

# Introduction to iOS development with Swift

## Lesson 5



**Adrien Humilière**  
Trainline

[adhumi+dant@gmail.com](mailto:adhumi+dant@gmail.com)



- Protocols
- App anatomy and life cycle
- Model View Controller
- Scroll views
- Table views

# Protocols



# Protocols

- Defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality
- Swift standard library defines many protocols, including these:
  - `CustomStringConvertible`
  - `Equatable`
  - `Comparable`
  - `Codable`
- When you adopt a protocol, you must implement all required methods.

# CustomStringConvertible

# Printing with CustomStringConvertible

```
let string = "Hello, world!"  
print(string) // Hello, world!
```

```
let number = 42  
print(number) // 42
```

```
let boolean = false  
print(boolean) // false
```

# Printing with CustomStringConvertible

```
class Shoe {  
    let color: String  
    let size: Int  
    let hasLaces: Bool  
  
    init(color: String, size: Int, hasLaces: Bool) {  
        ...  
    }  
}  
  
let myShoe = Shoe(color: "Black", size: 12, hasLaces: true)  
print(myShoe) // __lldb_expr_1.Shoe
```

```
class Shoe: CustomStringConvertible {  
    let color: String  
    let size: Int  
    let hasLaces: Bool  
  
    init(color: String, size: Int, hasLaces: Bool) {  
        ...  
    }  
  
}
```



```
class Shoe: CustomStringConvertible {
    let color: String
    let size: Int
    let hasLaces: Bool

    init(color: String, size: Int, hasLaces: Bool) {
        ...
    }

    var description: String {
        return "Shoe(color: \(color), size: \(size), hasLaces:
\(\hasLaces))"
    }
}
```

```
let myShoe = Shoe(color: "Black", size: 12, hasLaces: true)
print(myShoe) // Shoe(color: Black, size: 12, hasLaces: true)
```

# Equatable

# Comparing information with Equatable

```
struct Employee {  
    let firstName: String  
    let lastName: String  
    let jobTitle: String  
    let phoneNumber: String  
}  
  
struct Company {  
    let name: String  
    let employees: [Employee]  
}
```

# Comparing information with Equatable

```
let currentEmployee = Session.currentEmployee
let selectedEmployee = Employee(firstName: "Adrien",
    lastName: "Humilière", jobTitle: "Mobile engineer",
    phoneNumber: "415-555-9293")

if currentEmployee == selectedEmployee {
    // Enable "Edit" button
}
```

# Comparing information with Equatable

```
struct Employee: Equatable {  
    let firstName: String  
    let lastName: String  
    let jobTitle: String  
    let phoneNumber: String  
  
    static func ==(lhs: Employee, rhs: Employee) -> Bool {  
        // Equality logic  
    }  
}
```

# Comparing information with Equatable

```
struct Employee: Equatable {  
    let firstName: String  
    let lastName: String  
    let jobTitle: String  
    let phoneNumber: String  
  
    static func ==(lhs: Employee, rhs: Employee) -> Bool {  
        return lhs.firstName == rhs.firstName && lhs.lastName ==  
rhs.lastName  
    }  
}
```

# Comparing information with Equatable

```
let currentEmployee = Employee(firstName: "Adrien",  
    lastName: "Humilière", jobTitle: "Mobile engineer",  
    phoneNumber: "415-555-9293")  
let selectedEmployee = Employee(firstName: "Adrien",  
    lastName: "Humilière", jobTitle: "Customer support",  
    phoneNumber: "417-436-7384")  
  
if currentEmployee == selectedEmployee {  
    // Enable "Edit" button  
}
```



# Comparing information with Equatable

```
struct Employee: Equatable {  
    let firstName: String  
    let lastName: String  
    let jobTitle: String  
    let phoneNumber: String  
  
    static func ==(lhs: Employee, rhs: Employee) -> Bool {  
        return lhs.firstName == rhs.firstName && lhs.lastName ==  
rhs.lastName && lhs.jobTitle == rhs.jobTitle &&  
lhs.phoneNumber == rhs.phoneNumber  
    }  
}
```

# Comparable

# Sorting information with Comparable

```
let employee1 = Employee(firstName: "Ben", lastName: "Atkins")
let employee2 = Employee(firstName: "Vera", lastName: "Carr")
let employee3 = Employee(firstName: "Grant", lastName: "Phelps")
let employee4 = Employee(firstName: "Sang", lastName: "Han")

let employees = [employee1, employee2, employee3, employee4]
```

```
struct Employee: Equatable, Comparable {  
    let firstName: String  
    let lastName: String  
    let jobTitle: String  
    let phoneNumber: String  
  
    static func ==(lhs: Employee, rhs: Employee) -> Bool {  
        return ...  
    }  
  
    static func < (lhs: Employee, rhs: Employee) -> Bool {  
        return lhs.lastName < rhs.lastName  
    }  
}
```

```
let employees = [employee1, employee2, employee3, employee4,
employee5]

let sortedEmployees = employees.sorted(by:<)

for employee in sortedEmployees {
    print(employee)
}

// Employee(firstName: "Ben", lastName: "Atkins")
// Employee(firstName: "Vera", lastName: "Carr")
// Employee(firstName: "Sang", lastName: "Han")
// Employee(firstName: "Grant", lastName: "Phelps")
```

```
let employees = [employee1, employee2, employee3, employee4,
employee5]

let sortedEmployees = employees.sorted(by:>)

for employee in sortedEmployees {
    print(employee)
}

// Employee(firstName: "Grant", lastName: "Phelps")
// Employee(firstName: "Sang", lastName: "Han")
// Employee(firstName: "Vera", lastName: "Carr")
// Employee(firstName: "Ben", lastName: « Atkins")
```

# Codable

# Encoding and decoding objects with Codable

```
struct Employee: Equatable, Comparable, Codable {  
    var firstName: String  
    var lastName: String  
    var jobTitle: String  
    var phoneNumber: String  
  
    ...  
}
```



# Encoding and decoding objects with Codable

```
let ben = Employee(firstName: "Ben", lastName: "Atkins",
                    jobTitle: "Front Desk",
                    phoneNumber: "415-555-7767")

let jsonEncoder = JSONEncoder()
if let jsonData = try? jsonEncoder.encode(ben),
    let jsonString = String(data: jsonData, encoding: .utf8) {
    print(jsonString)
}
```

```
{"firstName": "Ben", "lastName": "Atkins", "jobTitle": "Front
Desk", "phoneNumber": "415-555-7767"}
```

# Protocol creation

# Creating a protocol

```
protocol FullyNamed {  
    var fullName: String { get }  
  
    func sayFullName()  
}  
  
struct Person: FullyNamed {  
    var firstName: String  
    var lastName: String  
}
```

# Creating a protocol

```
struct Person: FullyNamed {  
    var firstName: String  
    var lastName: String  
  
    var fullName: String {  
        return "\(firstName) \(lastName)"  
    }  
  
    func sayFullName() {  
        print(fullName)  
    }  
}
```

# Delegation

# Delegation

Enables a class or structure to hand off responsibilities to an instance of another type

```
protocol ButtonDelegate {  
    func userTappedButton(_ button: Button)  
}  
  
class GameController: ButtonDelegate {  
    func userTappedButton(_ button: Button) {  
        print("User tapped the \(button.title) button.")  
    }  
}
```

# Delegation

```
class Button {  
    let title: String  
    var delegate: ButtonDelegate? // Add a delegate property  
  
    init(title: String) {  
        self.title = title  
    }  
  
    func tapped() {  
        self.delegate?.userTappedButton(self)  
        // If the delegate exists, call the delegate  
        // function `userTappedButton` on the delegate  
    }  
}
```

# Delegation

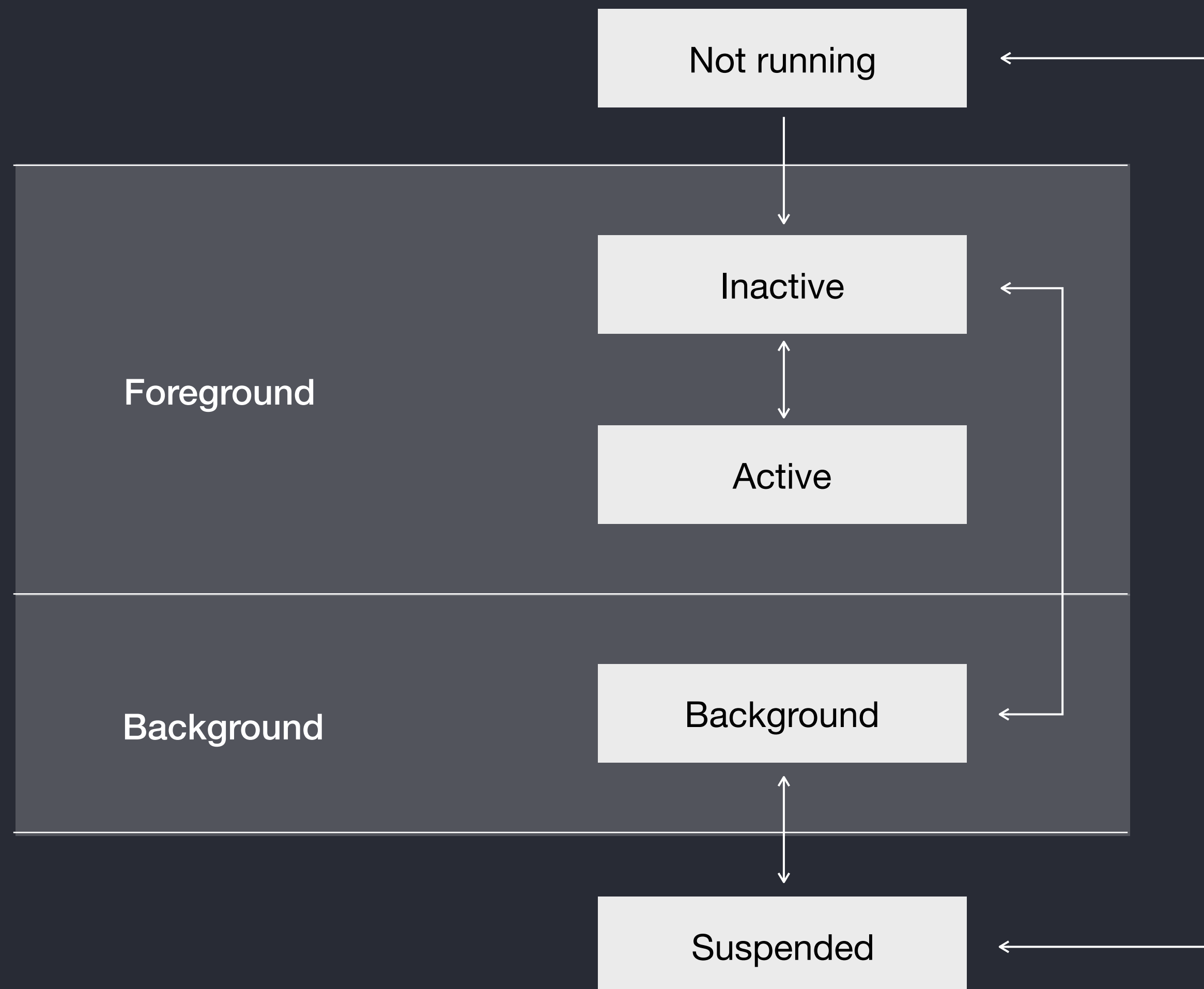
```
let startButton = Button(title: "Start Game")  
let gameController = GameController()  
startButton.delegate = gameController  
  
startButton.tapped()
```



# App Anatomy and Life Cycle



# App life cycle



# UIApplicationDelegate

- Did Finish Launching
- Will Resign Active
- Did Enter Background
- Will Enter Foreground
- Did Become Active
- Will Terminate

# UIApplicationDelegate

## Did Finish Launching

→ App has finished launching

```
func application(_ application: UIApplication,  
didFinishLaunchingWithOptions launchOptions:  
    [UIApplicationLaunchOptionsKey: Any]?) -> Bool {  
    return true  
}
```

→ Override point for customization after app launch

# UIApplicationDelegate

## Will Resign Active

- App is about to move from active to inactive state

```
func applicationWillResignActive(_ application: UIApplication) {}
```

- Can occur for certain types of temporary interruptions (such as an incoming phone call or SMS message)
- Can occur when the user quits the app and it begins the transition to the background state
- Use to pause ongoing tasks, disable timers, and invalidate graphics rendering callbacks

# UIApplicationDelegate

## Did Enter Background

- App is about to move from active to inactive state

```
func applicationDidEnterBackground(_ application: UIApplication) {}
```

- Use to release shared resources, save user data, invalidate timers, and store enough application state information to restore your application to its current state in case it's terminated later
- If your application supports background execution, this method is called instead of applicationWillTerminate: when the user quits

# UIApplicationDelegate

## Will Enter Foreground

- Called immediately before the `applicationDidBecomeActive` function

```
func applicationWillEnterForeground(_ application: UIApplication) {}
```

- Called as part of transition from the background to the active state
- Can be used to undo many of the changes made on entering the background

# UIApplicationDelegate

## Did Become Active

- App was launched by the user or system

```
func applicationDidBecomeActive(_ application: UIApplication) {}
```

- Restart any tasks that were paused (or not yet started) while the app was inactive
- If the app was previously in the background, optionally refresh the user interface



# UIApplicationDelegate

## Will Terminate

- App is about to be terminated

```
func applicationWillTerminate(_ application: UIApplication) {}
```

- Save data if appropriate
- See also `applicationDidEnterBackground:`

# UIApplicationDelegate

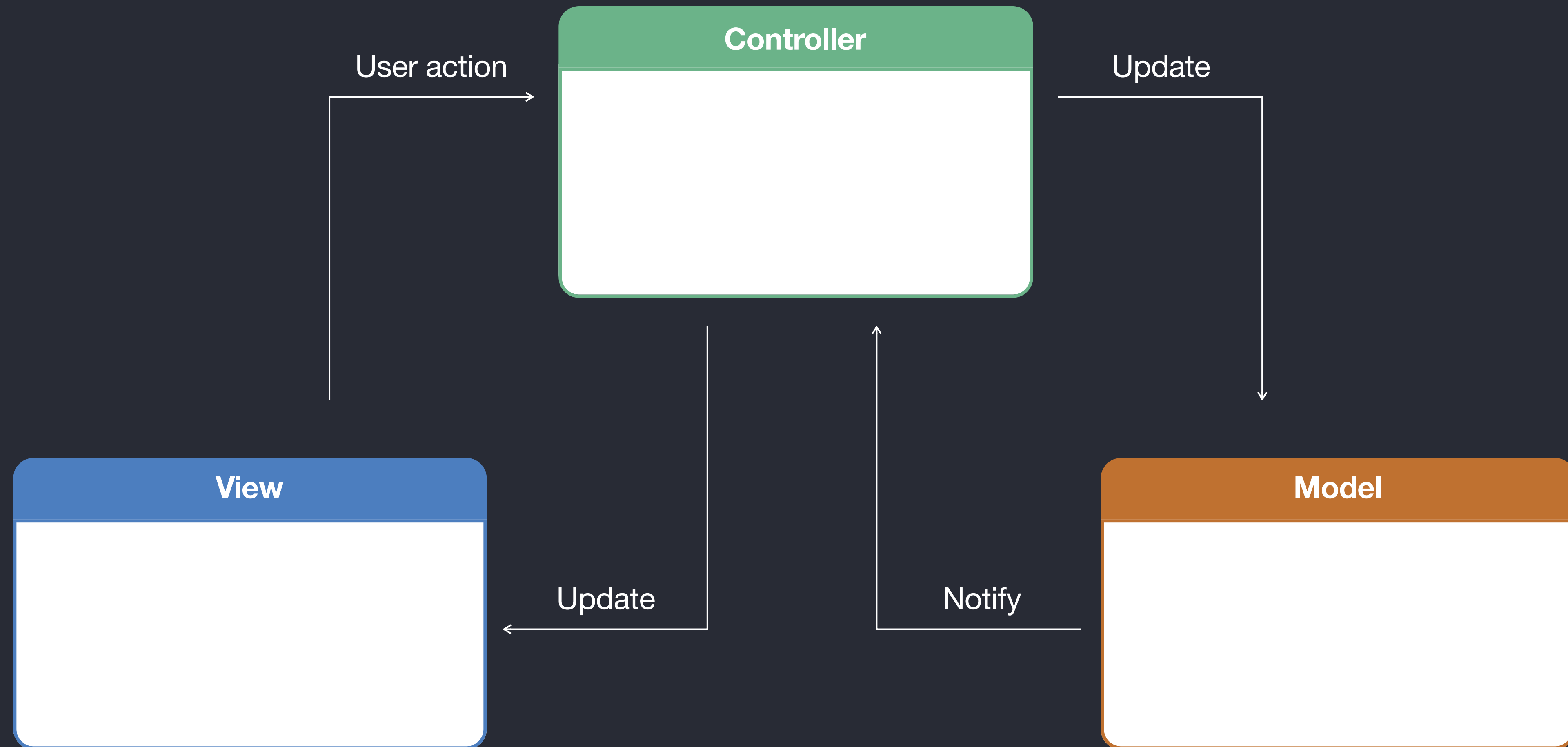
Which methods should I use?

