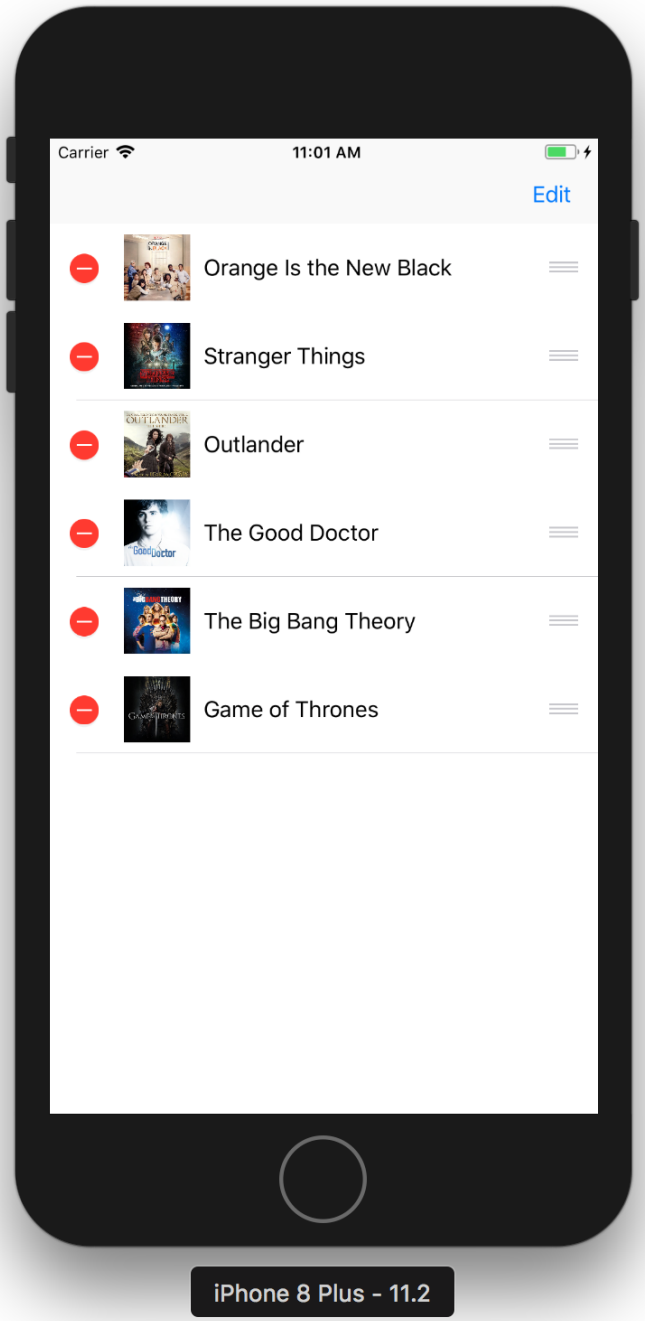
iOS Workshop Step by Step Guide

Creating an iOS app that displays a list of our favourite shows

Helen Brown & Erin Gallagher

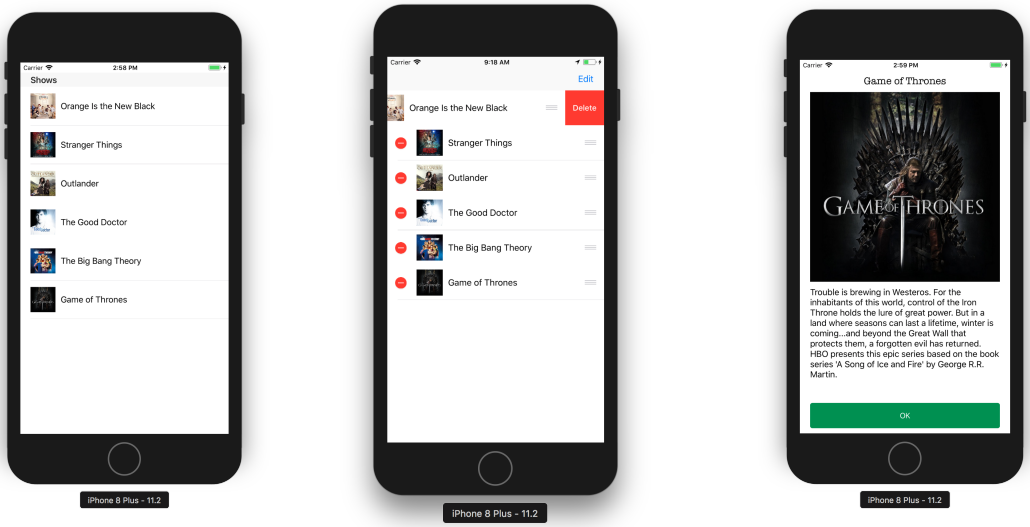


Getting Started

This is a beginner's tutorial on iOS development basics. The application you will learn how to build will maintaining a list of your favourite TV Shows that you can change the order of or delete content from and, when clicked, will open a pop-up with the show's details and a photo. To get started you will need Xcode 9.2 as well as Swift 4 which comes with it.

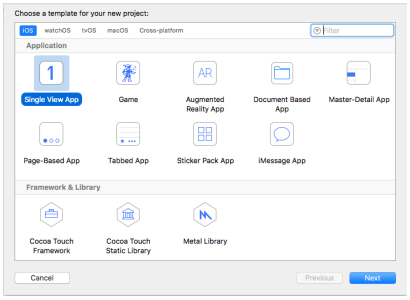
Xcode 9.2 Download Link: https://itunes.apple.com/ca/app/xcode/id497799835?mt=12

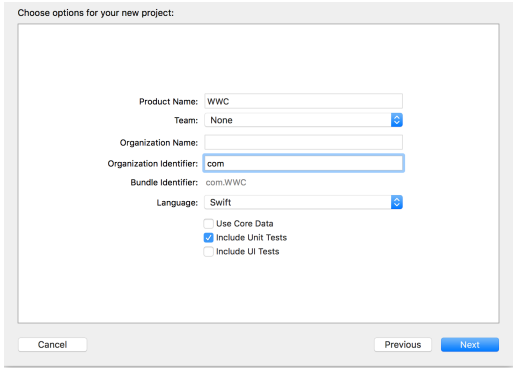
What Are We Working Towards?



Part 1: Creating a list of favourite TV Shows

Step 1: Creating A New Project & Adding a TableView

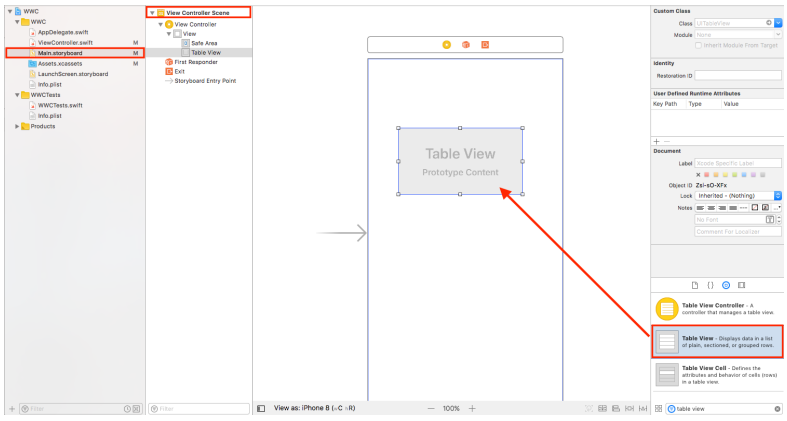




1. Start Xcode and click **Create a new Xcode Project**.
2. Choose the **Single View App** template and click **Next**.
3. Give your project a **Product Name**, we have chosen *WWC*.
4. You can leave **Team** as None, and **Organization Name** empty.
5. Ensure Swift is the selected **Language**. Keep only the **Include Unit Tests** box checked.
6. On clicking **Next,** choose where to save your project, and select **Create**.

Table View Basics

Add Table View Object

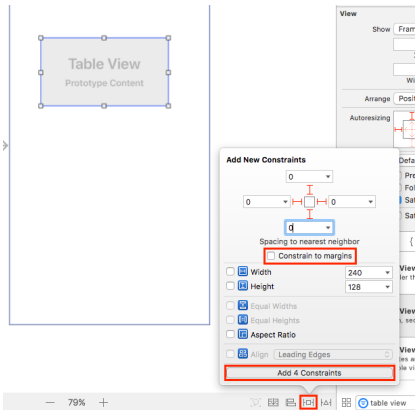


Click on **Main.storyboard** in the project navigator to view the first screen we will be working on. Expand the drop-down arrows under **View Controller Scene** until you can see the **Safe Area** component.

Next go to the **Object Library**  and filter on table view. **Click** and **drag** the **Table View** component onto your scene.

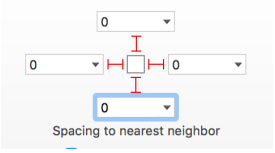
Safe Area Layouts are a convenient layout tool that allow the user to constrain their views to edges of the screen that are deemed “safe” - they may also have an additional view elements like a navigation bar that will also be presented on the screen, but you want to make sure your views don’t accidentally fall behind them.

Adding Table View Constraints

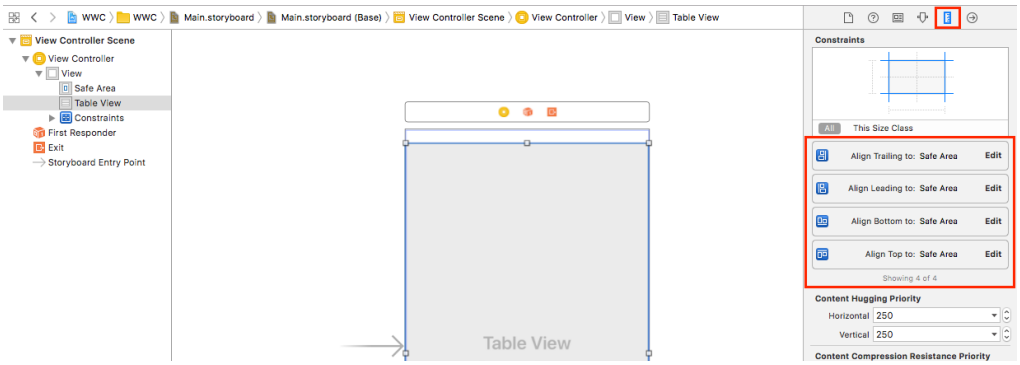


Select the Table View in the scene and while it’s selected click on **Add New Constraint** button in the bottom right hand corner of Xcode.

Change the **Top**, **Bottom**, **Right** and **Left** constraints to **0** and **uncheck** the **constrain to margins** check box. Finally, click **add 4 Constraints** button in order to add the constraints to the scene.



Note: To ensure all 4 constraints are included ensure the dotted red lines between the 4 boxes become solid red lines.



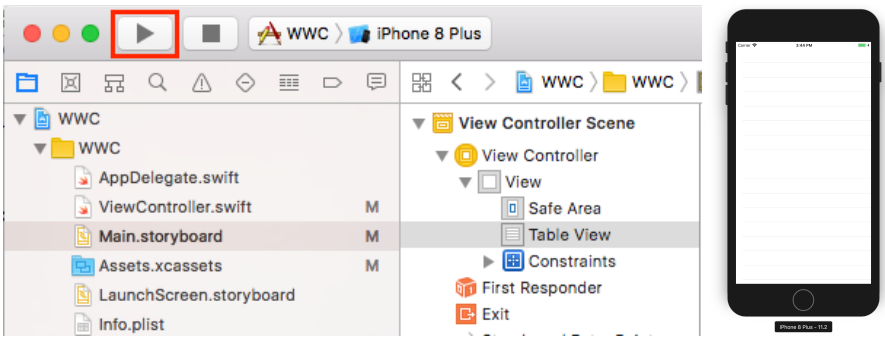
After adding the constraints confirm they were added correctly by navigating to the **Size Inspector** tab within the right-hand panel.

Under the **Constraints** panel you should see 4 boxes with the contents:

1. Align Trailing to: Safe Area

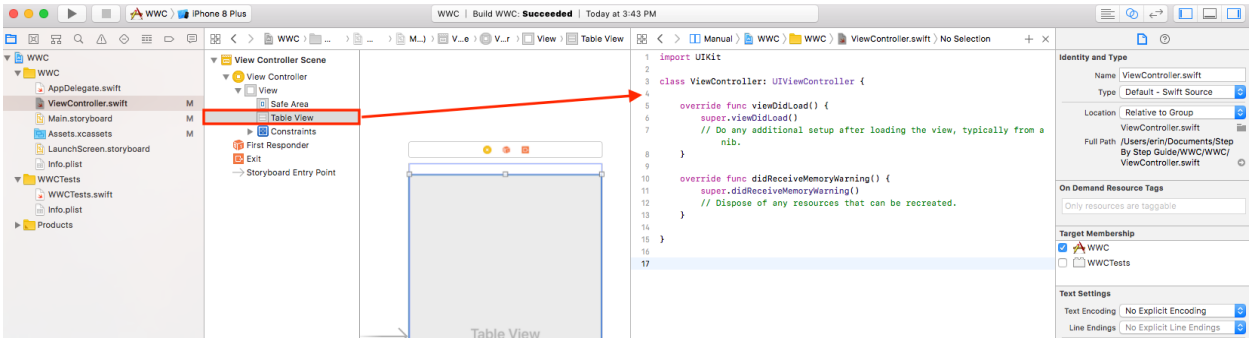
2. Align Leading to: Safe Area
3. Align Bottom to: Safe Area
4. Align Top to: Safe Area

Let's Build for The First Time!



