

Up to date for iOS 13,  
Xcode 11 & Swift 5.1



# Auto Layout by Tutorials

**FIRST EDITION**

Build Dynamic User Interfaces on iOS

By the **raywenderlich Tutorial Team**

Jayven Nhan & Libranner Santos

# Auto Layout by Tutorials

By Jayven Nhan & Libranner Santos

Copyright ©2020 Razeware LLC.

## Notice of Rights

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

## Notice of Liability

This book and all corresponding materials (such as source code) are provided on an “as is” basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

## Trademarks

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

# Table of Contents: Overview

About the Cover .....	10
What You Need.....	14
Book License.....	15
Book Source Code & Forums .....	16
<b>Section I: Beginning Auto Layout .....</b>	<b>17</b>
Chapter 1: Introducing Auto Layout.....	18
Chapter 2: Construct Auto Layout with the Interface Builder .....	28
Chapter 3: Stack View .....	64
<b>Section II: Intermediate Auto Layout .....</b>	<b>81</b>
Chapter 4: Construct Auto Layout with Code .....	83
Chapter 5: Scroll View .....	112
Chapter 6: Self-Sizing Views .....	125
Chapter 7: Layout Guides.....	154
Chapter 8: Content-Hugging & Compression- Resistance Priorities.....	163
Chapter 9: Animating Auto Layout Constraints .....	172
Chapter 10: Adaptive Layout .....	183
Chapter 11: Dynamic Type.....	221
Chapter 12: Internationalization & Localization.....	247
Chapter 13: Common Auto Layout Issues .....	257
<b>Section III: Advanced Auto Layout.....</b>	<b>281</b>

*Auto Layout*

Chapter 14: Under the Hood of Auto Layout .....	282
Chapter 15: Optimizing Auto Layout Performance... ...	306
Chapter 16: Layout Prototyping with Playgrounds ...	321
Chapter 17: Auto Layout for External Displays.....	342
Chapter 18: Designing Custom Controls .....	356
Conclusion .....	381



# Table of Contents: Extended

About the Cover .....	10
About the Authors .....	12
About the Editors .....	12
What You Need .....	14
Book License .....	15
Book Source Code & Forums .....	16
<b>Section I: Beginning Auto Layout.....</b>	<b>17</b>
Chapter 1: Introducing Auto Layout.....	18
What is Auto Layout? .....	19
Beginning Auto Layout.....	19
UIView and the View Hierarchy .....	20
Thinking in Auto Layout .....	21
Constraints .....	24
Intrinsic content size.....	25
Priorities .....	26
Chapter 2: Construct Auto Layout with the Interface Builder .....	28
Setting up the launch screen layout using a storyboard .....	29
Using Xibs.....	47
Challenges .....	62
Key points.....	63
Chapter 3: Stack View .....	64
Implementing a vertical stack view .....	65
Adding constraints to a stack view.....	65
Embedding views into a horizontal stack view .....	67
Alignment and distribution .....	69

Nesting stack views.....	76
When not to use stack views.....	78
Challenges .....	78
Key points.....	80
<b>Section II: Intermediate Auto Layout .....</b>	<b>81</b>
<b>Chapter 4: Construct Auto Layout with Code .....</b>	<b>83</b>
Launching a view controller from the storyboard in code .....	84
Launching a view controller without initializing storyboard .....	86
Building out a view controller's user interface in code .....	88
Auto Layout with visual format language .....	100
Benefits and drawbacks from choosing the code approach.....	109
Challenges.....	110
Key points .....	111
<b>Chapter 5: Scroll View.....</b>	<b>112</b>
Working with scroll view and Auto Layout.....	113
Adding the Options Menu to the Profile Screen .....	114
Challenge.....	123
Key points .....	123
<b>Chapter 6: Self-Sizing Views .....</b>	<b>125</b>
Accomplishing self-sizing views.....	126
Table views .....	128
Collection View.....	138
Key points .....	153
<b>Chapter 7: Layout Guides .....</b>	<b>154</b>
Available Layout Guides.....	155
Creating layout guides .....	157
Creating custom layout guides .....	158
Key points .....	162