- Start with the methods that will run when launching, reopening, or closing your app
  - `applicationDidFinishLaunchingWithOptions`
  - `applicationWillResignActive`
  - `applicationDidBecomeActive`
- Take advantage of the other three delegate methods as you become more experienced

# Model View Controller

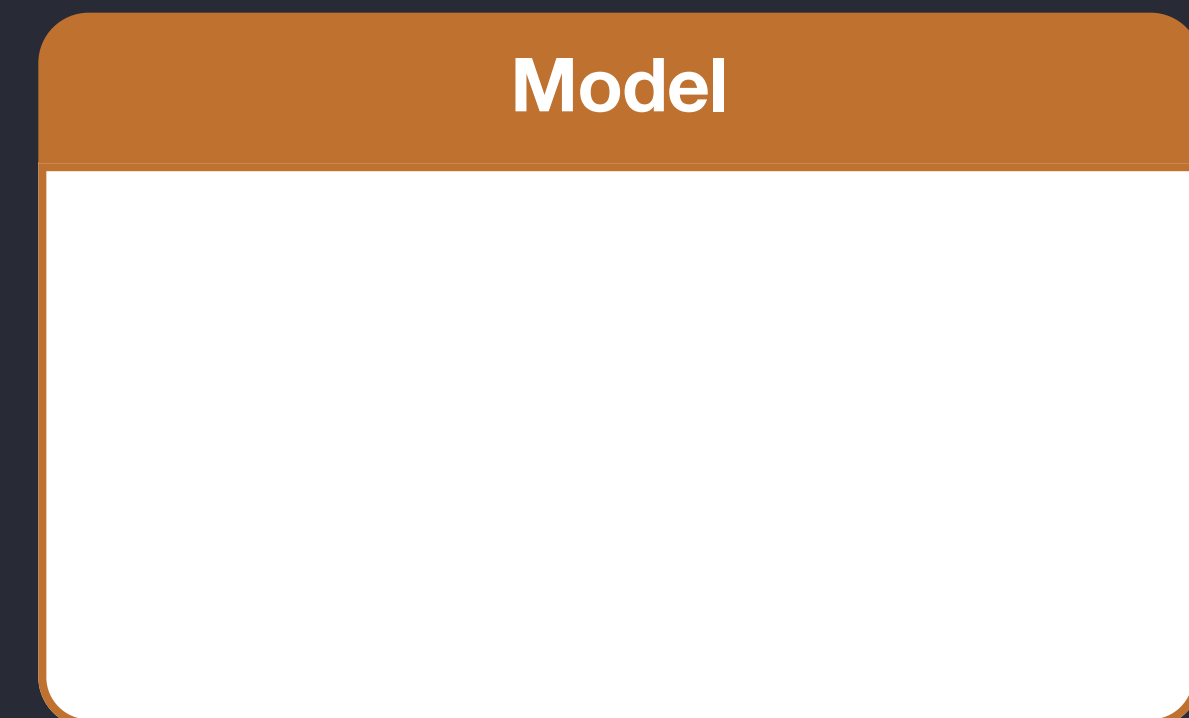


# Model View Controller

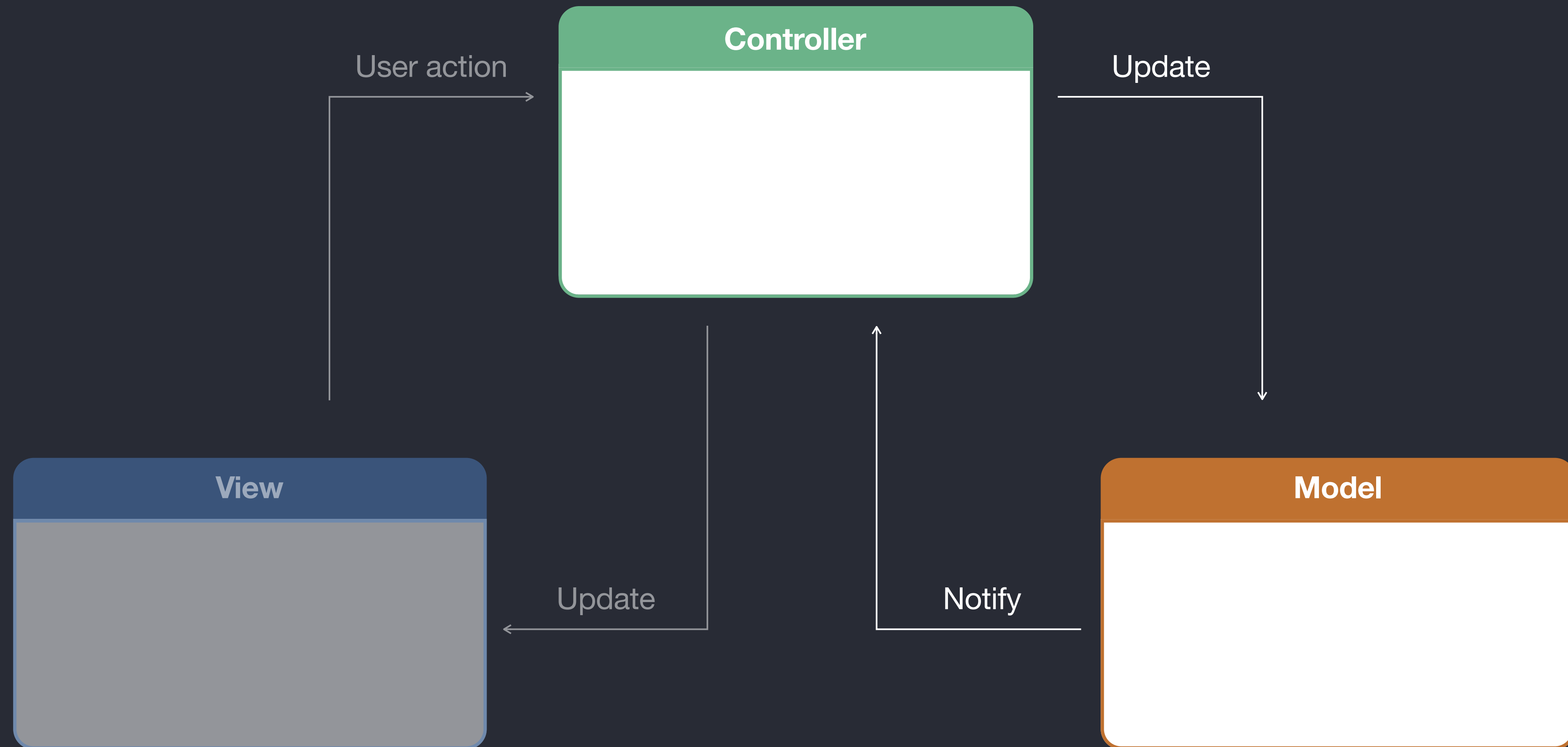


# Model objects

- Groups the data needed for a specific problem domain or a type of solution to be built
- Can be related to other model objects

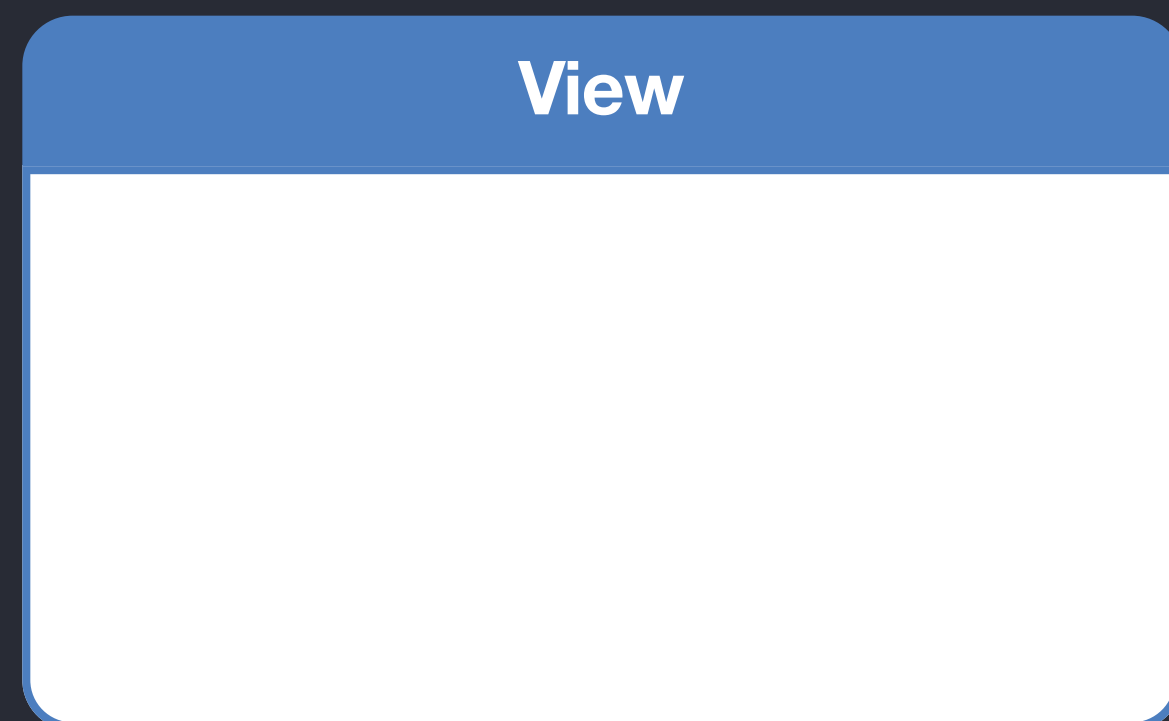


# Model objects

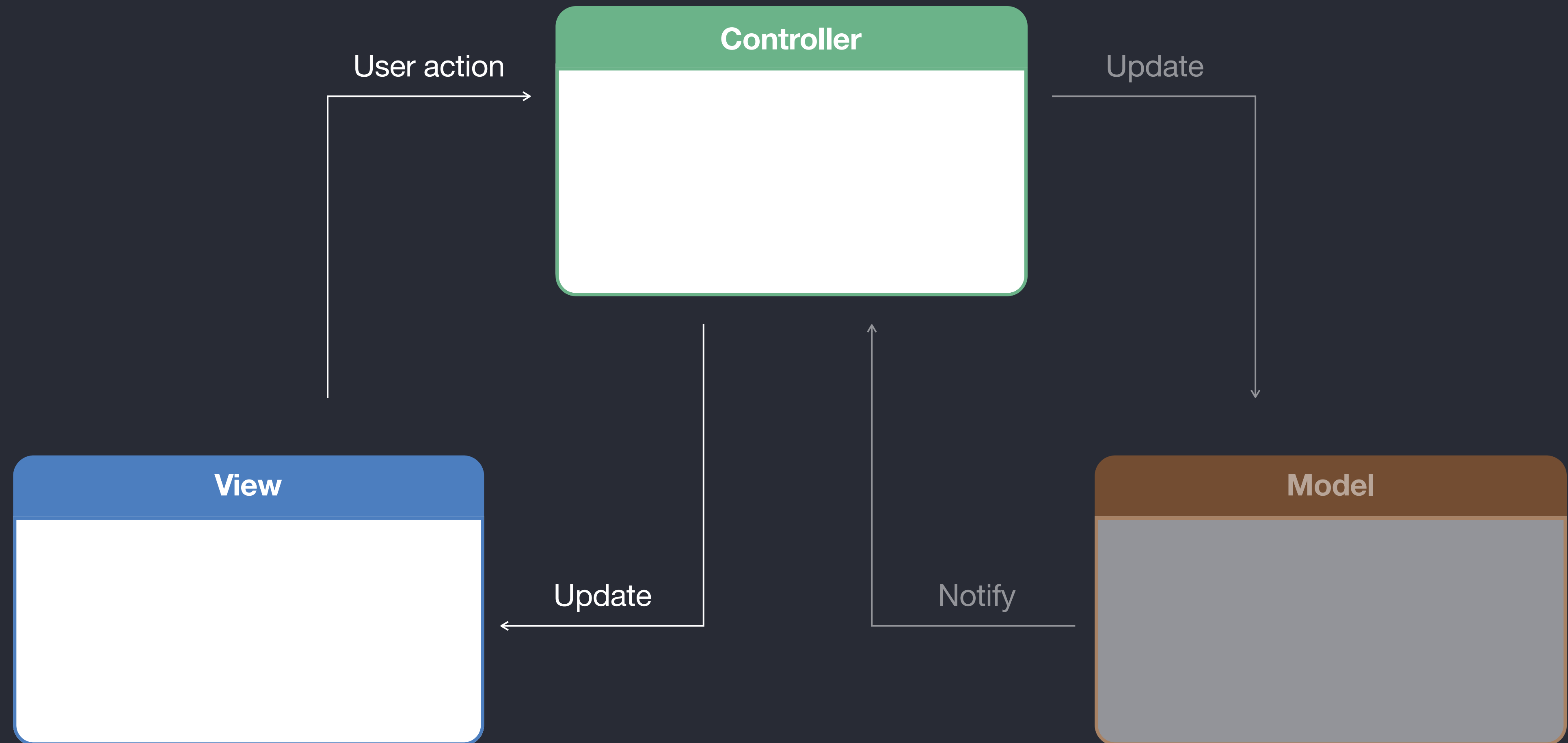


# Views

- Displays data about the app's model objects and allows user to edit the data
- Can be reused to show different instances of the model data