Click the **Build and the Run (or use COMMAND-R)** button at the top left-hand corner of Xcode. It looks like a play button. The iPhone 8 Plus simulator is selected by default and once the build has completed it will open in a separate window to display the application so far. Once loaded, you will see a white screen with lots of **Table View Rows**. If you click and drag on the simulator you are able to scroll through all the rows in the **Table View**.

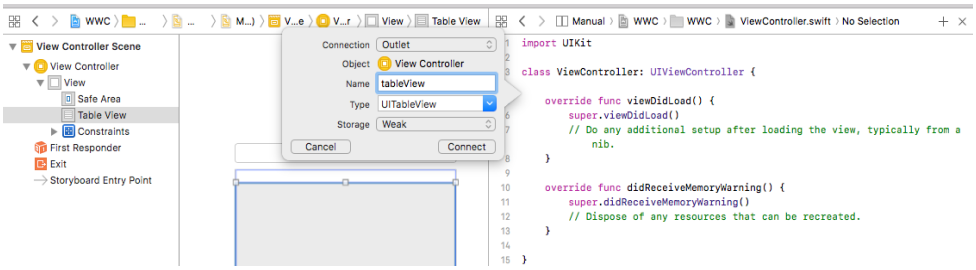
Step 2: Connecting a Table View to a View Controller



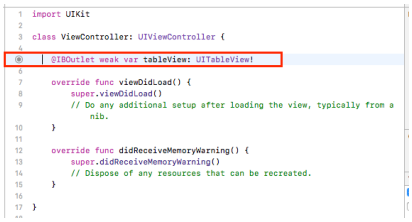
In order to add logic to the Table View we have to connect it to the View Controller.

For convenience we want to have 2 files displayed side by side. To do this start with the **Main.storyboard** file open**.** Next, hold the **Option Key** and click on **ViewController.swift** in the Project Navigator.

To connect the View Controller Scene with the ViewController class, find the **Table View** component under the **View Controller Scene** drop downs. Then hold the **Control key** while clicking the **Table View** component and drag it to the top of the **ViewController** class file beneath the class definition (left-hand file beneath line 3).



On releasing the cursor and the Control key, a grey pop up will appear providing options for creating an **Outlet**. Leave all fields as their default and provide the **Outlet** with a name, we used *tableView*. Once finish click the **Connect** button.



Now you will see a new line of code was added to your **ViewController** class. This line of code starting with **IBOutlet**, is the connection between your Table View component within the storyboard and your **ViewController** class.

The **dark grey circle** on the left-hand side replacing a line number shows you the connection. If this circle was not filled in the connection would be broken.

Step 3: Set up the Data Source

Change to the **ViewController.swift** file. We will start by deleting the entire `didReceiveMemoryWarning()` function because there is no need to override that functionality.

Also, switch to the **Standard Editor**  so you have some more space to work with.

```
override func viewDidLoad() {
     super.viewDidLoad()

     //table view properties
     tableView.dataSource = self
     tableView.tableFooterView = UIView(frame: CGRect.zero)
     tableView.estimatedRowHeight = 10
     tableView.rowHeight = UITableViewAutomaticDimension
}
```

The `viewDidLoad()` function is called once when the View Controller is created. During the creation process we want to add attributes to our **Table View** in order to control its properties. Above we added 4 lines below `super.viewDidLoad()` to add those required properties.

1. `dataSource` – provides the View Controller with the ability to give the table data
2. `tableFooterView` – configures the footer with an empty view
3. `estimatedRowHeight` – set to **10**
    a. must be provided in order to allow for the View Controller's auto layout to determine the correct dynamic size given the content of each cell
4. `rowHeight` – set to **UITableViewAutomaticDimension**

```
// ...

} //Closing bracket for ViewController class

extension ViewController: UITableViewDataSource {

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        //Only 1 TV show exists as a placeholder
        return 1
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
{
        //Provide a placeholder cell for the row
        return UITableViewCell()
    }

    func tableView(_ tableView: UITableView, titleForHeaderInSection section: Int) -> String? {
        //Table View name displayed in header
        return "Shows"
    }
}
```

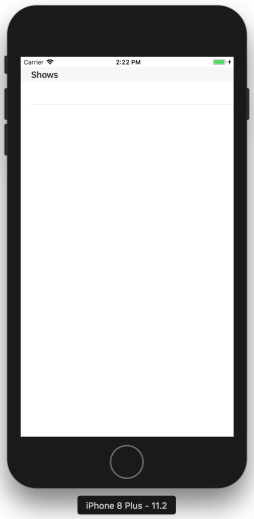
Now that we have allocated a dataSource we use an extension to provide it with data. The Table View dataSource provides a number of methods which allow you to provide it with your data in a way it can interpret.

Above we add the **UITableViewDataSource** extension outside of the ViewController class (below the last closing bracket "}"). To conform to the required methods of the **UITableViewDataSource** protocol we must implement two functions:

1. `numberOfRowsInSection`
2. `cellForRowAt`

In addition to those three we will add a fourth function to provide the Table View with a title:

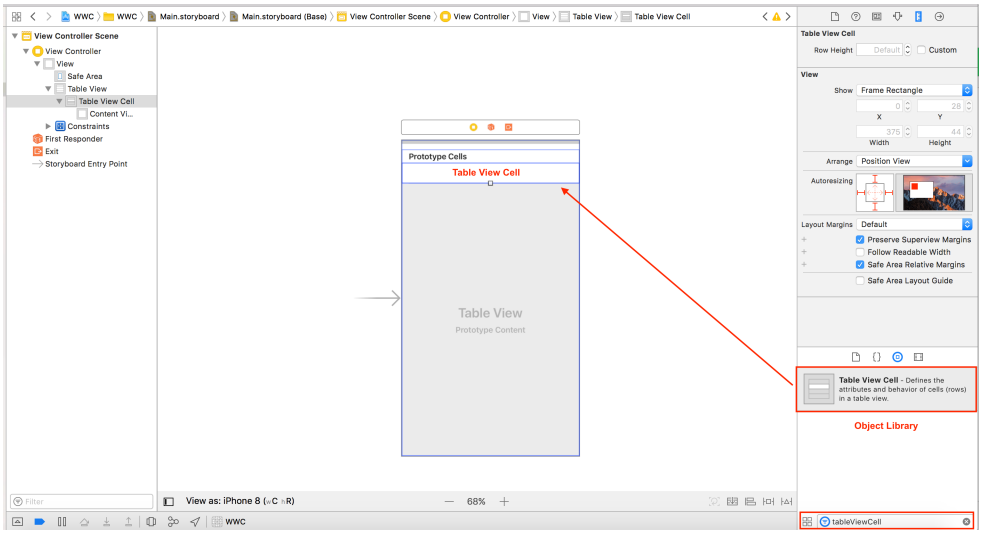
3. `titleForHeaderInSection`



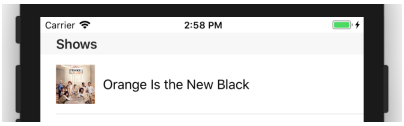
Now **Build** the project and take a look at the results so far.

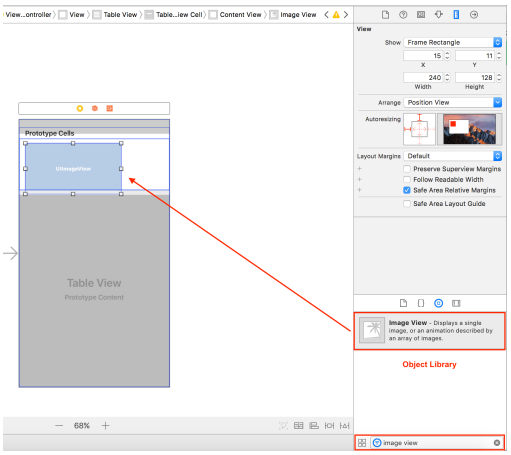
At the top of the simulator you can see the title which was set to *Shows*. Below the title there is now only one **Table View Cell**. There is only one because we returned *1* for the number of sections and *1* for the number of rows in that section. Finally, the cell is empty because we create an empty cell using `UITableViewCell()` function.

Step 4: Creating A Table View Cell for TV Show Content

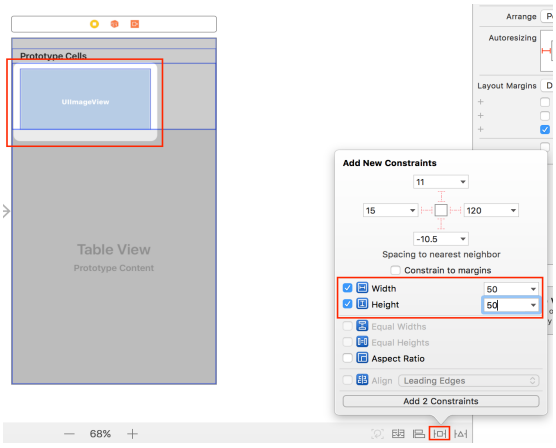


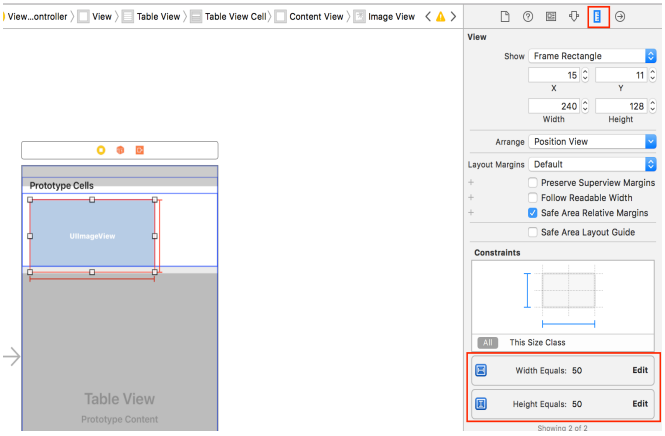
Return to the **Main.storyboard** file. In the **Object Library** filter on *Table View Cell*. Similar to adding the Table View to the scene in a previous step, drag the **Table View Cell** component onto the Table View.





In order display an image and label within the Table View Cell (finished product image above on left) we need to add some more components. In the **Object Library** filter on *Image View*. Next drag the **Image View** component onto the **Table View Cell**.





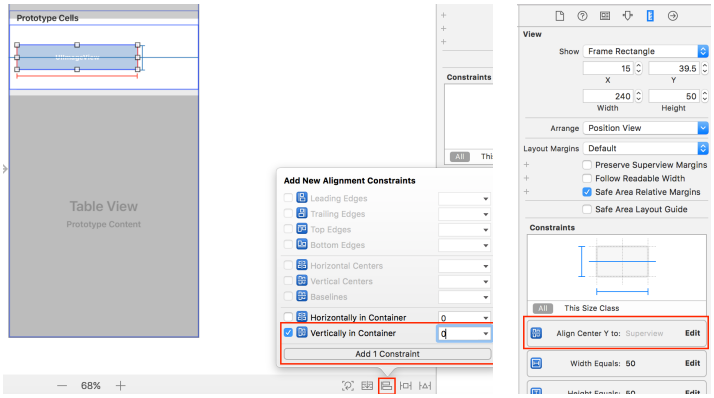
Next, we have to add constraints in order to have the image view centered and with correct height and width. First, select the **Image View** and the click on the **Add New Constraints** button. Check the height and width and then set both to *50*. Finally Click the **Add 2 Constraints** button.

To confirm the constraints were added correctly check the **Size Inspector** tab.

Under the **Constraints** panel you should see 2 boxes with the contents:

1. Width Equals: 50
2. Height Equals: 50

Note: The **Image View** will highlight red to indicate it requires more constraints. Don't worry, we will add those next.



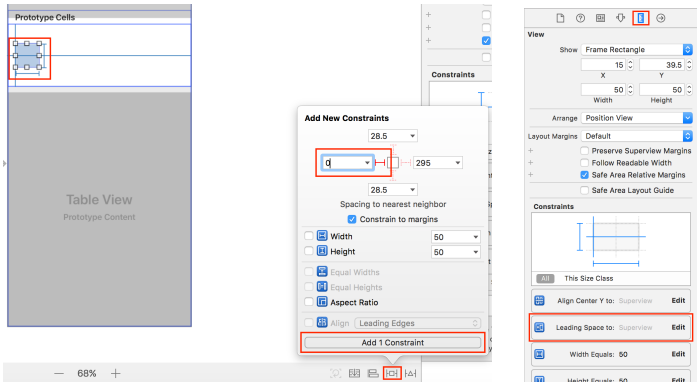
To center align the **Image View** vertically within the **Table View Cell** select the **Image View** and then click on the **Add New Alignment Constraints** button.

Check the **Vertically in Container** box and leave the value as *0*. Then click the **Add 1 Constraint** button.

To confirm the constraints was added correctly check the **Size Inspector** tab.

Under the **Constraints** panel you should see a box with the contents:

1. Align Center Y to: Superview



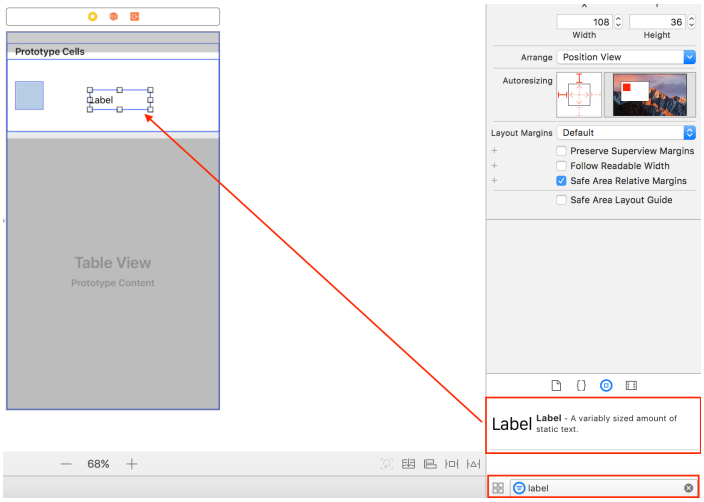
The last constraint required for the **Image View** is it's **leading** or left constraint. Select the **Image View** and then click on the **Add New Constraint** button.