<b>Chapter 8: Content-Hugging &amp; Compression-Resistance Priorities.....</b>	<b>163</b>
Intrinsic Size and Priorities.....	164
Practical use case.....	165
Challenge.....	171
Key points .....	171
<b>Chapter 9: Animating Auto Layout Constraints.....</b>	<b>172</b>
Animate Auto Layout using Core Animations.....	173
Key points .....	182
<b>Chapter 10: Adaptive Layout.....</b>	<b>183</b>
One storyboard to run them all .....	184
Size classes .....	187
Adaptive presentation .....	201
UIKit and adaptive interfaces .....	205
Adaptive images.....	215
Challenge.....	220
Key points .....	220
<b>Chapter 11: Dynamic Type .....</b>	<b>221</b>
Why Dynamic Type?.....	221
Setting the preferred font size .....	223
Making labels support Dynamic Type .....	225
Making custom fonts support Dynamic Type .....	232
Managing layouts based on the text size .....	234
Supporting Dynamic Type with UITableView.....	236
Supporting Dynamic Type with UICollectionView .....	237
Challenges.....	245
Key points .....	246
<b>Chapter 12: Internationalization &amp; Localization .....</b>	<b>247</b>
What's internationalization and localization?.....	247
Auto Layout and internationalization .....	248

Previewing languages in Interface Builder .....	252
Key points .....	256
<b>Chapter 13: Common Auto Layout Issues.....</b>	<b>257</b>
Understanding and solving Auto Layout issues .....	257
Common performance issues .....	276
Key points .....	280
<b>Section III: Advanced Auto Layout.....</b>	<b>281</b>
<b>Chapter 14: Under the Hood of Auto Layout.....</b>	<b>282</b>
Alignment rectangles versus frame rectangles.....	283
Fitting Auto Layout in the big picture .....	287
Solving constraints .....	289
Solving equality constraints .....	289
Solving linear programming problems .....	292
Cassowary.....	298
Key points .....	305
<b>Chapter 15: Optimizing Auto Layout Performance .....</b>	<b>306</b>
Betting safe on performance with Interface Builder .....	307
Factoring in the render loop.....	307
Why update constraints.....	309
Constraints churn .....	310
Unsatisfiable constraints.....	317
Constraints dependencies.....	317
Advanced Auto Layout features and cost .....	318
Apple optimizing UIKit .....	319
Key points .....	319
<b>Chapter 16: Layout Prototyping with Playgrounds .....</b>	<b>321</b>
Getting started with playgrounds .....	322
Creating a custom button .....	330
Moving code to the Sources folder .....	334

Working with more complex custom views .....	334
Key points .....	341
<b>Chapter 17: Auto Layout for External Displays .....</b>	<b>342</b>
Getting started .....	344
Building a layout for an external display.....	344
Configuring an external display window .....	346
Handling an existing external display connection.....	348
Connecting a new external display.....	349
Disconnecting an external display.....	350
Accommodating external display resolutions.....	351
Key points .....	355
<b>Chapter 18: Designing Custom Controls.....</b>	<b>356</b>
Getting started .....	357
Applying adaptive layout to a custom view.....	358
Integrating a custom view from NIB to storyboard .....	363
Visualizing XIB/NIB files in storyboards .....	364
Preparing custom views for Interface Builder .....	366
Making a custom UIButton.....	368
Making a custom UIControl .....	370
Implementing accessibility features.....	376
Key points .....	380
<b>Conclusion.....</b>	<b>381</b>

# About the Cover

While the *Marrus orthocanna* looks as though it belongs to some other alien world, the jellyfish is found in the deep waters of the Arctic and other cold oceans. Though delicate, it is large — 1–2 meters — and translucent, blushing hints of color across the body.