# Views





# Controllers

→ Acts as the messenger between views and model objects

→ Types:

View controllers

Model controllers

Helper controllers

# Model Controllers

Helps control a model object or collection of model objects

Three common reasons to create a model controller:

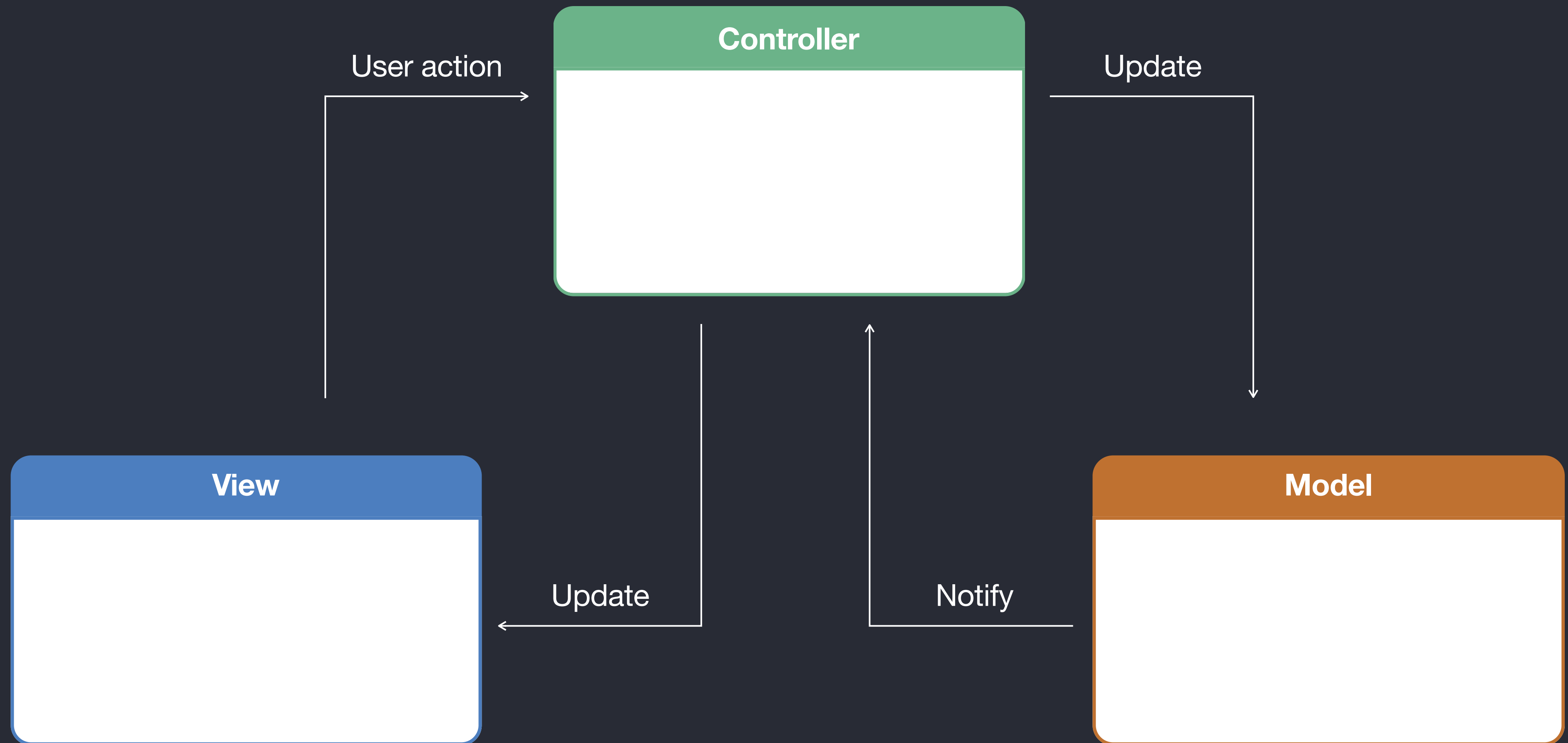
- Multiple objects or scenes need access to the model data
- Logic for adding, modifying, or deleting model data is complex
- Keep the code in view controllers focused on managing the views

Crucial in larger projects for readability and maintainability

# Helper Controllers

- Useful to consolidate related data or functionality so that it can be accessed by other objects in your app

# Controllers



# Example

# Meal tracker example

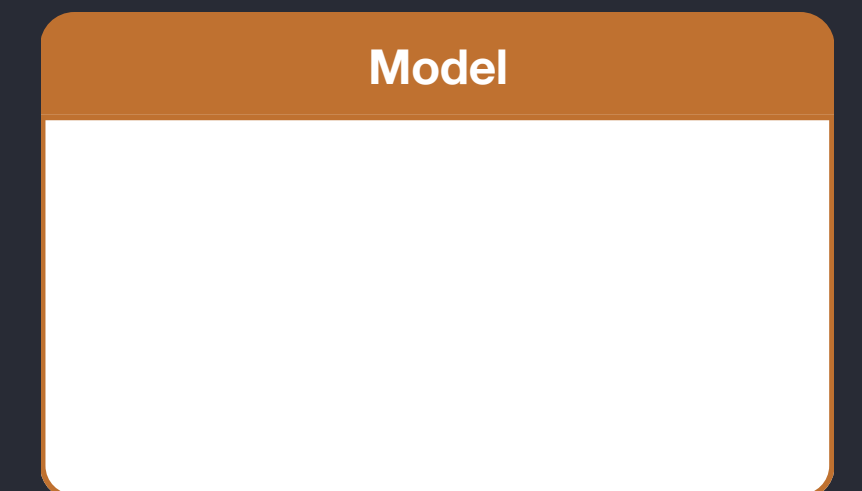
Creating an app to track eaten meals

- What should be in a "Meal" model object?
- What views are needed to display meals?
- How many controllers makes sense?

# Meal tracker example

Meal:

- Name
- Photo
- Notes
- Rating
- Timestamp



# Meal tracker example

```
struct Meal {  
    var name: String  
    var photo: UIImage  
    var notes: String  
    var rating: Int  
    var timestamp: Date  
}
```

Model

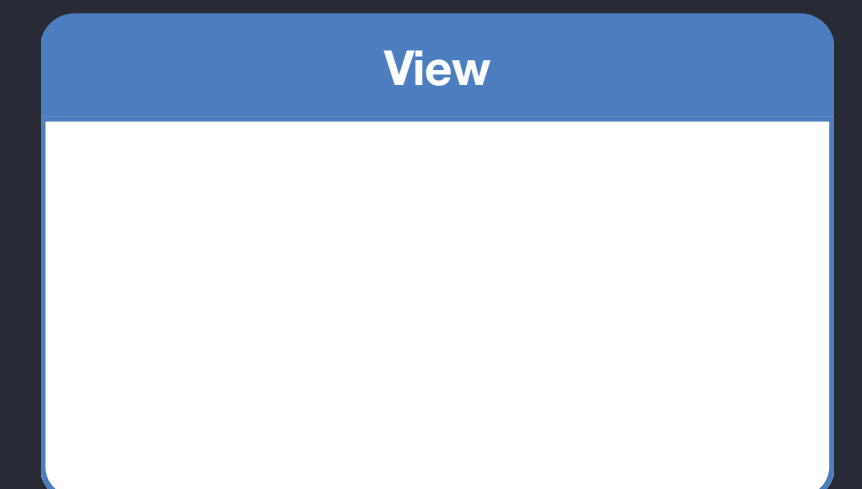


# Meal tracker example

Two possible views:

- List of all tracked meals
- Details of each meal

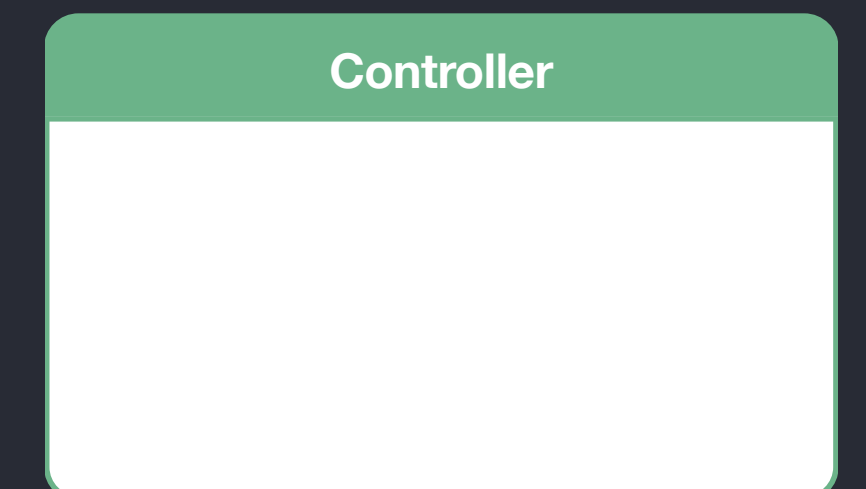
Each needs a view controller class



# Meal tracker example

Minimum of two controllers:

- List view
- Detail view



# Meal tracker example

```
class MealListTableViewController: UITableViewController {  
  
    var meals: [Meal] = []  
    @IBOutlet weak var tableView: UITableView!  
}
```



Controller

A diagram of a Controller object, represented as a rounded rectangle with a green header bar containing the text 'Controller' and a white body.

# Meal tracker example

```
class MealListTableViewController: UITableViewController {  
    var meals: [Meal] = []  
  
    func saveMeals() {...}  
  
    func loadMeals() {...}  
}
```



Controller

A diagram of a Controller component, represented as a rounded rectangle with a green header bar containing the text 'Controller' and a white body area.

```
class MealListTableViewController: UITableViewController {  
  
    let meals: [Meal] = []  
  
    override func viewDidLoad() {  
        // load the meals and set up the table view  
    }  
  
    // Required table view methods  
  
    override func tableView(_ tableView: UITableView,  
numberOfRowsInSection section: Int) -> Int {...}  
  
    override func tableView(_ tableView: UITableView,  
cellForRowAt indexPath: IndexPath) -> UITableViewCell {...}
```

```

    // Navigation methods
    override func prepare(for segue: UIStoryboardSegue, sender:
Any?) {
        // Pass the selected meal to the MealDetailViewController
    }
    @IBAction func unwindToMealList(sender: UIStoryboardSegue) {
        // Capture the new or updated meal from the
MealDetailViewController and save it to the meals property
    }

    // Persistence methods
    func saveMeals() {
        // Save the meals model data to the disk
    }

    func loadMeals() {
        // Load meals data from the disk and assign it to the
meals property
    }
}

```

# Meal tracker example

```
class MealDetailViewController: UIViewController {...}
```



Controller

A diagram of a Controller component, represented as a rounded rectangle with a green header bar containing the word "Controller" and a white body.

```
class MealDetailViewController: UIViewController,
UIImagePickerControllerDelegate {

    @IBOutlet weak var nameTextField: UITextField!
    @IBOutlet weak var photoImageView: UIImageView!
    @IBOutlet weak var ratingControl: RatingControl!
    @IBOutlet weak var saveButton: UIBarButtonItem!

    var meal: Meal?

    override func viewDidLoad() {
        if let meal = meal {
            update(meal)
        }
    }

    func update(_ meal: Meal) {
        // Update all outlets to reflect the data about the meal
    }
}
```



```
// Navigation methods

    override func prepare(for segue: UIStoryboardSegue,
sender: Any?) {
        // Update the meal property that will be accessed by
the MealListTableViewController to update the list of meals
    }

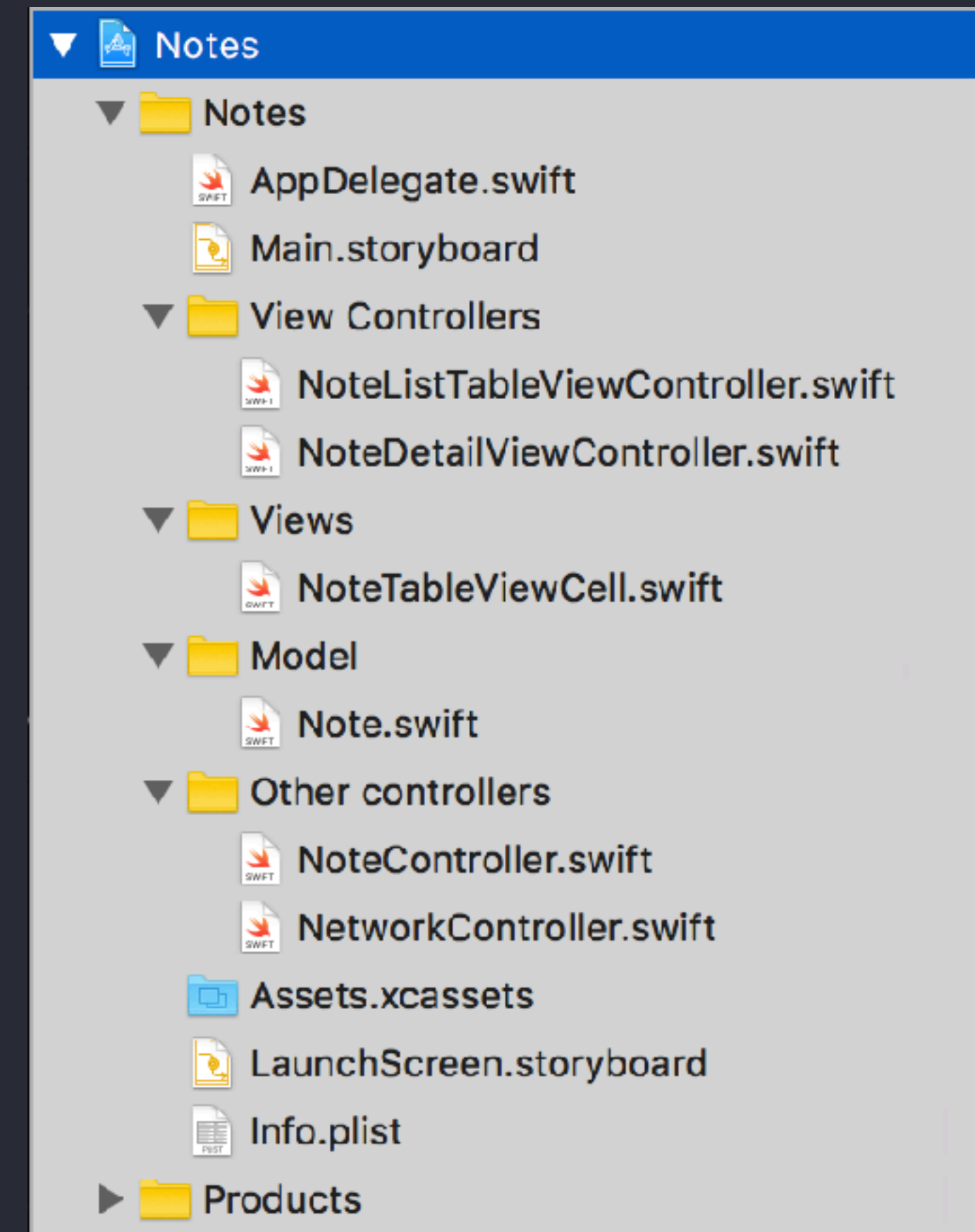
    @IBAction func cancel(_ sender: UIBarButtonItem) {
        // Dismiss the view without saving the meal
    }
```