Enter 0 in the left box and ensure the red dotted line become solid. Next, leave the **Constrain to margins** box checked to leave in the default margin on the table view cell. Finally Click the **Add 1 Constraint** button.

To confirm the constraints was added correctly check the **Size Inspector** tab.

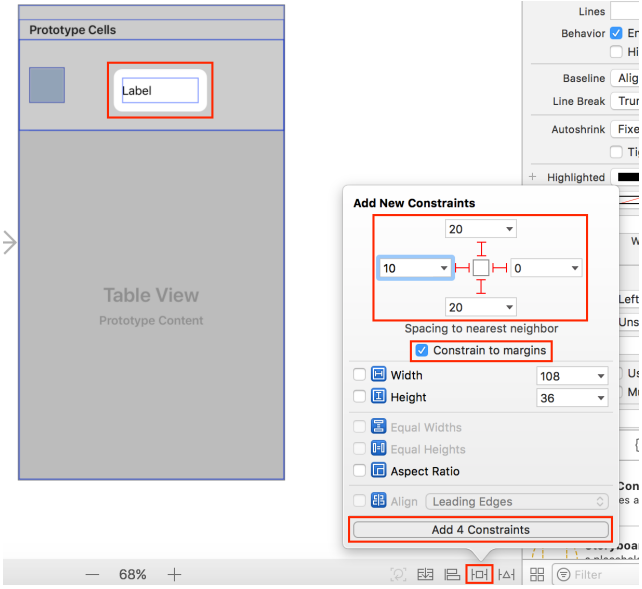
Under the **Constraints** panel you should see a box with the contents:

1. Leading Space to: Superview



Next, we will add the Label to the cell in order to display the TV Show name. In the **Object Library** filter on *label*.

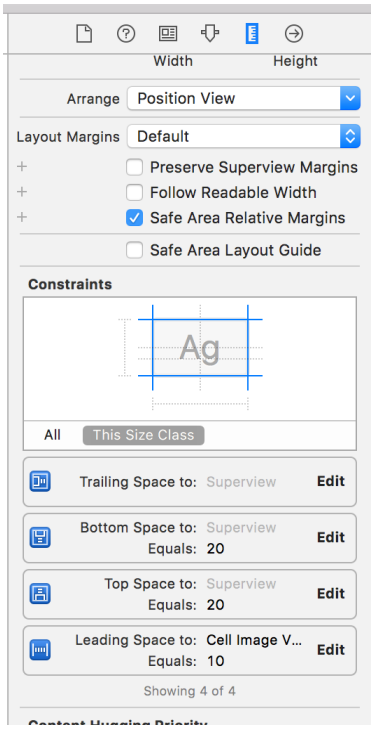
Next drag **Label** component onto the **Table View Cell**.



In order to position the label correctly in the field we need to add constraints. The **Label's** left side should be **10px** from the **image view**. its top and bottom edges should be **20px** from **Table View Cell**. And finally, the **Label's** right edge should be **0px** from the **Table View Cell's** margin.

While selection the label component, click on the **Add New Constraints** button to input those values.

Finally click **Add 4 Constraints** to add the constraints to the **Label.**

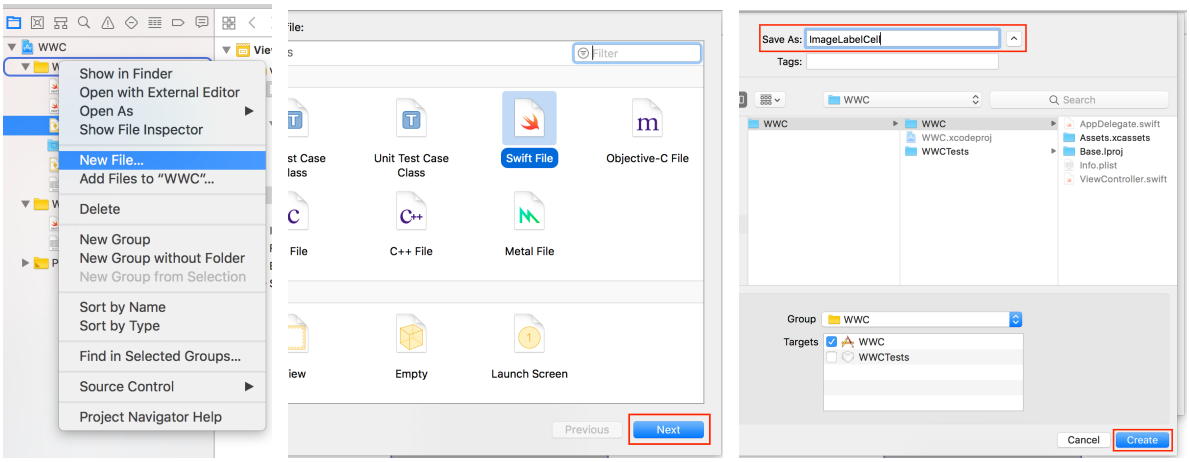


To confirm the constraints was added correctly check the **Size Inspector** tab.

Under the **Constraints** panel you should see 4 boxes with the contents:

1. Trailing Space to: Superview
2. Bottom Space to: Superview, Equals: 20
3. Top Space to: Superview, Equals: 20
4. Leading Space to: Image View, Equals 10

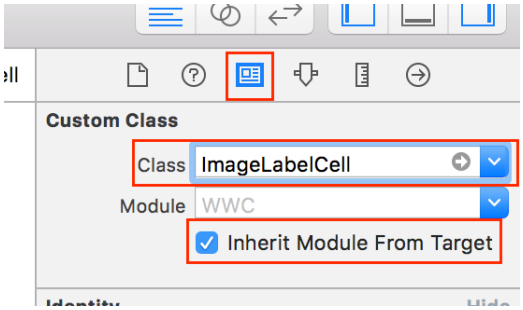
Step 5: Creating a Table View Cell Class



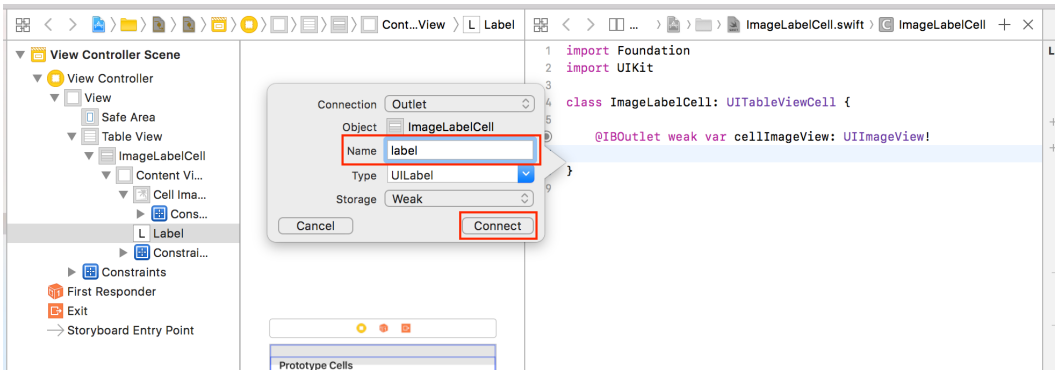
In order to connect the **Image** and the **Label** logic to the **View Controller** we must create a class for the cell. **Right Click** the WWC folder holding your ViewController.swift and Main.storyboard file and select **New File**. Select **Swift File** and then click **Next**. Finally Create a name for your Table View Cell and click **Create**. We chose *ImageLabelCell.*

```
import Foundation
import UIKit

class ImageLabelCell: UITableViewCell { }
```

Next, add the class definition and import **Foundation** and **UIKit**.



Next go to the **Attributes Inspector** and add *ImageLabelCell* as the **Class**. Then check the **Inherit Module From Target** checkbox.



In order to set a value for the label and an image in the Table View Cell we need to create a connection between the **ImageLabelCell Class** and the **Main.storyboard file**. To do so we need to create an **IBOutlet** for both.

To create the outlet **Option click** the **Image** component and drag it into the ImageLabelCell Class.
Provide the image with a name, we chose *cellImageView*, and click **Connect**. Do the same thing with the label using the **option click** to drag it to the class. Once completed your file should have **2 IBOutlets** with filled in circles.

Note: To get the split screen view, start with the Main.storyboard file open and then **Option click** the ImageLabelCell.swift file.

```
import Foundation
import UIKit

class ImageLabelCell: UITableViewCell {
       @IBOutlet weak var cellImageView: UIImageView!
       @IBOutlet weak var label: UILabel!

       func setup(labelName: String?, imageName: String?) {
        // Set the label of the cell to be the name we passed in
        self.label.text = labelName

        // Make sure we have the show's picture name before we try creating an image out of it
        guard let iconName = imageName else { return }
        self.cellImageView.image = UIImage(named: iconName)
    }
```

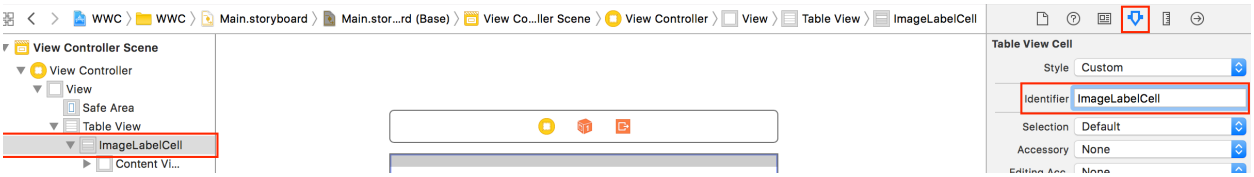
Next we will create a `setup()` function that sets the label name and image for the cell. Both attributes can be set to any text of type **String**. The **imageName** attribute accepts a file name and then converts that into a **UIImage**. The **label** attribute accepts text and sets it within the label exactly as given.

Switch to the **Standard Editor**  so you have some more space to work with.

Step 6: Adding our ImageLabelCell into our Tableview

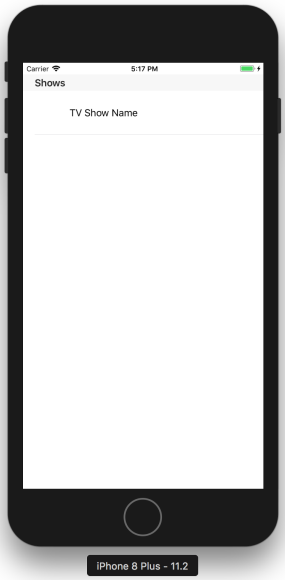
We need to register the the cell to be used by the tableView. We do this by adding a unique identifier to the cell. We like to use the cell name for this identifier since we know the cell name has to be unique.

While selecting the **Table View Cell**, Click the **Attributes Inspector** and add *ImageLabelCell* as the **Identifier**.



Now we can have our tableView use the cell. In the `CellForRowAt()` function in the **ViewController** class, remove `return UITableViewCell()`from the file and add the code above. Next, we will set the show name as *TV Show Name* as a placeholder and leave the image empty (`nil`).

```
/* cellForRowAt will be called as many times as you have “numberOfRows". */
    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

        let cell = tableView.dequeueReusableCell(withIdentifier: "ImageLabelCell") as! ImageLabelCell

        // Grab the show from the model
        let tvShow = shows[indexPath.row] as Show

        // Set the label using the placeholder string
        let tvShowName = “TV Show Name"
       cell.setup(labelName: tvShowName, imageName: nil)
        return cell
    }
```

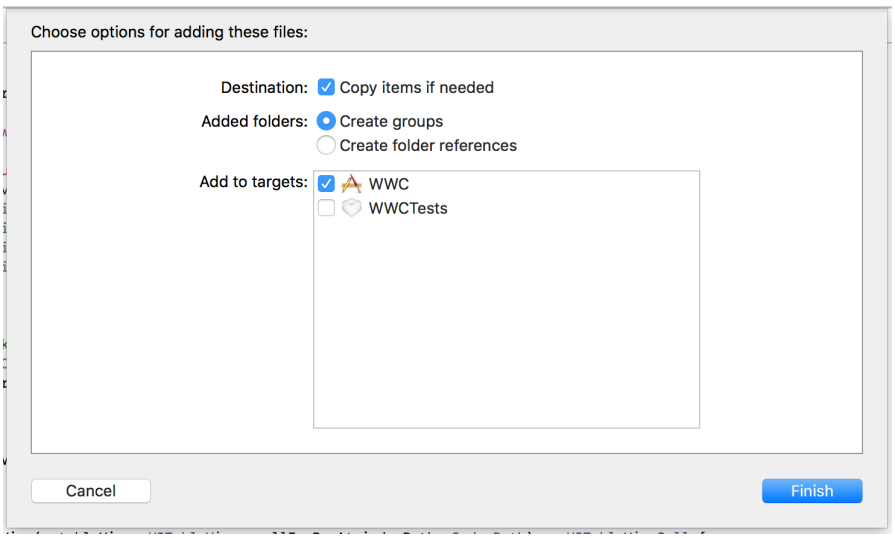


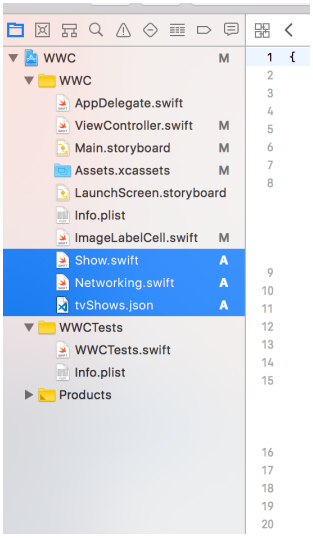
Select **Build** and see the Results! 