Beyond its science-fiction appearance, the most amazing thing about this jellyfish is that it is *many* jellyfish. The *Marrus orthocanna* is a colonial animal. Like bees or some flocks of birds, it is comprised of multiple animals who live together for some evolutionary benefit. In the case of this jellyfish, the separate zooids are linked together by one long stem, working in unison to live.

Like this creature, Auto Layout boasts impressive interdependence and coordination, while the separate components are related by constraints and attached to a parent view. Both Auto Layout and the *Marrus orthocanna* show us the elegance and efficiency that can be achieved when independent pieces work together to thrive.



# Dedications

"To my family. Thank you for your unconditional love and support."

*— Jayven Nhan*

"To my family, because they are the reason for all the good I do."

*— Libranner Santos*

## About the Authors



**Jayven Nhan** is an author of this book. He is an Apple scholar who contributes his best work to passion, fitness training, and nutrition. Passion makes problem-solving an enjoyment. Fitness training keeps him from staring at his Macbook, unrequited love. Nutrition gives him the epic energy he needs to power his day. He enjoys meeting passionate developers from all around the world. Outside of coding, you may find him listening to audiobooks and podcasts, reading, or watching YouTube videos.



**Libranner Santos** is an author of this book. He is a Software Engineer, with a passion for teaching at all levels and always eager to learn new things. Over the last few years, he has worked with many companies all over the world as a mobile developer. He's the co-founder of saestech.com. When he's not programming, you can probably find him at the basketball court, dancing salsa or reading a book.

## About the Editors



**Jerry Beers** is the tech editor of this book. He is co-founder of Five Pack Creative, a mobile development company specializing in iOS development. He is passionate about creating well-crafted code and teaching others. You can find his company's site at [fivepackcreative.com](http://fivepackcreative.com).



**Tammy Coron** is the editor of this book. She is an independent creative professional. As an author, editor, illustrator, podcaster, and indie game developer, Tammy spends a lot of her time creating content and teaching others. For more information about Tammy, visit her website at [tammycoron.com](http://tammycoron.com).



**Richard Critz** is the final pass editor of this book. He is the iOS Team Lead at raywenderlich.com and has been doing software professionally for nearly 40 years, working on products as diverse as CNC machinery, network infrastructure, and operating systems. He discovered the joys of working with iOS beginning with iOS 6. Yes, he dates back to punch cards and paper tape. He's a dinosaur; just ask his kids. On Twitter, while being mainly read-only, he can be found [@rcritz](#). The rest of his professional life can be found at [www.rwcfoto.com](#).

## About the Artist



**Vicki Wenderlich** is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.



# What You Need

To follow along with this book, you'll need the following:

- A Mac running **macOS Mojave** (10.14) or later. Earlier versions might work, but they're untested.
- **Xcode 11 or later.** Xcode is the main development tool for iOS. You'll need Xcode 11 or later for the tasks in this book. You can download the latest version of Xcode from Apple's developer site here: [apple.co/2asi58y](https://apple.co/2asi58y)

If you want to try things out on a physical iOS device, you'll need a developer account with Apple, which you can obtain for free. However, all the sample projects in this book will work just fine in the iOS Simulator bundled with Xcode, so a paid developer account is completely optional.



# Book License

By purchasing *Auto Layout by Tutorials*, you have the following license:

- You are allowed to use and/or modify the source code in *Auto Layout by Tutorials* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Auto Layout by Tutorials* in as many apps as you want, but must include this attribution line somewhere inside your app: “Artwork/images/designs: from *Auto Layout by Tutorials*, available at [www.raywenderlich.com](http://www.raywenderlich.com)”.
- The source code included in *Auto Layout by Tutorials* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Auto Layout by Tutorials* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an “as is” basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action or contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.



