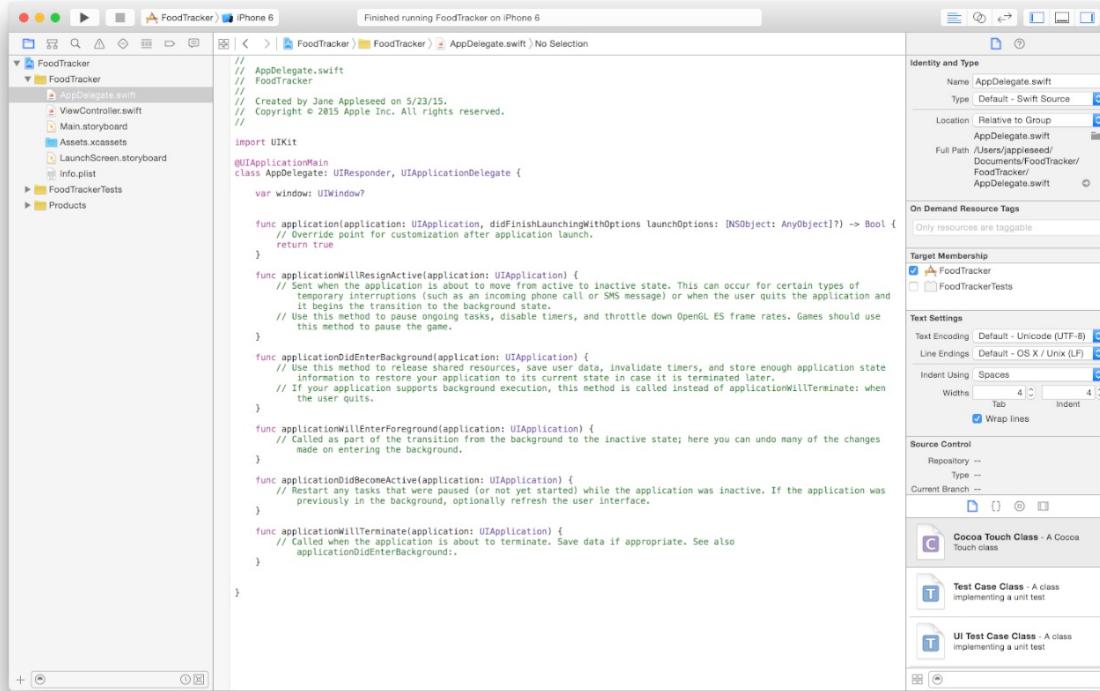
1. 确保工程导航在导航区是打开的。

工程导航(project navigator)显示了你工程的所有文件。如果工程导航没有打开，则点击最左边的导航条按钮。(除此之外，还可以选择 View > Navigators > Show Project Navigator)



2. 如果必要，通过点击工程导航栏里的FoodTracker文件夹旁边的三角形来打开FoodTracker文件夹。
3. 选择AppDelegate.swift文件。

Xcode将会在窗口的主编辑区中打开该文件的源代码。



除此之外，你还可以通过双击AppDelegate.swift文件，在新的窗口中打开它。

## AppDelegate(应用代理)源文件

AppDelegate.swift文件包含两个主要的功能：

- 它在你的app中创建了入口和一个分发输入事件的运行循环（run loop）。这个工作是通过UIApplicationMain属性（@UIApplicationMain）完成的；这个属性位于该文件内容的上方。UIApplicationMain创建了一个应用对象（application object）管理app生命周期，同时也会创建一个应用代理（app delegate）对象—这个对象在下面会作描述。
- 它创建了一个描述这个app代理对象的蓝图的AppDelegate类。App代理会创建一个绘制app内容和提供响应app内状态改变的窗口，而AppDelegate类是你写定制的app层级代码的地方。

AppDelegate类包含一个独立地属性：window。通过这个属性，app代理能够跟踪绘制了所有app内容的窗口。Window属性是一个可选值，也就是说在某些情况下，它可能没有值(nil)。

```
var window: UIWindow?
```

AppDelegate类同样包含了一些重要方法的实现模板。这些预定义的方法允许应用对象（application object）和app代理对话。

```

func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool
func applicationWillResignActive(application: UIApplication)
func applicationDidEnterBackground(application: UIApplication)
func applicationWillEnterForeground(application: UIApplication)
func applicationDidBecomeActive(application: UIApplication)
func applicationWillTerminate(application: UIApplication)

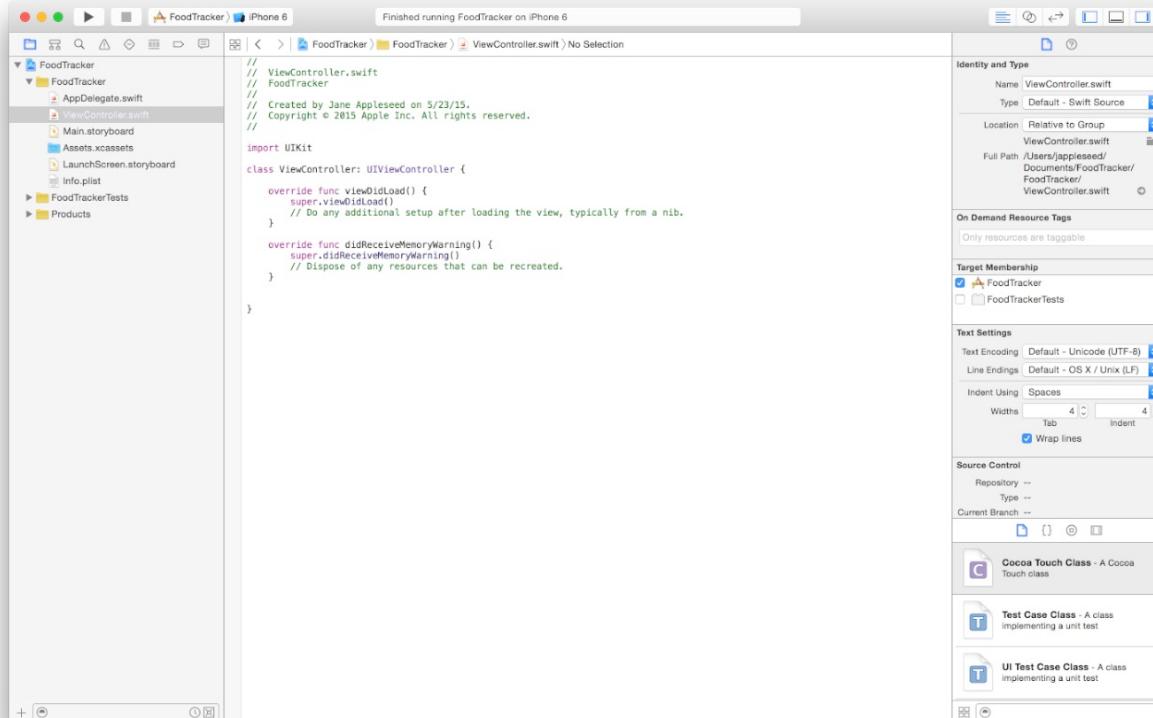
```

在app状态改变的时候—比如，app启动、转至后台运行、终止等，应用对象会调用app代理中的响应方法，使得它能够有机会做出合适的响应。你不需要做任何特别的事情来确保这些方法在正确的时间调用—应用对象会为你做这部分的处理工作。

每一个这些自动实现的方法都有一个默认的行为。如果你保持这些模板实现为空或者从AppDelegate类中把它们删除，那么在这些方法调用的时候，你都会得到它们默认的行为。在这些方法调用的时候，使用这些方法模板来增加的自定义代码也会得到执行。在本教程中，你不会使用到任何自定义的app代理代码，所以你不必改动AppDelegate文件。

## 视图控制器源文件

单页应用模板还有另外一个源码文件：ViewController.swift。在工程导航中选择 ViewController.swift 可以查看该文件。



这个文件定义了一个自定义的UIViewController的子类，名字为ViewController。现在，这个类只是简单地继承了UIViewController中定义的行为。要重写或者扩展这些行为，你需要重写在UIViewController中定义的方法（正如你在ViewController.swift文件中看到的viewDidLoad()和didReceiveMemoryWarning()方法所做的一样），或者实现你自定义的方法。