# Reminder

- Model-View-Controller is a useful pattern
- More than one way to implement it
- Everyone has their own style
- Yours will evolve as you gain experience

# Project organization

- Use clear, descriptive filenames
- Create separate files for each of your type definitions
- Write your code as if complete strangers are going to read it
- Group files to help organize your code



# Scroll views



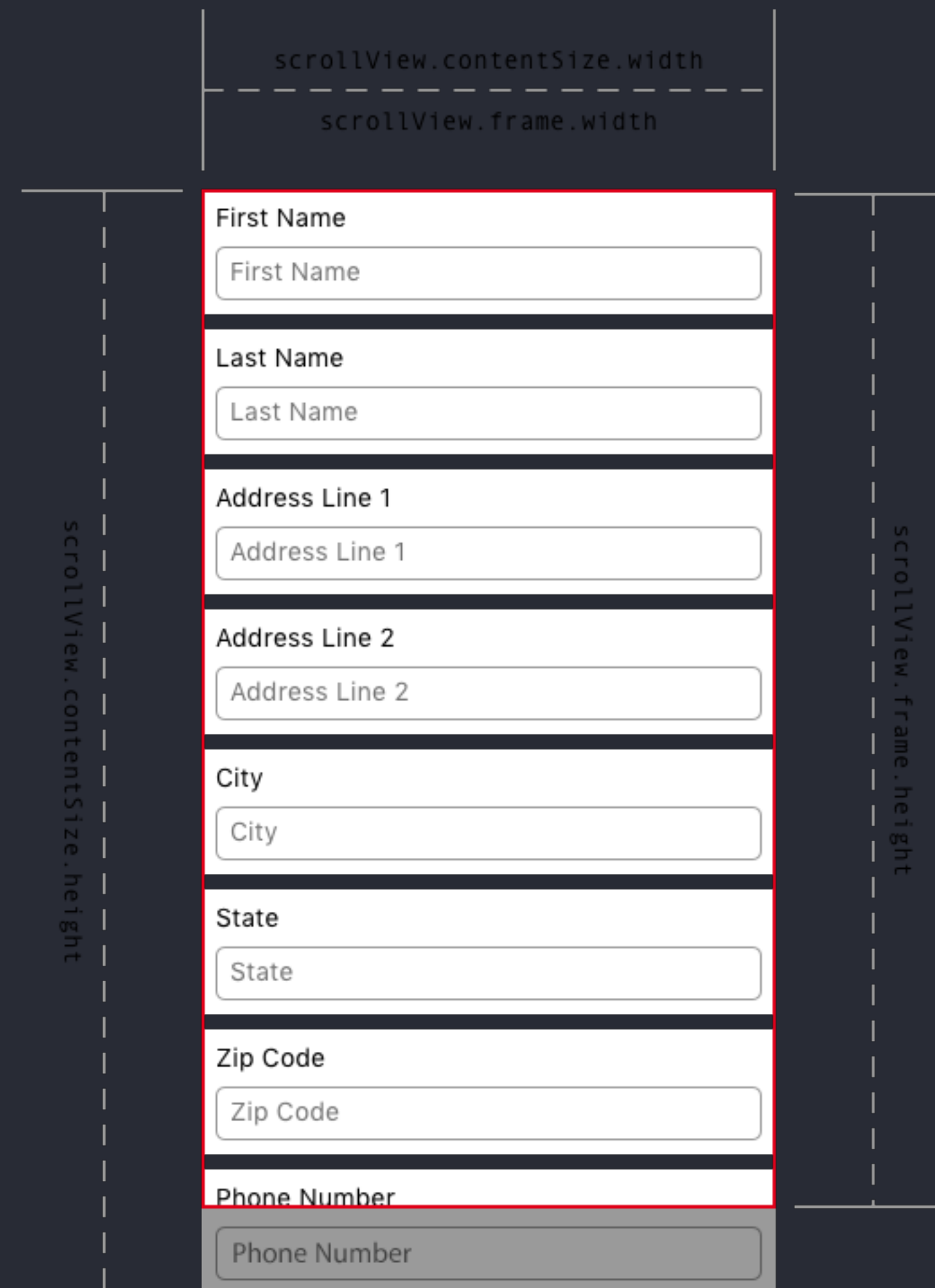
# Scroll views



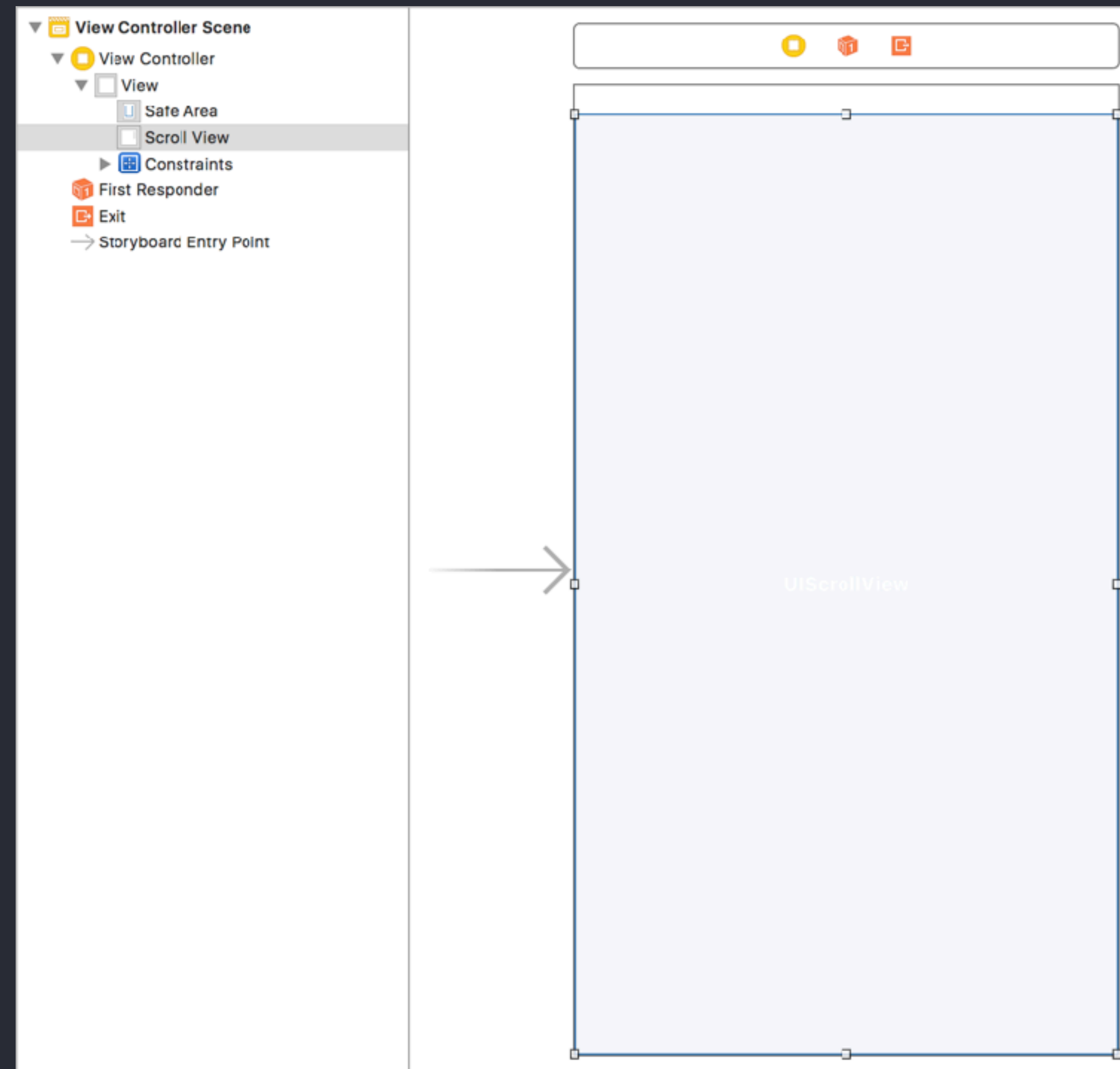
# UIScrollView

- For displaying more content than can fit on the screen
- Users scroll within the content by making swiping gestures
- Content can optionally be zoomed with a pinch gesture
- UIScrollView needs to know the size of the content

# UIScrollView



# Scroll views in Interface Builder





# Scroll views in Interface Builder

**Add New Constraints**

0

0 0

0

Spacing to nearest neighbor

☐ Constrain to margins

---

☐ Width 375

☐ Height 667

---

☐ Equal Widths

☐ Equal Heights

☐ Aspect Ratio

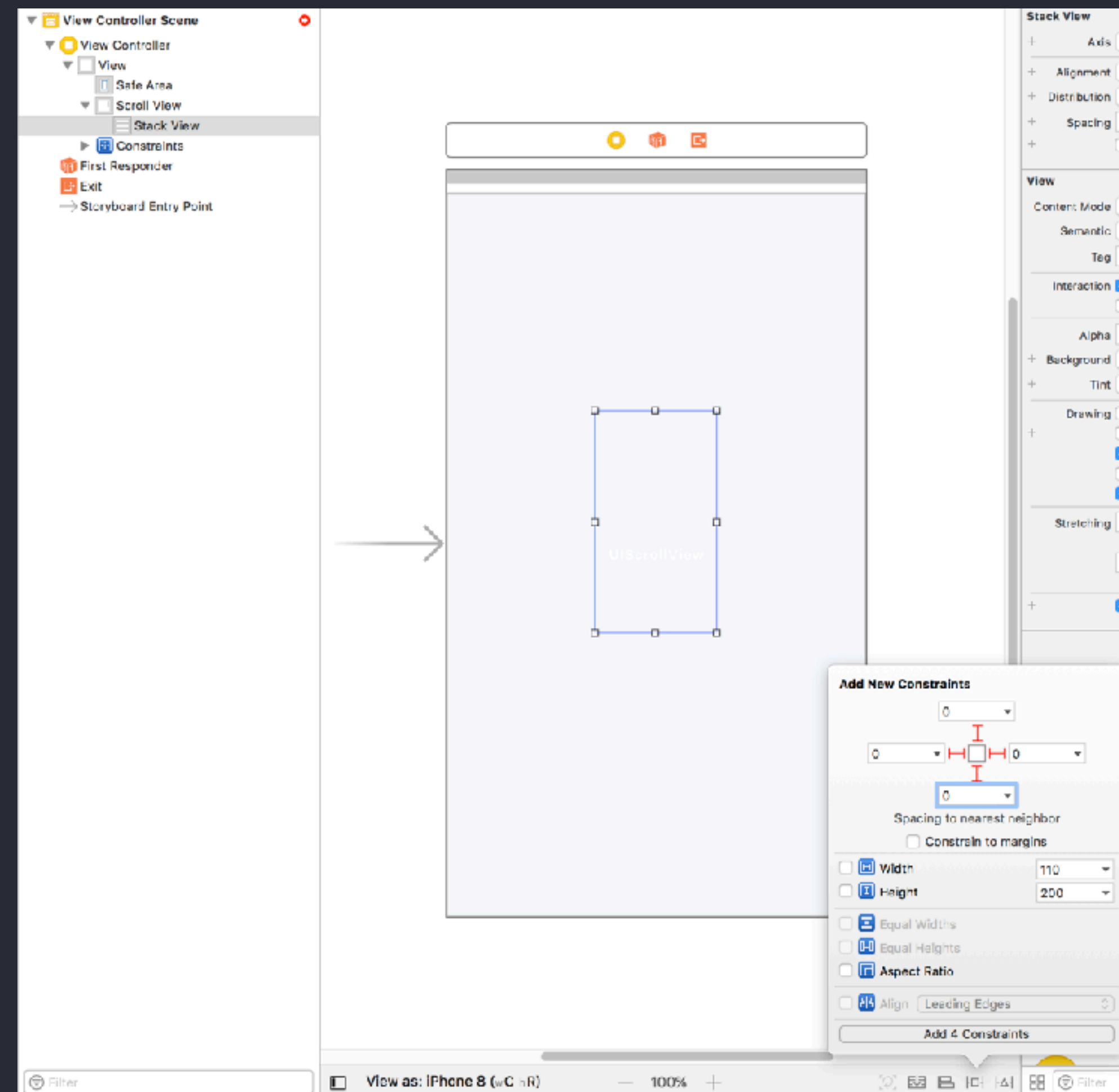
---

☐ Align Leading Edges

---

Add 4 Constraints

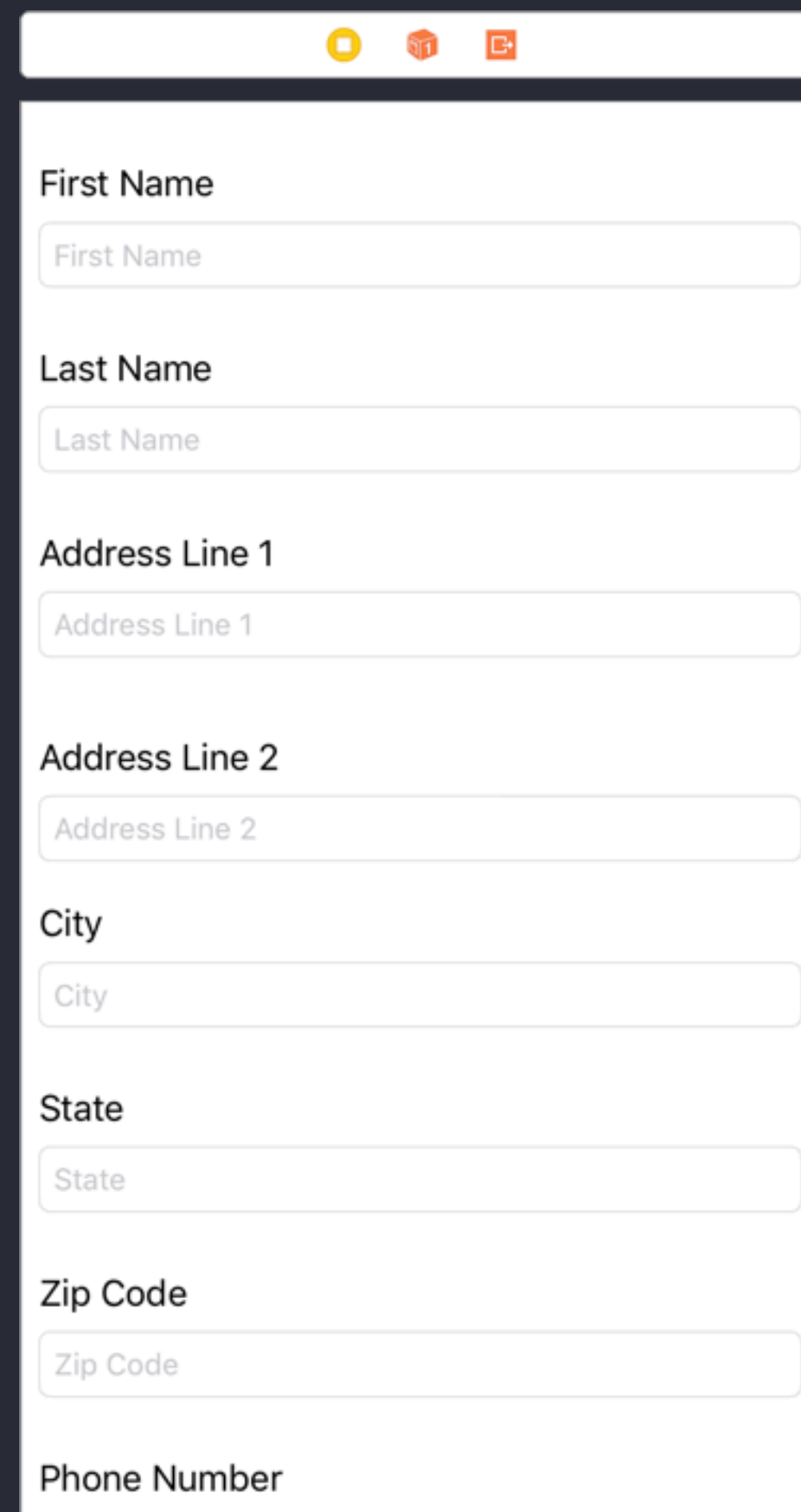
# Scroll views in Interface Builder



# Scroll views in Interface Builder

```
imageView.centerXAnchor.constraints(equalTo: scrollView.contentLayoutGuide.centerXAnchor)  
imageView.centerYAnchor.constraints(equalTo: scrollView.contentLayoutGuide.centerYAnchor)
```