# Book Source Code & Forums

## If you bought the digital edition

This book comes with the source code for the starter and completed projects for each chapter. These resources are shipped with the digital edition you downloaded here:

- <https://store.raywenderlich.com/products/auto-layout-by-tutorials>

## If you bought the print version

You can get the source code for the print edition of the book here:

- <https://store.raywenderlich.com/products/auto-layout-by-tutorials-source-code>

And if you purchased the print version of this book, you're eligible to upgrade to the digital editions at a significant discount! Simply email support@razeware.com with your receipt for the physical copy and we'll get you set up with the discounted digital edition version of the book.

## Forums

We've also set up an official forum for the book here:

- <https://forums.raywenderlich.com>.

This is a great place to ask questions about the book or to submit any errors you may find.



# Section I: Beginning Auto Layout

Start your journey into Auto Layout with the foundations you need. Specifically, you'll cover:

- **Chapter 1: Introduction:** Get an overview of Auto Layout and how to start “thinking in Auto Layout”. Learn what constraints are and the formula that defines them. Meet other key concepts such as intrinsic content size and priorities.
- **Chapter 2: Construct Auto Layout with the Interface Builder:** Learn to use Xcode’s visual tool, Interface Builder, to construct a user interface complete with all Auto Layout constraints. See how to preview your interface on multiple device sizes and orientations. Then, learn to create .xib files to describe a smaller — and potentially reusable — subset of a UI.
- **Chapter 3: Stack View:** Learn about `UIStackView`, an intelligent view container that makes many layout decisions for you. Build a complex layout using nested stack views. See how using stack views can eliminate the need for you to manually construct Auto Layout constraints.

# 1 Chapter 1: Introducing Auto Layout

By Jayven Nhan

Gone are the days of building user interfaces based on a single device with a known screen size. Today, with so many different devices available from Apple, you need to design and build your apps for multiple screen sizes. You also need to think about designing with dynamic content in mind.

In other words, it shouldn't matter whether you run your apps on an iPhone 11 Pro Max or an iPad 12.9" — either way, it should look good. The same holds true for dynamic content. Whether it's a single line of text or twenty lines of text, your UI needs to adapt. With an adaptive layout, you can create a visually stunning app on all devices.

This is where Auto Layout comes in.



# What is Auto Layout?

Auto Layout is a constraint-based layout system designed for building dynamically sized user interfaces. It lets you create user interface layouts that dynamically adapt for all screen sizes without manually setting the frame of every view.

For a complete Auto Layout system, every view must have a defined position and size. In other words, Auto Layout has to know how to position and size every user interface element in different circumstances. You'll learn more about this later.

Auto Layout helps developers adapt user interface layouts for external and internal changes. Some examples include:

- Different screen sizes.
- iPadOS multitasking features like Slide Over, Split View and Picture in Picture.
- Device rotation.
- Larger text support.
- Internationalization support.
- Dynamic content support.

# Beginning Auto Layout

Before you jump into learning about Auto Layout, it's important to understand that Auto Layout isn't automatic; you need to do some work to get it right. Some of the topics this book covers include:

- Thinking in auto layout (relativity).
- Dynamically sizing a `UITableViewCell` / `UICollectionViewCell`.
- Using Auto Layout with `UIStackView` and how to optimize for minimal constraints.
- Constraint content hugging and compression resistance priorities.
- Constructing Auto Layout programmatically.

To get started, you'll learn about the foundational layer of any `UIKit` user interface running on iOS/iPadOS.