虽然这些模板中存在didReceiveMemoryWarning()方法，但是在本教程中，你不需要对其做任何的实现，所以你尽管删掉它。

到此，你的ViewController.swift代码看起来应该像下面这样：

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

}
```

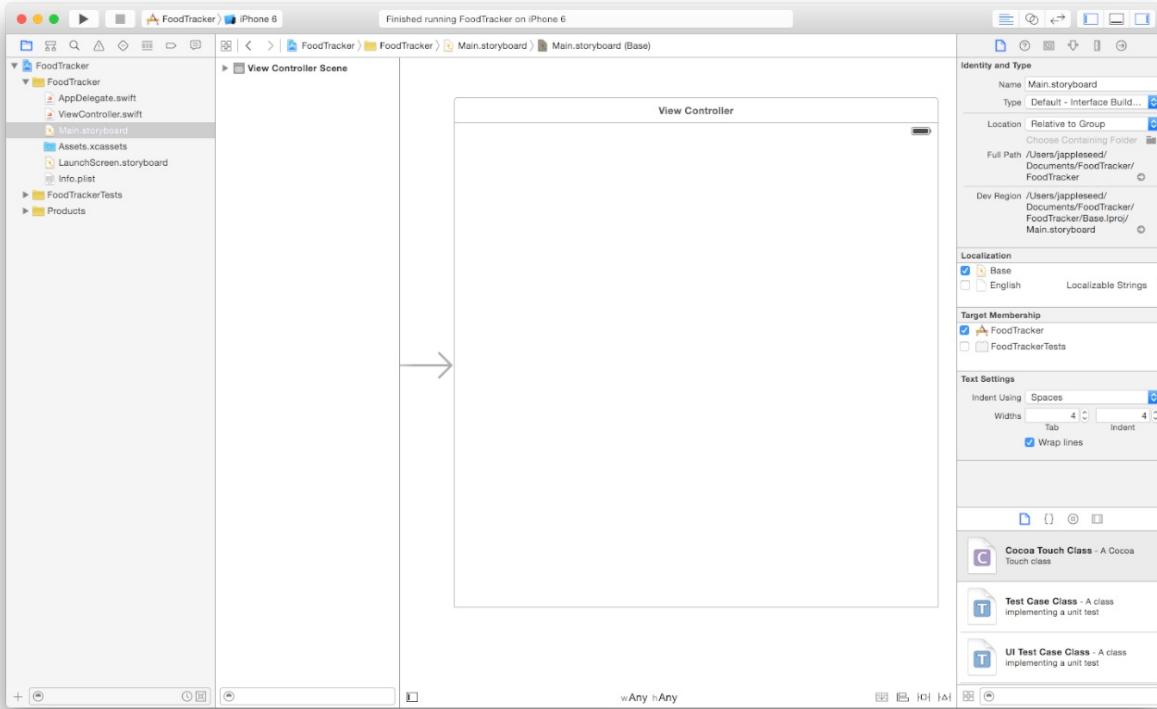
在稍后的教程中，你将会开始在这个源码文件中写代码。

## 打开你的**Storyboard**

你即将开始在你app的storyboard中进行工作。Storyboard是你app的用户界面的可视化呈现工具，它能展示内容界面和在内容界面之间的切换关系。使用storyboard可以对app进行布局以驱动你的app。在你构建界面的时候，你将会清晰地看到你正在构建的东西，得到对于你正在做的东西的即时的反馈，同时能够马上看到你对用户界面的更改。

### 步骤：打开你的**storyboard**

- 在工程导航中，选择Main.storyboard。Xcode会在界面设计器（Interface Builder）中打开storyboard。界面设计器是Xcode的可视化界面编辑器，位于编辑区中。Storyboard的背景是一块画布（canvas）。你可以使用这个画布增加和布局UI元素。你的storyboard看起来应该像这样：



这时候，你的app中的storyboard包含了一个场景（**scene**），这个场景展示了你app中一屏幕的内容。画布中，指向场景左边的箭头是storyboard的入口（**entry point**），其表示当app启动的时候，这个场景是第一个被加载的。现在，你在画布中看到的场景包含了一个被视图控制器管理的单个视图。你很快将会学习到更多关于视图和视图控制器的作用方面的知识。

当你在iPhone6模拟器中运行你的app的时候，该场景中的视图就是你在设备界面中看到东西。但是，当你在画布中看这个场景的时候，你会意识到它没有iPhone6屏幕的准确尺寸。这是因为在画布中的场景是通用的，它能够适用于所有的设备和方向。你可以使用这个特性创建一个自适应的界面---自适应界面能够根据当前的设备和方向，自动地调整界面以使其友好地显示。

## 创建基础的UI

现在是时候构建基本的界面了。你将会为你的食谱app—FoodTracker创建一个UI，它能允许你在你的app上添加一条食谱。

Xcode提供了一个对象库，通过这个库你可以添加一个storyborad文件。这个库中的一部分是能在用户界面中显示的元素，比如按钮（**buttons**）和文本框（**text fields**）。另外一部分如视图控制器（**view controllers**）和手势识别器（**gesture recognizers**）只是定义你的app的行为而不会在屏幕中显示。