Since no image has been provided it is left blank, but *"TV Show Name"* can be seen in the Table View Cell in the Label position.

Step 7: Adding Sample Data

For this tutorial we wanted to display a variation of data so we are using the **iTunes API** to retrieve data on popular TV Shows. Networking and JSON parsing is more of an intermediate level iOS Development skill so we will be including details within the Appendix. For now we are going to drag three files into the project to complete these task for us.



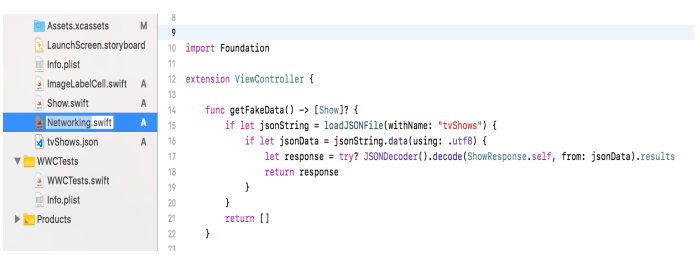


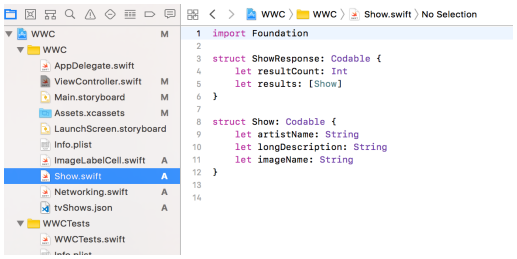
Drag the three following files into the WWC folder:

1. Shows.swift
2. Networking.swift
3. tvShows.swift

On dragging each file into the project, a popup will appear and ensure the **Destination checkbox** is checked, **Create Groups radio button** is selected and **WWC** is the **target checkbox selected**. Then click finish.

Note: See the Appendix for more details on these three files





For this tutorial we will be using stubbed JSON data from the iTunes API. **tvShows.swift** contains a sample JSON the iTunes API could retrieve. **Shows.Swift** provides a model for the data we will be using and displaying in our app.

JSON (JavaScript Object Notation) is a way of organizing and presenting data that is easy for humans to read and write, and easy for software to create and parse (sort through). A lot of times apps will need data that is kept in remote data storage systems. To use this data, they need to call them, and then make sense of what the system sends back as a response. To do this, they agree on the format in which the data will be sent and received. One of the formats commonly used is JSON. See "tvShows.json" for an example.

A show, for our example, has three attributes:

1. artistName – Name of the TV Show
2. longDescription – TV Show Description, long version
3. imageName – image's file name

In order to get those shows from the JSON file to our ViewController we will use the `getFakeData()` function from the **Networking.swift** file.

```
// Our "dataSource" or "model" is the data that we want to display
var shows: [Show]?

override func viewDidLoad() {
    super.viewDidLoad()

    // tableView properties
    tableView.dataSource = self
    tableView.tableFooterView = UIView(frame: CGRect.zero)
    tableView.estimatedRowHeight = 10
    tableView.rowHeight = UITableViewAutomaticDimension

    // get tv show data from our JSON
    shows = getFakeData()
```

In order to introduce the data into our View Controller we will call `getFakeData()` within the `viewDidLoad()` function. This function is called once when the application first loads and will add the TV show data to a variable called **shows.** Next, we have to add this variable to our class which will contain a list of TV shows for the Table View. Add this line above the `viewDidLoad()` function. Since it is possible the iTunesAPI could return no results leave shows as an optional variable.

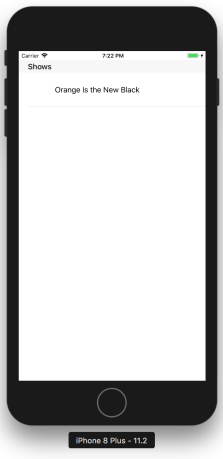
Accessing Data for Display

```
/* cellForRowAt will be called as many times as you have "numberOfRows". Each time it is called,
     * we will make a new cell, grab ONE show from the model, and use that show's data to populate
the cell.
     */
    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
{
        // Make sure our shows has data, otherwise we'll just return an empty tableViewCell
        guard let shows = shows else { return UITableViewCell() }

        // If 'shows' has some tv shows in it, make a cell
        let cell = tableView.dequeueReusableCell(withIdentifier: "ImageLabelCell") as!
ImageLabelCell

        // Grab the show from the model
        let tvShow = shows[indexPath.row] as Show

        // Set the label and the image using the data from our chosen tvShow
        let tvShowName = tvShow.artistName
        cell.setup(labelName: tvShowName, imageName: nil)
        return cell
    }
```

The final steps are to access the **artistName** attribute from the **Show** Object Model and add it to the **Label** field of our **ImageLabelCell**. Use a **guard** to ensure that the shows variable is not empty before trying to retrieve the artist name attribute. Then replace *"TV Show Name"* with `shows[indexPath.row].artistName`. **indexPath.row** provides an integer for the show number we are trying to display.

Select Build and see the Results!

Since we have set `numberOfRowsInSection()` (number of TV shows) to one, only the first show's title will be displayed in the list.

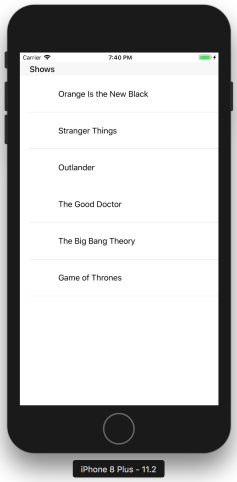


Step 8: Determining the Number of TV Shows

At this point there is still only 1 show displayed because we have set the number Table View Rows to equal one. In order to display all the TV shows within the Table View we must count how many were returned from the ITunes API JSON.

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        // Make sure our shows has data, otherwise return 0 rows
        guard let shows = shows else { return 0 }

        // If shows exist, count the number of tv shows we put in the list and return that many rows.
        return shows.count
```

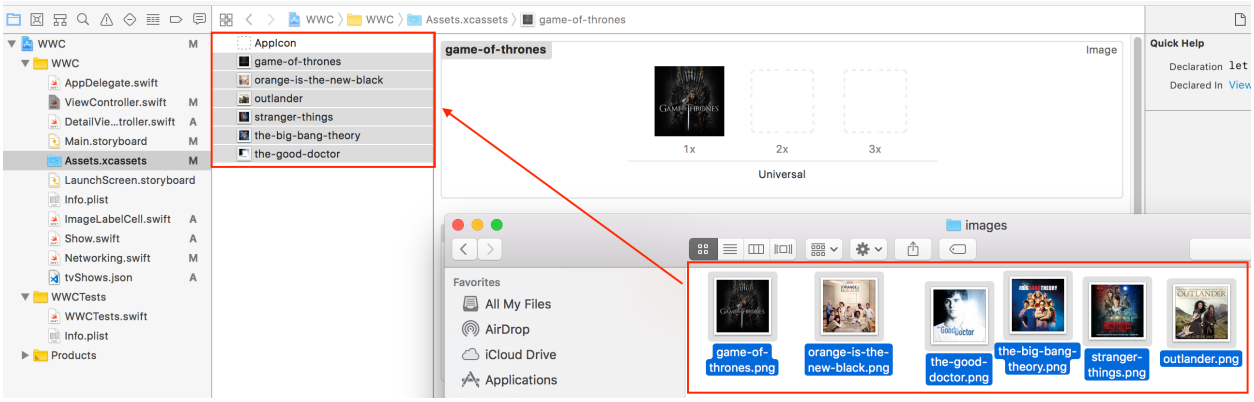
Similar to the `cellForRowAt()` function, we must use a **guard** to ensure the **shows** object exists before trying to count them. If the **Shows** object does not exist **return 0** because there are no TV shows to display. Next, if TV shows do exist, you want to count the number by using the `.count` function.



Select **Build** and see the Results!

Step 9: Adding Image Assets to A Project

The Shows Object contains the list of TV shows with the **imageName** for the TV show pictures. Since we are not making a live Network call for this tutorial we are going to add the image assets to the project manually.



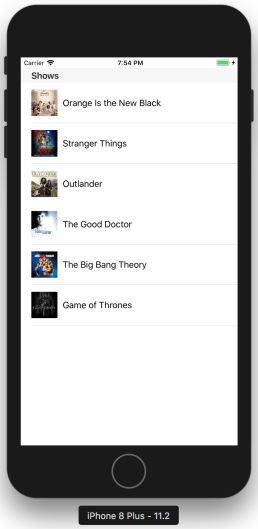
Select All the Images from the resources folder provided and drag them into the Assets right panel. The file names have been automatically added and the images have been place in the **1x** position.

Note: Provided with this tutorial is a **folder with Image assets**. Use these images for this step.

Connecting Image File names to Image Assets

```
    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
        // Make sure our shows has data, otherwise we'll just return an empty tableViewCell
        guard let shows = shows else { return UITableViewCell() }

        // If 'shows' has some tv shows in it, make a cell
        let cell = tableView.dequeueReusableCell(withIdentifier: "ImageLabelCell") as!
ImageLabelCell

        // Grab the show from the model
        let tvShow = shows[indexPath.row] as Show

        // Set the label and the image using the data from our chosen tvShow
        let tvShowName = tvShow.artistName
        let imageName = tvShow.imageName
        cell.setup(labelName: tvShowName, imageName: imageName)
        return cell
    }
```

Now that the assets are in the project we need to set the **imageName Show** attribute to the Table View Cell. In the **ViewController** class add one more line to the `cellForRowAt()` function. This line sets the imageName for the associated Show.

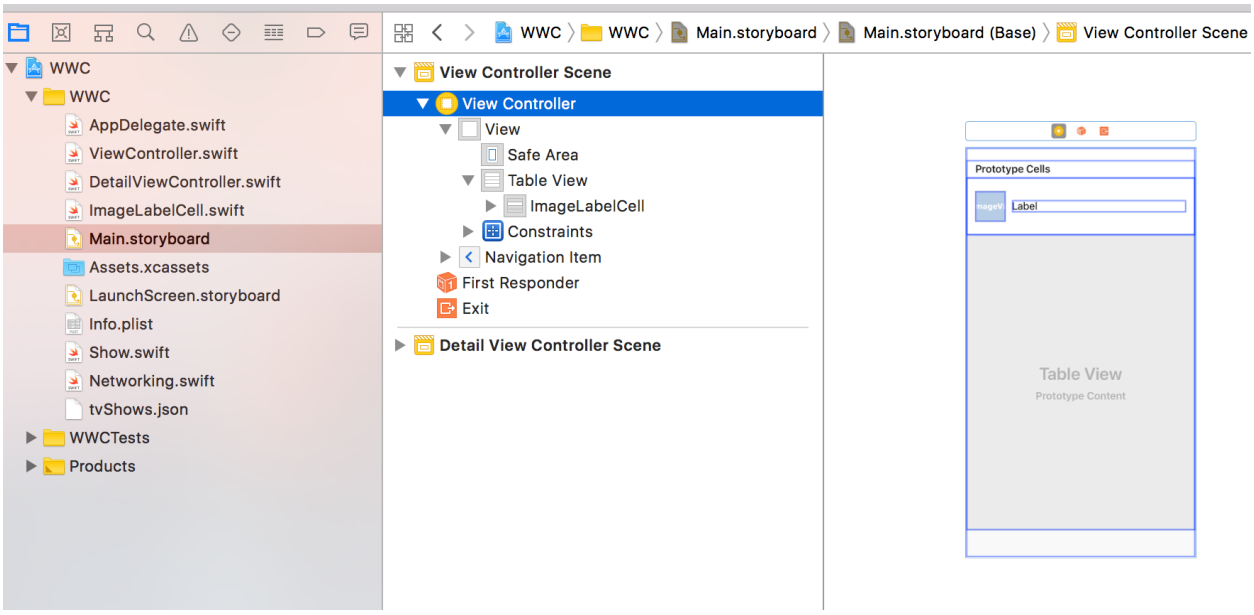


Select **Build** and see the Results!

Part 2: Deleting from your list & moving cells

Step 1: Add in an Edit button

In the Main.storyboard file, find your View Controller Scene and **select your View Controller** so you see the controller highlighted in the storyboard scene like so:



From your Xcode toolbar select **Editor > Embed in > Navigation Controller**

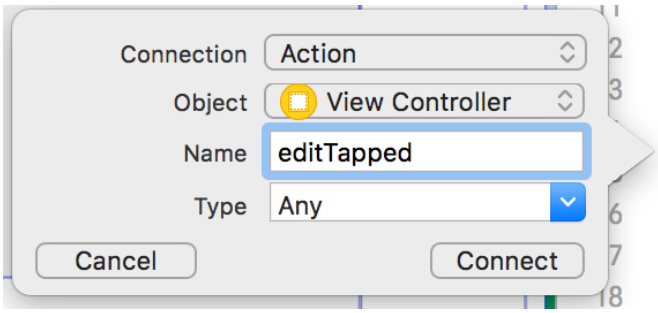
A navigation controller is responsible for presenting your content view controllers, but also has some views of its own. One of these views is what we call a “navigation bar” which you will now see at the top of the controller when you run your app. We will add our “edit” button here.

In your object library, drag a **Bar Button Item** to the right side of your navigation bar, and call it “Edit”.

Select your new Navigation Controller, and under the Attributes Inspector hit the checkbox that says “Is Initial View Controller”. This just tells the storyboard to go here first when you launch the app.

Step 2. Hook up the edit action

Use the assistant editor  with your Main.storyboard and your ViewController.swift



Control-drag your edit button to your View Controller and create an Action connection called “editTapped”. Xcode will create an action function called “editTapped” that will be called every time the edit button is tapped. We will use it to toggle the editing functionality on our view.

In our ViewController’s `override func viewDidLoad()` function, set the tableView to default to not editing when the app starts:

```
// set the original state of the tableview to not be editing
        self.tableView.isEditing = false
```

And in the editTapped function, set the editing functionality to be the *opposite* of whatever it is currently:

```
@IBAction func editTapped(_ sender: Any) {
        // set the isEditing property to be the opposite of whatever it is already.
        // this allows us to toggle between editing and not editing
        self.tableView.isEditing = !self.tableView.isEditing
    }
```

Run the app and see how it looks!

Step 3. Delete a show

Sick of seeing that one show in your list that we added into our default list that you just can’t stand? Let’s remove it.

When you ran the app with the edit mode, you probably saw that the iOS platform added in a delete button for you! It just doesn’t do anything yet.

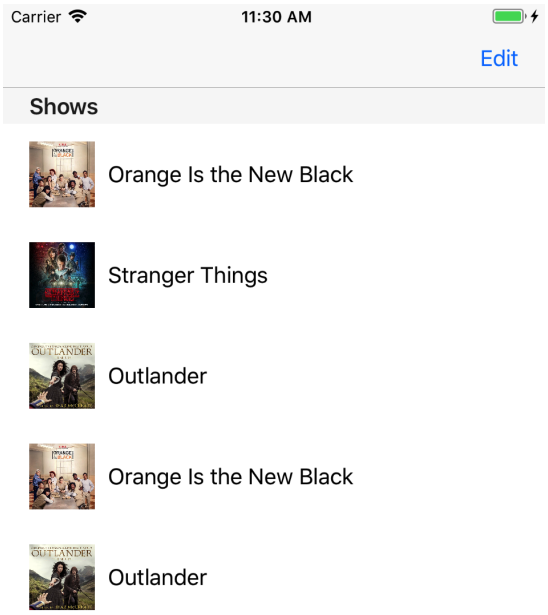
Below your ViewController’s viewDidLoad() function, add in the following function:

commitEditingStyle

```
func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
        //check to see if editing style is for deleting a row
        if editingStyle == .delete {

            //if true, remove the show from our list of shows
            shows?.remove(at: indexPath.row)

            //have the tableView delete the cell we selected
            tableView.deleteRows(at: [indexPath], with: .fade)
        }
    }
```

Run the app

Step 4. Change the order



Currently, the order of the shows is displayed according to how it is read from the JSON file.

But we can make a few quick changes so that the user can shuffle the order the shows are displayed in.

In ViewController, under the commit editingStyle function you just wrote, enable the moving functionality for your cells by writing introducing the following two functions:

canMoveRowAt

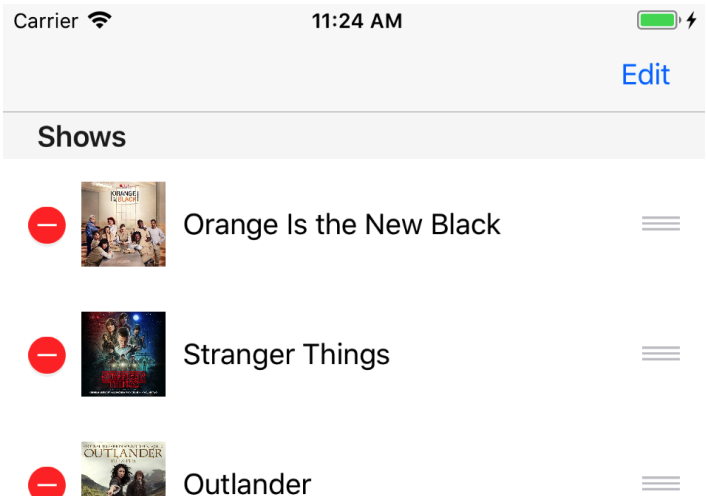
```
func tableView(_ tableView: UITableView, canMoveRowAt indexPath: IndexPath) -> Bool {
        /* In this function, we could check conditions on which rows are allowed to
move, and which rows are not allowed to move. For example, to stop the user from
moving the first row:

         if indexPath.row == 0 {
            return false
         } else {
            return true
         }
         However, we're going to allow movement of all the rows, so we simply return true
         */
        return true
    }
```

moveRowAt

```
func tableView(_ tableView: UITableView, moveRowAt sourceIndexPath: IndexPath, to
destinationIndexPath: IndexPath) {
        // Update your model's reference to the moved object
    }
```

Run the app

You'll see just by including these two function calls, when you click on our "Edit" button, we now have 3 lines on the right of each cell indicating that we can click and drag them to different positions in the list!



Move some cells around.

Try scrolling up so some cells get hidden from the screen, and then let go so the view springs back.

Do you see any problems?

Some of the rows are now repeating content! This is because we're not updating the tableview's data source. Let's do that.

Step 5. Update the data source

Remember that the tableview loads its cells from our "shows" array, so our "shows" array acts as a data source. When we move rows around we need to tell our datasource that the position of those items changed. Otherwise, when the tableview tries to load new cells when the user scrolls it will ask the datasource for the item it should be showing for that row, and the datasource will show the original list we made. Let's update the moveRowTo function to update our data!

```
func tableView(_ tableView: UITableView, moveRowAt sourceIndexPath: IndexPath, to
destinationIndexPath: IndexPath) {
        // Update your model's reference to the moved object

        // grab a reference to the show that the user is trying to move
        if let showToMove = shows?[sourceIndexPath.row] {

            // remove the object from its old location in the model 'shows'
            shows?.remove(at: sourceIndexPath.row)

            // insert the object into its new location in the model 'shows'
            shows?.insert(showToMove, at: destinationIndexPath.row)
        }
    }
```

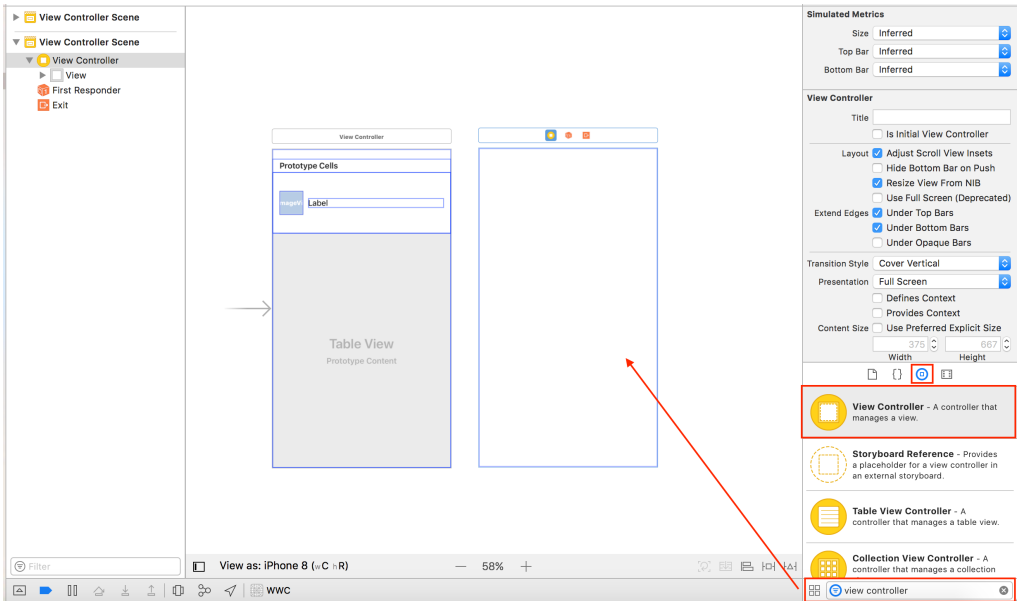
Run the app

Now no matter how much you move things around and then scroll, the tableview keeps an updated reference to its location!

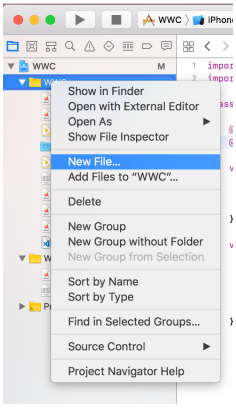
Part 3: Show a popover of the show's details

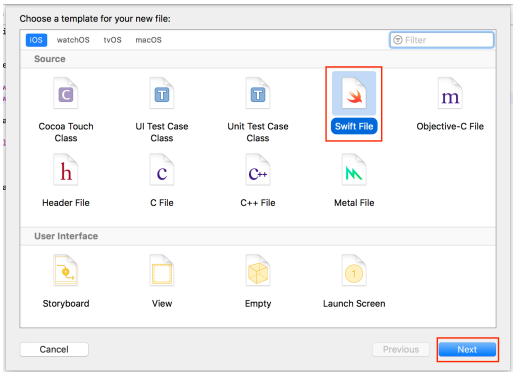
Step 1: Adding the TV Show Details View Controller

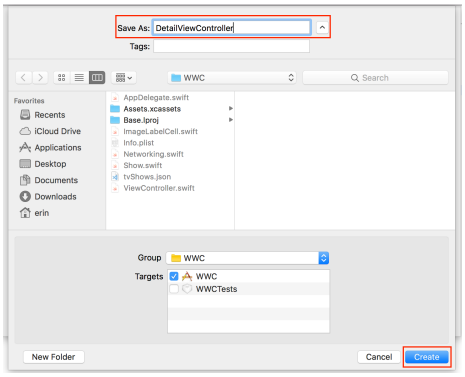
Part 3 of the workshop will be to add a popup view for the TV Show Details. This view will appear when clicking on a TV Show from the List of TV Shows.



First, we need to Add a new View Controller to **Main.storyboard** scene. In the **Object Library** Filter on *View Controller*. Drag the **View Controller** Component into the scene beside the first View Controller.



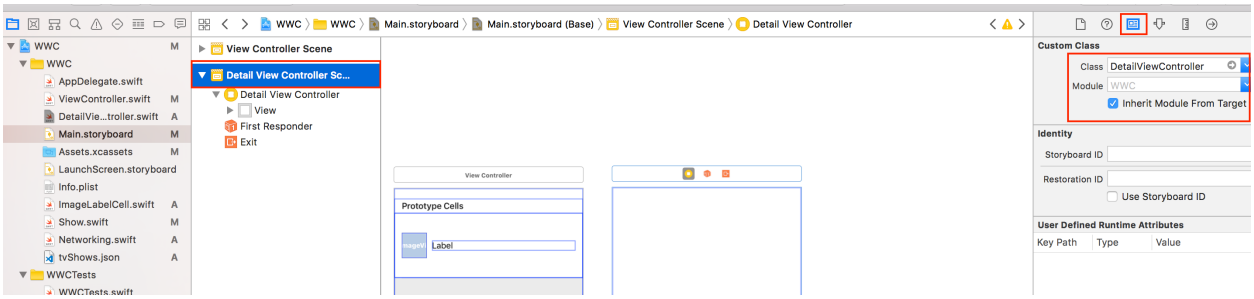




Next, we need to **create** a new **View Controller** for the TV Show Details View that pops over the TV Show List. **Right Click** the WWC folder and select **New File** from the menu. The Select **Swift File** and click **Next**. Finally, provided the new file with a name, we chose *DetailViewController*, and then click **Create**.

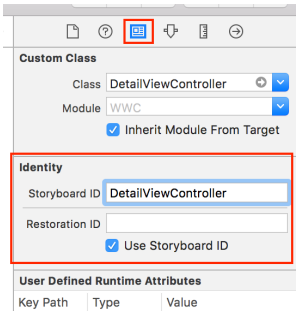
```
import Foundation
import UIKit

class DetailViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

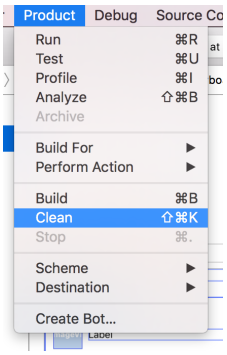
Now we need to add the basic file setup for the **DetailsViewController** class. This includes the **class definition**, adding the `Foundation` and `UIKit` **imports** and **overriding** the `viewDidLoad` function.



On returning to the **Main.storyboard** file we can now **connect** the newly added **View Controller** to the **ViewControllerDetails** Class. While Selecting the newly added View Controller Scene, Click on the **Identity Inspector** and set the Class name to be the name of the view Controller, for us that's *DetailViewController*. Then select the **Inherit Module From Target** checkbox.