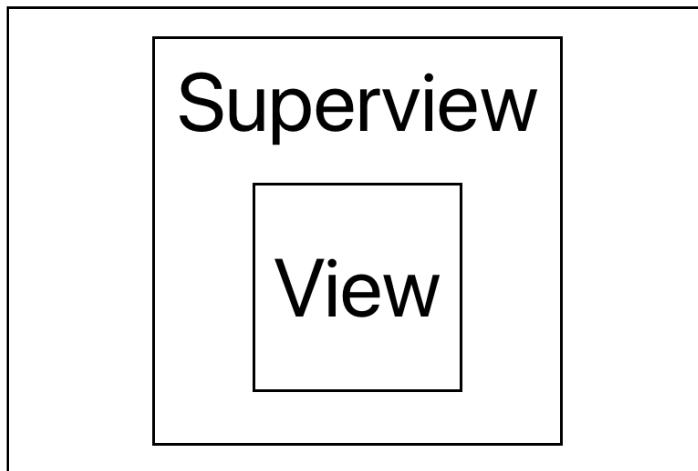
# UIView and the View Hierarchy

`UIView` is the foundation of all user interfaces built using `UIKit`. Every user interface is built on top of `UIView`. This includes `UIButton`, `UILabel` and `UITextField`, all of which are built on top of `UIView`.

Your app's views are arranged in a hierarchy. Each view is contained within a parent and each view may contain one or more child views. Another name for the parent view is **Superview**.

## Superview

The superview is the view that contains a child view.



Every view has a superview, and every screen has a view. The standard view of a screen is the view controller's view. It's the base layer containing all of the user interfaces on a single screen. You can consider this view as a blank canvas because you can put any drawings on top of it.

Each view controller's view can have a different size based on many variables. For example, an iPhone 11 Pro Max full-screen view controller's view is 414×896 points in portrait while an iPad Pro 12.9" full-screen view controller's view is 1024×1366 points in portrait.

# Thinking in Auto Layout

When you think about Auto Layout, it's helpful to think of the word **relative**. When using Auto Layout, remember that you're working on a blank canvas with predefined width and height known as the superview.

Given an app design, you'll want to know how to implement the design using Auto Layout. Before you start adding views onto the superview, it's helpful to ask yourself Auto Layout questions to begin thinking in Auto Layout.

Starting with a view's position, you could ask questions like:

1. In what way is the view's position relative to other views?
2. As the superview grows or shrinks, will the view's position change? If so, what's the pattern?
3. Is the view's content static or dynamic? If dynamic, how do you want the view to position as content grows or shrinks?
4. Do you want to place any limitations on how far or close a view is relative to another?

Questions relating to a view's size might be:

1. Do you want your view's size to be relative to other views? If so, what's the pattern?
2. As the superview grows or shrinks, will the view's size change? If so, what's the pattern?
3. Is the view content static or dynamic? If dynamic, how do you want the view's size to change as content grows or shrinks?
4. Do you want to place any limitations on how much a view can grow or shrink?

These questions are crucial to consider when implementing Auto Layout.

Next, you'll go through a thought process of thinking in Auto Layout, given an app design.

## Example

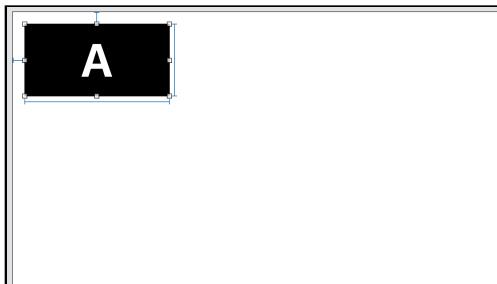
The following example shows a simple app design with two labels, Label A and Label B.



Think about how you might implement Auto Layout for this type of app design. As you do, consider that there's more than one way to implement the same user interface using Auto Layout.

The first view you'll think about implementing in Auto Layout is Label A.

You add Label A onto the superview. The superview looks like this:



To apply the concept of relativity, start asking some of the Auto Layout questions. Continue down your list of questions until the view's position and size requirements are met.

### Auto Layout questions: Position (Label A)

Starting with the questions for the view's position:

Q: In what way is Label A's position relative to other views?

A: Label A's leading edge is 16 points away from the superview's leading edge. Label A's top edge is 16 points away from the superview's top edge.

The view's two relationships you give to Auto Layout satisfy the position requirement.

### Auto Layout questions: Size (Label A)

With the view's position questions answered, you're ready to move onto asking questions about its size.

Q: Do you want your view's size to be relative to other views? If so, what's the pattern?

A: No.

Q: As the superview grows or shrinks, will the view's size change? If so, what's the pattern?