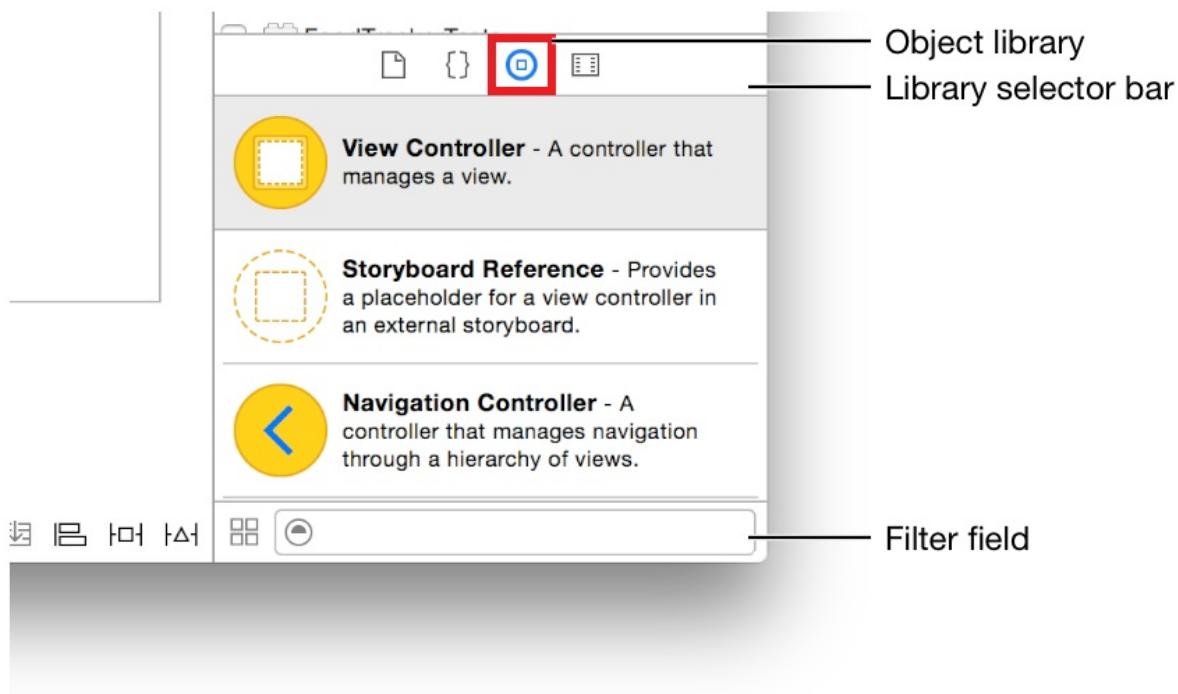
在UI中显示的元素就是总众所周知的视图(**views**)。视图为用户显示需要的内容，其是构建你的UI的基础块，并以一种清晰的、优雅的、实用的方式呈现你的内容。视图有很多内置的实用的行为，包括在屏幕上显示它们自身，以及响应用户的输入。

在iOS中，所有的视图对象都是UIView或者其子类。很多UIView的子类在外观和行为上是高度定制的。在你的场景（scene）中，从添加一个UIView子类的文本框（UITextField）开始你的构建工作。文本框能够让用户输入一段单行的文本，而这段文本将会作为你食谱的名字。

添加一个文本框到你的场景中

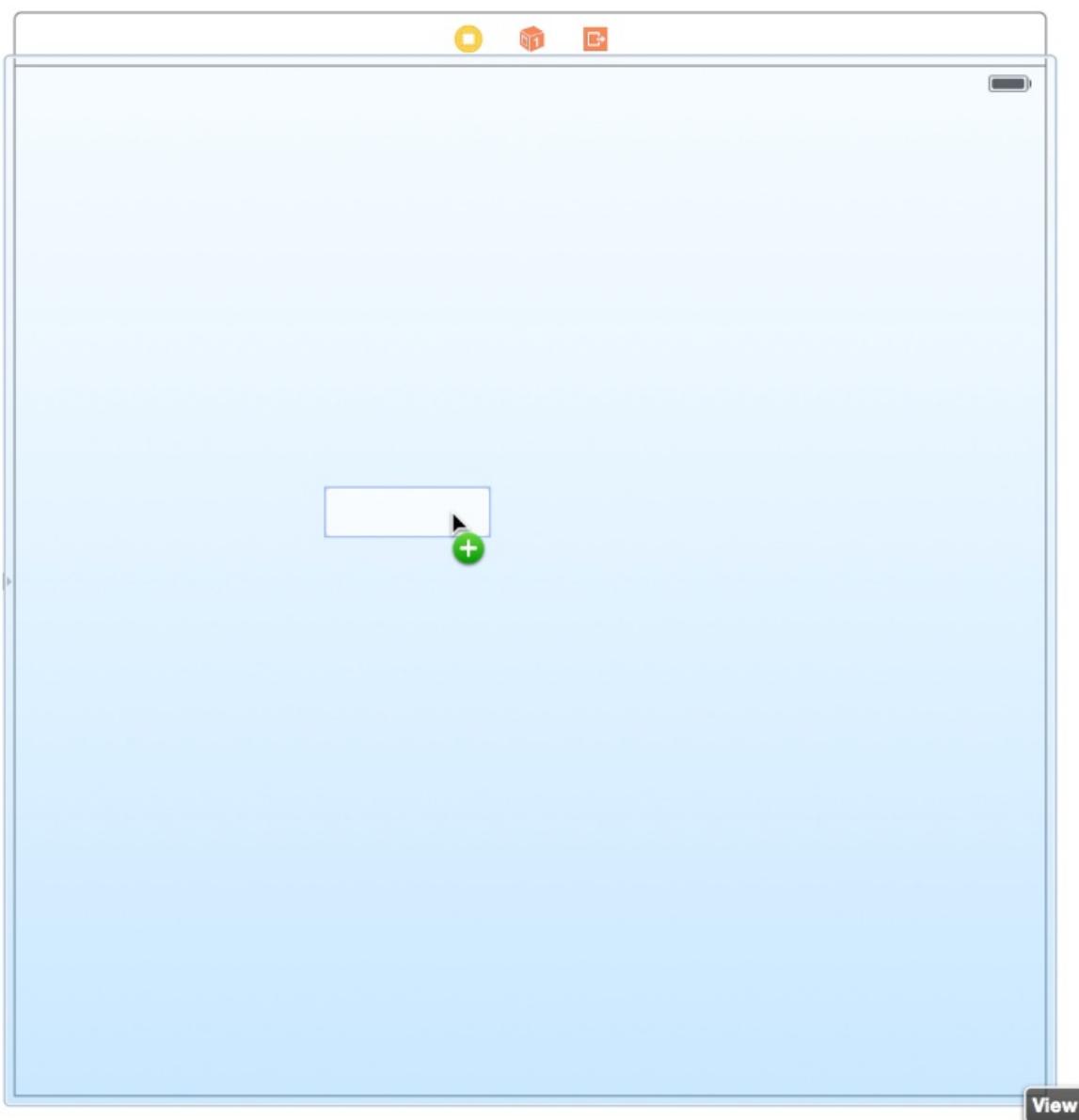
1. 打开对象库(Object library)。

对象库位于Xcode右侧工具区的下方。如果你没有看到对象库，可以点击库选择条下从左算起的第三个按钮。（另外一种方式：选择 View > Utilities > Show Object Library.）



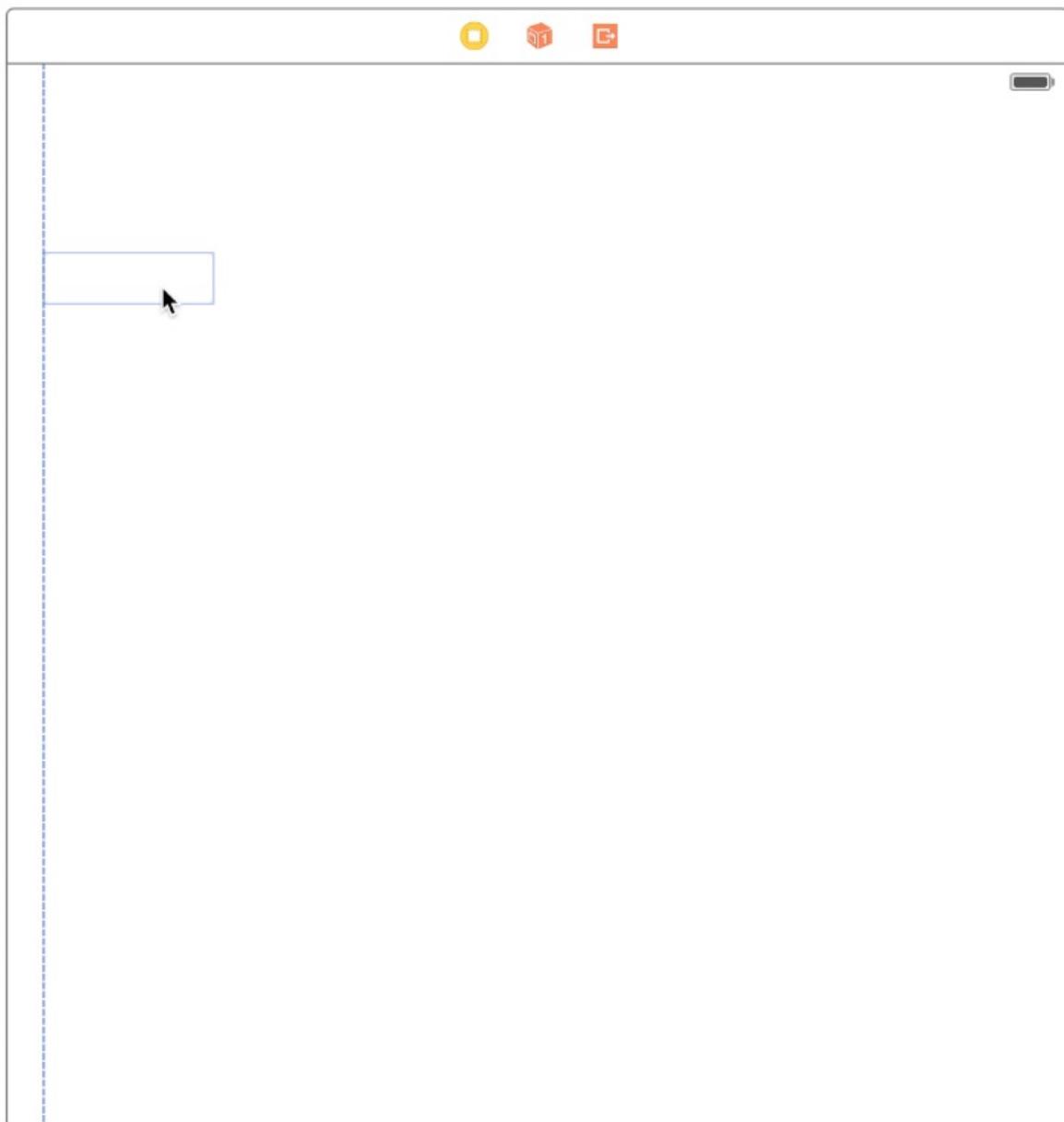
在出现的列表中，显示了每一个对象的名字，描述和可视化的图标。

2. 在对象库中，在筛选区输入text field快速查找文本框对象。
3. 从对象库中拖动一个文本框对象到你的场景（scene）中。



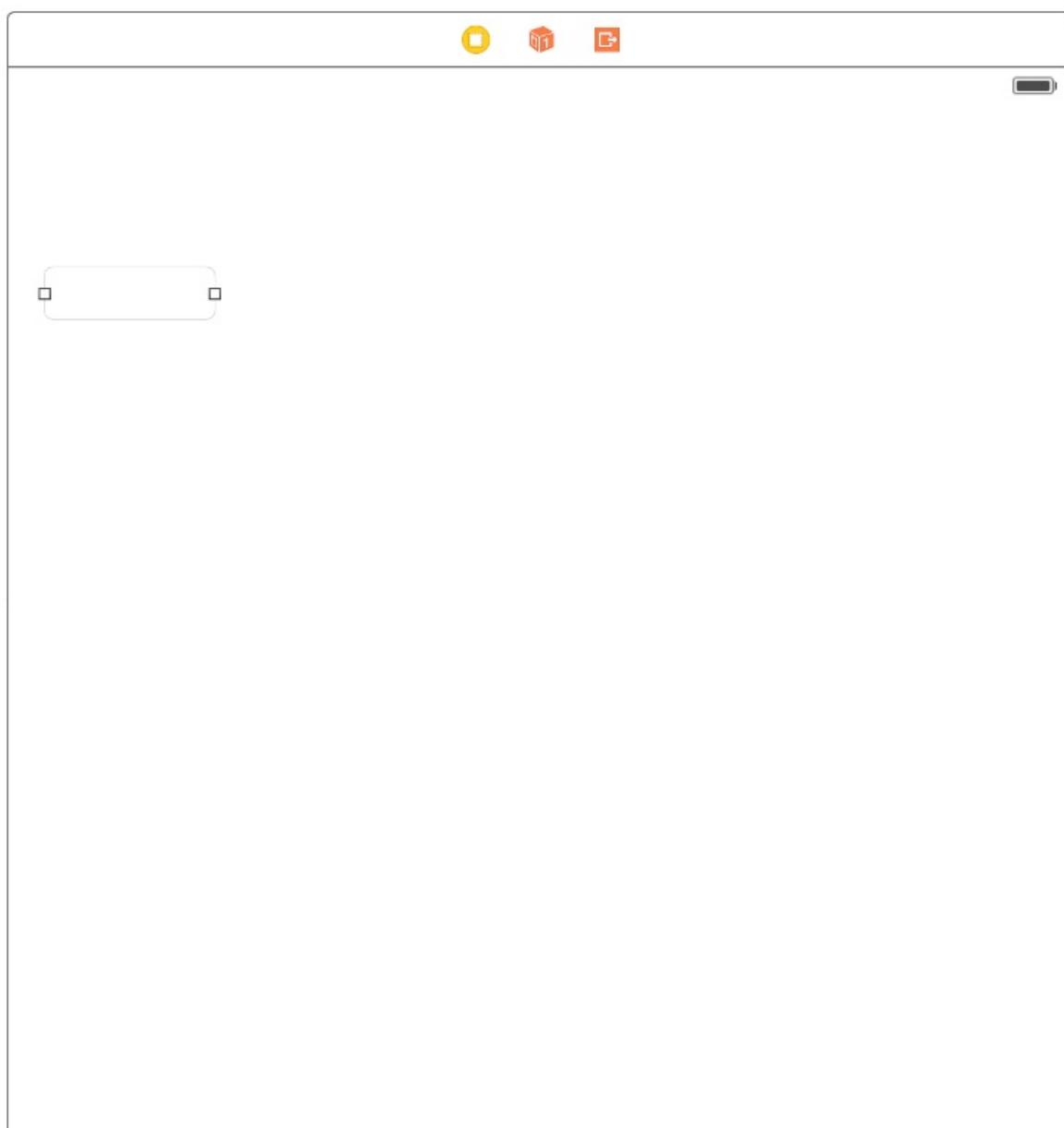
如果有必要，通过选择 `Editor > Canvas > Zoom` 来进行缩放。

4. 拖动文本框使其位于场景的上半部分，并使其和左边界对齐。当你看到如图所示的效果的时候，停止拖动。

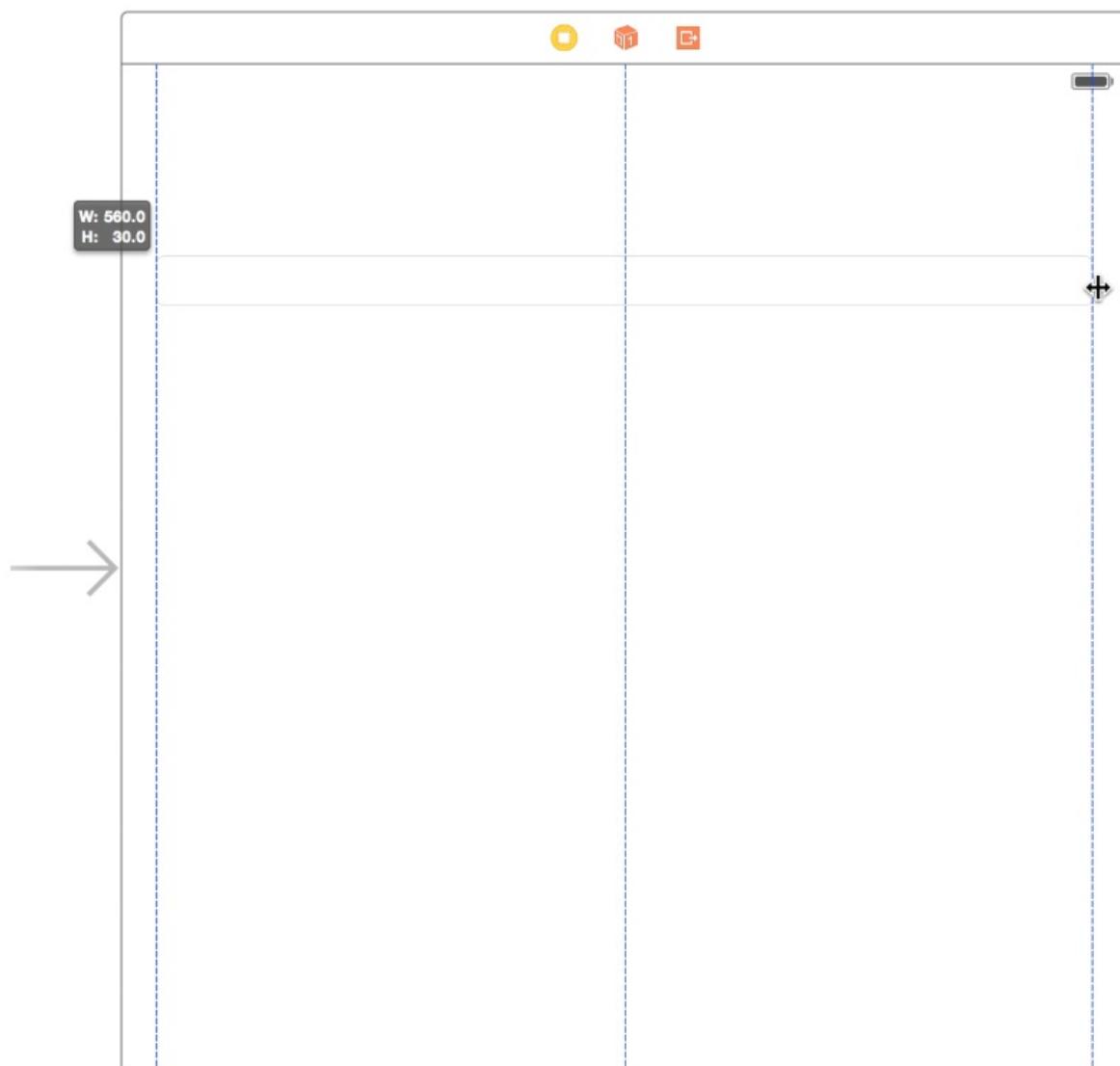


蓝色的布局辅助线能帮助你放置文本框。布局辅助线只有当你拖动或者改变对象尺寸的时候才会出现；当你释放了文本框的拖动的时候，这些辅助线会消失。

5. 如果有必要，可以点击文本框以显示用于调整尺寸的控制点。你可以通过拖动文本框的调整尺寸的控制点来实现对UI的尺寸调整。控制点位于元素的边框上，外观上是一些白色的小正方形。当你选择了文本框的时候，这些白色的小正方形就会出现。在这种情况下，文本框应该已经选上了，因为你刚刚才把它拖动完成。如果你的文本框像下面这个一样，你就可以对它进行调整大小了；否则，你应在画布上先选上它。



6. 调整文本框的左右边界，直到你看到三条垂直方向的布局辅助线：左对齐，水平居中对齐，右对齐。

