iOS Dev (in Swift)

What I wish I had been told instead of “RTFM”

Alex Popov • @alexpopovjr • hello@alexpopov.ca

About Me

Principal iOS dev @ Shelfie

I'm an SFU student but currently I'm working at Shelfie as the main iOS dev.

- Software systems
- SFU Surrey is beautiful
- 1 year
- Completely rewrote the entire app
- Made it a lot more stable, thanks to Swift's safety
- ship updated every week or two, mega sprints
- get a drink with me after if you want horror stories about working in a start up

Coming Up

- Where to start
- High-level Concepts
- Best Practices

First, everything you do before writing an app

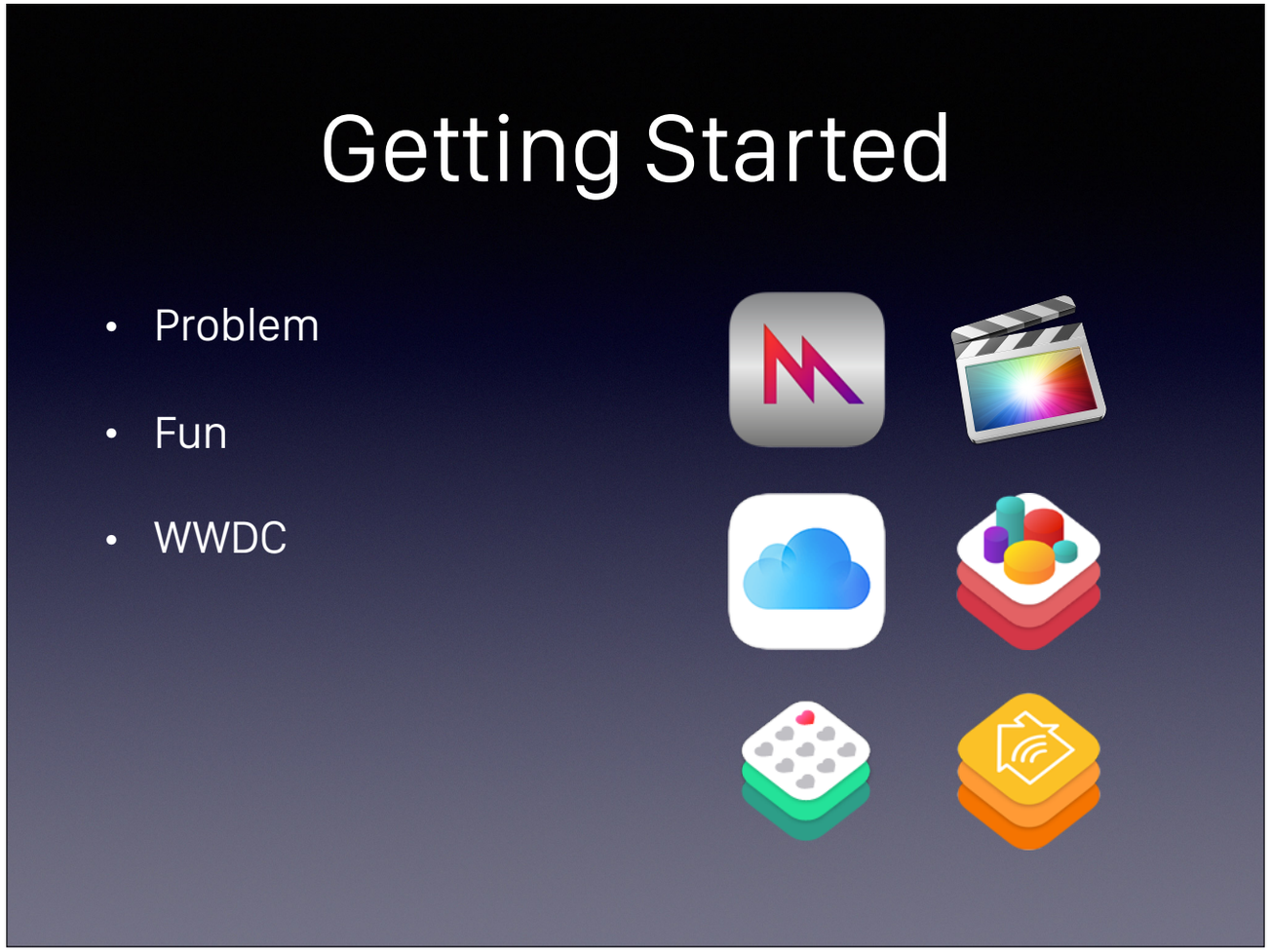
- Installing tools
- Choosing an idea
- Which language to use

Some of the building blocks of iOS and how to use them.