A: No.

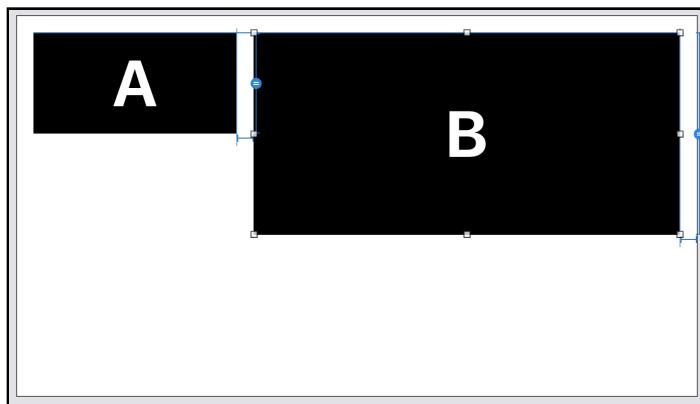
Q: Is the view content static or dynamic? If dynamic, how do you want the view's size to change as content grows or shrinks?

A: Label A will be statically sized. Label A's width is 200 points, and its height is 100 points.

The view's constant values you give to Auto Layout satisfy the size requirement. You're ready to move on to Label B.

### Auto Layout questions: Position (Label B)

You add Label B onto the superview. The superview looks like this:



Again, apply the concept of relativity. Start asking some of the Auto Layout questions, and continue down the list of questions until the view's position and size requirements are met.

Like before, start with the questions about its position:

Q: In what way is Label B's position relative to other views?

A: Label B's leading edge is 24 points away from Label A's trailing edge. Label B's top edge aligns with Label A's top edge. Label B's height is two times Label A's height.

The view's two relationships you give to Auto Layout satisfy the position requirement. You can safely move on to the view's size requirement.

### Auto Layout questions: Size (Label B)

Ask yourself the same size questions as before, but this time consider how your answers apply to Label B.

Q: Do you want your view's size to be relative to other views? If so, what's the pattern?

A: Yes, for width and height. Label B's trailing edge is 16 points away from the superview's trailing edge.

The view's two relationships you give to Auto Layout satisfy the size requirement. Congratulations, you now have your first complete Auto Layout system.

Label A will position at the same relative distance from the superview's upper left corner. It will also have a static size. Label B positions at a relative distance to Label A. Label B will have a dynamic width based on the screen width and a dynamic height based on Label A's height.

Next, you'll learn how to communicate your view's position and size to Auto Layout using constraints.

## Constraints

You use Auto Layout constraints to layout your user interfaces dynamically in Auto Layout. These constraints communicate a view's position and size to Auto Layout using math. Constraints are a series of linear equations. A constraint is made up of the following linear equation:

**Item A's Attribute = Multiplier \* Item B's Attribute + Constant**



When you add a constraint, you add a linear equation. It's a way to express relationships or constants. You can think of adding constraints as a way to express relationships or constants mathematically to the layout engine.

When expressing constant values to Auto Layout, the linear equation sets Item B's attribute and the multiplier to zero. For expressing constant values, the linear equation simply looks like this:

### **Item A's Attribute = Constant**

## Attributes

An attribute is a view feature Auto Layout uses to create a relationship between two views. There are two types of attributes: location and size. Location attributes include a view's leading, trailing, top and bottom edges. Size attributes include the view's width and height. Different user interfaces have different sets of attributes.

## Multiplier

A multiplier is a ratio you can apply to a relationship between two item attributes. For example, you can have Label B be twice the height of Label A.

## Constant

Constant is a fixed value that you add to a constraint. For example, Label A's leading edge spaces 16 points away from the superview's leading edge.

For a deep dive into the Auto Layout engine, read Chapter 14, "Under the Hood of Auto Layout."

There are a few more key concepts that impact how Auto Layout works. These include intrinsic content size and priorities.