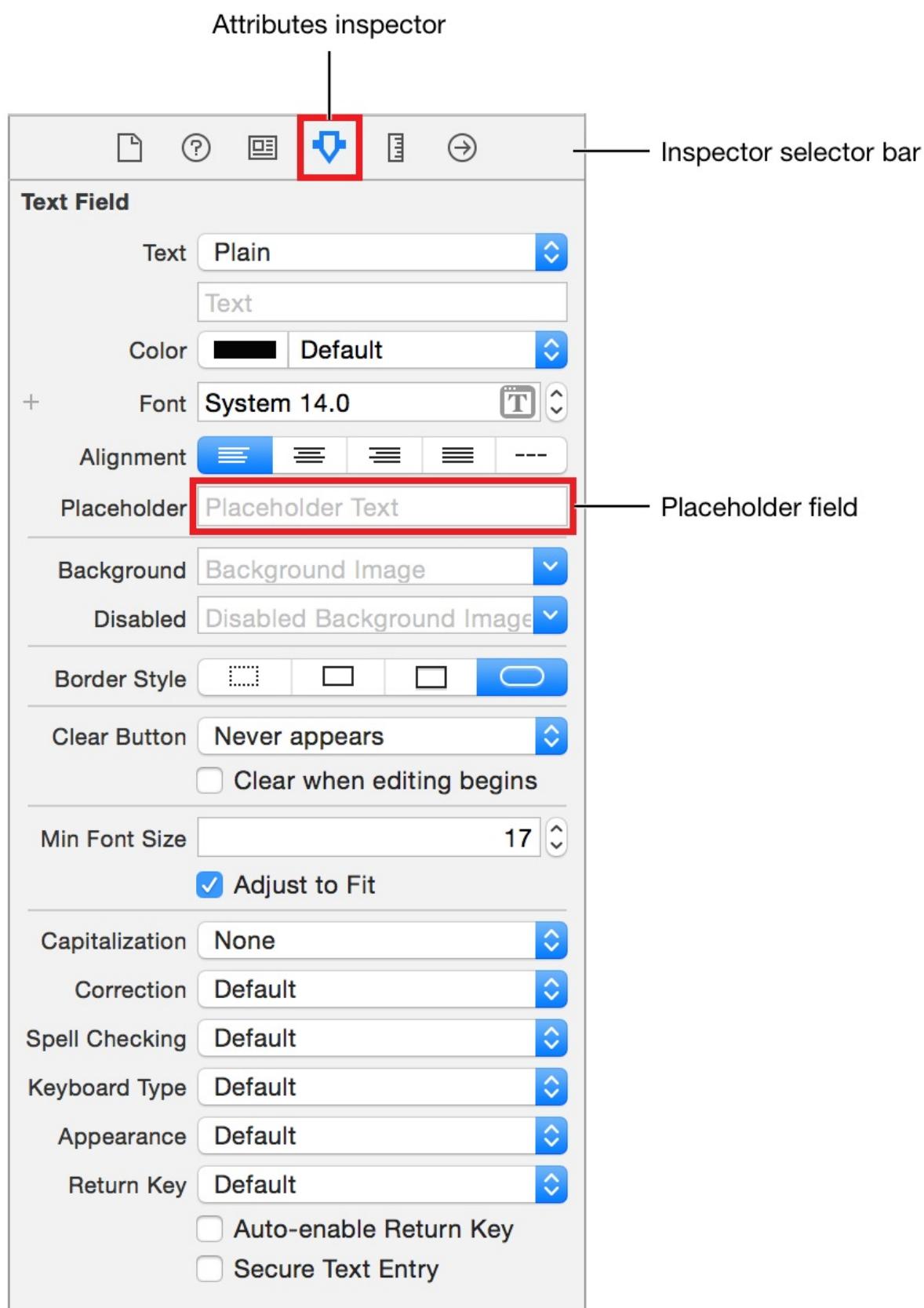


虽然你已经在场景中放置好文本框了，但是对于这个文本框应该输入什么内容，还没有任何的指示提供给用户。使用文本框的占位符（placeholder）可以提示用户输入一个新食谱的名字。

## 配置文本框占位符文本

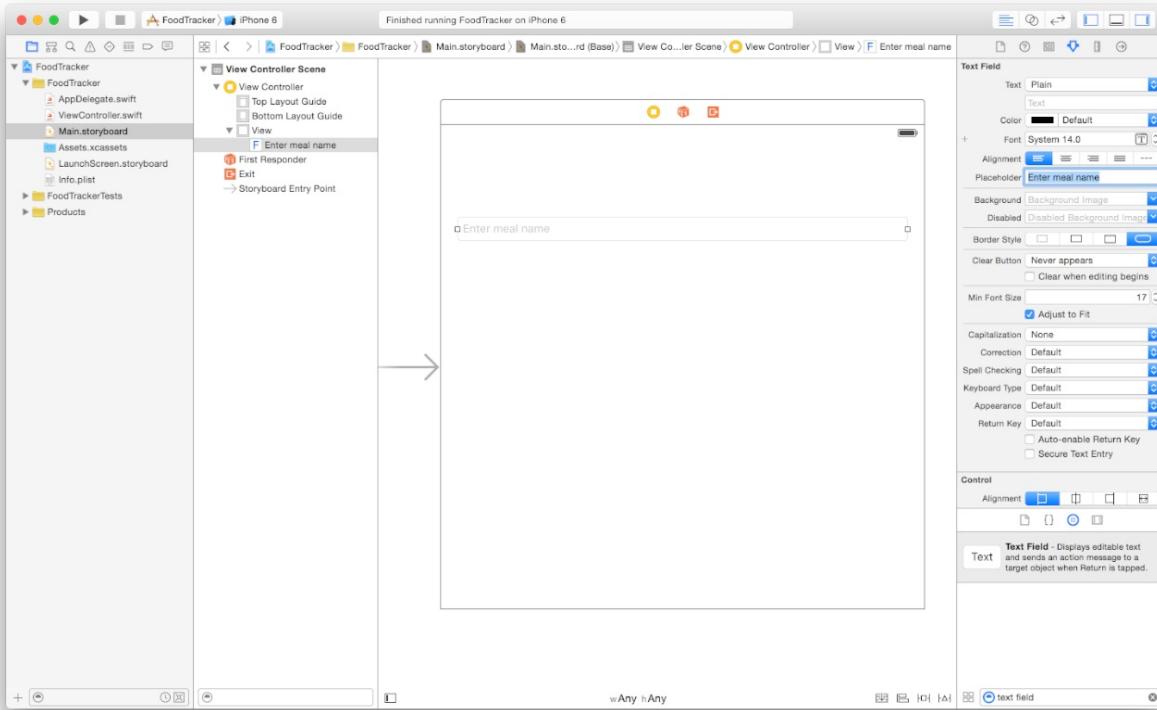
1. 选中文本框后，打开工具栏的属性检查面板。

当你点击了选项条中从左到右算起的第四个按钮之后，就会出现属性检查面板，它能让你编辑storyboard中对象的属性。



2. 在属性检查面板中，找到 Placeholder 标签的区域，输入“Enter meal name”。
3. 按下Return，在文本框中就会显示新的占位符文本。

现在，你的场景应该像下面这样：



当你在编辑文本框的属性的时候，你还可以在用户选择文本框的时候，更改系统默认键盘。

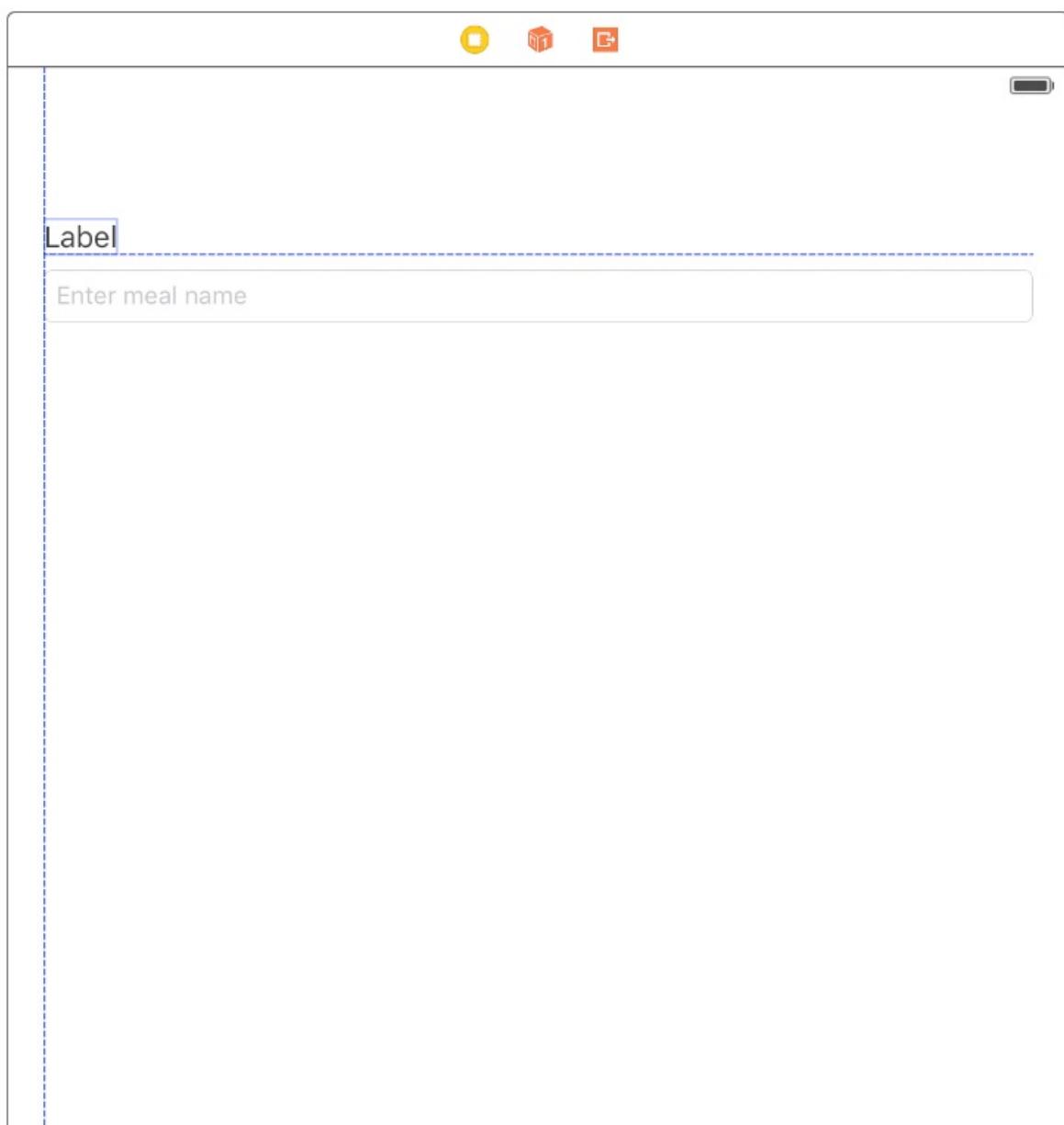
## 配置文本框对应的键盘类型

1. 确保文本框仍在选中状态。
2. 在属性检查面板中，找到Return Key标签所在的域，选择 Done。把它改成 Done 之后，这个更改能让键盘上的 Return 键更加突出。
3. 在属性检查面板，勾选上 Auto-enable Return Key。这个更改会让文本框在用户输入文本之前，禁用Done键，确保用户不会输入一个空的食谱名字。

接下来，添加一个标签(**UILabel**)到场景的上方。标签是没有交互的，它只是单纯地在UI中显示静态的文本。为了帮助你理解如何定义在UI中元素的交互，你将会让这个标签以使其显示用户在文本框中输入的内容。这是一个很好的方法，用以检测文本框正确地处理用户的输入。

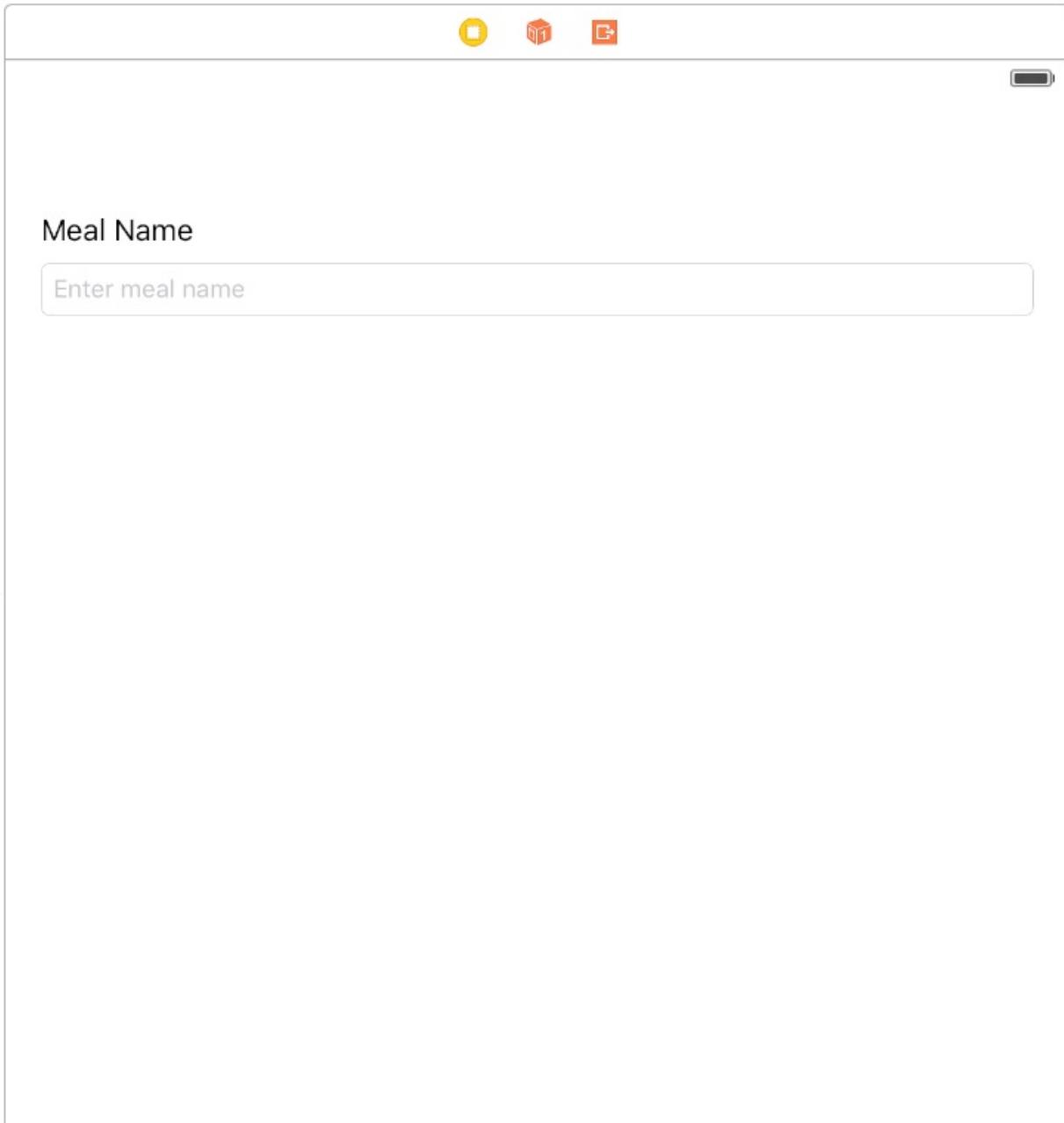
## 在你的场景中添加一个标签

1. 在对象库的搜索框中，输入label以快速查找Label对象。
2. 从对象库中，拖动label对象到你的场景中。
3. 拖动标签使其位于文本框的上方并使其向左对齐。当你看到像下面这样的效果时，停止拖动。



4. 属性这个标签，并输入**Meal Name**。
5. 按下**Return**键，标签中就会显示新的文本。

你的场景应该像下图这样：

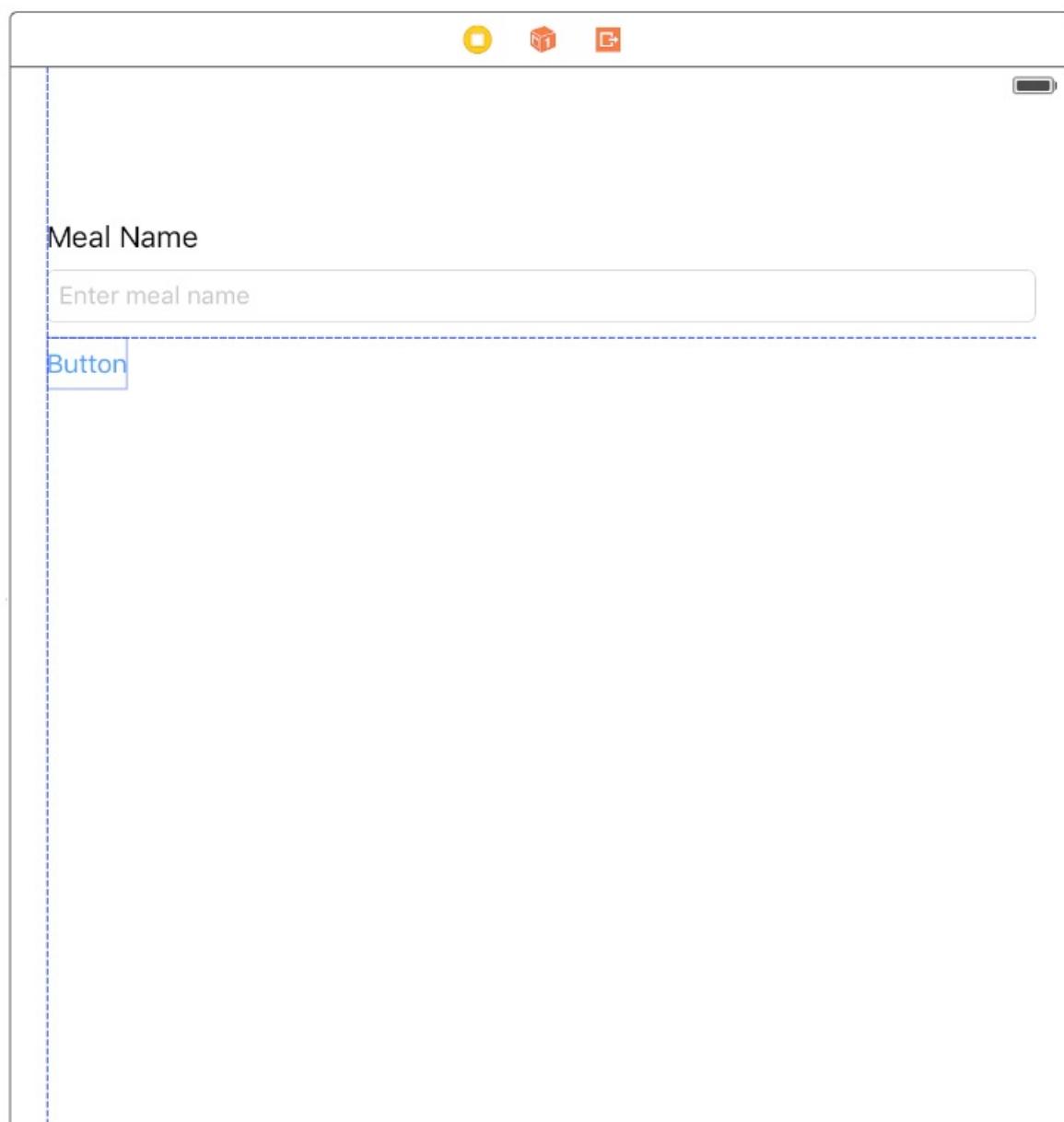


现在，在你的UI中添加一个按钮(`UIButton`)。按钮是可交互的，所以用户可以点击它并触发你定义的行为。稍后，你将会创建一个行为用以重设标签的值为默认值。

## 在你的场景中添加一个按钮

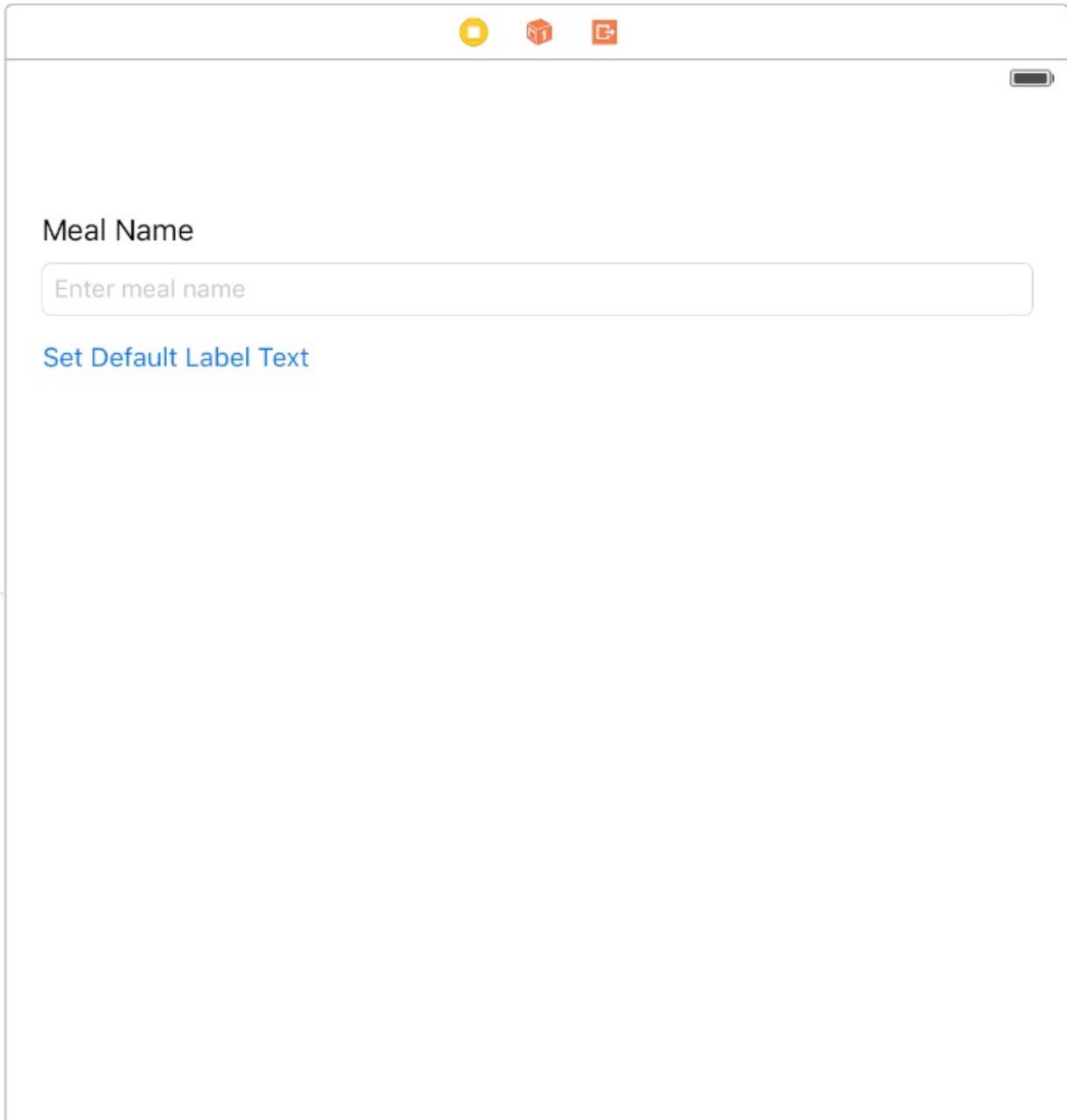