# Scroll views in Interface Builder



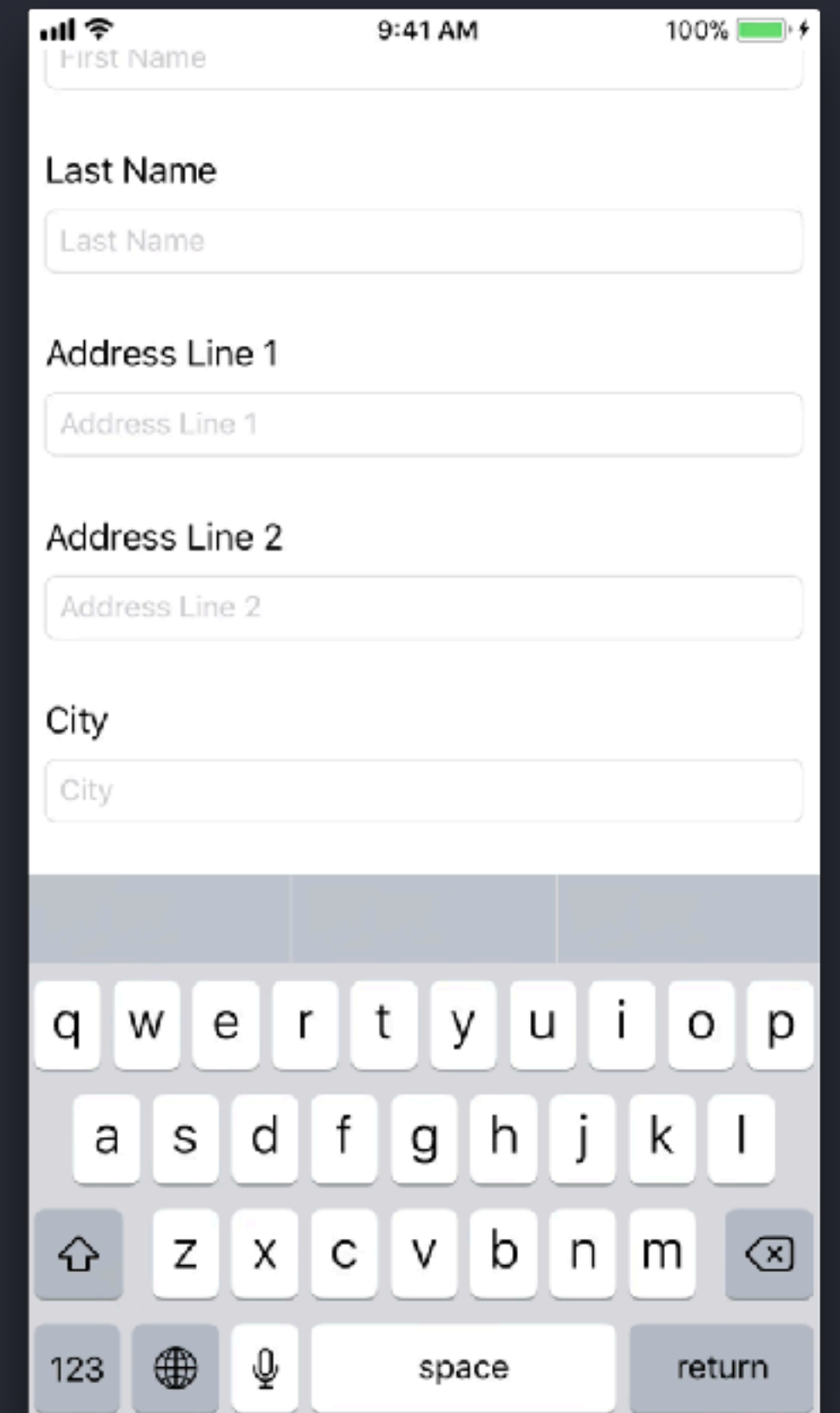
A vertical scroll view containing a form with the following elements:

- First Name**
- Last Name**
- Address Line 1**
- Address Line 2**
- City**
- State**
- Zip Code**
- Phone Number**

# Keyboard issues

- Sent a notification when the keyboard has been shown or will be hidden
- Register for keyboard notifications

```
func registerForKeyboardNotifications() {  
    NotificationCenter.default.addObserver(self,  
    selector: #selector(keyboardWasShown(_:)),  
    name: .UIKeyboardDidShow, object: nil)  
    NotificationCenter.default.addObserver(self,  
    selector: #selector(keyboardWillBeHidden(_:)),  
    name: .UIKeyboardWillHide, object: nil)  
}
```

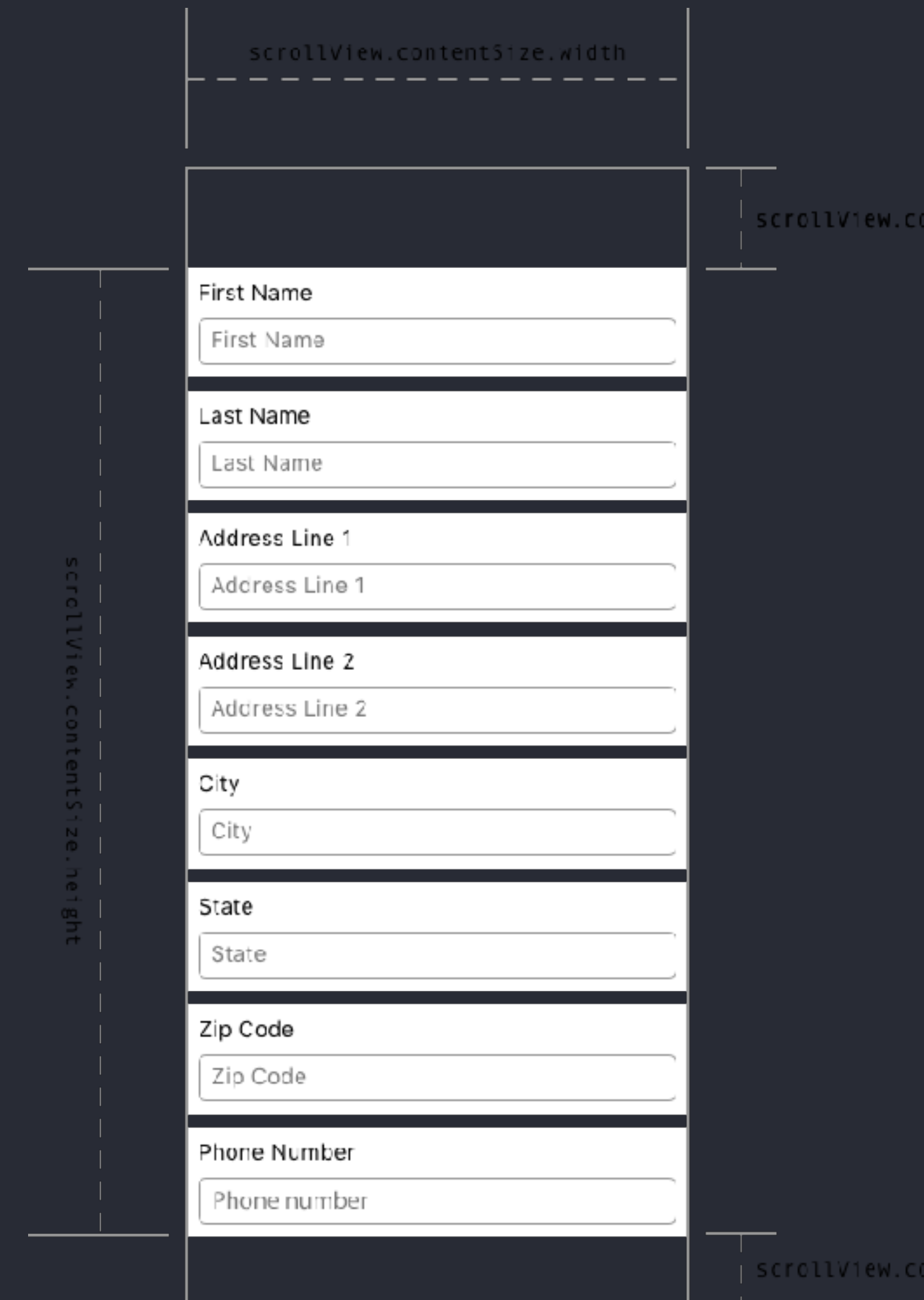


```
func keyboardWasShown(_ notification: NSNotification) {  
    guard let info = notification.userInfo,  
          let keyboardFrameValue =  
info[UIKeyboardFrameBeginUserInfoKey] as? NSValue else { return }  
  
    let keyboardFrame = keyboardFrameValue.cgRectValue  
    let keyboardSize = keyboardFrame.size  
  
    let contentInsets = UIEdgeInsetsMake(0.0, 0.0,  
keyboardSize.height, 0.0)  
    scrollView.contentInset = contentInsets  
    scrollView.scrollIndicatorInsets = contentInsets  
}  
  
func keyboardWillBeHidden(_ notification: NSNotification) {  
    let contentInsets = UIEdgeInsets.zero  
    scrollView.contentInset = contentInsets  
    scrollView.scrollIndicatorInsets = contentInsets  
}
```

# Content insets

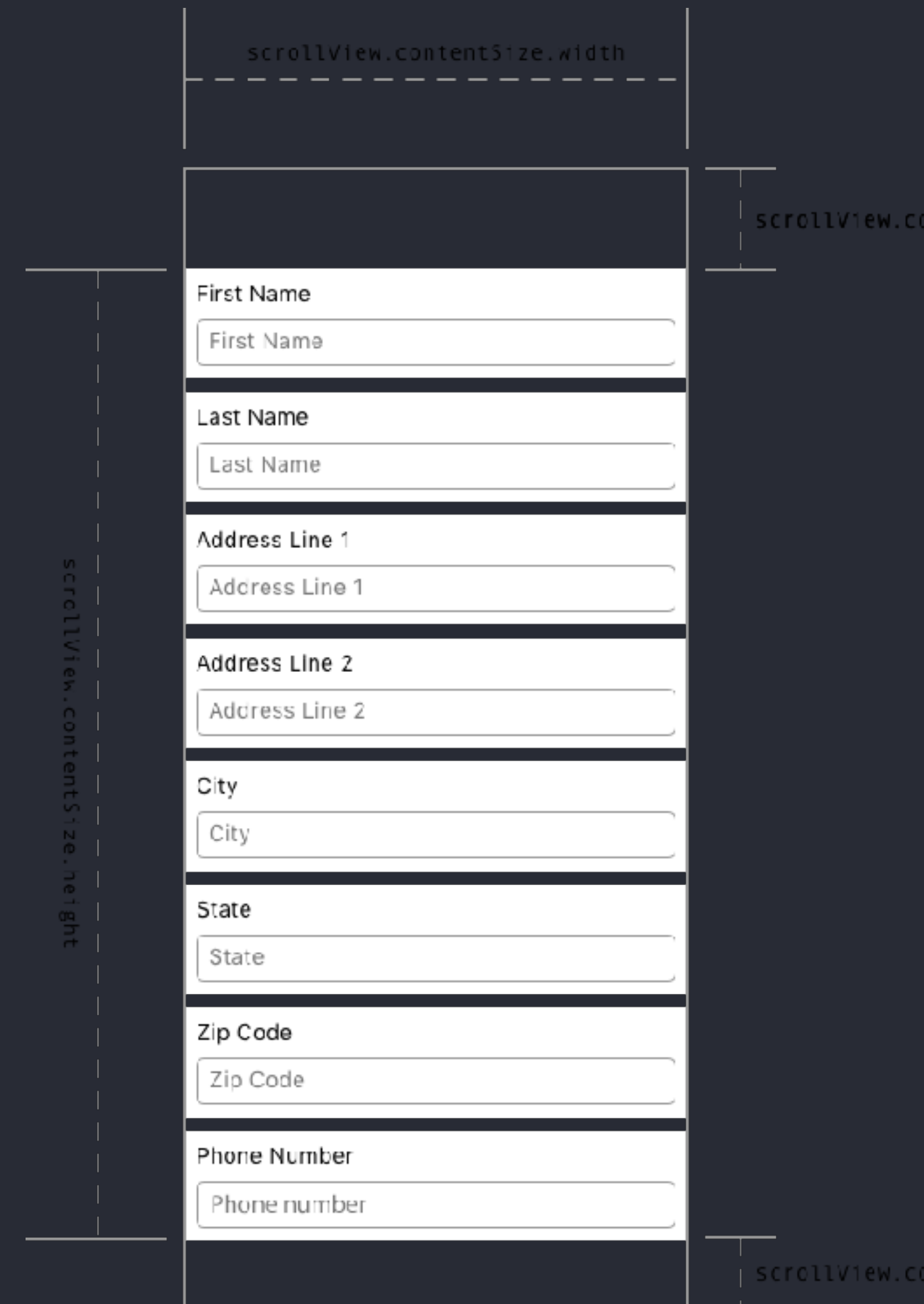
- Allows you to pad the content at the top and bottom of the scroll view
- Useful if you have toolbars floating above your scroll view

```
scrollView.contentInset.top  
                           .bottom  
                           .left  
                           .right
```