## Intrinsic content size

Intrinsic content size is a size a view naturally takes up based on its content. It's like a balloon. The more air you put inside it, the bigger it becomes and vice versa. Some views have an intrinsic content size, while others don't because they don't contain defined size content.

# Priorities

There are times where Auto Layout is unable to satisfy all of the constraints applied in a view hierarchy. In situations like this, you can encounter scenarios where views fight for space. Using priorities, you can help Auto Layout resolve these layout decisions.

## Constraint priority

Constraint priority is about which constraint is more important for Auto Layout to satisfy. Constraint priority is defined using a value between 1 and 1000. A constraint priority between 1 and 999 is non-required. A constraint priority of 1000 is required. When you add a constraint, the default constraint priority is 1000, meaning it's required.

## Content hugging priorities

Content hugging priority helps Auto Layout decide which view gets stretched larger than its intrinsic size. Each view has a horizontal and vertical content hugging priority. A view with a higher content hugging priority fights harder from being stretched and vice versa.

When choosing between two views and deciding which view to stretch, the layout engine looks into the content hugging priorities. The view with the lower content hugging priority stretches.

A view's default content hugging priority varies depending on its type and how you instantiate the view. When you instantiate a view using Interface Builder versus programmatically, the default content hugging priority can differ.

## Compression resistance priorities

Compression resistance priority helps Auto Layout decide which view shrinks smaller than its intrinsic size. Each view also has a horizontal and vertical compression resistance priority. A view with a higher compression resistance priority fights harder from being shrunk and vice versa.

Each view has a default 750 horizontal and vertical compression resistance priorities. Unlike the content hugging priorities, all view objects have identical default compression resistance priorities regardless of the view type or instantiation method.



When choosing between two views and deciding which view to shrink, the layout engine looks into the compression resistance priorities. The view with the lower content compression resistance priority shrinks.

For more information on constraint priorities, read Chapter 8, “Content Hugging and Compression Resistance Priorities.”

Now that you’ve gotten an overview of Auto Layout, you’re ready to dive into more detail. Enjoy the journey!

# Chapter 2: Construct Auto Layout with the Interface Builder

By Jayven Nhan and Libranner Santos

Interface Builder is a graphical user interface (GUI) for developers to create an app's user interface (UI). With Interface Builder, developers can drag and drop UI objects onto a design canvas, and then use Auto Layout to create adaptive user interfaces for different screen sizes.

With Interface Builder, you can build most of your app's UI without writing a single line of code. In the end, you'll end up with an Interface Builder file, saved as either a storyboard or a .nib/.xib file.

In this chapter, you'll learn:

- To use a storyboard to layout your app's UI.
- The benefits and drawbacks of using storyboards.
- To use .nib/.xib to layout your app's UI.
- The benefits and drawbacks of using .nib/.xib files.



# Setting up the launch screen layout using a storyboard

Open **MessagingApp.xcodeproj** in the **starter** folder. Under the Storyboards group, open **LaunchScreen.storyboard**. You'll see the following:



You want the background image view to fill the edges of the view controller's view. The current storyboard preview device is the iPhone 8. On this device, the UI looks fine. But it's time to scrutinize how the launch screen looks on other devices and screen sizes.

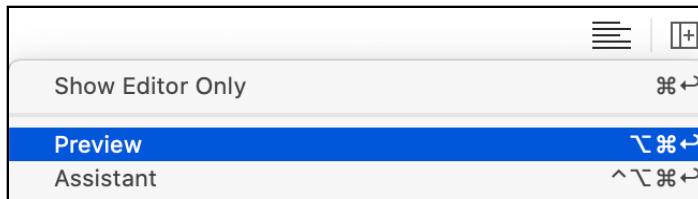
## Previewing layouts on multiple screen sizes

Interface Builder's Preview mode lets you see your screen layout on various devices without running the project on a simulator or device. Preview is useful, but you still have to keep in mind its limitations.