1. 在对象库的搜索框中，输入 `button` 以快速地查找Button对象。
2. 从对象库中拖动一个Button对象到你的场景中。
3. 拖动该按钮对象，使其位于文本框的下方，并使其左对齐。

当你看到像下面这样的效果时，停止拖动。



4. 双击该按钮，并输入Set Default Label Text。
5. 按下Return键，按钮中就会显示新的文本。

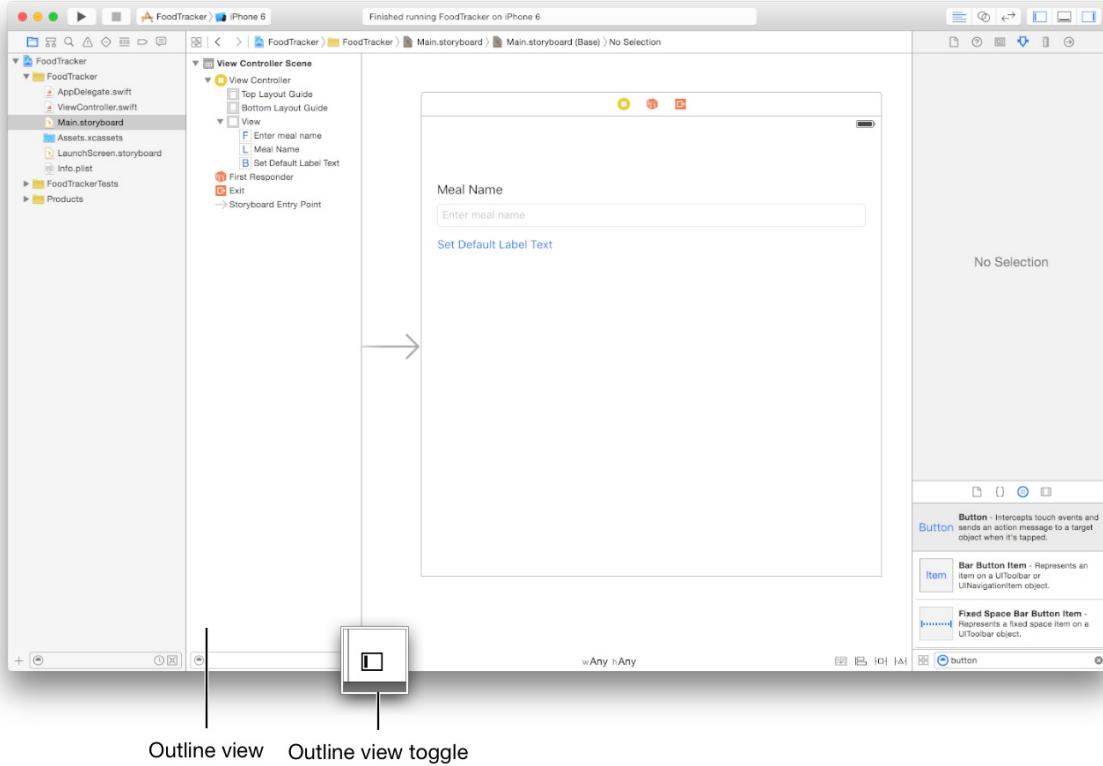
至此，你的场景应该像下面这样：



理解你在场景中添加的元素如何布局是很有用的。通过结构图，查看你在场景中所添加的UI元素。

## 查看结构图

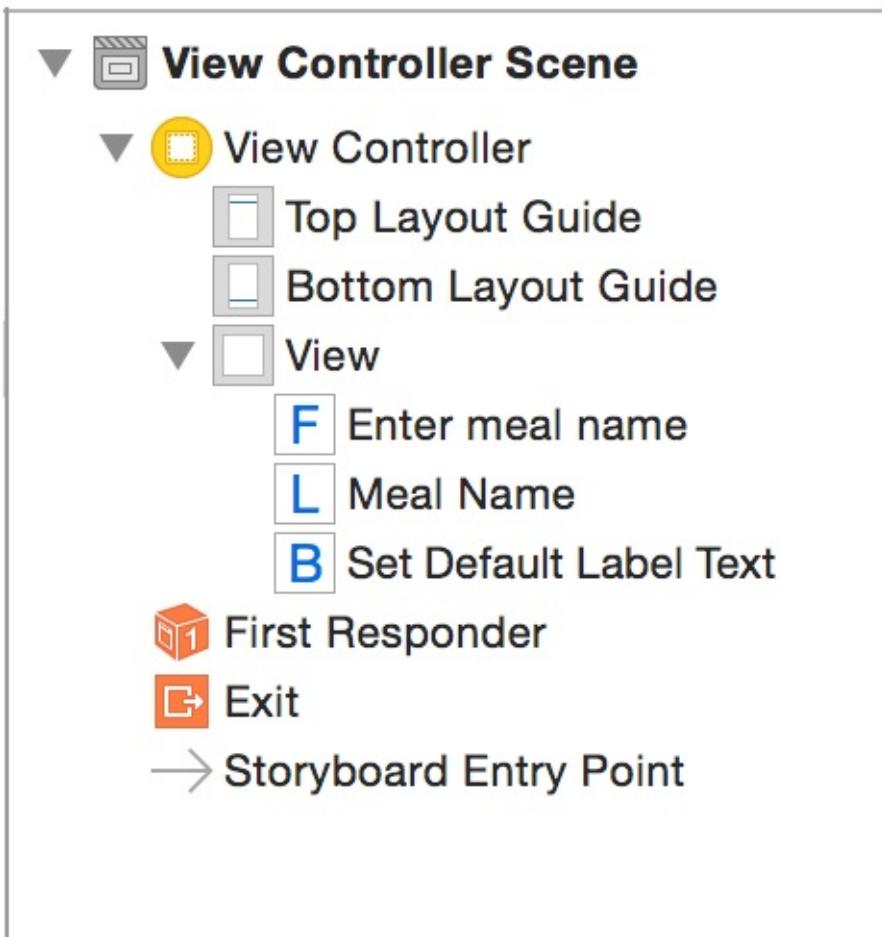
1. 在你的storyboard中，找到结构视图切换按钮。



- 如果结构视图已经折叠起来，可以点击切换按钮以展开结构视图。你可以根据需要，通过这个切换按钮以折叠或展开结构视图。

结构视图位于画布的左边，它展示了你的storyboard中对象的结构层次。你应该能看到刚刚添加到storyboard中的文本框，标签，和按钮。但是，为什么你添加到其中的元素被包含在view中？也就是说，在另外的一个视图中？

视图不仅仅在屏幕上显示它们自身，它们还可以响应用户的输入，可以作为其它视图的容器。视图在层次结构中的排列称为视图层次（view hierarchy）。视图层次定义了视图与视图之间的关系。在视图层次内部，被另外一个视图包含的视图称为子视图（subview），而包含别的视图的视图称为这些视图的父视图。一个视图可以有多个子视图，而只能拥有一个父视图。



通常来说，每一个场景会有自己独有的视图层次。每一个视图层次的上方是内容视图（content view）。在上图的场景中，内容视图是一个View，它是视图控制器的顶层视图。文本框，标签和按钮是内容视图的子视图。你放置到场景中所有其它的视图都将会是这个内容视图的子视图（尽管它们本身可以被包含在其它子视图中）。

## 预览界面

周期性地预览你的app，确保所有的東西都像你期望的那样。你可以使用辅助编辑器（assistant editor）预览app界面。辅助编辑器是一个备用的编辑器，其跟主界面并排地显示。

### 预览界面步骤

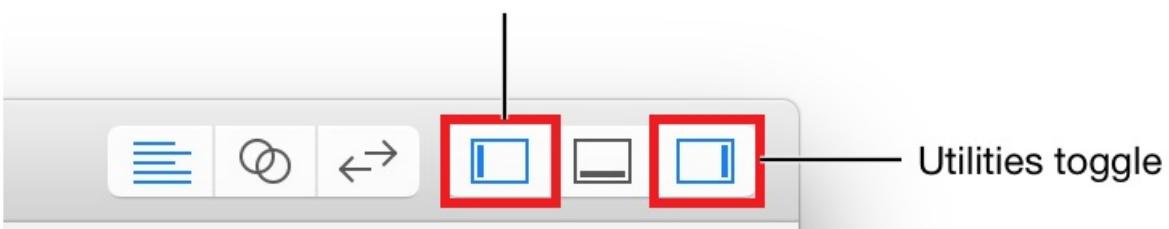
1. 在Xcode中，点击位于右上角工具条中的Assistant按钮，以打开辅助编辑器。

## Assistant editor



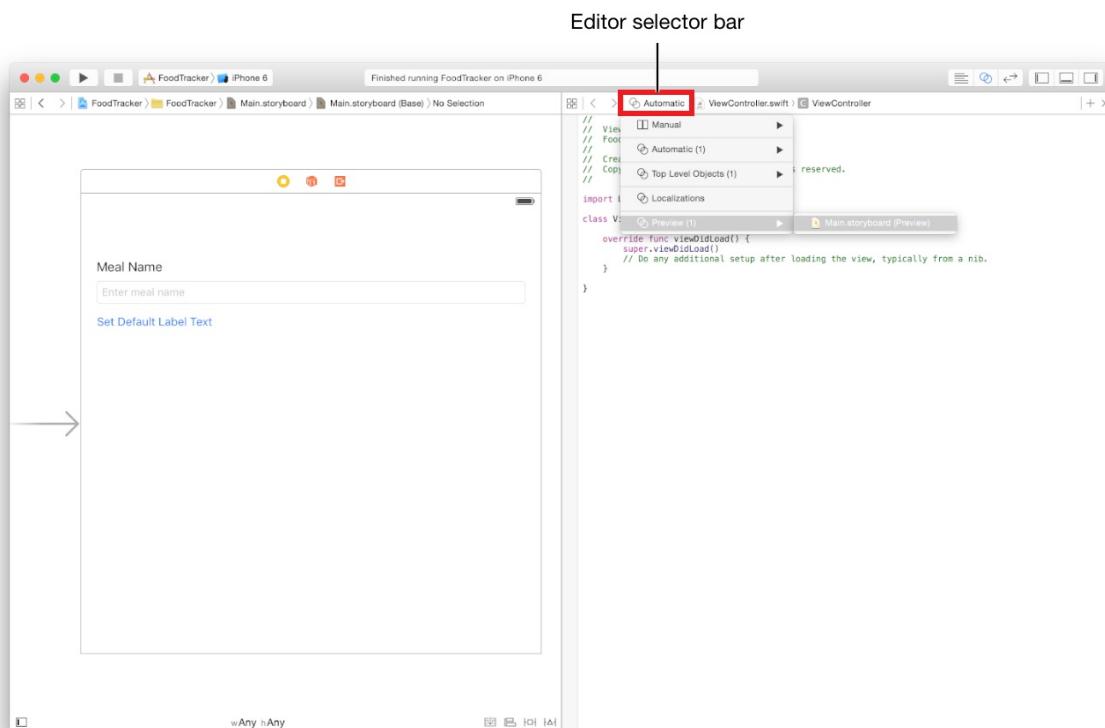