# Scroll indicator

```
let contentInsets = UIEdgeInsetsMake(0.0, 0.0,  
    keyboardSize.height, 0.0)  
scrollView.contentInset = contentInsets  
scrollView.scrollIndicatorInsets = contentInsets
```





# Table views

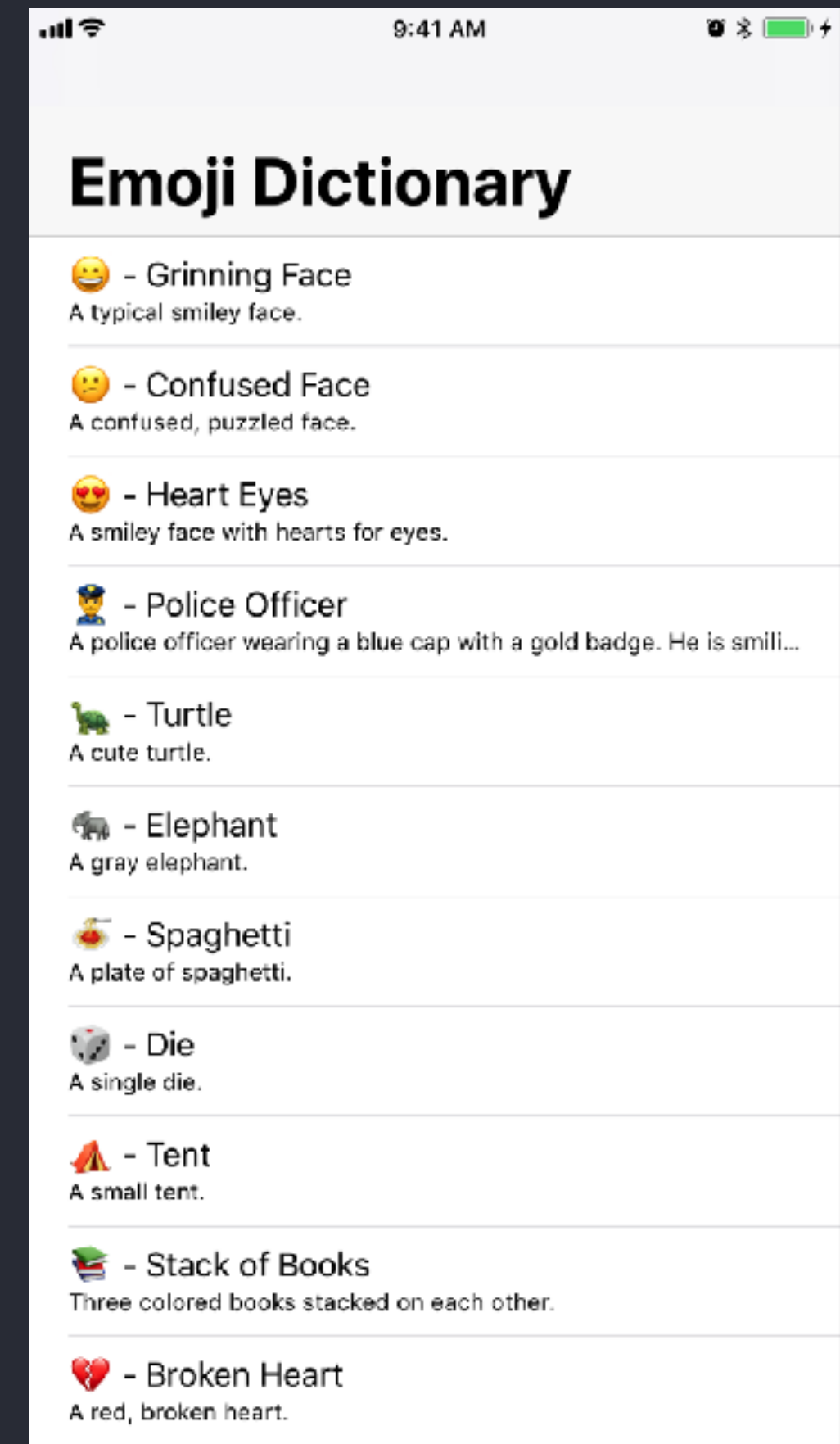


# Table views

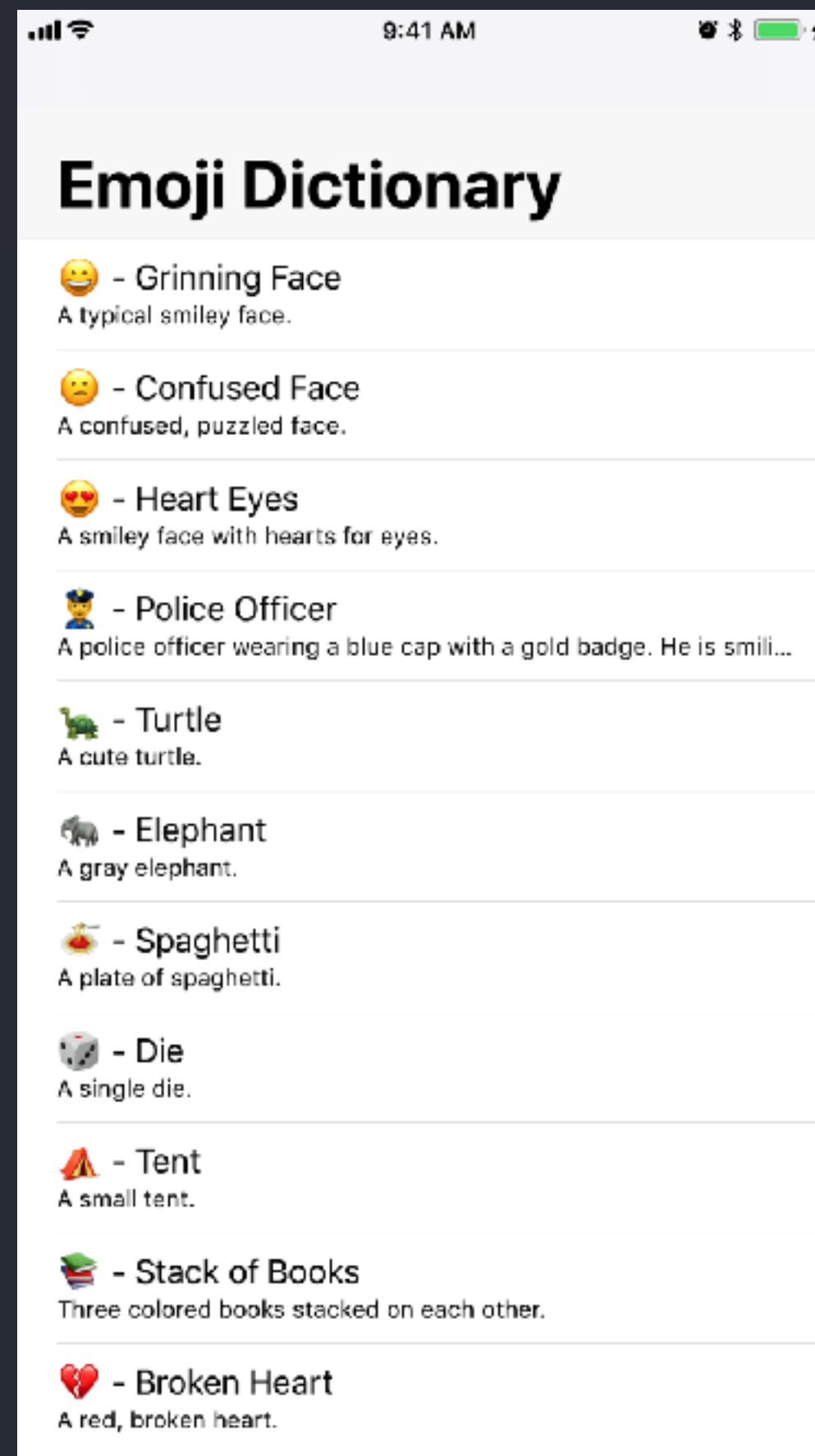
An instance of the UITableView class

A subclass of UIScrollView

- Displays a list of items
- Displays one or possibly thousands of data objects
- Presents vertical scrolling and single-column, multiple rows
- Provides customizable options



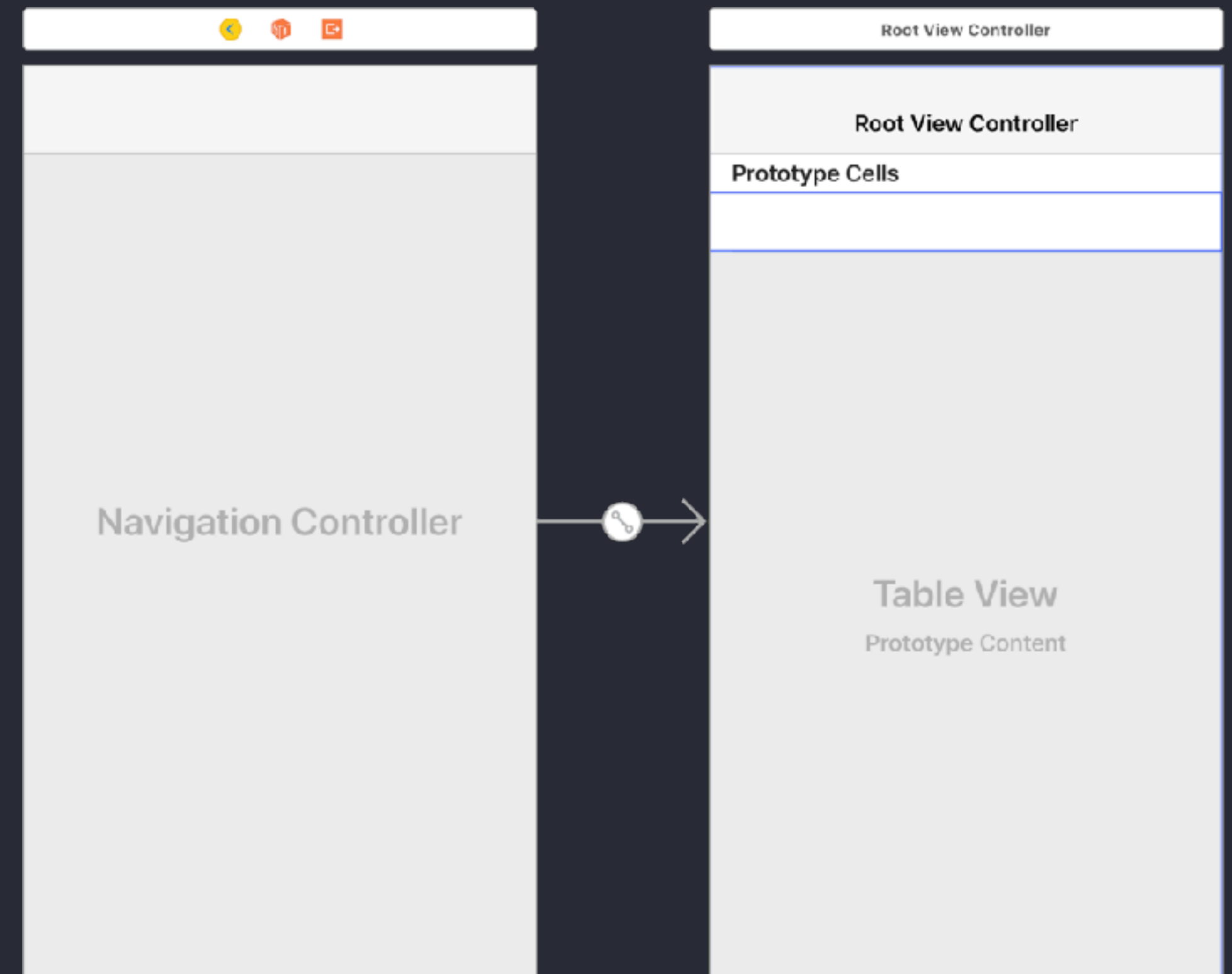
# Table views



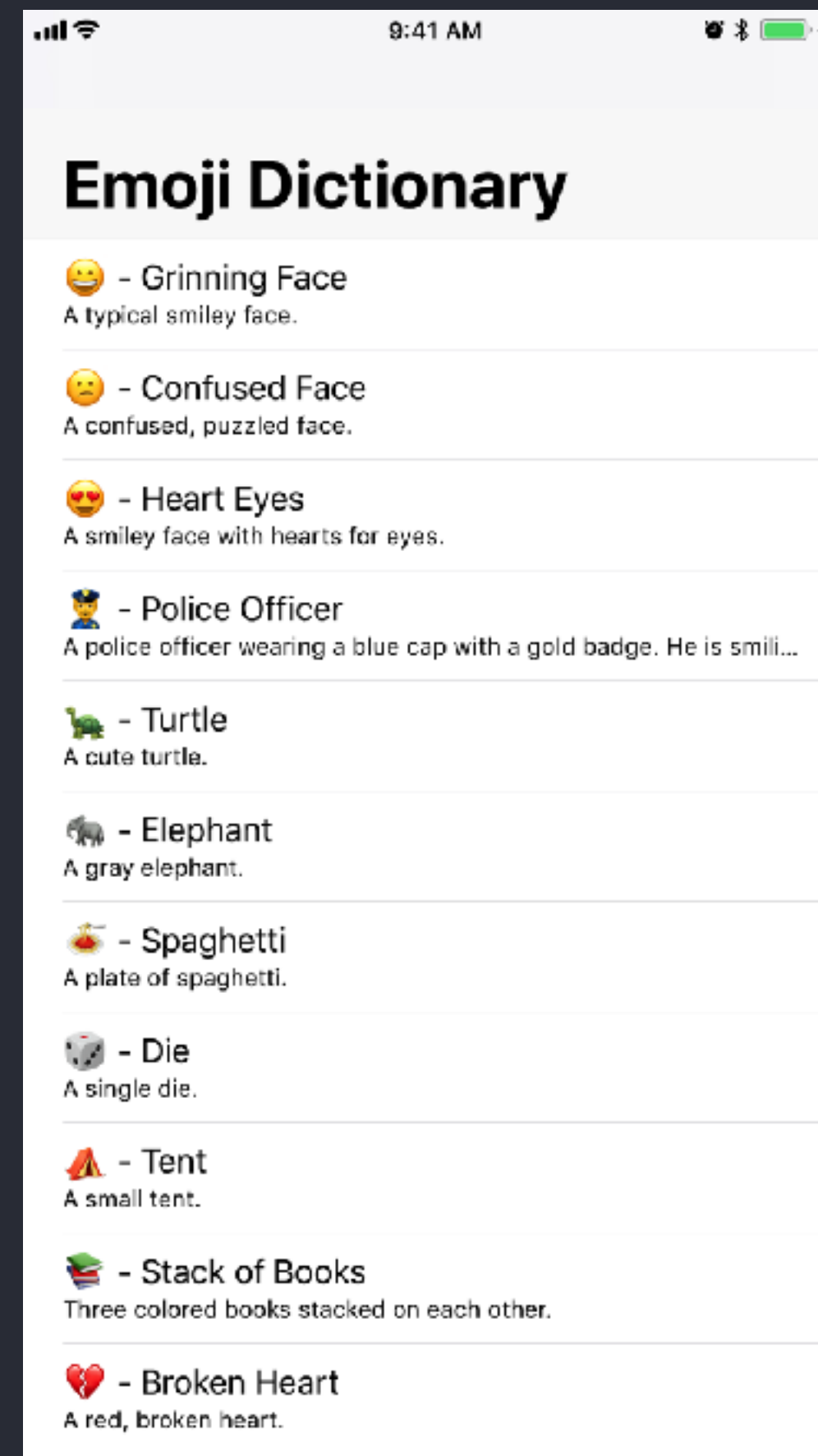
# Anatomy of a table view

Two possible approaches to add table views:

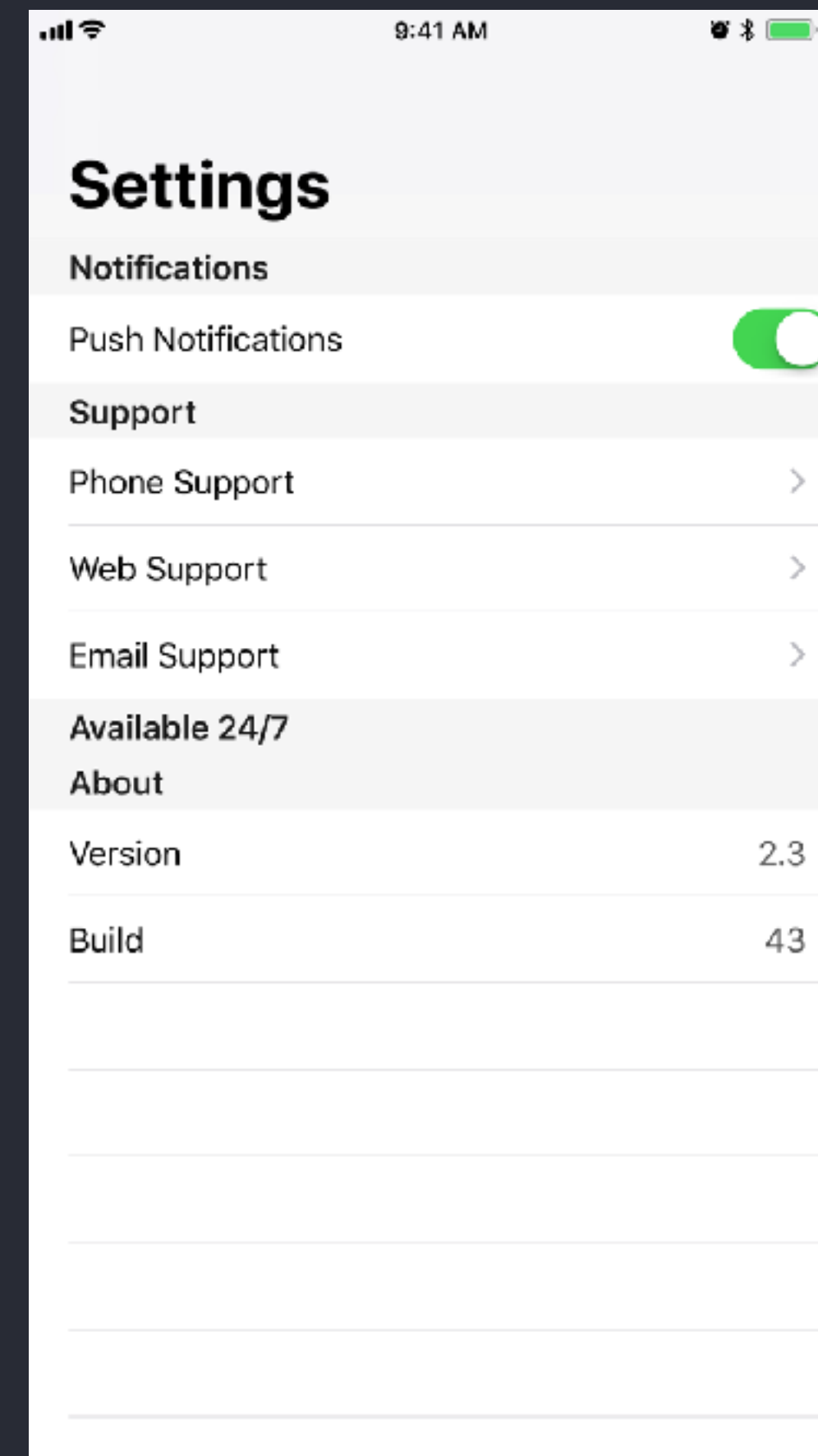
- Add a table view instance directly to a view controller's view
- Add a table view controller to your storyboard



# Anatomy of a table view



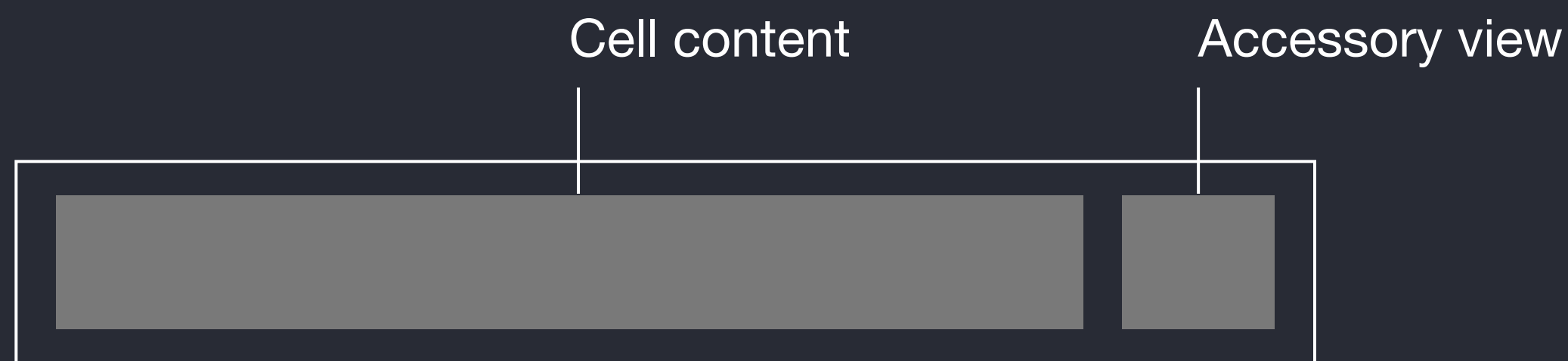
Plain



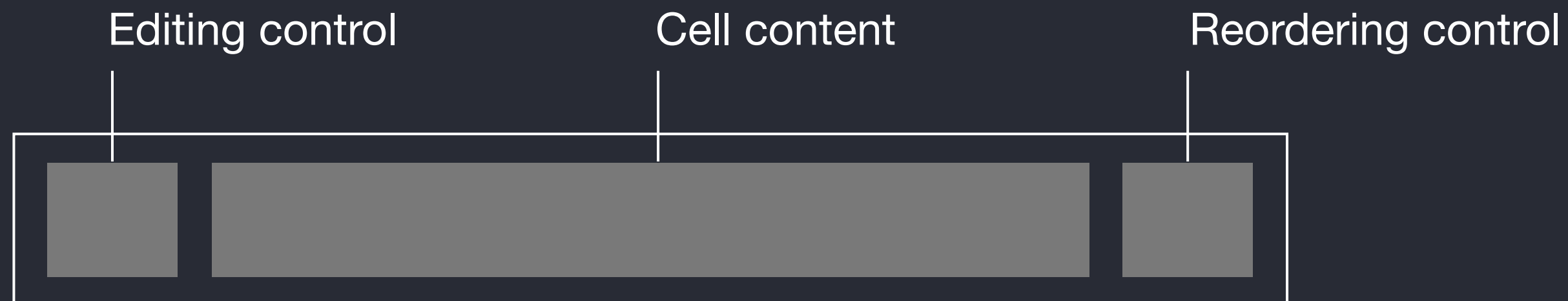
Grouped

# Anatomy of a table view - cells

Every row is represented with a table view cell



In editing mode, the cell content shrinks



# Anatomy of a table view - cells

`UITableViewCell` class defines three properties for cell content

Cell property	Description
<code>titleLabel</code>	<code>UILabel</code> for the title
<code>detailTextLabel</code>	<code>UILabel</code> for the subtitle
<code>imageView</code>	<code>UIImageView</code> for an image

# Anatomy of a table view - cells

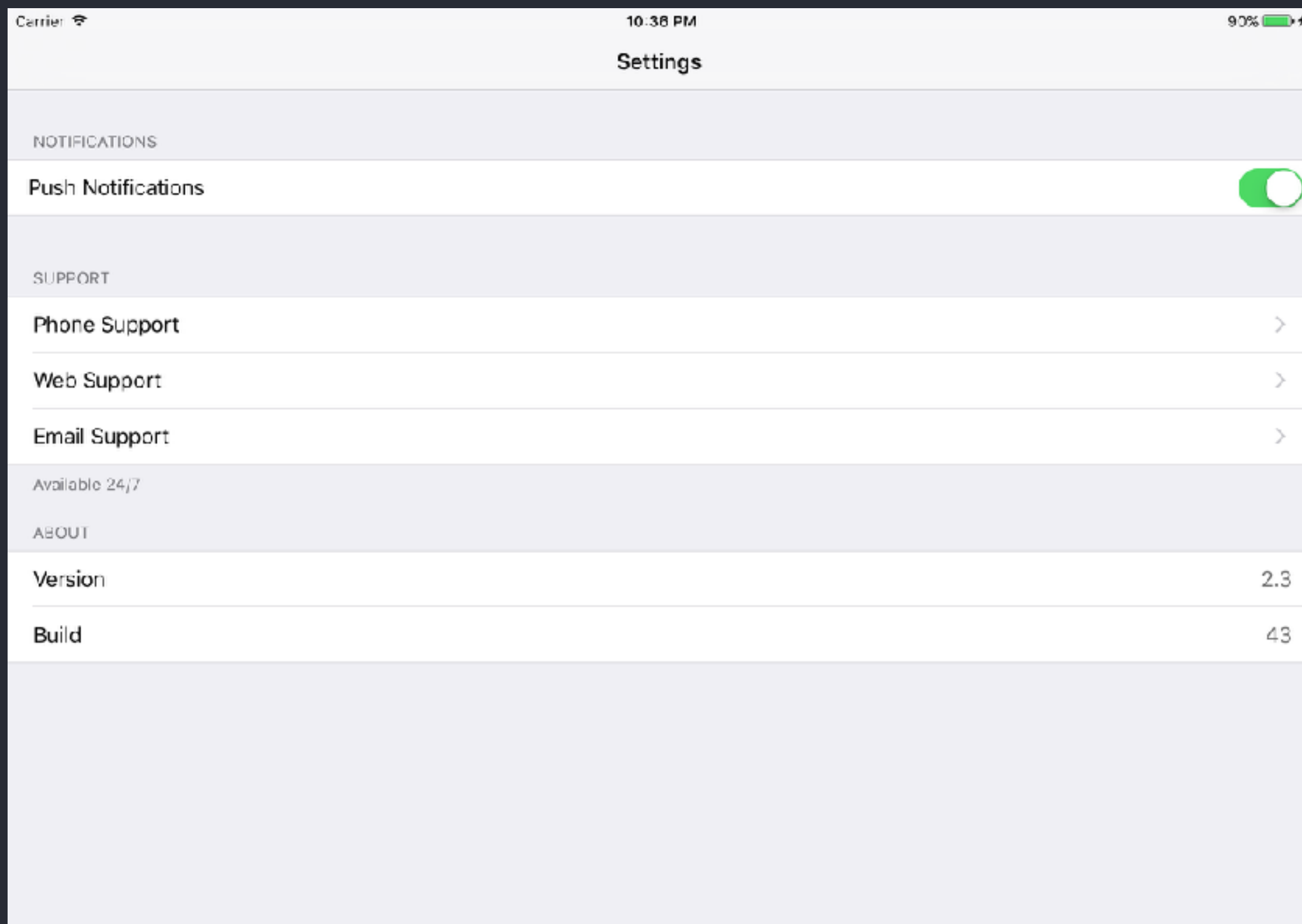
Storyboard	Programmatic enum name	Displays
Basic	.default	textLabel, imageView
Subtitle	.subtitle	textlabel, detailTextLabel, imageView
Right detail	.value1	textlabel, detailTextLabel, imageView
Left detail	.value2	textLabel, detailTextLabel