- MVC
- Autolayout

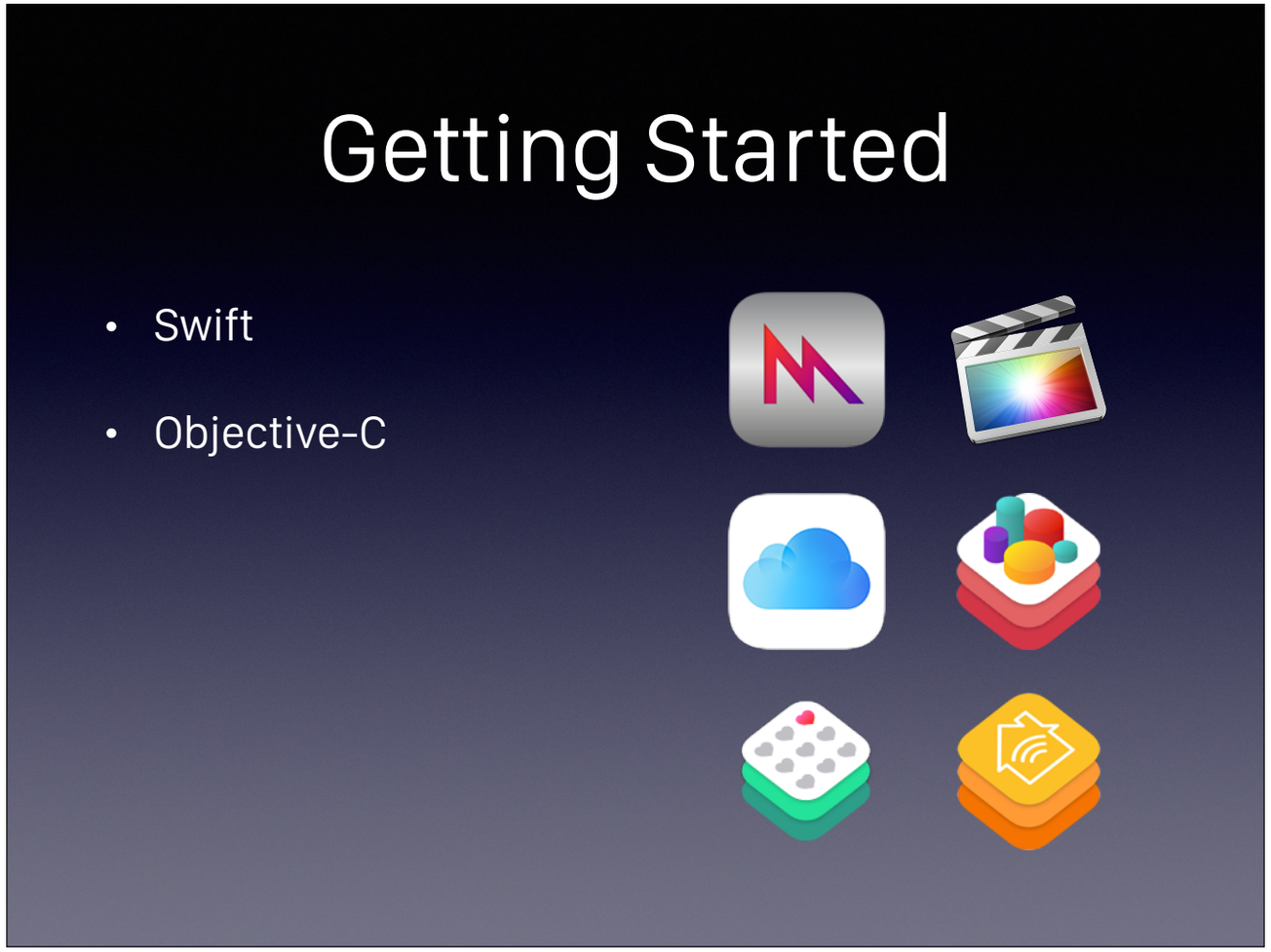
Best Practices and things I wish someone had told me

The order of these steps isn’t critical. There’s two things you’re going to want to do:
Download Xcode and Sign up for a developer account.
Xcode can be downloaded from the Mac AppStore or through Apple’s developer download page. Can also use AppCode if you like JetBrains IDEs.
The Developer Program isn’t necessary, but you’re probably going to want to be able to hold what you built. It’s free. It’s $100 a year to actually publish an app on the AppStore.



Now that everything's installed, what kind of app do you want to make?

- Is there a problem you have?
    - Are current apps not solving it perfectly?
    - Likely other people have the same problem
- Do something fun
    - Games
    - Audio/Visual stuff
    - Music creation
- WWDC is the perfect time to start writing your first app
    - New technologies released
    - Less market competition for new stuff
        - iOS AdBlockers
    - More exploration



You should use Swift.

- Safe
- Modern
- Functional
- New APIs

Getting Started

- Swift
- Objective-C
  - Low-level
  - AVFoundation
  - Games

Swift isn’t good for some tasks

- C-based stuff is a huge hassle
- Working with Audio/Video stuff is C based
- Simple games in Sprite/SceneKit are fine, other libraries are difficult to deal with

Note on Languages

- Writing code is easy
- Software engineering is hard

What language you use doesn’t matter.
You could even skip ObjC/Swift and use Xamarin with C# or ReactNative with JS
You can learn any language over the course of a weekend, become comfortable in a month.
Frameworks, libraries and software design is hard.

- Cocoa is huge
- Cocoa is archaic

Must understand how the overall system works

Building Blocks

- iOS
- AppDelegate
- ViewController
- View
- Model

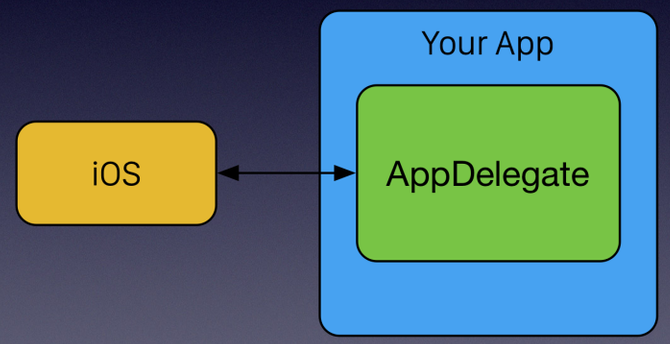
At a simple level, iOS looks like this.

- Operating System, which manages resources and outside interactions
- AppDelegate, which is your link to the OS
- ViewControllers, which contain views and coordinate interactions
- Views, display content and receive touch
- Models, for processing and holding data

Let’s break these down to see what the interactions between them are like

Building Blocks

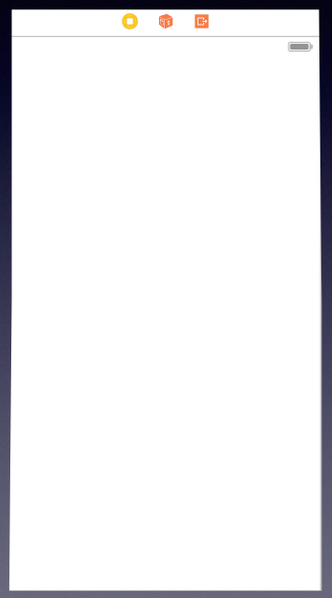
- iOS
- AppDelegate
- ViewController
- View
- Model



User taps on your app icon
Loads your app, and tells the AppDelegate it started.
From there you do whatever you want.
Will get into that later.