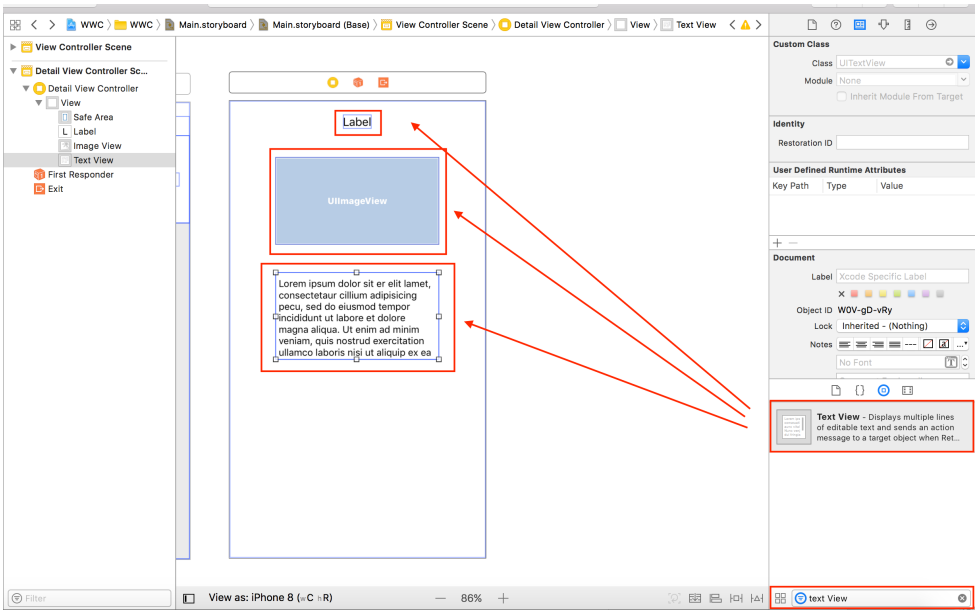
Next provide the scene with a **Storyboard ID** which we will use to identify this view controller in the future. Ensure the **Use Storyboard ID Checkbox** is checked.



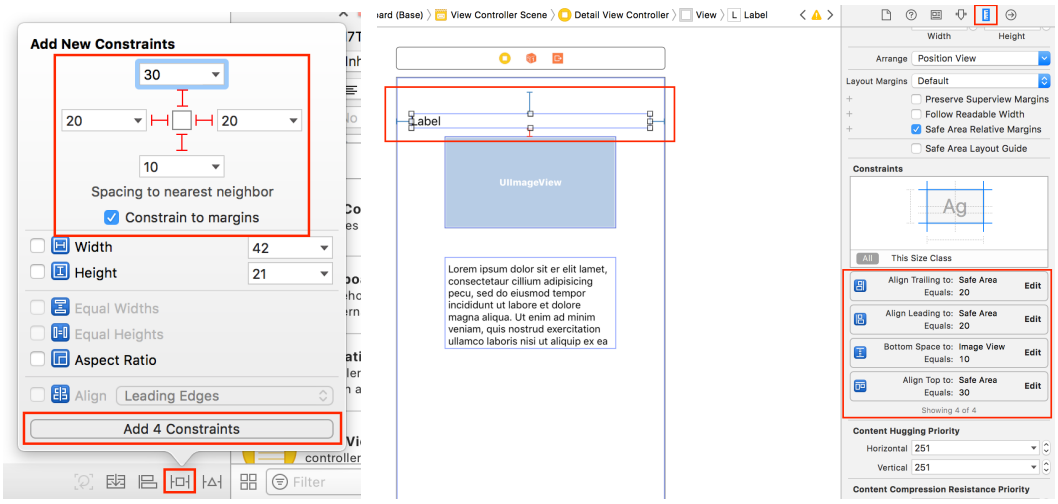


Note: If Xcode cannot find the View Controller Class you may need to clean the project. Do this by Going to the **Product Menu** and then select **Clean**.

Adding Components to the Details View Controller Scene



Next, we will add the **Label**, **Image** and **Text View** components to the Details View Controller Scene. Using the **Object Library**, Filter on the three components (label, image and textview) and **drag** them into the scene one at a time.

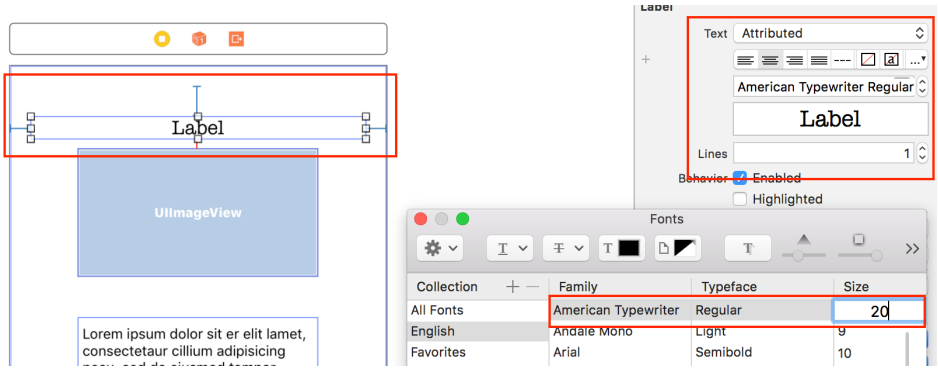


First, we will add constraints to the Label Component. While selecting the Label, click on the **Add New Constraints** button at the bottom of Xcode. Select the **Top constraint** to equal **30px**, **Bottom Constraint** to the Image View to equal **10px**, set the leading and trailing constraints (left and right) to equal **20px**. Ensure the **Constraint to** margins checkbox is selected, then click the **Add 4 Constraints** button.

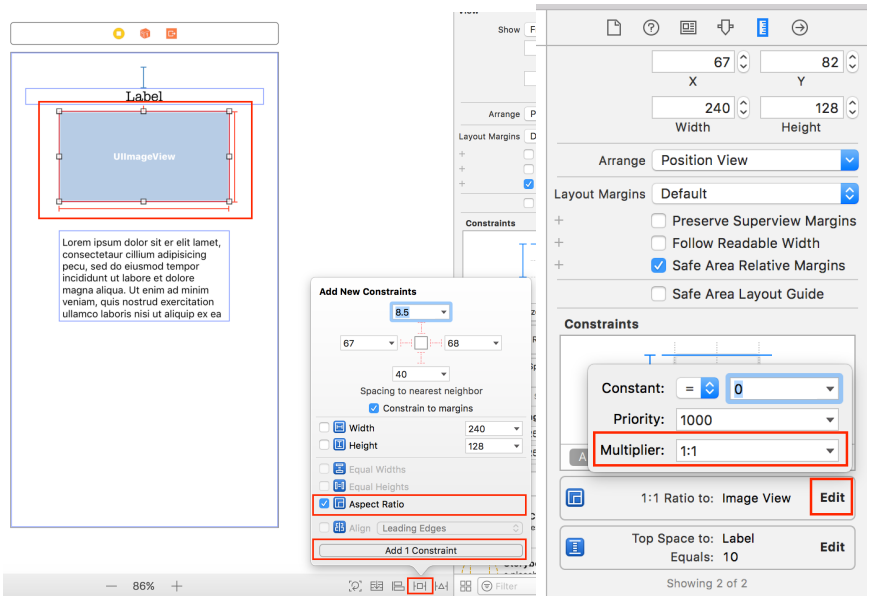
After adding the constraints confirm they were added correctly by navigating to the **Size Inspector** tab within the right-hand panel.

Under the **Constraints** panel you should see 4 boxes with the contents:

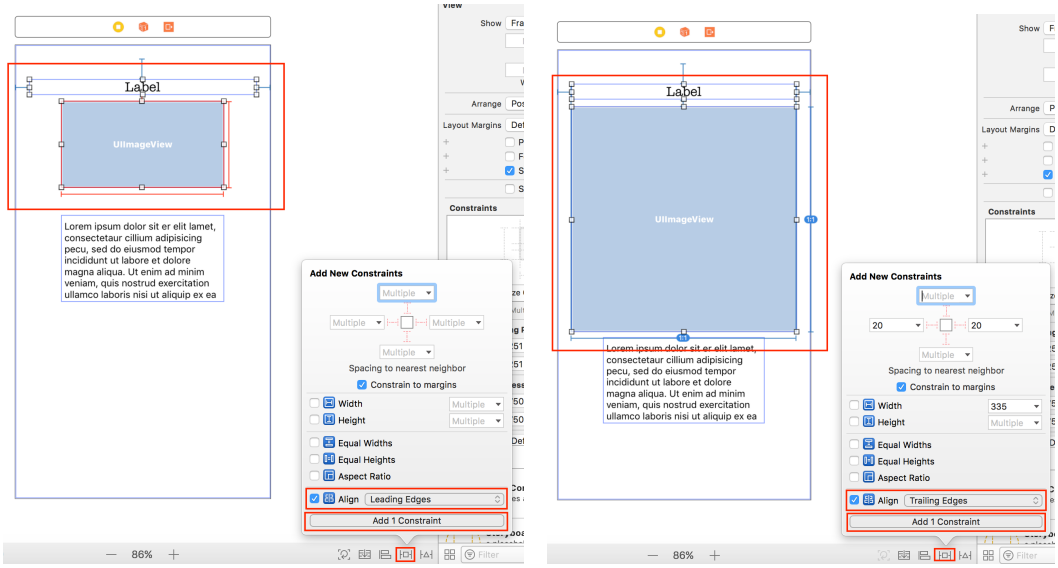
1. Align Trailing to: Safe Area, Equals 20
2. Align Leading to: Safe Area, Equals 20
3. Align Bottom to: Safe Area, Equals 10
4. Align Top to: Safe Area, Equals 30



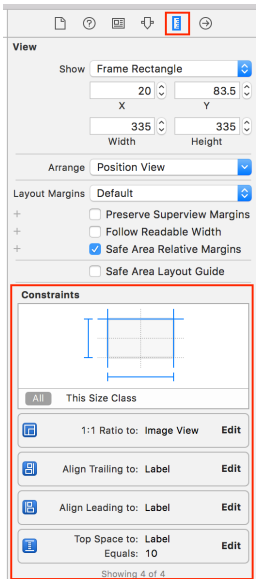
Next, we want to change the font of the label and center the text. While selecting the Label click on the **Attributes Inspector.** Change the **Text drop down** to **Attributed** and select the **Center aligning** button. Next Select the Font Field and choose a font family, Typeface and font Size. We chose *American Typewriter, Regular* and *20pt* font.



Now we will add Constraints to the Image View Component. First, we want to set the **Aspect Ratio** of the image to be **1:1** since our assets are square. While selecting the Image View component, click on the **Add New Constraints** button and check the **Aspect Ratio** checkbox. Next, go to the **Size Inspector** tab and click the **Edit** button on the **Ratio To: Image View** constraint. Change the multiplier to **1:1**.



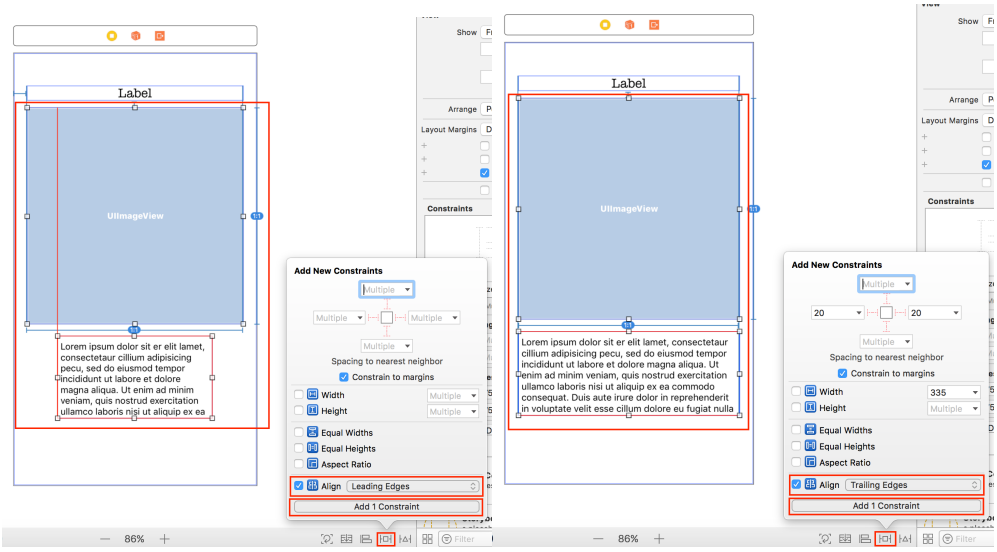
Next, we want to add constraints that align the left and right edges of the Image View to the Label. On Selecting **Both** the Label and the Image View components, select the **Add New Constraint** button and check the **Align** checkbox. Use the drop down to select **Leading Edges** and then click the **Add 1 Constraint Button**. Then, with both the Label and the Image View components selected, click on the **Add New Constraint** button again and check the **Align Checkbox**, this time use the trop down to select **Trailing Edges** and then click the **Add 1 Constraint** button.



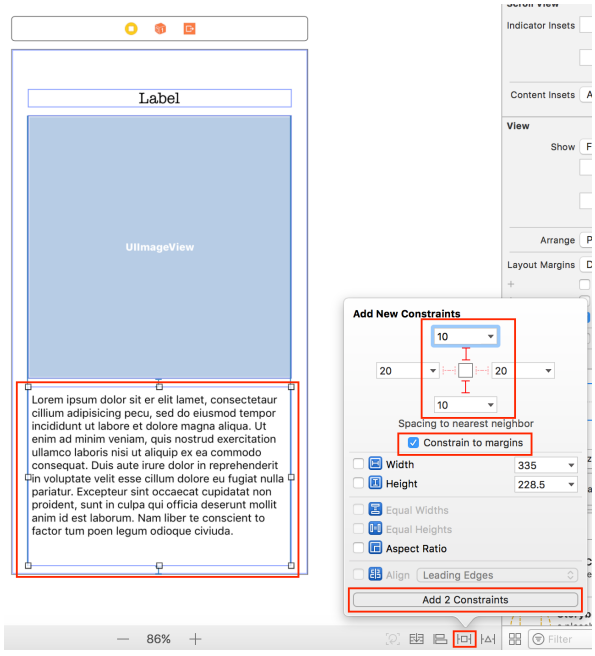
After adding the constraints confirm they were added correctly by navigating to the **Size Inspector** tab within the right-hand panel.