2. 如果需要更多的工作空间，你可以通过点击在Xcode工具栏中的Navigator和Utilities按钮隐藏工程导航栏和工具区。

### Navigator toggle

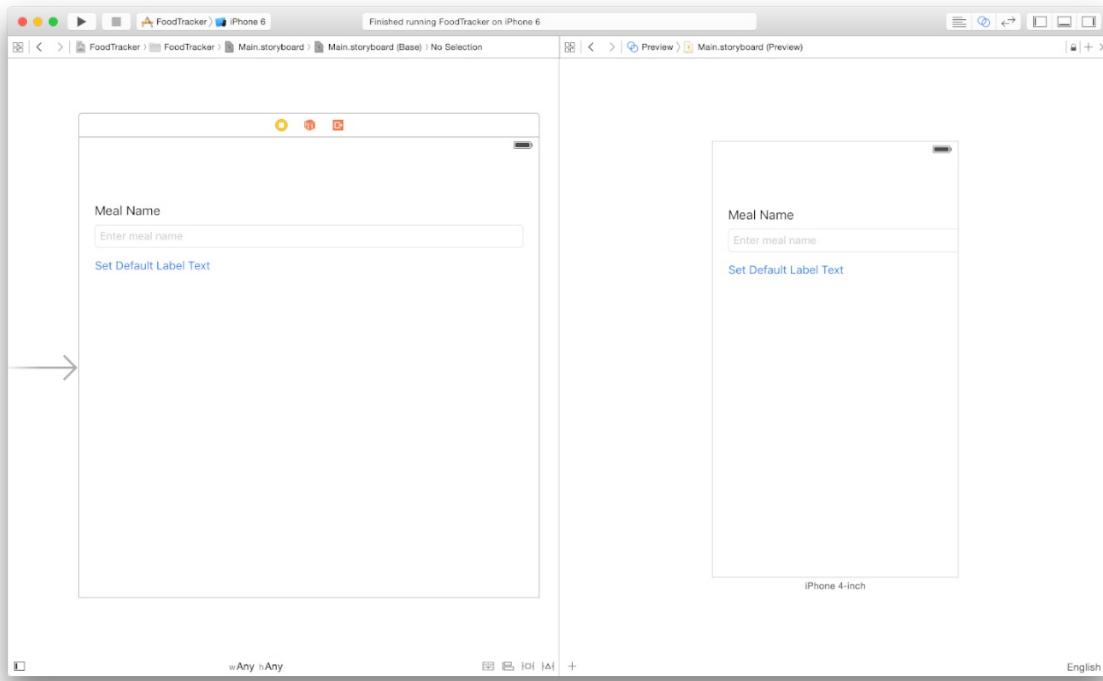


除此之外，你还可以将轮廓视图隐藏。

3. 在位于辅助编辑器上方的编辑器选项条中，把辅助编辑器从Automatic切换到Preview > Main.storyboard。



如你在辅助编辑器中看到的，这个文本框看起来不大准确，它向右溢出了边界。但是，你在storyboard中看到的界面是正常的，为什么在iPhone的预览下就不正常了呢？



在这之前，你已经知道你其实创建的是一个自适应的界面，这个自适应的界面会在不同尺寸的iPhone和iPad下进行缩放。你在storyboard中看到的默认的场景展示的是你所设计的界面的通用版本。这里，你将需要指定界面在不同的屏幕尺寸下应该如何进行自适应。例如，当界面缩小到一个iPhone尺寸时，这个文本框应该也同时缩小。当界面放大到一个iPad的尺寸时，这个文本框也应该随着放大。你可以通过自动布局（Auto Layout），很简单地指定这些种类的界面。

## 采用自动布局(Auto Layout)

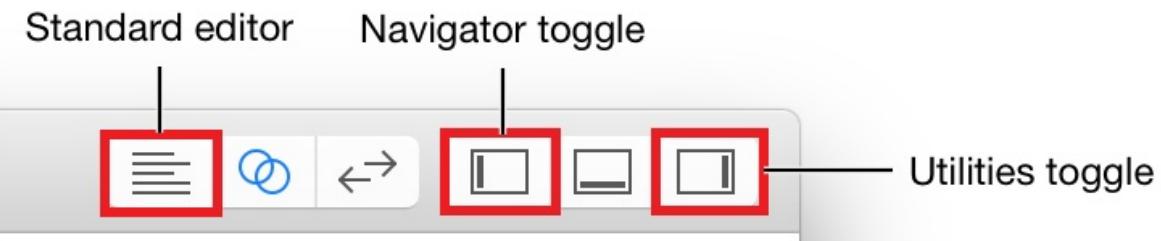
自动布局是一个强大的布局引擎，它能帮助你毫无压力地设计自适应的布局。你只需要为场景中需要定位的元素描述你意图，让布局引擎决定如何最完美地实现你的意图。你将会使用约束（constraints）表达你的意图，而约束是一些布局规则，其能解释一个元素相对于另一个元素如何定位，描述尺寸大小，或者当某些东西减少了两个元素的可用空间的时候，这两个元素应该如何缩小。

和自动布局相伴而生的其中一个非常有用的工具是堆视图（stack view，UIStackView），它能辅助你处理自动布局。堆视图提供一个流线型的界面，用以在一个行或者列上布局一个视图的集合。堆视图能让你利用自动布局的能力，创建能动态适应设备方向，屏幕尺寸和任何可用空间的改变的用户界面。

你可以很简单地把你现有的界面包在一个堆视图中，通过添加必要的约束使堆视图在不同的情形下正确地显示。

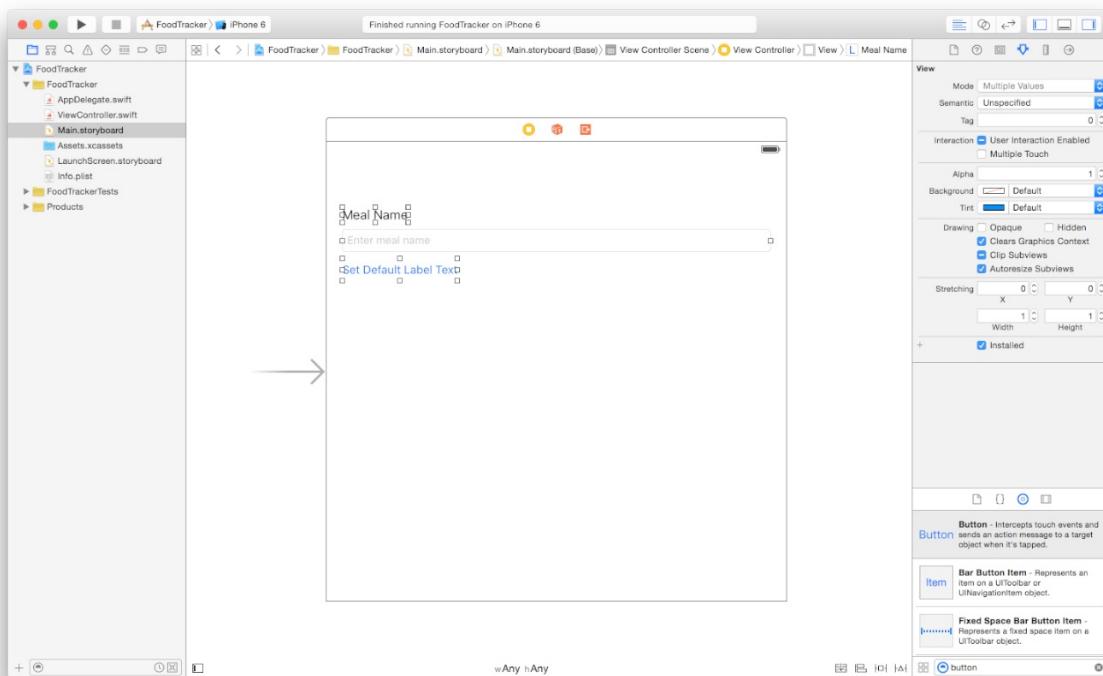
在食谱场景中添加自动布局约束

1. 通过点击Standard按钮返回到标准编辑器中。

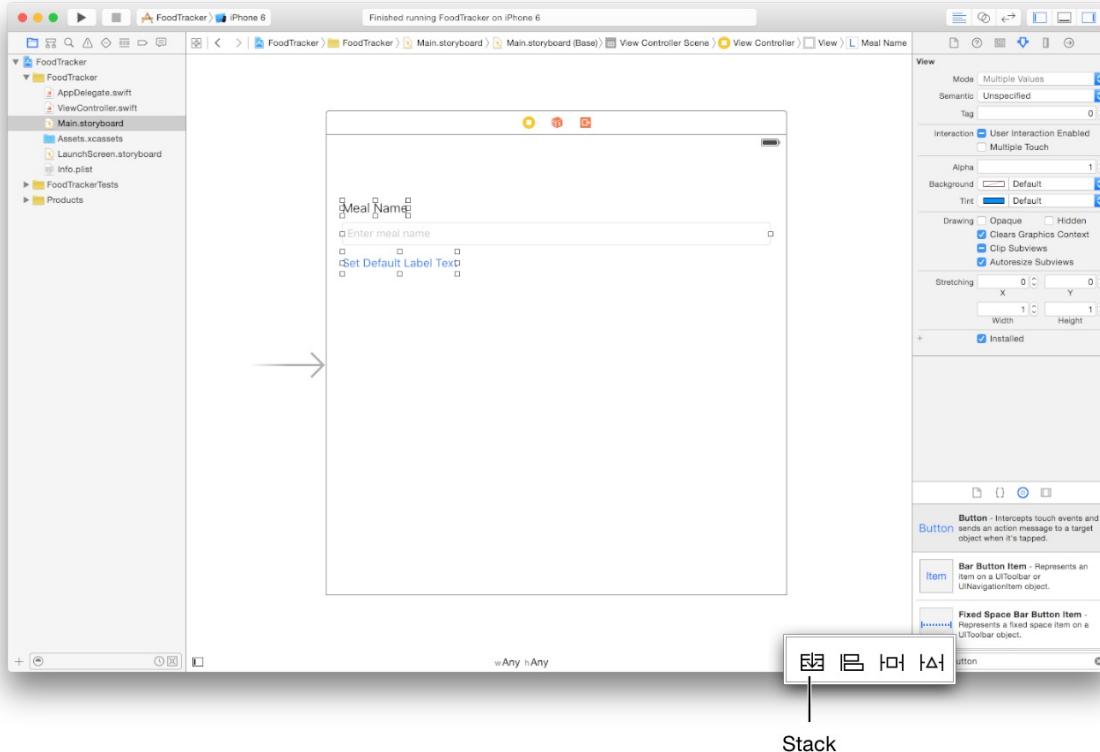


通过点击在Xcode工具栏中的Navigator和Utilities按钮，展开工程导航和工具区。

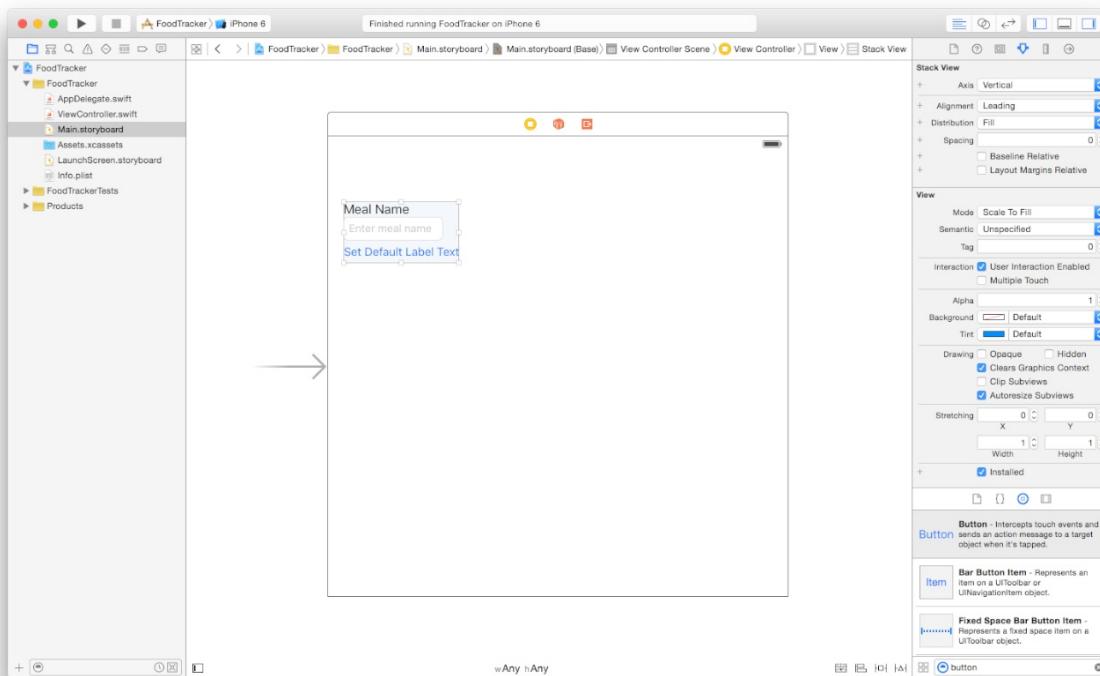