Building Blocks

- iOS
- AppDelegate
- ViewController
- View
- Model



Step away from that for a moment. You're mostly going to be dealing with

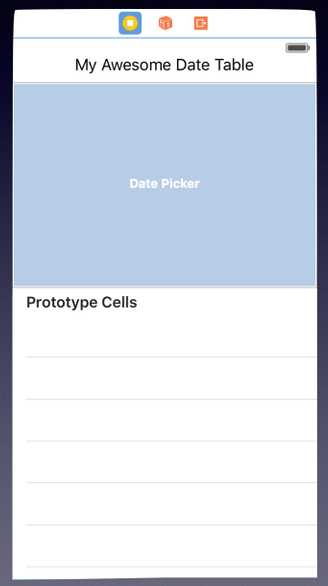
- Controllers
- Views
- Models

Hopefully everyone is familiar with MVC

- Always has a view
- Can present other view controllers

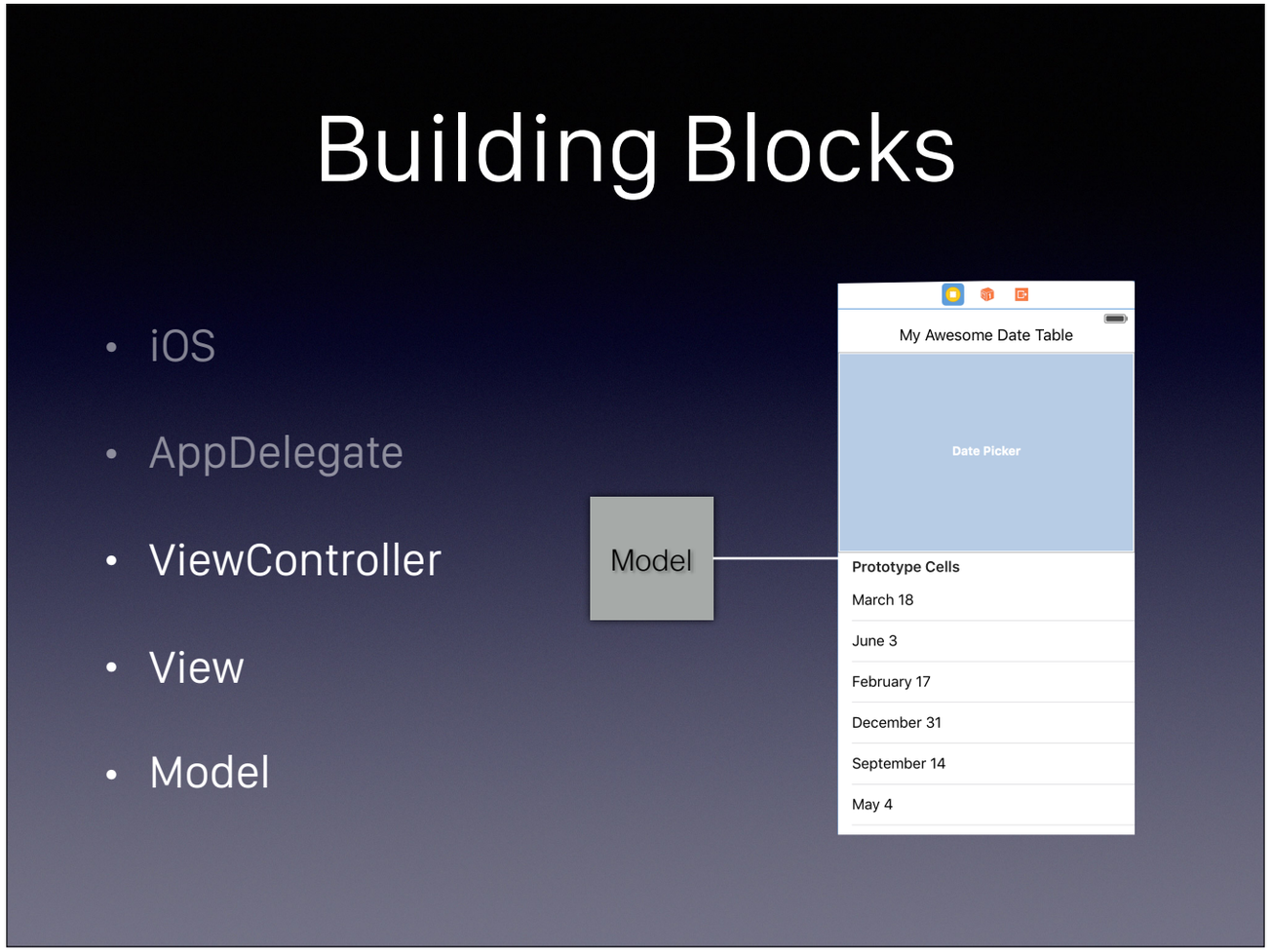
Building Blocks

- iOS
- AppDelegate
- ViewController
- View
- Model



Views are what you see
Views should be dumb
- Know how to present data
- Know what's touchable

Views are composed into a hierarchy
Here we have:
- The ViewController view
    - A title label
    - A date picker
    - a tableview
        - TableViewCells
            - Labels
            - Images



To populate our cells, we need data.
That data goes in a custom model you write.