Under the **Constraints** panel you should see 4 boxes with the contents:

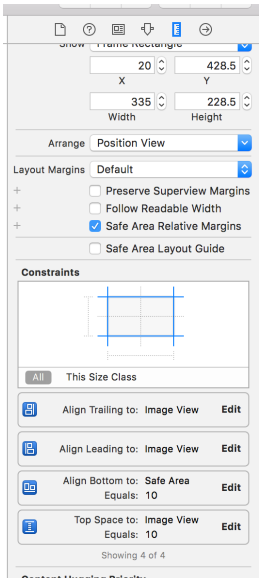
1. 1:1 Ratio to: image view
2. Align Trailing to: Label
3. Align Leading to: Label
4. Top Space to: Label, Equals 30



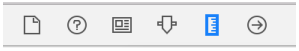
The last constraints added will be applied to the Text View. First, we want to align the leading and trailing edges to the Image's edges. We achieve this by selecting both the **Text View** and the **Image** and the clicking the **Add New Constraints** button. Next, check the **Align checkbox** and use the dropdown to select **Leading Edges**. Then click the **Add 1 Constraint** button. Repeat those steps for the trailing edge except on checking the **Align Checkbox** use the drop down to select **Trailing Edges**.



The last Step is to add the **Top** and **Bottom Constraints**. Select the **Text View** and then click on the **Add New Constraints** button. Input **10px** as the **Top** and **Bottom Constraint** and then ensure the **Constraint to Margins checkbox** has been checked. Finally click the **Add 2 Constraints** button to add the constraints to the **Text View**.



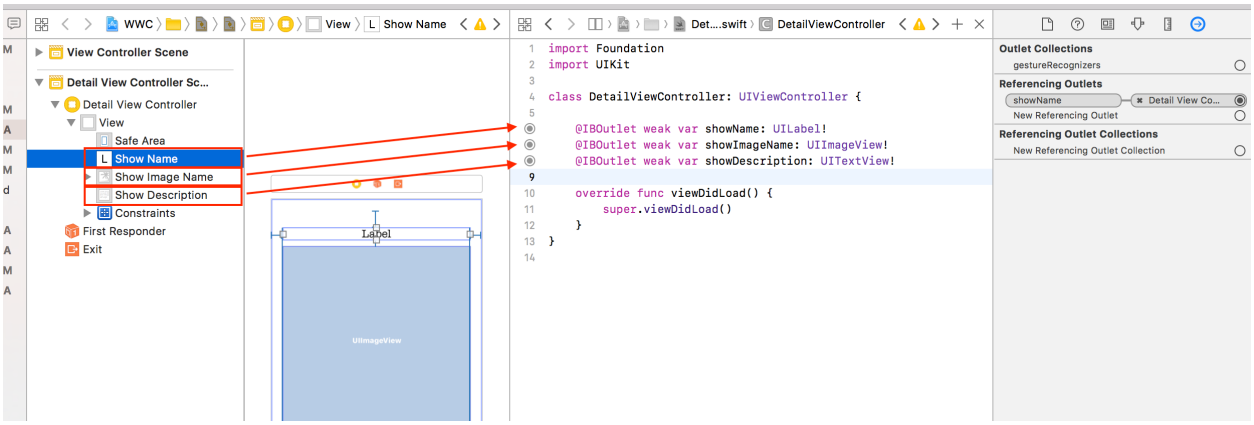
After adding the constraints confirm they were added correctly by navigating to the **Size Inspector** tab within the right-hand panel.

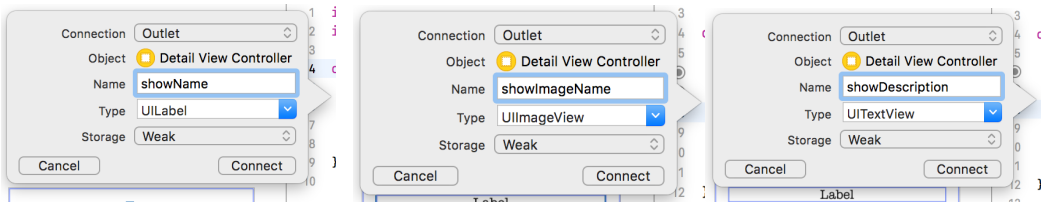


Under the **Constraints** panel you should see 4 boxes with the contents:

1. Align Trailing to: Image View
2. Align Leading to: Image View
3. Align Bottom to: Safe Area, Equals 10
4. Top Space to: Image View, Equals 10

Step 2: Adding Attributes to DetailViewController





In order to connect the storyboard components and the view controller we need to create **IBOutlets.** Find the **Label**, **Image** and **TextView** components in the **Detail View Controller Scene** dropdown. While holding down the **Control key**, click on the components one at a time and drag them to the **DetailViewController** class beneath the class definition. On releasing the click a grey pop-up will appear where you can provide each outlet with a name. We use *showName*, *showImageName* and *showDescription*.

Note: To get the split screen view, start with the Main.storyboard file open and then **option click** the DetailViewController.swift file.

Another Note: When trying to create an IBOutlet if you get an error, try restarting Xcode and cleaning your project.

```
//public optional tv show attributes
var tvShowName: String?
var tvShowImageName: String?
var tvShowDescription: String?

override func viewDidLoad() {
     super.viewDidLoad()
     //setup the label, image and textView with content on initial load
     setup()
}

func setup() {
     //set the showName, showImageName and showDescription
     showName.text = tvShowName
     showImageName.image = UIImage(named: tvShowImageName!)
     showDescription.text = tvShowDescription
}
```

Now that the Outlets are in the Details View Controller we want to provide them with content when the view controller first loads. In order to accomplish this we will create a `setup()` function. On calling this function each outlet will retrieve it's content from the public accessible variables `tvShowName, tvShowImageName, tvShowDescription`.

Step 3: Using the Table View Delegate

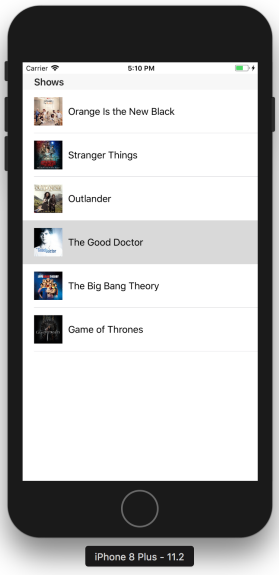
```
override func viewDidLoad() {
     super.viewDidLoad()
     tableView.dataSource = self
     tableView.delegate = self
     //continued ...
```

Next, we need to set up the pop over which is displayed when a TV show is selected. To do this we need to create a **delegate** for the Table View. A delegate is used to assign a job which occurs on clicking a Table View row. In the `viewDidLoad()` assign the **Table View Delegate** to **Self**.

```
extension ViewController: UITableViewDelegate {

    func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
        //check if tv shows exist
        guard let shows = shows else {
            //if tv shows do not exist, deselect the row, don't display pop-up
            self.tableView.deselectRow(at: indexPath, animated: true)
            return
        }

        //find view Controller by its storyboard ID
        let movieDetailVC =
            self.storyboard?.instantiateViewController(withIdentifier:
                "DetailViewController") as? DetailViewController

        //set tv show name, image and description using the selected tv show
        movieDetailVC?.tvShowName = shows[indexPath.row].artistName
        movieDetailVC?.tvShowImageName = shows[indexPath.row].imageName
        movieDetailVC?.tvShowDescription = shows[indexPath.row].longDescription

        //deselect row after selection
        self.tableView.deselectRow(at: indexPath, animated: true)

        //presentation style and action
        movieDetailVC?.modalPresentationStyle = .popover
        self.present(movieDetailVC!, animated: true, completion: nil)
    }
}
```

Then in the `didSelectRowAt()` function of the Table View **pass** the **TV Show data** to the **DetailViewController** through its **Public Attributes** and then present the View Controller as a **popover**. Don't forget to deselect the cell after the cell selection was handled in order to remove the grey highlight.

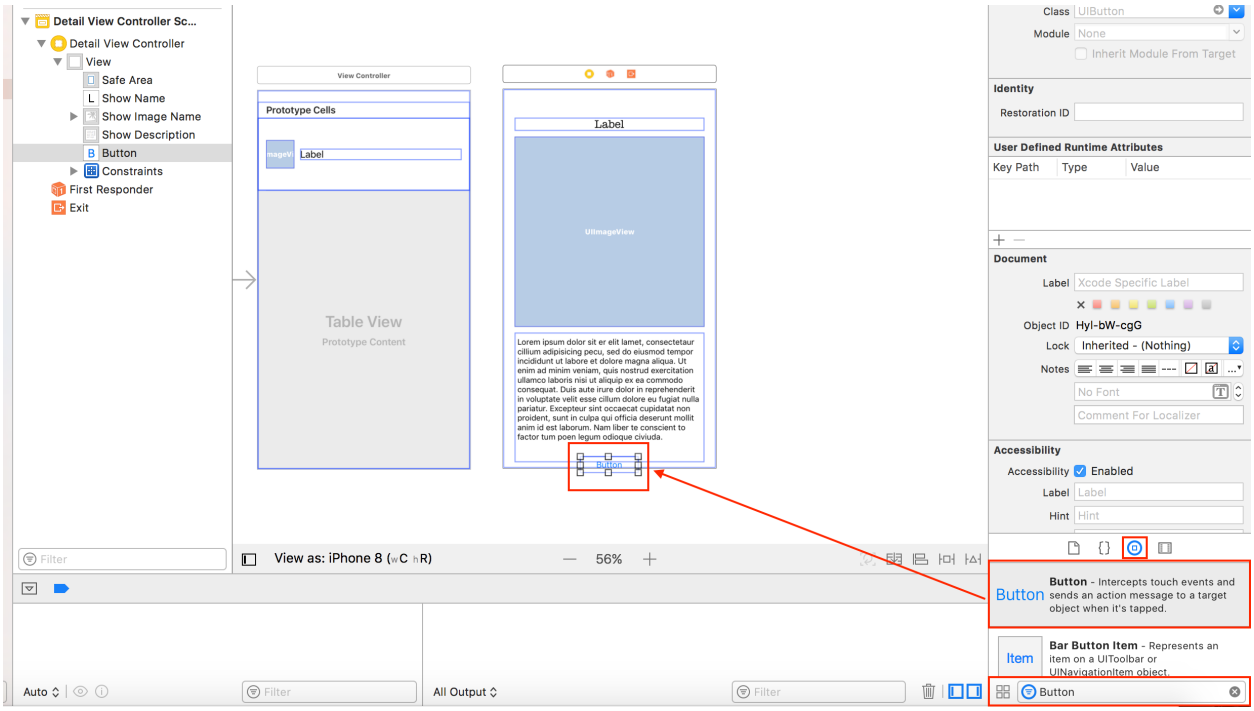
Select **Build** and see the Results!



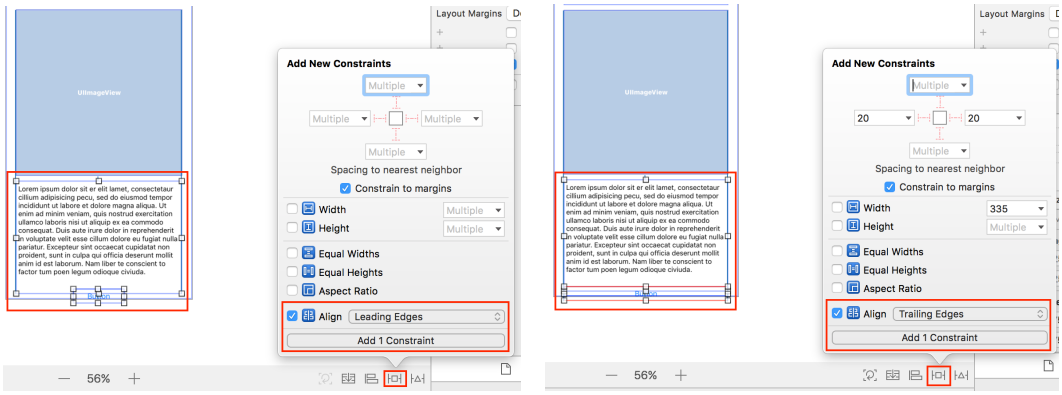
Step 4: Adding the Dismiss Button

You may have noticed that on opening the pop-over you user is stuck on the screen. This is because there is no dismiss button. Next, we will be adding that button and its associated functionality.