2. 在你的键盘中按下Shift键的时候，选择文本框（text field），标签（label）和按钮（button）。



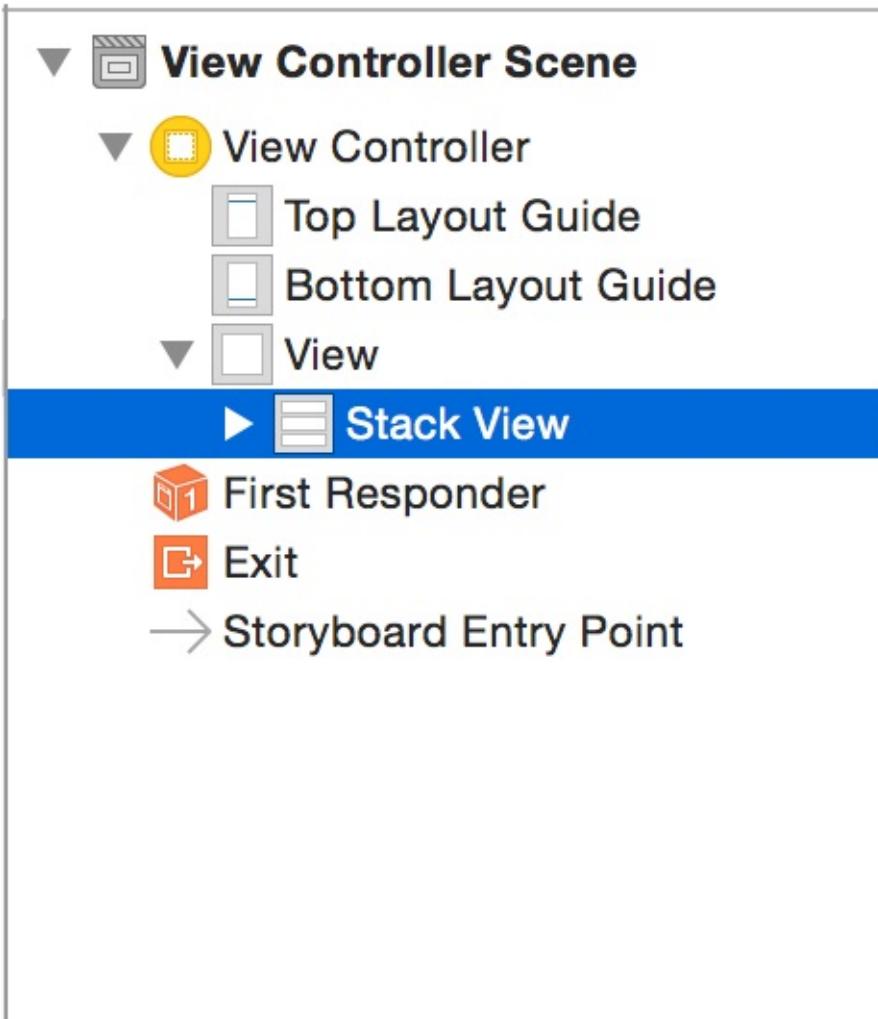
3. 在画布的右下角，点击 Stack 按钮。（除此之外，也可以选择 Editor > Embed In > Stack View）。



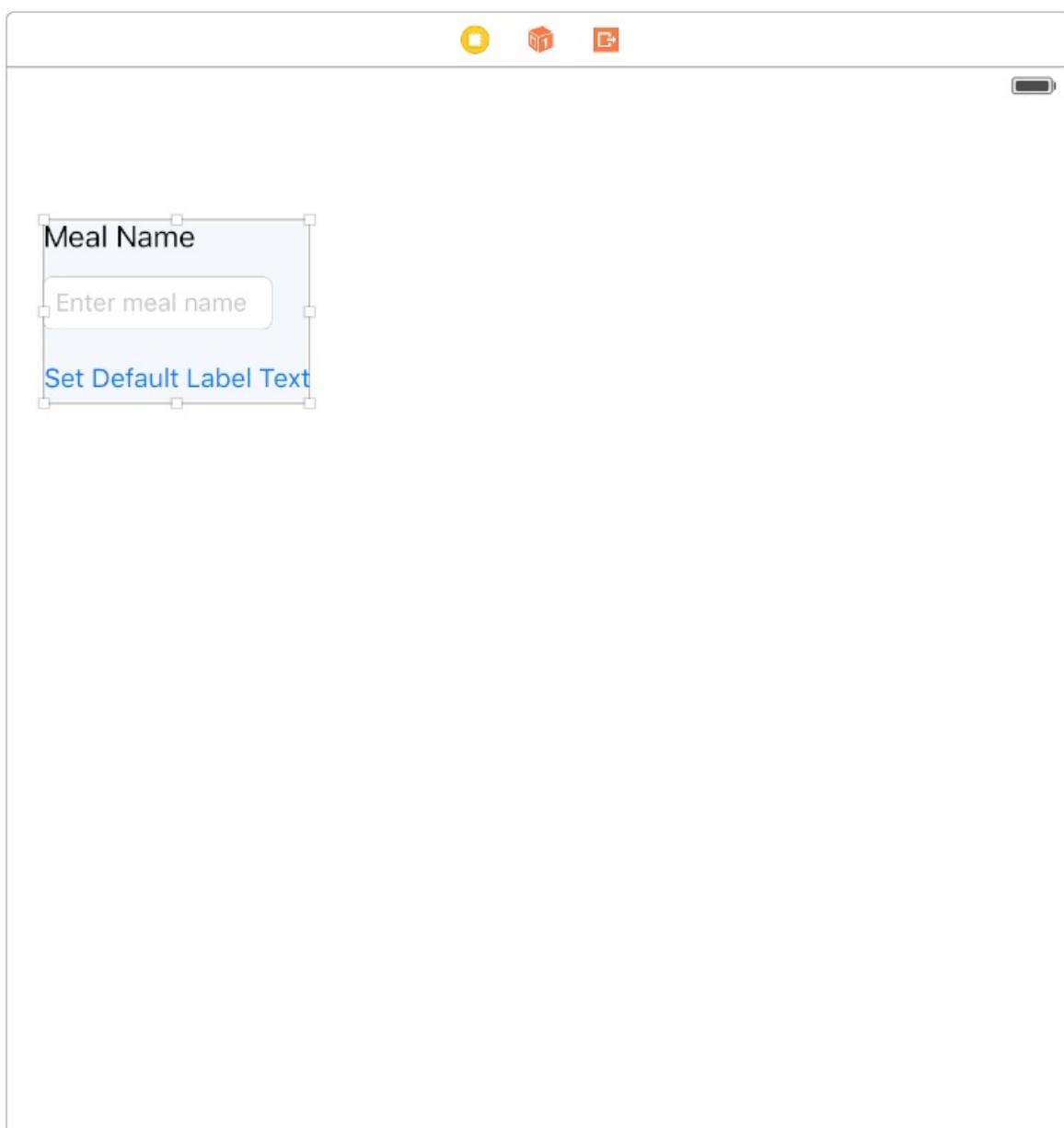
Xcode会把这些UI元素包在一个堆视图中，把它们堆在一起。Xcode会分析你现有的布局，计算出这些元素应该在纵向堆叠，而不是横向。



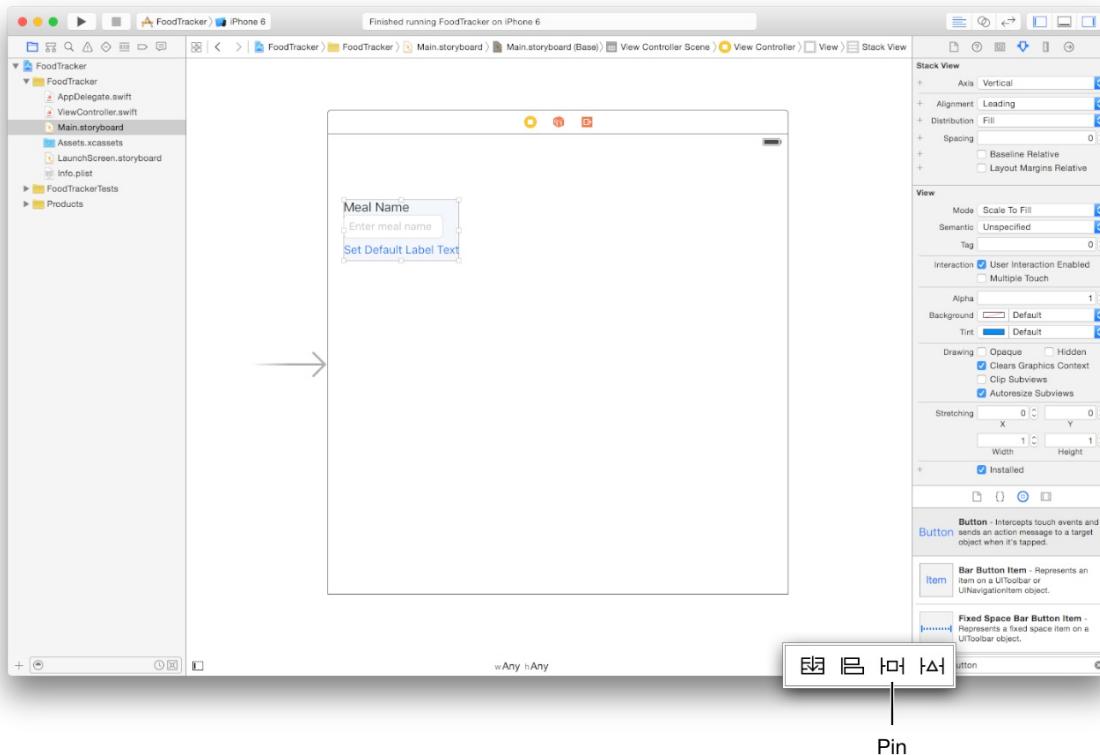
4. 如果有必要，可以打开轮廓视图，然后选择 Stack View 对象。



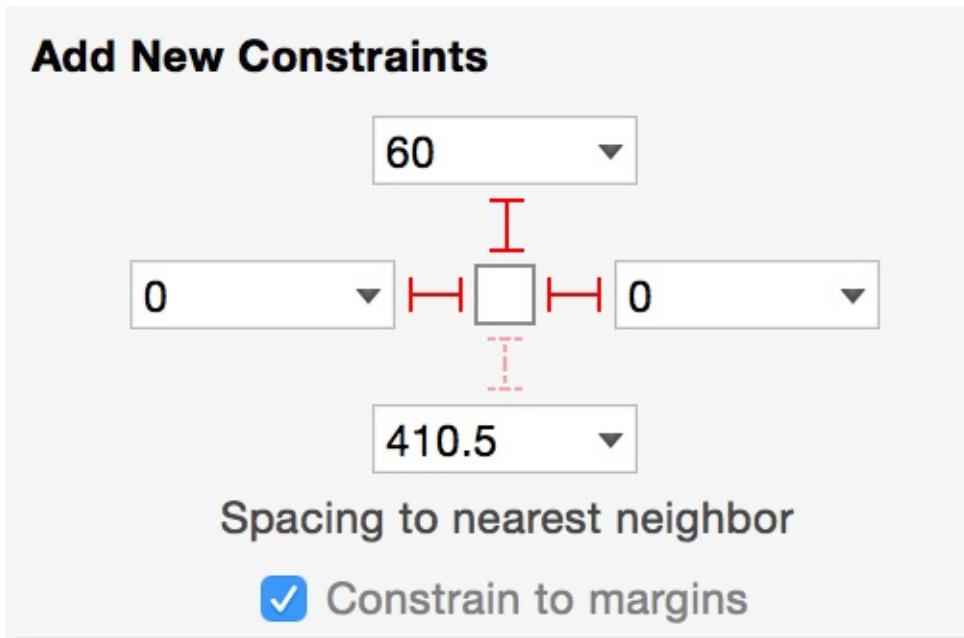
5. 在属性检查器中，在 Spacing 框中输入12，然后按下回车。你将会看到这些UI元素在纵向铺展开，而这个堆视图也同样被放大了。



6. 点击位于画布右下方的Pin，打开Pin菜单。

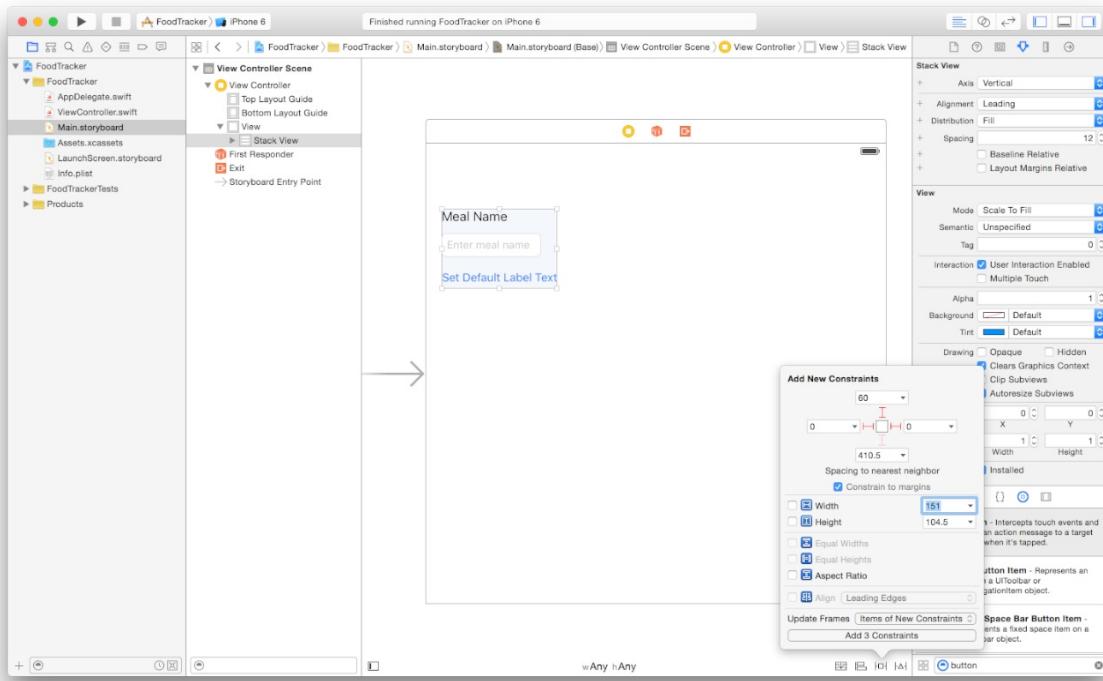


- 在“Spacing to nearest neighbor”上方，通过点击水平的两个和上面纵向的一个约束以选上它们。选上后，它们会变成红色。



这些约束表明了相对于左、右和顶部近邻元素的间距。在这个上下文中，近邻的情况表示最接近自身的元素的边界，这可能会是父视图，同级视图，或者是一个边距。因为“Constrain to margins”选框已经选上了，所以这个堆视图在这种情况下将会约束它的父视图的左、右和上边距，从屏幕的边界中预留少量的空间。

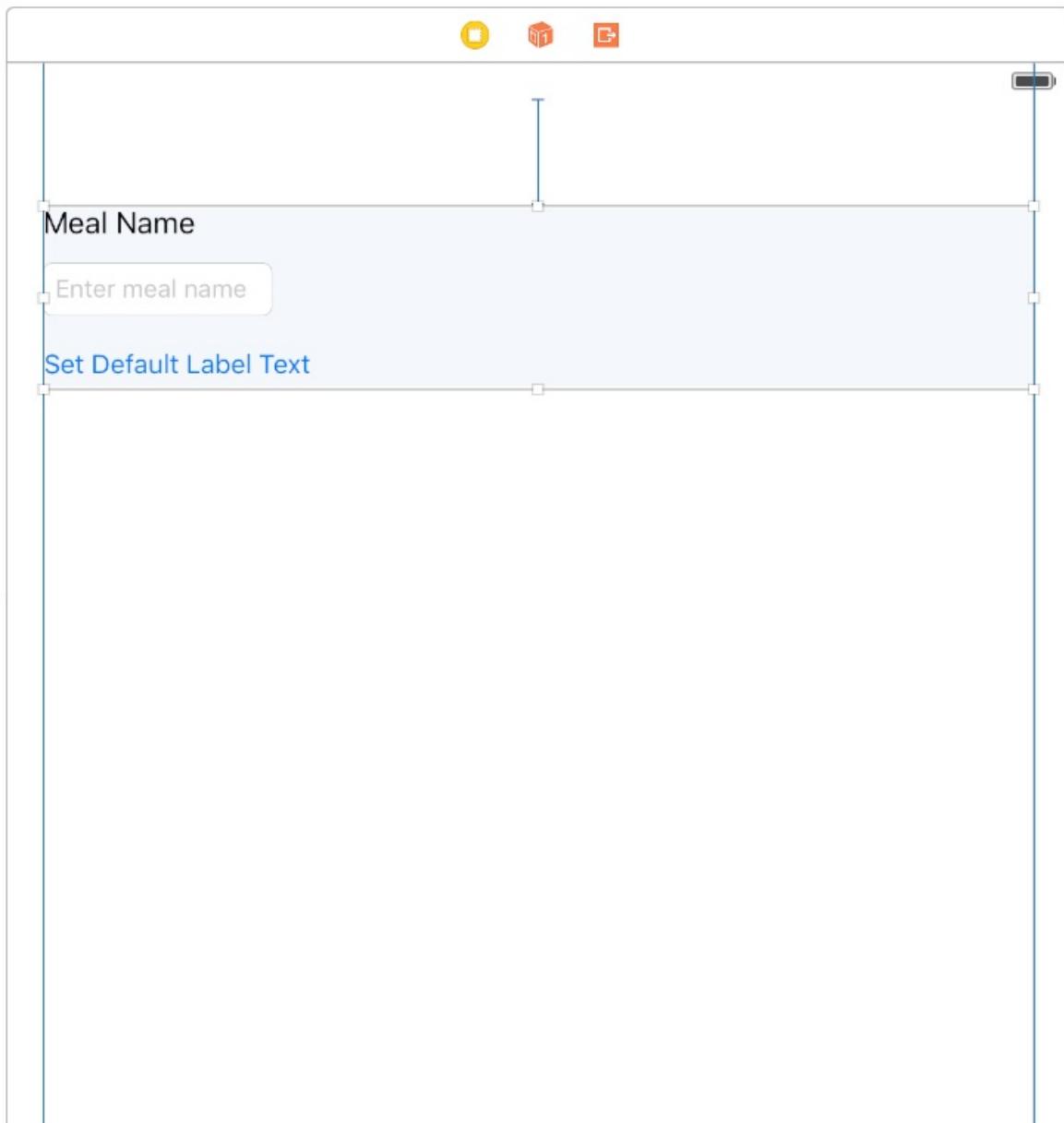
- 在左右框中分别输入0，在上方框中输入60。
- 在弹出的Update Frames旁边，选择Items of New Constraints。  
这个Pin菜单看起来应该像这样：



10. 在Pin菜单中，点击Add 3 Constraints按钮。

Add 3 Constraints

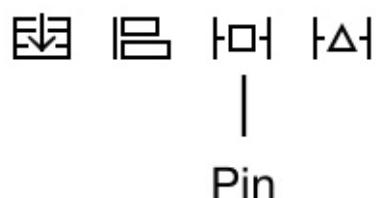
这个食谱UI看起来应该像这样：



你会发现文本框没有像之前那样伸展到场景的边缘，所以你稍后将会修复这个问题。

#### 调整食谱场景中的文本框宽度

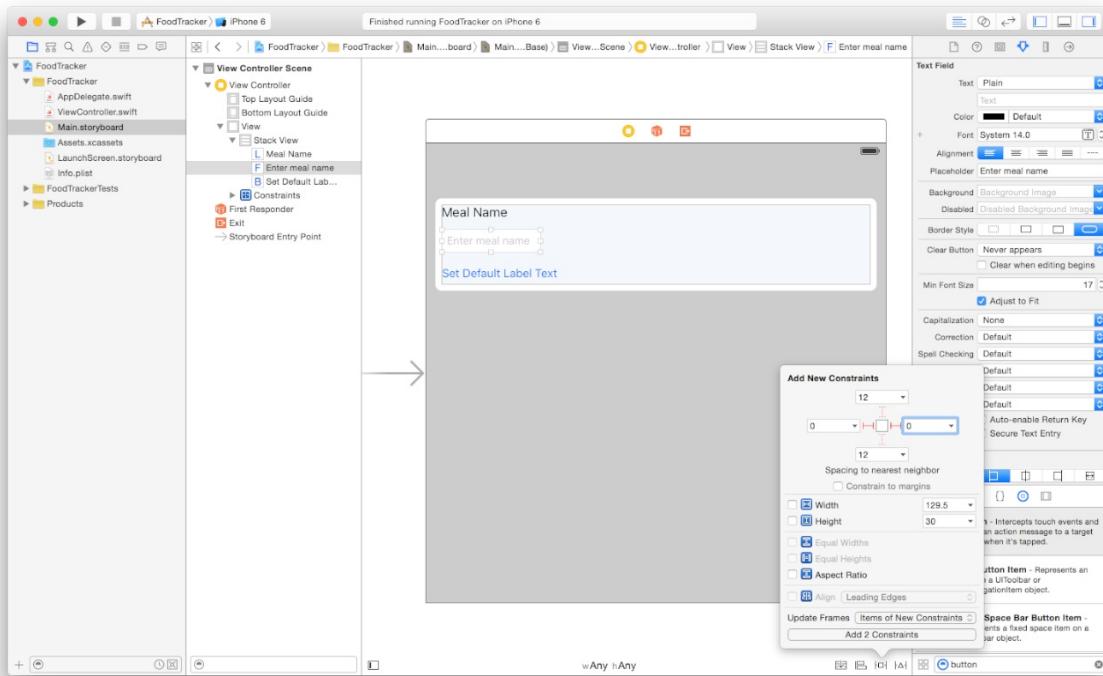
1. 在你的storyboard的食谱场景中，选上文本框。
2. 在画布的右下方，打开Pin菜单。



3. 在“Spacing to nearest neighbor”中，点击两个水平的约束以选上它们。选上后，它们会变成红色。