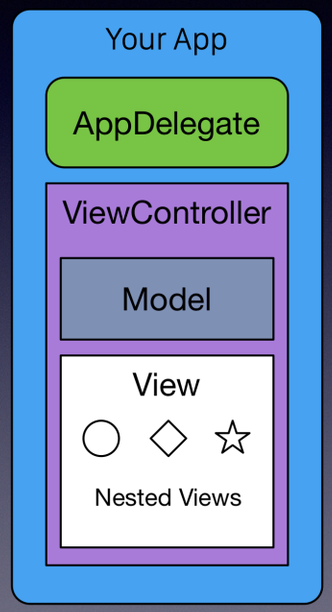
- Model stores data
- View controller asks it for that data
- Gives data to TableView

If a user scrolls through the Date Picker (a view):

- Date Picker tells ViewController it’s changing
- Once it stops, it tells the ViewController it stopped on a value
- ViewController takes that value and tells the Model
- Model updates its internal storage
- ViewController takes new data
- Updates the tableView

Building Blocks

- iOS
- AppDelegate
- ViewController
- View
- Model

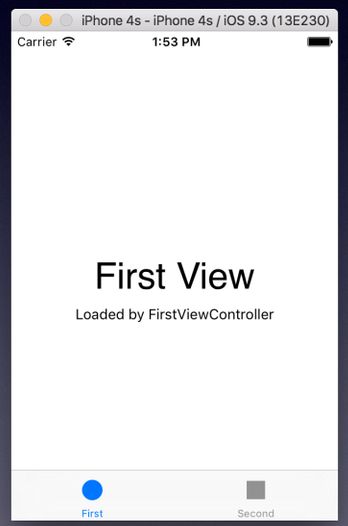


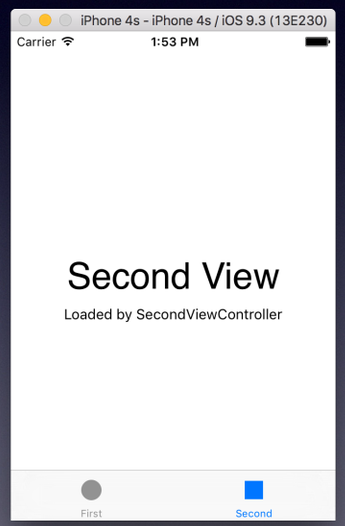
So if we look over everything again, the OS launches our app.
Communicates through the AppDelegate

- Notifications
- Be backgrounded
- Orientation changes

AppDelegate starts up our first ViewController, which coordinates the display and processing of data
Views form a hierarchy.

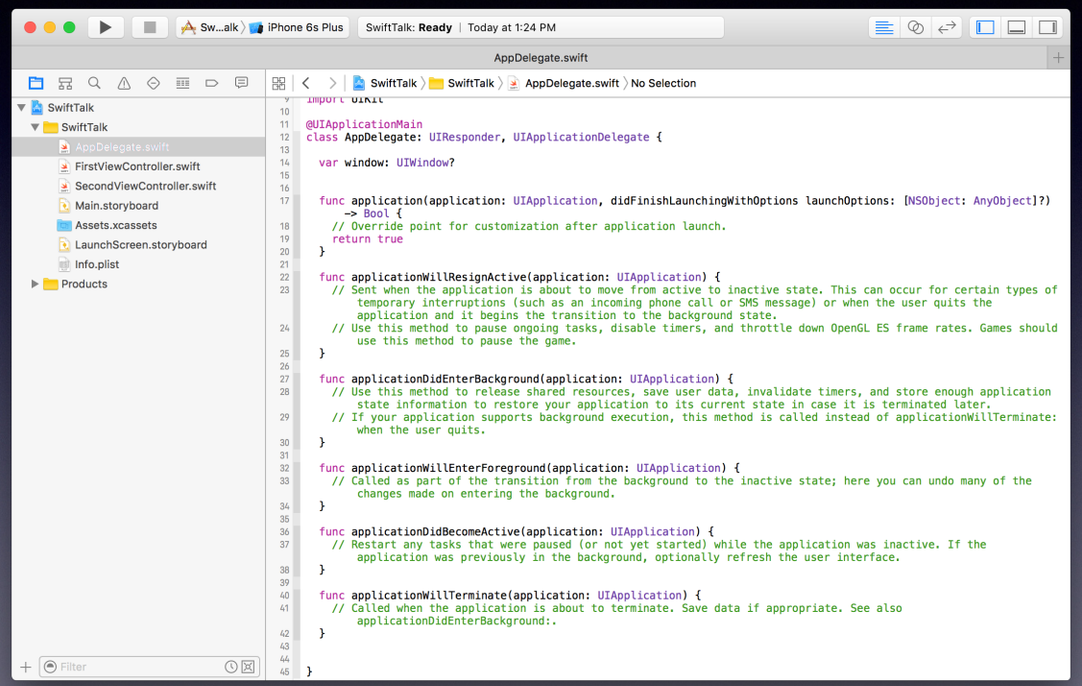
Your First App





So you launch Xcode, create a new Project, and select the Tabbed Application starting point.
It'll look like this. Standard TabBar, each tab shows a different ViewController.

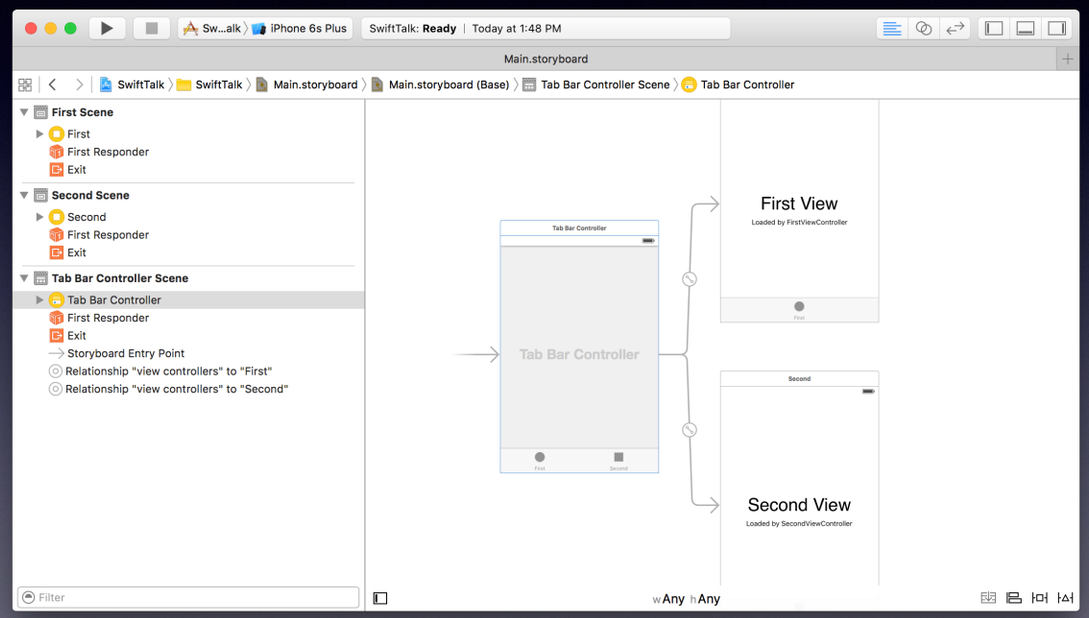
Your First App



Xcode has created:

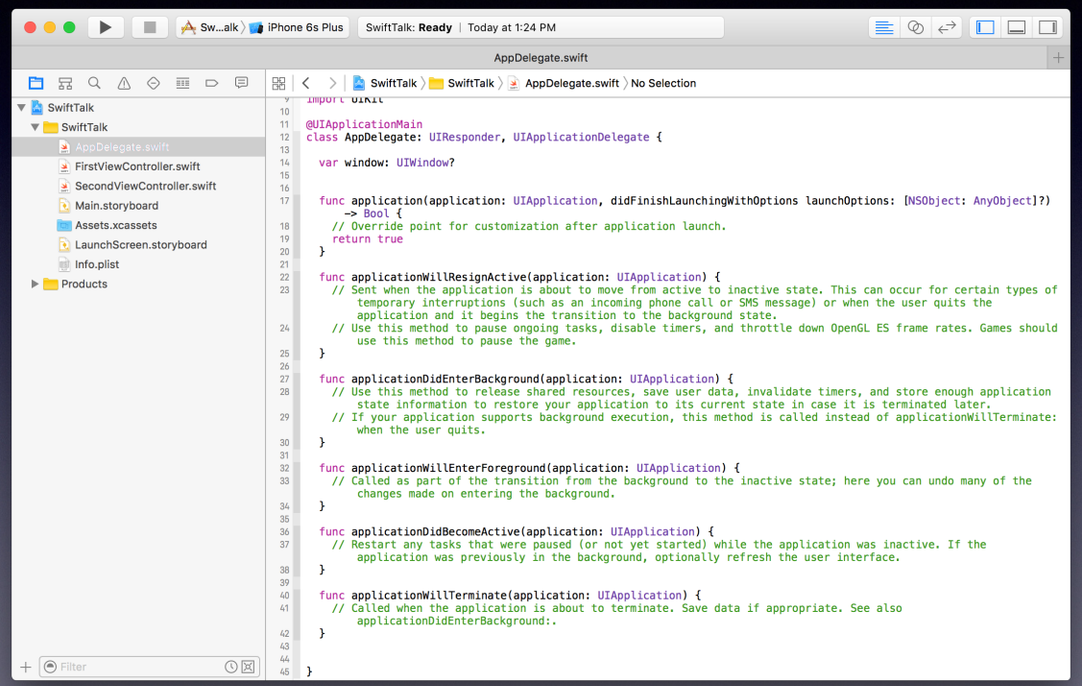
- AppDelegate (necessary)
- Two view controllers (for our TabBarController)
- A Storyboard
- An assets folder
- Info.plist

Your First App



Storyboards are a visual way of doing layout and navigation hierarchies
On the left we have the hierarchy information
On the right we have the Storyboard itself
We can see that there is a TabBarController, with the two view controllers.
ViewControllers can have other nested ViewControllers. This is a powerful way of hierarchies interactions.

AppDelegate



So let’s go back to the AppDelegate for a second.
Lots of methods telling you what the Application is doing.

- enter background
- enter foreground
- will terminate
- did finish launching

AppDelegate

Entry point:

```
application(_, didFinishLaunchingWithOptions:)
```

- Initial View Hierarchy
- Singletons
- Services

Want to keep as little code as possible here
Don’t want it to become monolithic.
Did finish launching is the entry point

In the app Xcode created for us, nothing happens here because it launches the app from a Storyboard. But you could do the same thing in code.

- Won’t go into Launch Options here
- Singletons like LoginManagers, AudioManagers, maybe the Game Engine
- Services: Facebook integration, account verification, analytics, notification registry.

View Controllers

```
import UIKit

class FirstViewController: UIViewController {

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

}
```

So let’s see what an ViewController class looks like.

View Controllers

```
import UIKit

class FirstViewController: UIViewController {

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

}
```

We import UIKit

- you'll have this import on almost all your files.
- Over 170 classes in UIKit
- Notable classes: NSResponder, UIView, UIViewController
- Apple's classes are prefixed NS- or UI-. The NS comes from NeXTStep, UI mostly came about with iOS

View Controllers

```
import UIKit

class FirstViewController: UIViewController {

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

}
```

FirstViewController inherits from UIViewController.
All controllers are UIViewControllers. Different types:

- UINavigationControllers, like in Messages app or Mail app
- UITableViewControllers, if a view only consists of a table
- UITabBarController

UIViewController class provides us with a ton of functionality

- View management
- Storyboard interactions so you don’t have to code many things
- Presenting/managing other controllers
- Layout hooks

View Controllers

```
import UIKit

class FirstViewController: UIViewController {

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

}
```

One of the methods provided: ViewDidLoad
This is where you'll do the majority of your setup