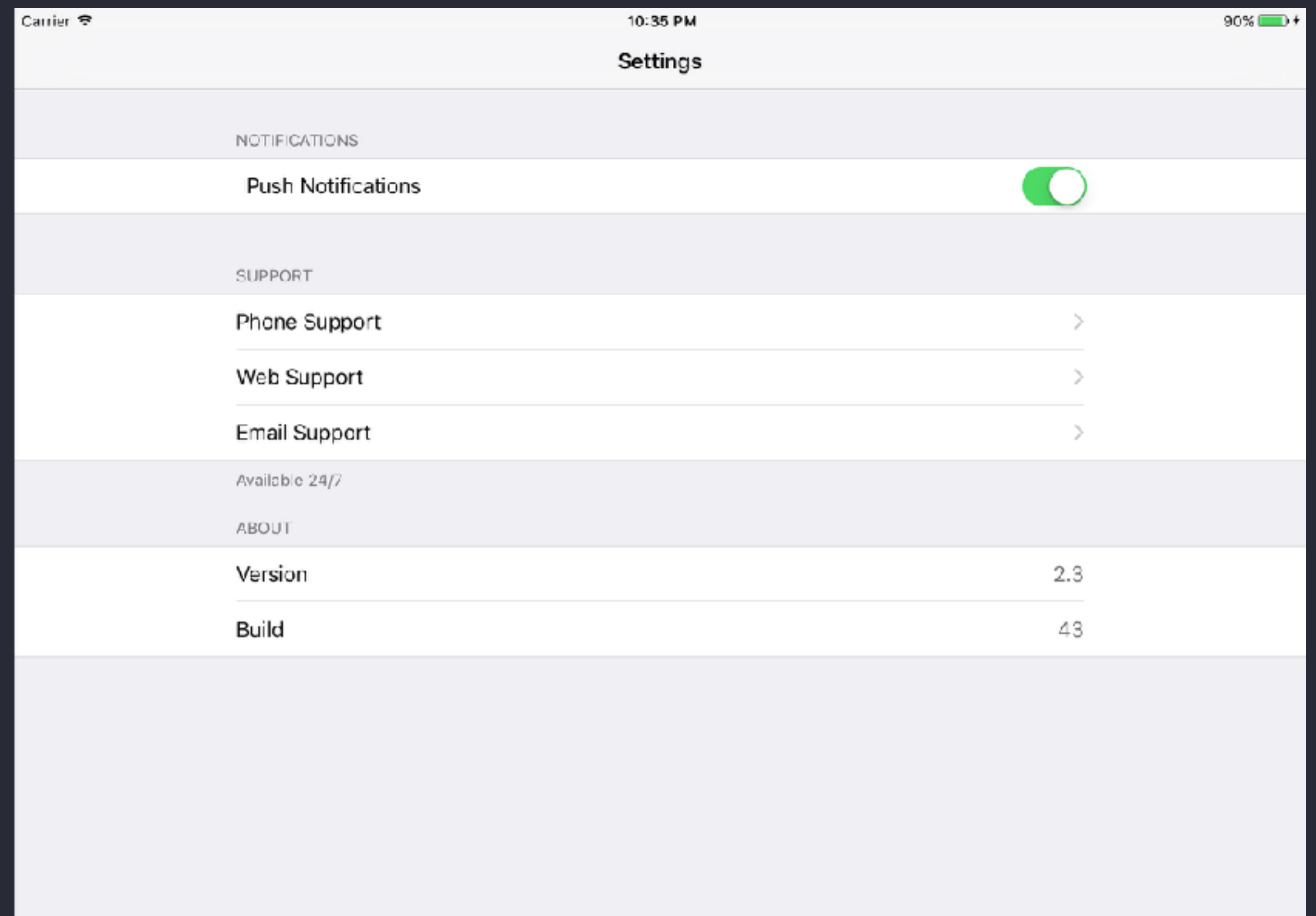
# Anatomy of a table view - cells

→ Set `tableView.cellLayoutMarginsFollowReadableWidth` to `true`

Default



Adjusted



# Index paths

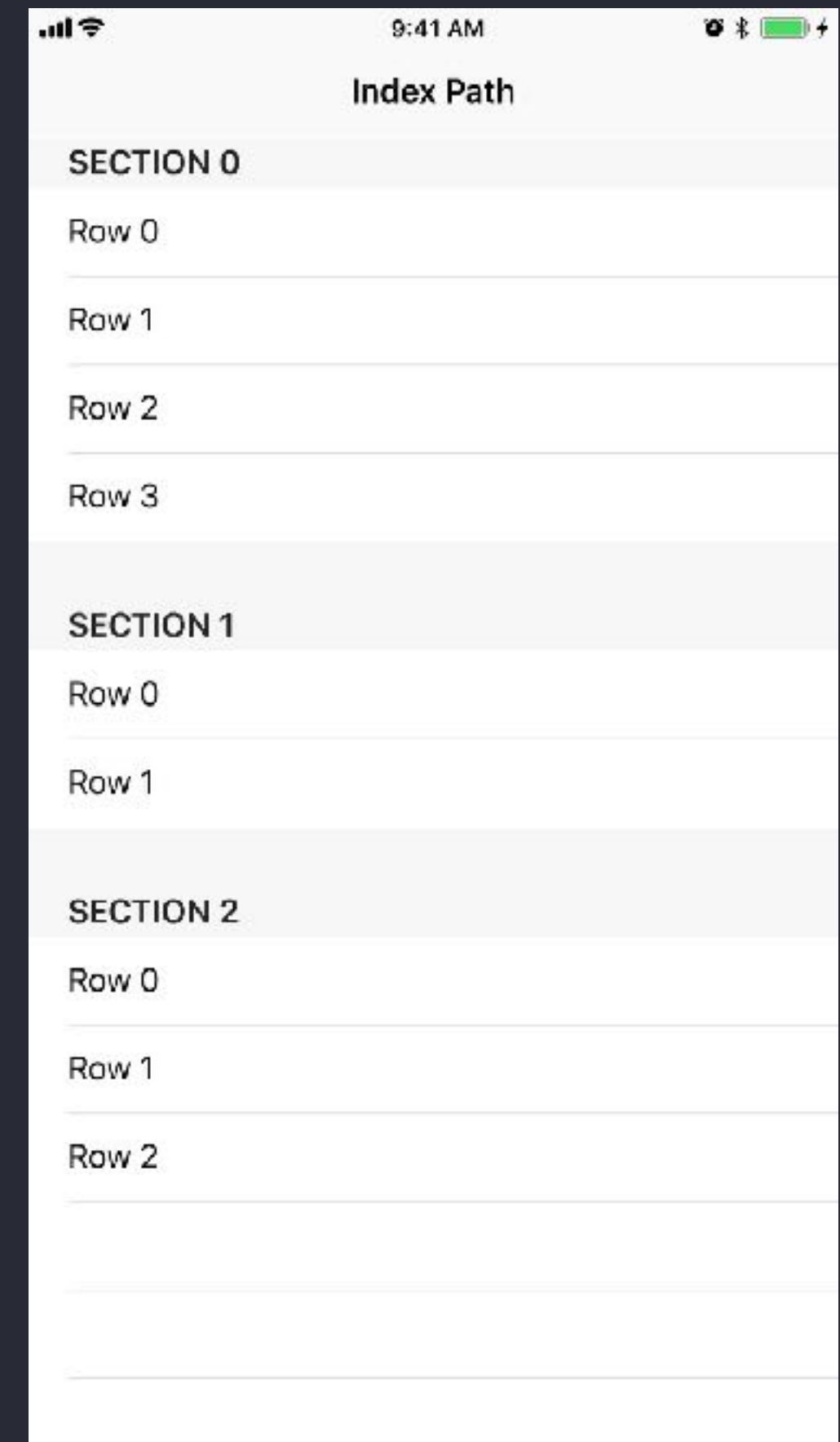
Points to a specific row in a specific section

Accessible through the row and section properties

→ `indexPath.row`

→ `indexPath.section`

Values are zero-based



# Arrays and table views

- Collection of similar data
- Typically backed by a collection of model objects

```
var emojis: [Emoji] =  
[Emoji(symbol: Character("😊"), name: "Grinning Face", description: "A  
typical smiley face.", usage: "happiness"),  
  Emoji(symbol: Character("😞"), name: "Confused Face", description: "A  
confused, puzzled face.", usage: "unsure what to think; displeasure"),  
  Emoji(symbol: Character("😍"), name: "Heart Eyes", description: "A  
smiley face with hearts for eyes.", usage: "love of something;  
attractive")]
```

# Arrays and table views

## Cell dequeuing

- Table views only load visible cells
- Saves memory
- Allows for a smooth flow when scrolling

```
let cell: UITableViewCell =  
tableView.dequeueReusableCell(withIdentifier: "Cell", for:  
indexPath)
```

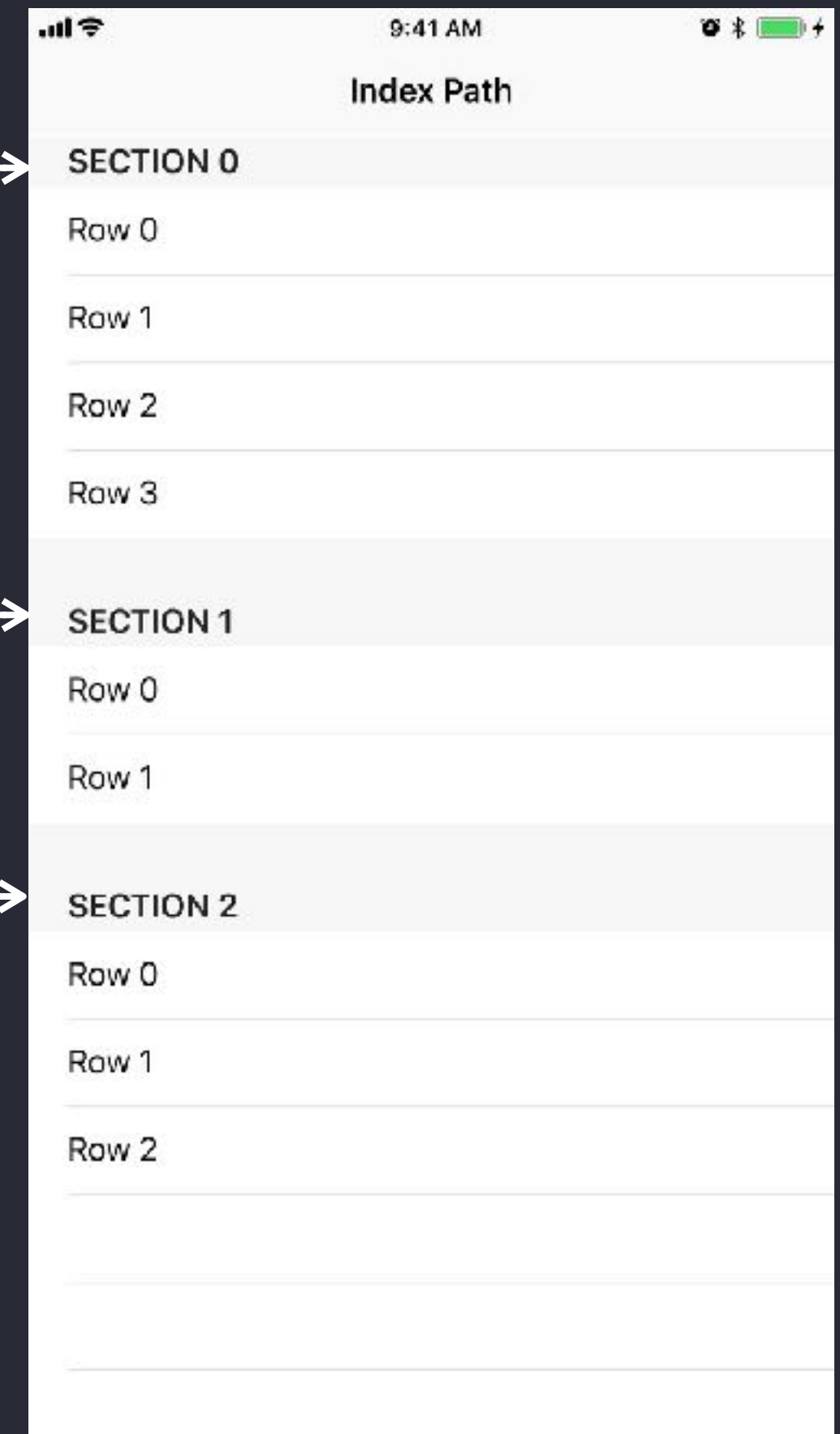
# Table view protocols

- UITableViewDataSource  
Provides data for populating sections and rows
- UITableViewDelegate  
Customizes appearance and behavior

# UITableViewDataSource

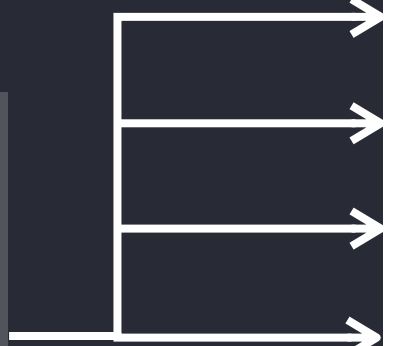
```
optional func numberOfSections(in  
tableView: UITableView) -> Int
```

- If function isn't provided, the table view assumes one section



# UITableViewDataSource

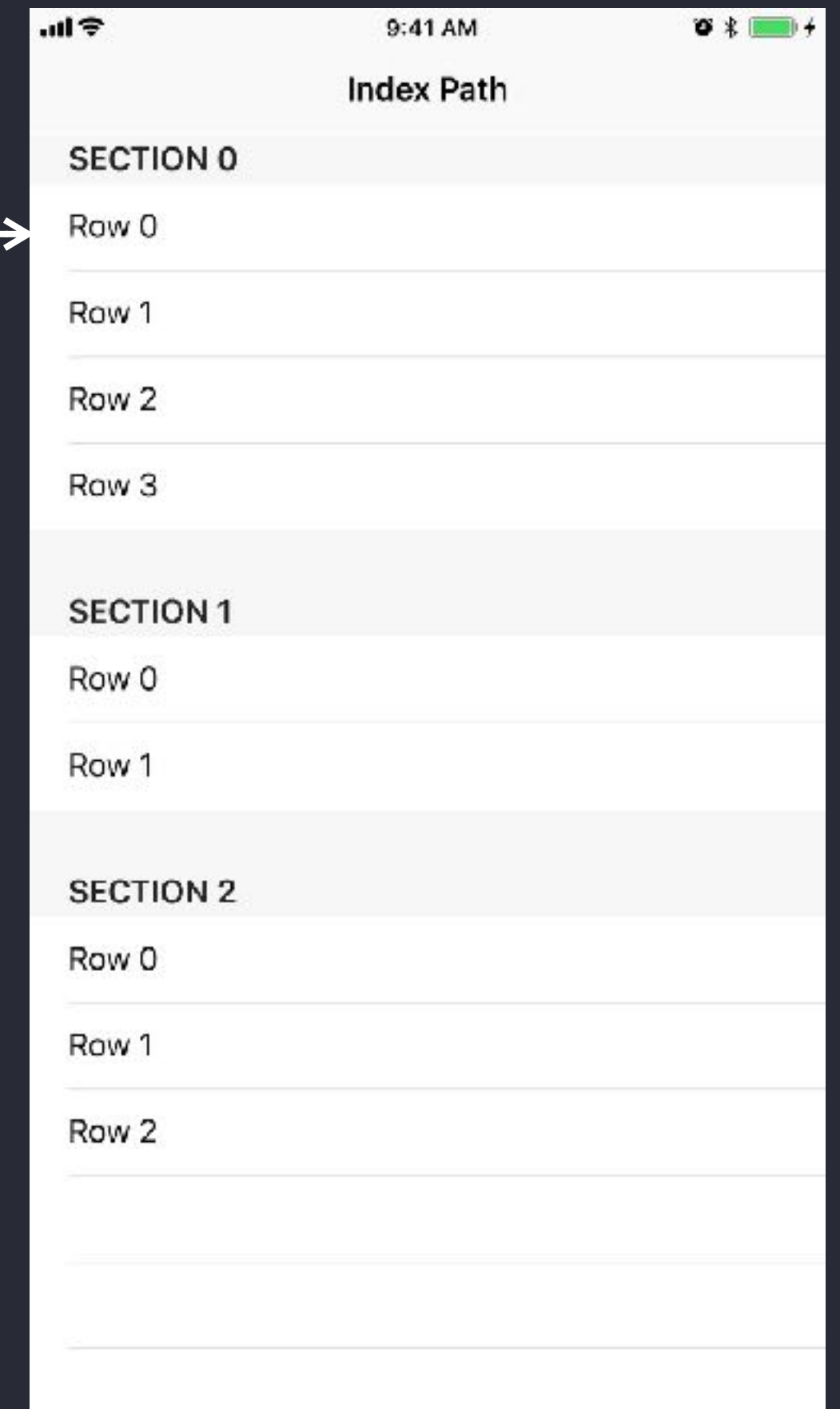
```
func tableView(_ tableView: UITableView,  
    numberOfRowsInSection section: Int) -> Int
```



Index Path
SECTION 0
Row 0
Row 1
Row 2
Row 3
SECTION 1
Row 0
Row 1
SECTION 2
Row 0
Row 1
Row 2

# UITableViewDataSource

```
func tableView(_ tableView: UITableView,  
    cellForRowAt indexPath: IndexPath) ->  
    UITableViewCell
```





# UITableViewDelegate

- Responding to accessory view interaction

```
tableView(_:accessoryButtonTappedForRowWith:)
```

- Responding to user interaction

```
tableView(_:didSelectRowAt:)
```

# Reload data

```
reloadData()
```

# The End.