4. 在左右两个框中，输入0。

5. 在弹出的Update Frames旁边，选择Items of New Constraints。

这个Pin菜单看起来应该像这样：

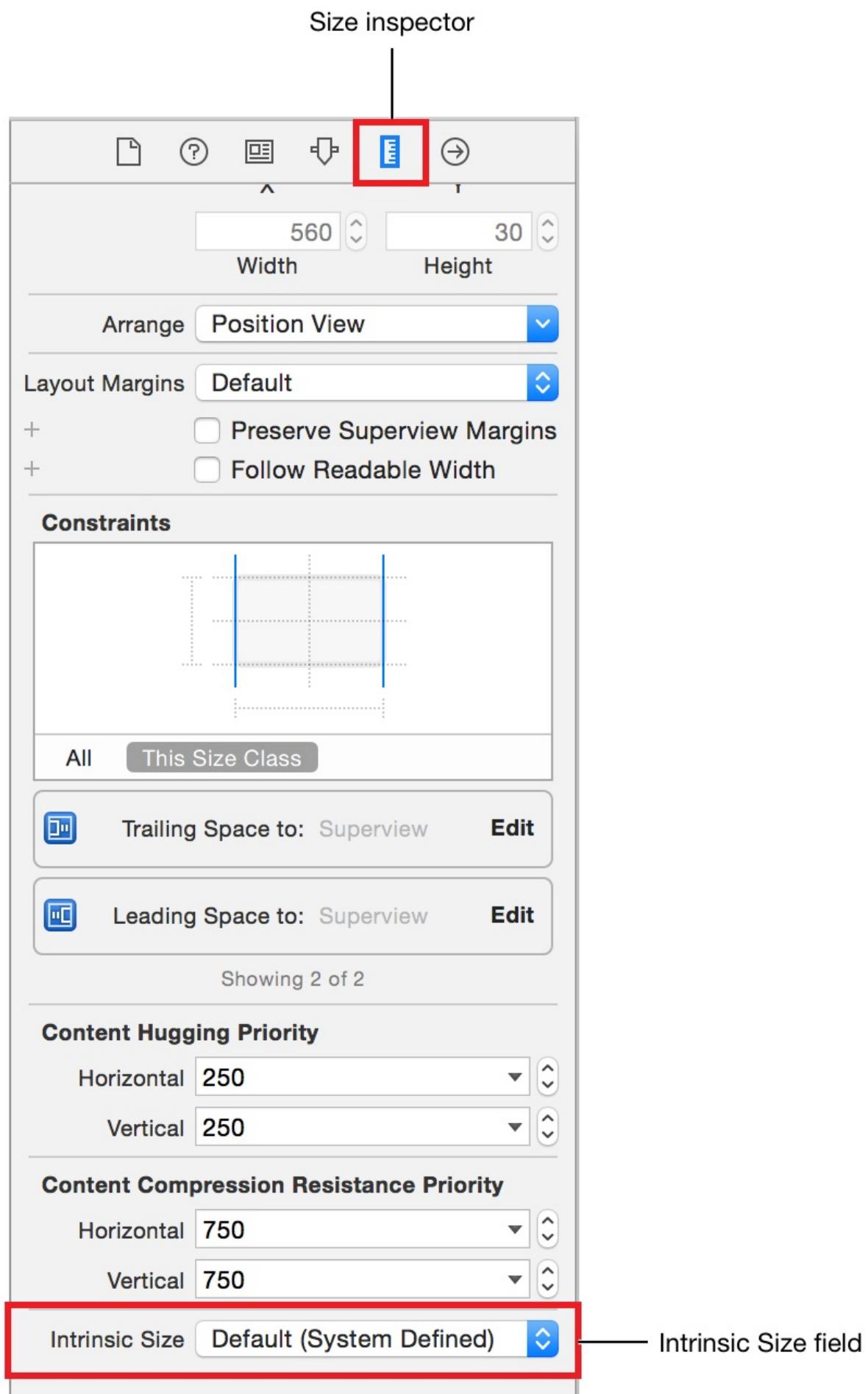


6. 在Pin菜单中，点击Add 2 Constraints按钮。

Add 2 Constraints

7. 选上文本框后，在工具区中打开尺寸检查（Size inspector）。

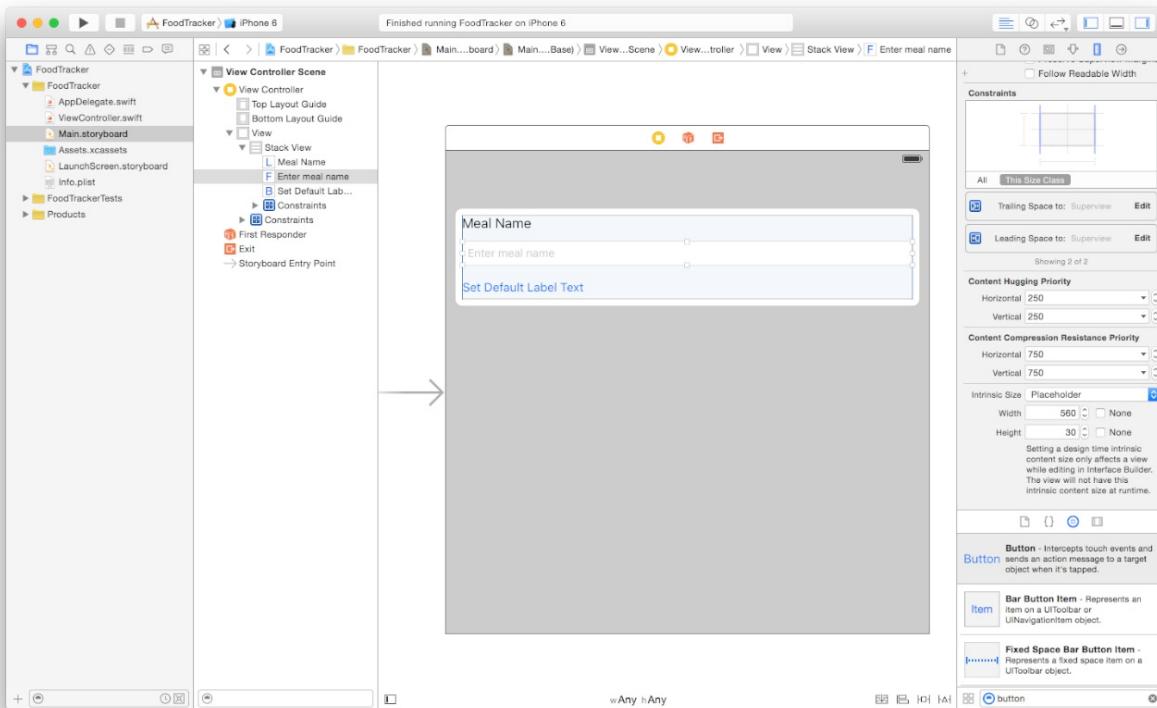
当你选上检查器选条中的第五个按钮后，尺寸检查器将会出现。它能让你编辑storyboard中对象的尺寸和位置。



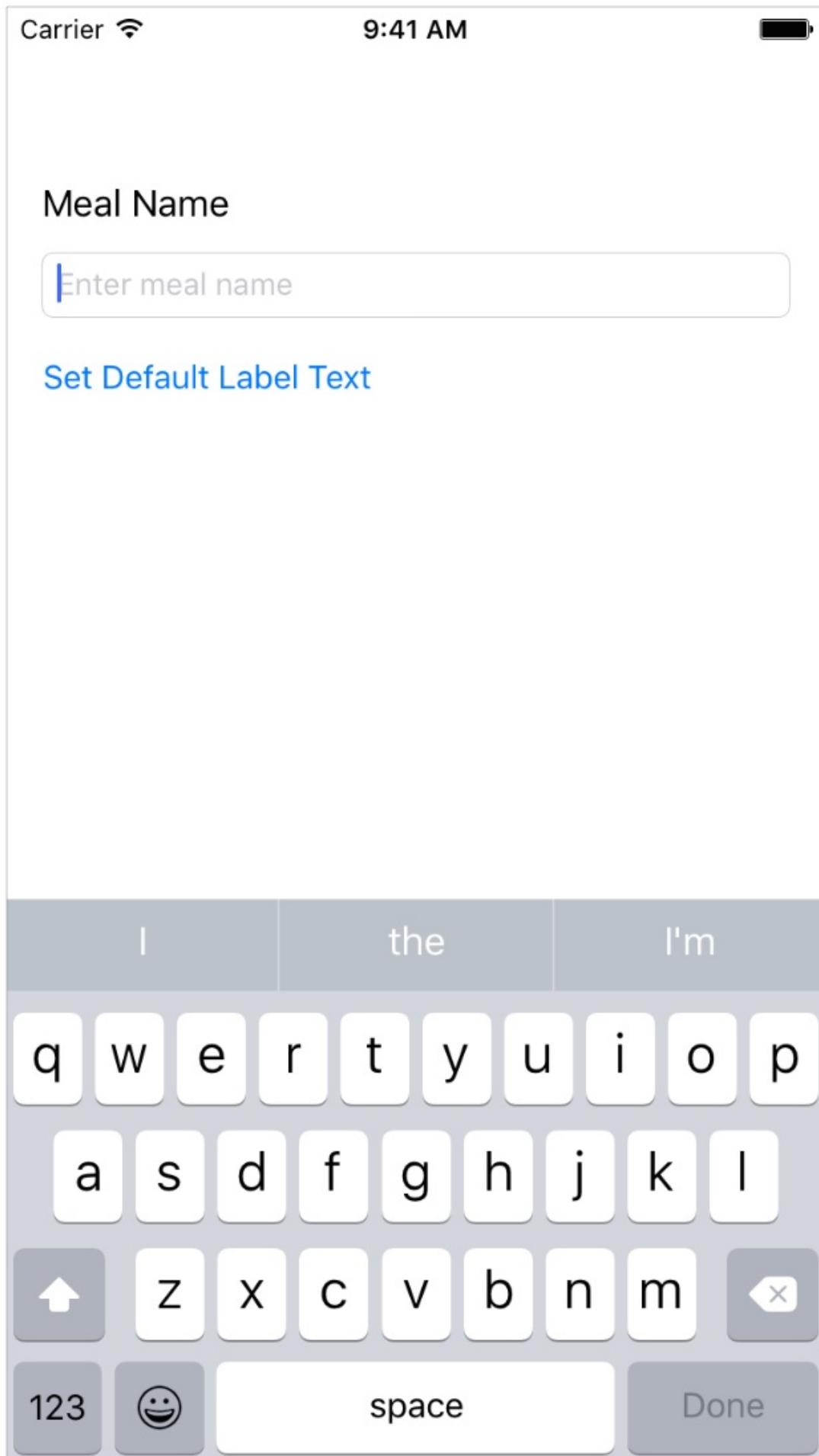
8. 在 Intrinsic Size 框中，选择 Placeholder。（这个框位于尺寸检查器的最下方，所以你需要滚到下方。）

文本框是基于它们自身的内 容确定大小的，这些内容定义了它们的固有的内 容尺寸（**intrinsic content size**）。固有的内 容尺寸表明在视图中显示所有内容需要的最小尺寸。如果你需要为不同的尺寸设计一个UI，你可以指定UI元素一个占位的固有的内 容尺寸。现在，这个文本框的内容是它的占位符文本，但是用户输入的实际文本可能比这个更长。

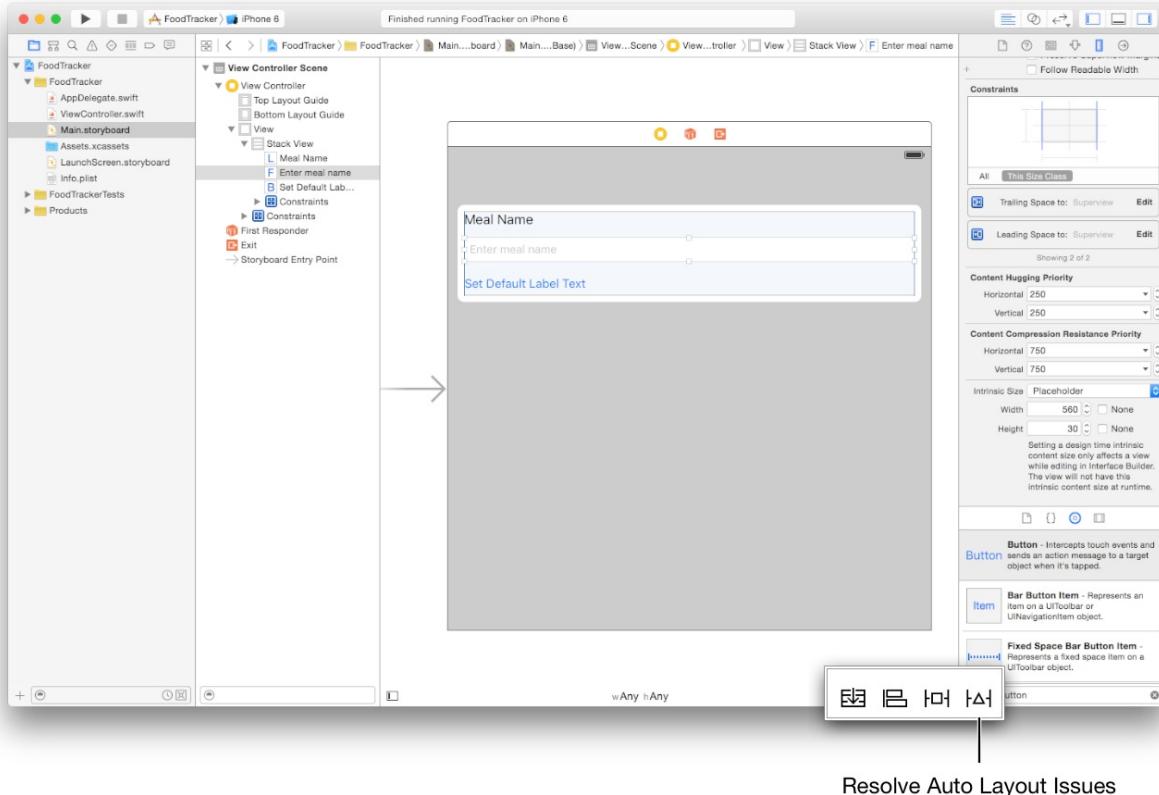
现在这个食谱场景的UI看起来应该像这样：



检验成果：在模拟器中运行你的app。这个文本框应该不会溢出屏幕的边界了。你应该可以点击文本框并使用键盘输入文本（如果你喜欢，可以通过按下Command+K弹出软键盘）。如果你翻转设备（command + 左箭头或者command + 右箭头），或者在一个不同的设备中运行app，文本框将会根据设备的方向和屏幕大小放大或者缩小文本框。注意，在横屏的时候，状态条会消失。



如果你没有得到预料中的结果，可以使用自动布局调试功能来帮助你。点击“Resolve Auto Layout Issues”图标，然后选择“Reset to Suggested Constraints”让Xcode通过一个有效的约束集来更新你的界面。或者点击“Resolve Auto Layout Issues”然后选择“Clear Constraints”以移除UI元素上所有的约束，之后再尝试一遍上面的步骤。



虽然这个场景还没有太多东西，但是已经搭建了基础的用户界面和功能了。确保你的布局在一开始就是健壮和可拓展的，以保证你有一个坚实的基础来继续添砖加瓦。

# 第四章：将**UI**与代码连接