- Custom UI
- loading your data
- setting up model and view

Nibs (from comment) is the Storyboard file format

Views and Layout

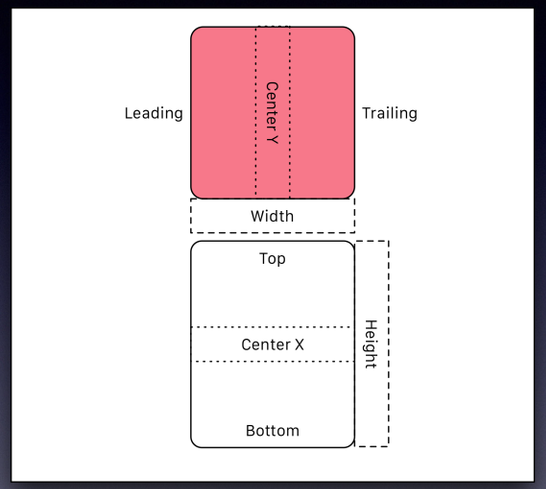
- Dumb
- Imported from UIKit
- Autolayout
- Constraints

Views don’t have much internal logic.
Apple has views that help you get started:

- UITableViewCell
- UIButton
- UIProgressIndicatorView

Layout views using Autolayout. Despite “auto” in the name, it’s not doing any black magic
Not Dreamweaver

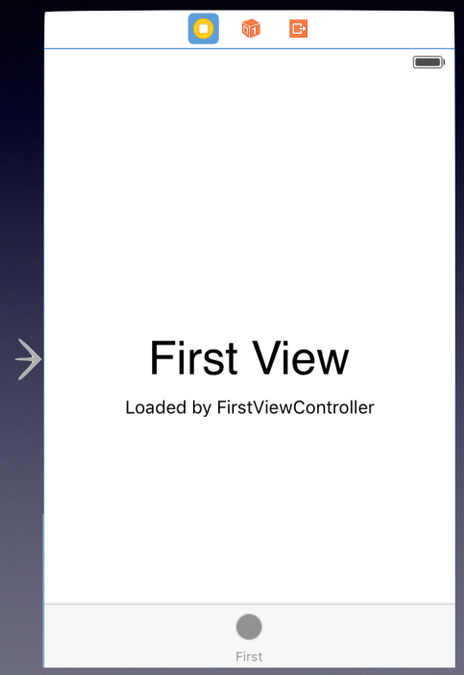
Views and Layout



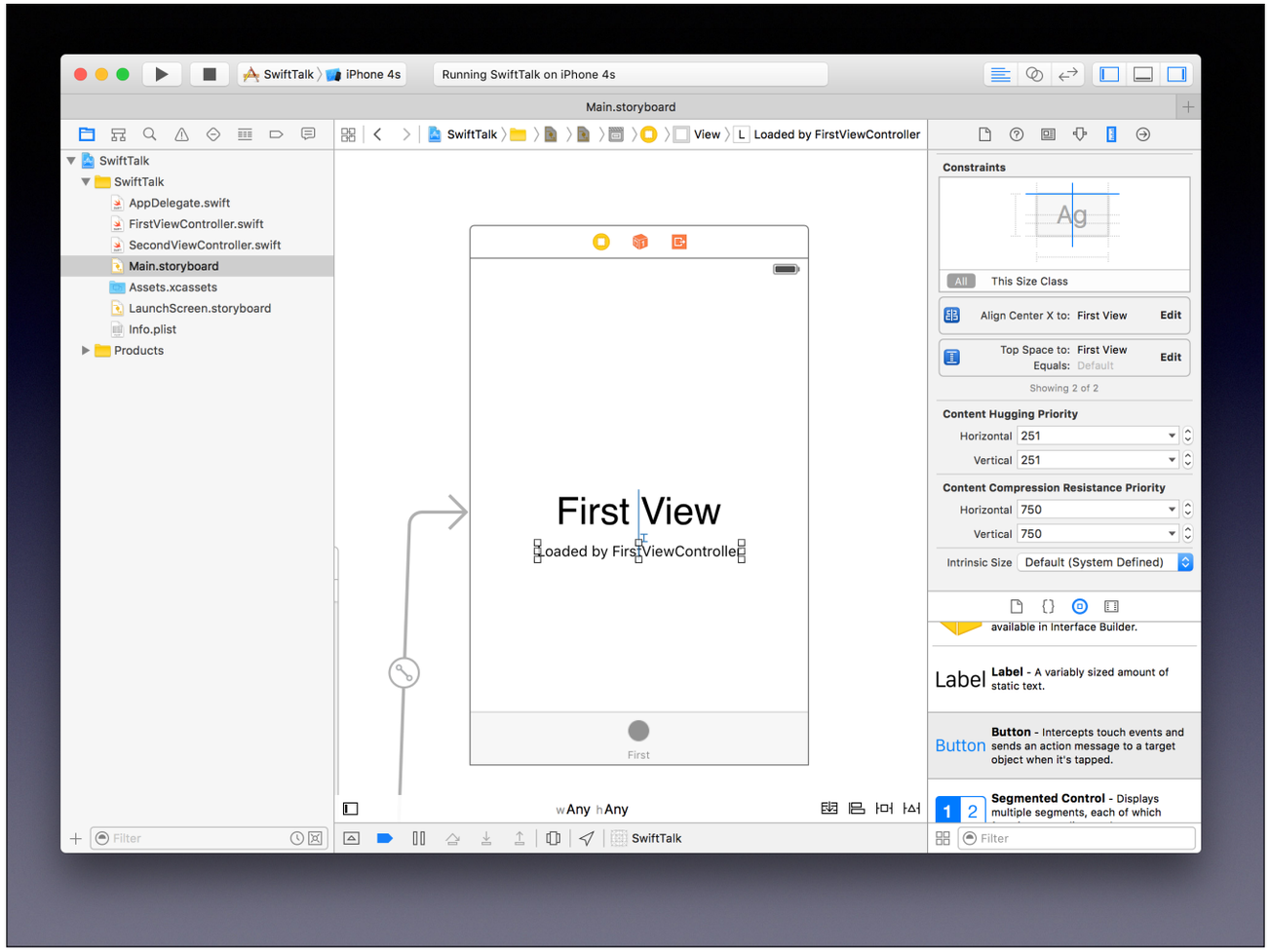

```
Y = m*X + B

WhiteBox.Top = 1.0 * PinkBox.Bottom + 20
```

Constraints use properties of the views to compute layout.
You can orient a view's property relative to another view's property.