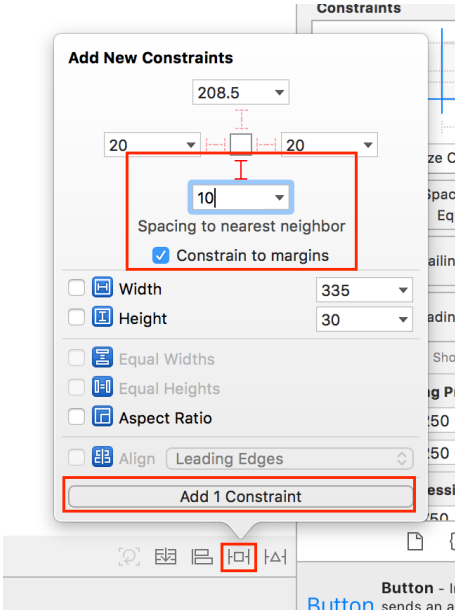
Adding Constraints to the Button



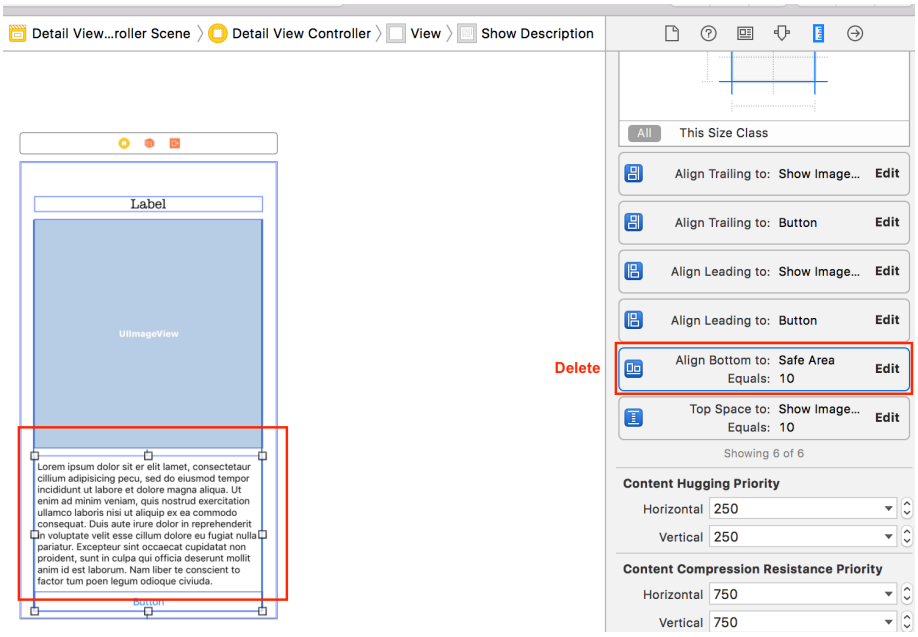
On the **Main Storyboard** filter in the object library for *Button*. **Drag** the button into the **Detail View Controller Scene**.



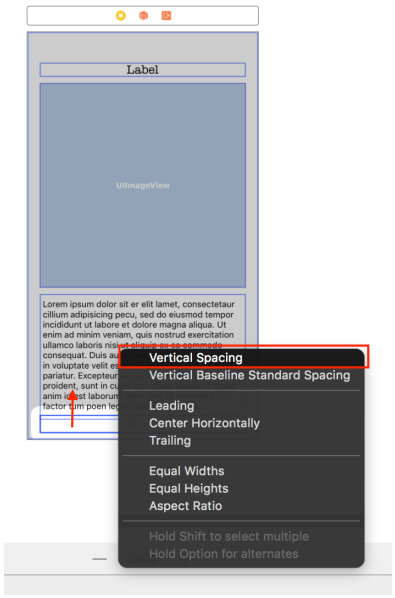
Now that the button is in the scene we will add constraints. First select the **Text View** and the **Button** and then click the **Add New Constraints button**. Check the **Align Checkbox** and then use the drop down and select **Leading Edges.** Then click the **Add 1 Constraint Button**. Follow those exact steps except this time select **Trailing Edges**. These 2 constraints align the left and right side of the button with the Text View's left and right side.



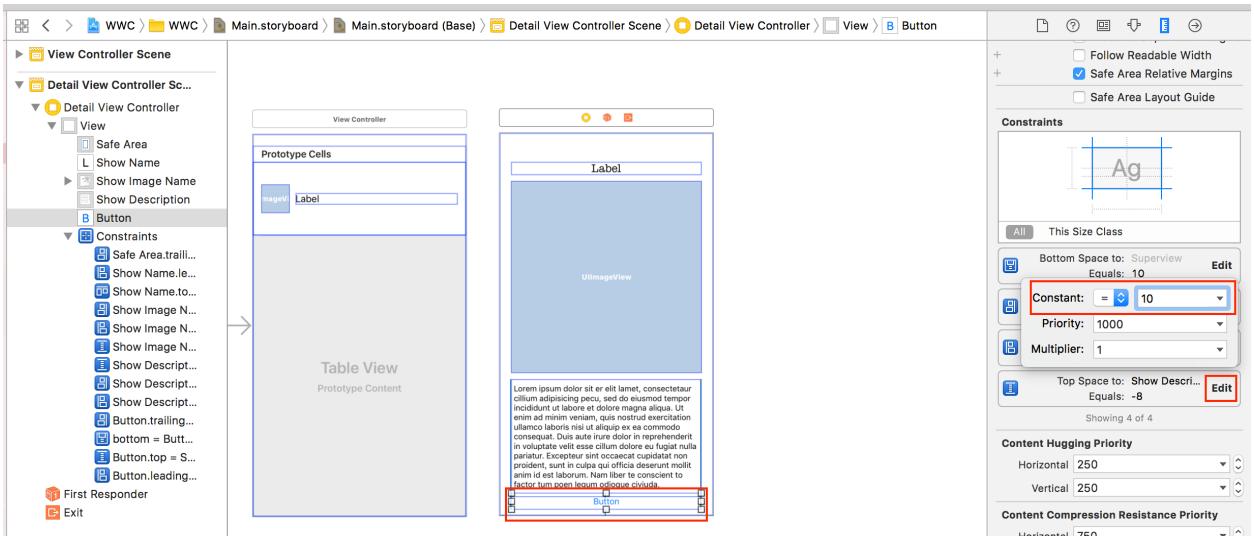
With only the button selected click the **Add New Constraints** button and input **10** as the **bottom constraint**. Ensure the **Constrain to margins Checkbox** is selected and then click **Add 1 Constraint**.



Since we want the **Text View** to be above the button we want to **delete** the **Text View's Bottom Constraint.** In the **Size Inspector** tab, go to the **Constraints** section and **Delete Align Bottom to: Safe Area**.



Now the **Text View** does not have a bottom constraint and the **Button** does not have a Top Constraint. To Constraint them to each other **hold the Control key** while selecting the **Button** and **drag the cursor** to the **Text View**. On releasing the cursor a black pop-up appears and select **Vertical Spacing.**

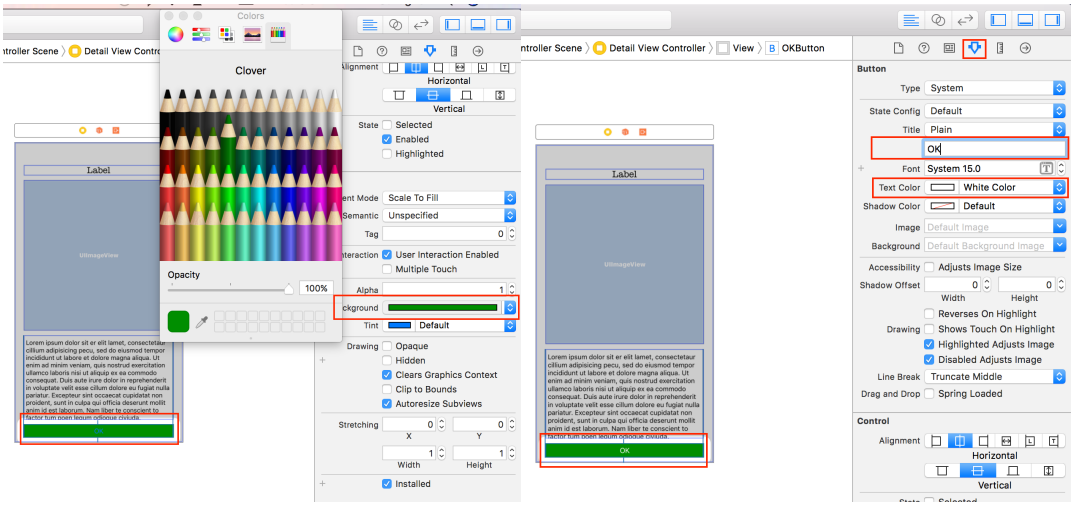


Using this method for constraining will create a constraint that matches the components current positions in the scene. Navigate to the **Size Inspector** tab and you can see the top space of the button is -8. Click the **Edit** button and change the **Constant** to **10**. Now the Button's **top Edge** is **10px** from the Text View's **Bottom Edge**.

After adding the constraints confirm they were added correctly by navigating to the **Size Inspector** tab within the right-hand panel.

Under the **Constraints** panel you should see 4 boxes with the contents:

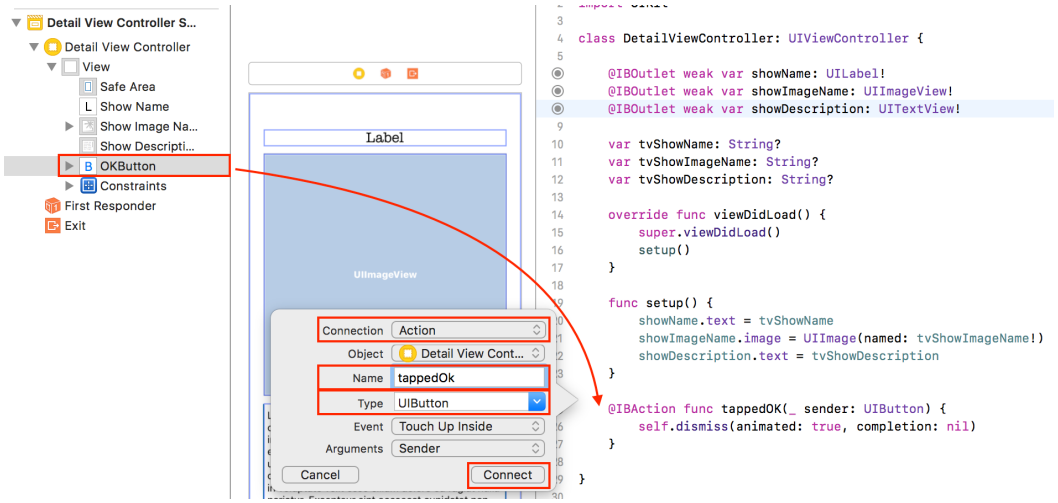
1. Bottom Space to: Superview, Equals 10
2. Align Trailing to: Show Description
3. Align Leading to: Show Description
4. Top Space to: Show Description, Equals 10



The final step is giving the button some colour and text. To add background colour go to the **Attributes Inspector** and select the **Background Dropdown**. Next go to the top of the attributes inspector and change the **Text Colour** to **White** and the **Button** text to **OK**.

Connecting the Button to the Details View Controller

```
@IBAction func tappedOK(_ sender: UIButton) {
      self.dismiss(animated: true, completion: nil)
}
```

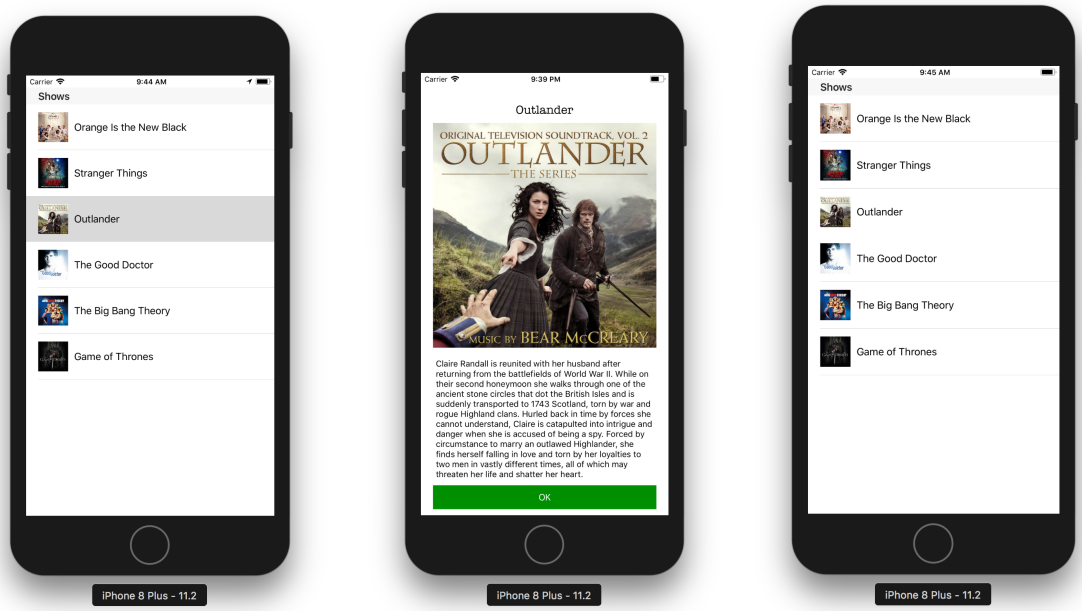


Similar to creating an IBOutlet for the different properties we will create an **IBAction** to control the action associated with clicking the OK Button. **Control Click** and **Drag** the cursor to the **DetailViewController** class. On releasing the cursor a grey pop-up will appear to set the properties for the action. Use the **Connection Drop Down** and select **Action**. Give the action a

name, we chose *tappedOk*. Use the **Type Drop Down** to select **UIButton** and then ensure the **Event** type is **Touch Up Inside**. Once completed Click **Connect**. The **IBAction** function will now appear.

Note: To get the split screen view, start with the Main.storyboard file open and then **option click** the DetailViewController.swift file.

The action associated with this button is dismissing the **DetailViewController** pop-up. Add the above line of code within the **tappedOK Action** to dismiss to pop-up with animation.



Select **Build** and see the Results!
Now on Clicking **OK** the pop-up dismisses and returns you to the TV Show List.