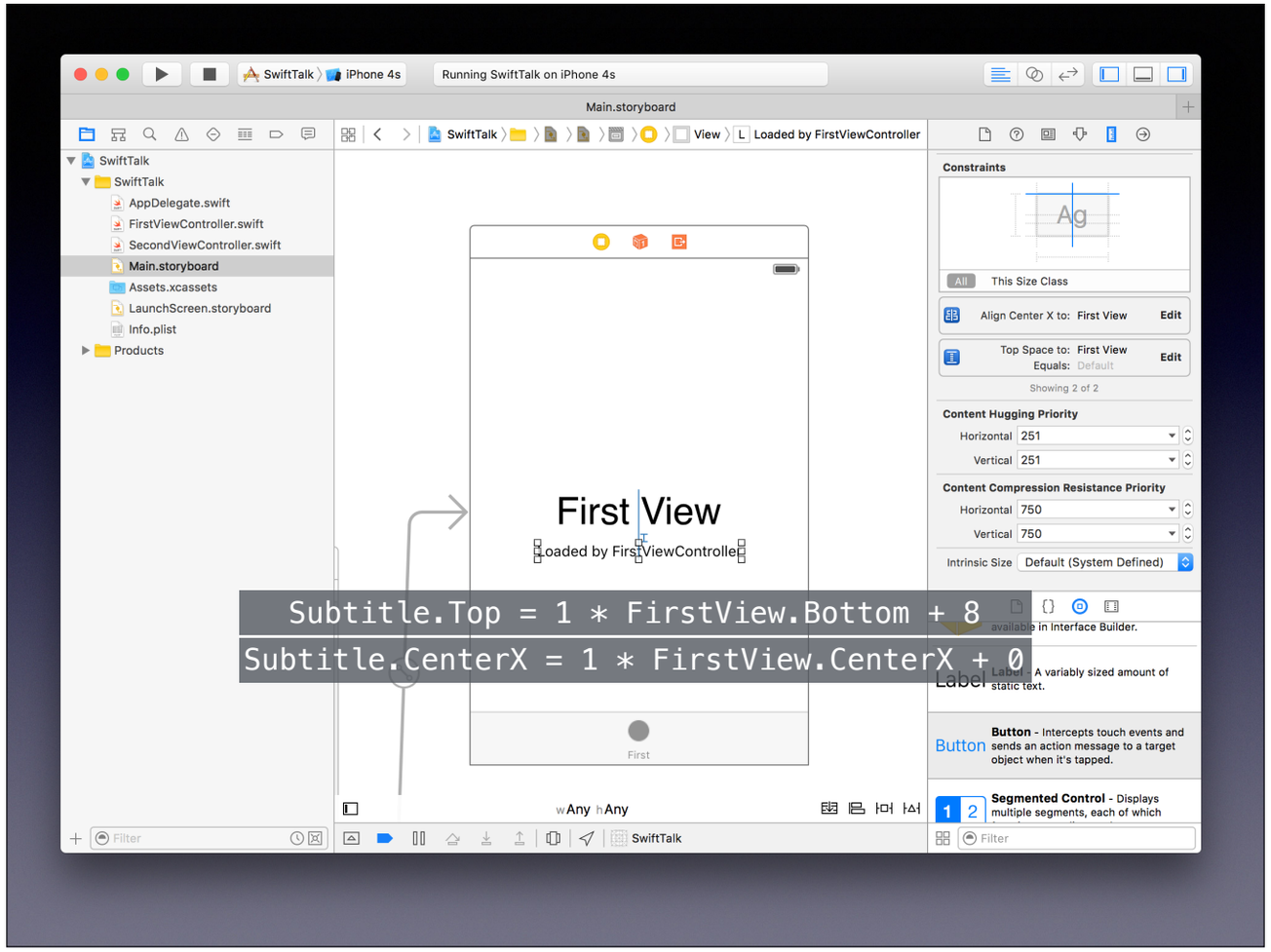
Views and Layout



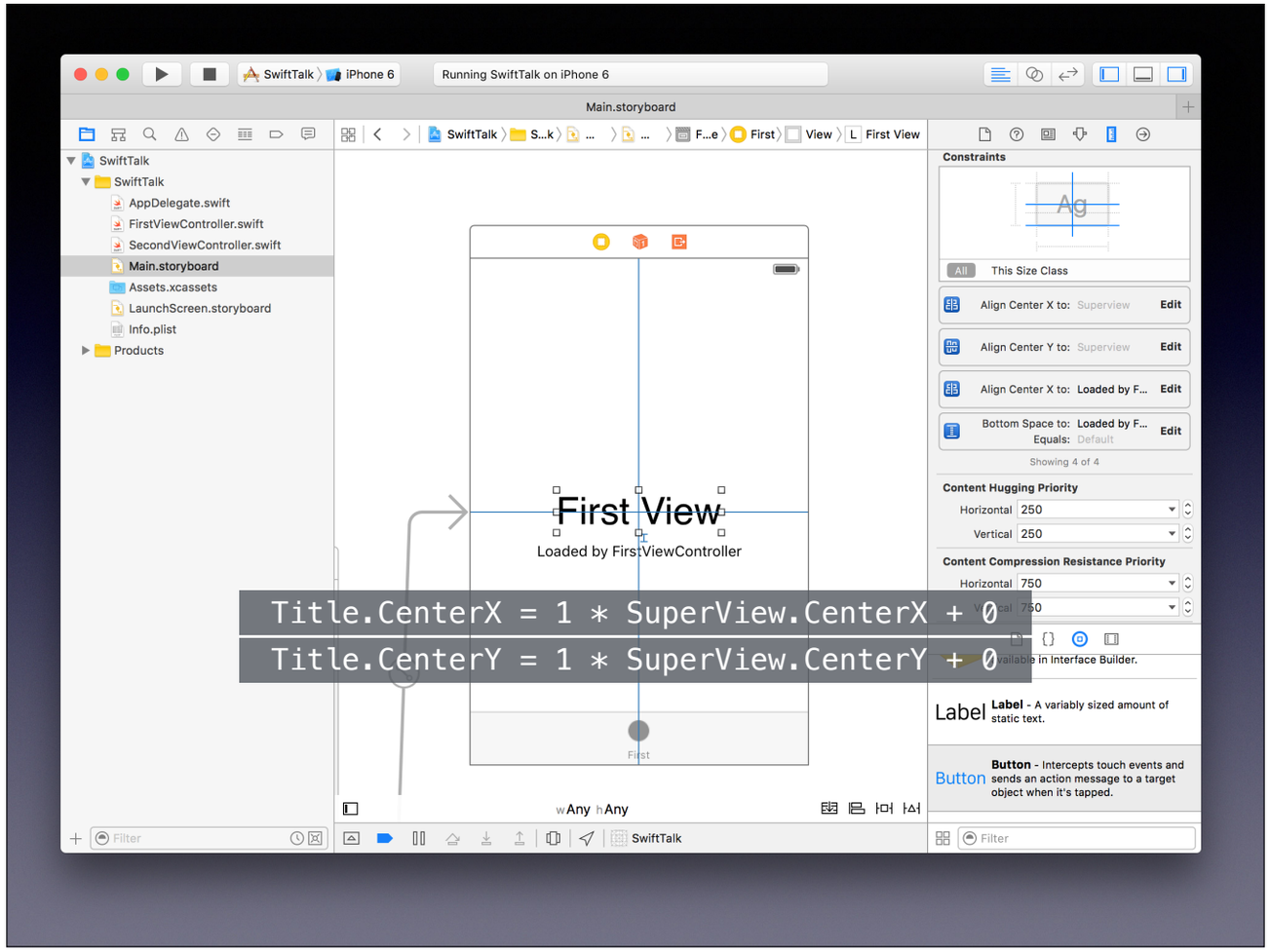
So let’s take a look at our First View.
How does the text know where to go?
How will it look on different sized phones?



If we click on the smaller label, we see constraints.
Right menu: Top constraint and CenterX
Aligned to FirstView label



If we click on the smaller label, we see constraints.
Right menu: Top constraint and CenterX
Aligned to FirstView label



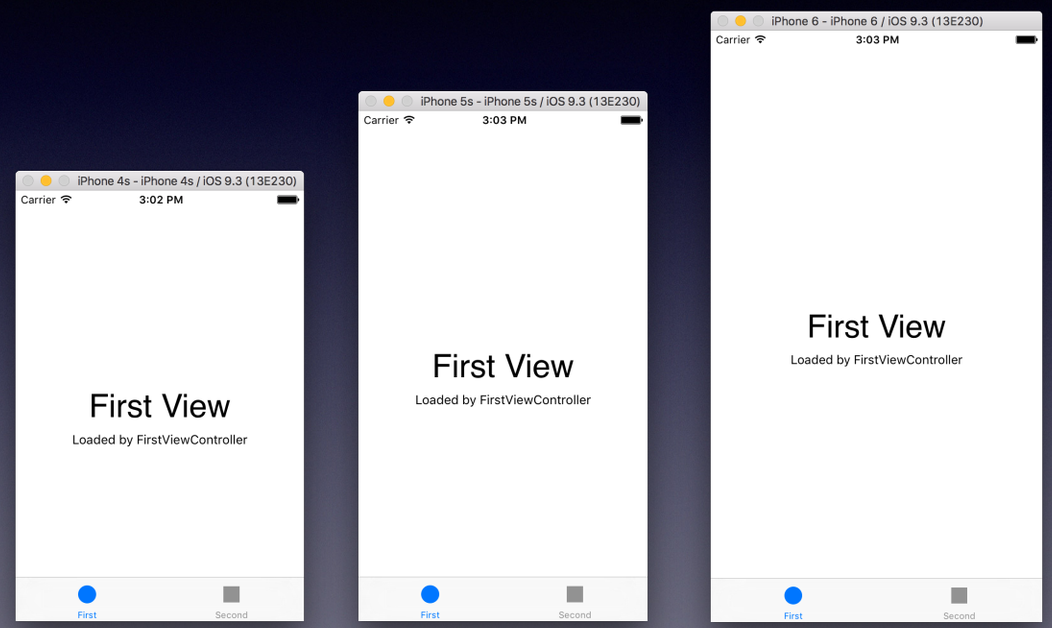
Similarly, the FirstView is aligned CenterX to superview CenterX, CenterY to superView CenterY

Views and Layout

- Must be unambiguous
  - X
  - Y
  - Width
  - Height

Need to unambiguously specify the frame
Notice that we only specified X and Y for the labels
Width and Height are determined by the content
    - Called intrinsic content size
Views without an intrinsic content size need Width and Height as well.

Views and Layout



So we can see it scaling elegantly across devices.
Autolayout is your friend.
Can also be done in code, not only Interface Builder

Building Blocks

- iOS
- AppDelegate
- ViewController
- View
- Model
- AutoLayout

At a simple level, iOS looks like this.

- Operating System, which manages resources and outside interactions
- AppDelegate, which is your link to the OS
- ViewControllers, which contain views and coordinate interactions
- Views, display content and receive touch
- Models, for processing and holding data

Let’s break these down to see what the interactions between them are like

Best Practices & Tips

- Structs
- Protocols
- Strongly-typed
- map-filter-reduce
- Futures

Use Value types for most data modelling
- compose objects from smaller objects
- avoid hierarchies
- embrace immutability

Protocols for polymorphism
- introduce polymorphism into structs via protocols
- use extensions to allow many objects to share code
- compose protocols from other protocols as required

Strongly typed:
- Avoid JSON if you can
- Avoid NSNotifications
- Avoid untyped Key-Value dictionaries if at all possible.

Learn the basic higher-order functions
- avoid for-loops
- make code composable
- reduce side-effects

Futures and Promises
- ties in with functional thinking
- monadic
- makes async declarative

Questions

Alex Popov • @alexpopovjr • hello@alexpopov.ca