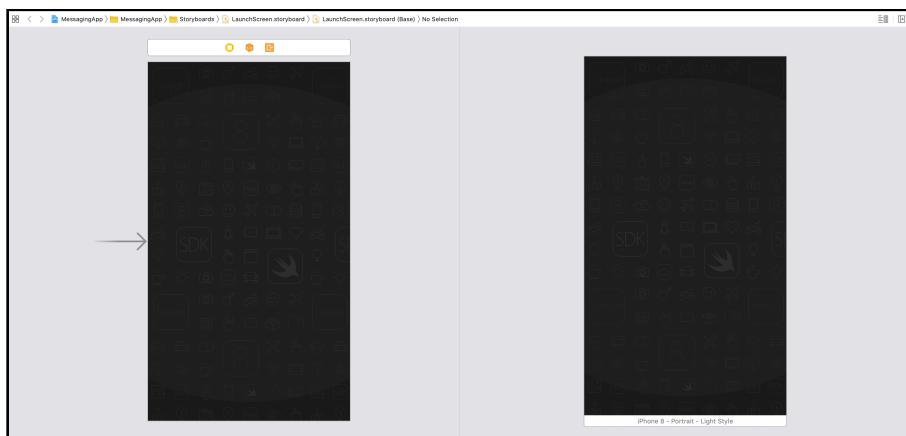
Although Preview is a handy tool, you should take what you see with a grain of salt. It's important to understand that Preview mode shows design time layouts, and design time layouts don't always match runtime layouts. This means that when you run your app, the actual view you see on a device can differ from the preview because of different runtime variables.

Nonetheless, using preview is a helpful starting point. When your layout isn't runtime dependent, preview accurately depicts how a view will look on various screen sizes. For this reason, Preview mode is perfect for seeing how the launch screen looks on different screen sizes.

On the top-right hand corner of the editor, click the **Adjust Editor Options** button. From the drop-down menu, click **Preview**.



After the selection, you should see preview on the right panel:



**Tip:** Give in-use panels more space by hiding unused panels.

Press **Command-0** to show/hide the Navigator panel.

Press **Command-Option-0** to show/hide the Inspectors panel.

You can delete devices from the preview panel. This is advantageous when you want to preview a different set of devices or arrange the preview devices in a different order. You'll do the latter.

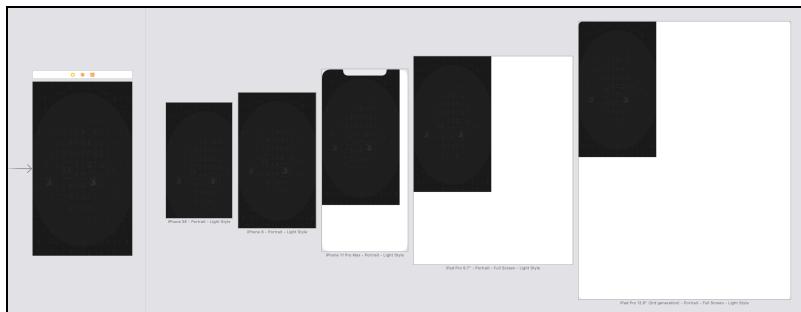
On the preview panel, select the **iPhone 8 device**. Press **delete** to remove the device. Then, click **+** for a list of devices you can preview.



Add the following devices to **preview**:

- iPhone SE
- iPhone 8
- iPhone 11 Pro Max
- iPad Pro 9.7" - Full Screen
- iPad Pro 12.9" (3rd generation) - Full Screen

Now, your workspace window will look like this:



You've discovered a layout problem! For devices larger than iPhone 8, the background image view doesn't fill the view's edges. You're going to fix this.

The launch screen user interface is not yet able to adapt itself to different screen sizes. At the moment, the background image view is positioned at a fixed coordinate and size. It's likely that your app's users will use more than just the iPhone 8. It can vary greatly, which means this is a great opportunity to use Auto Layout.

## Adding constraints

The first thing you'll learn is how to add constraints to a view in Interface Builder.

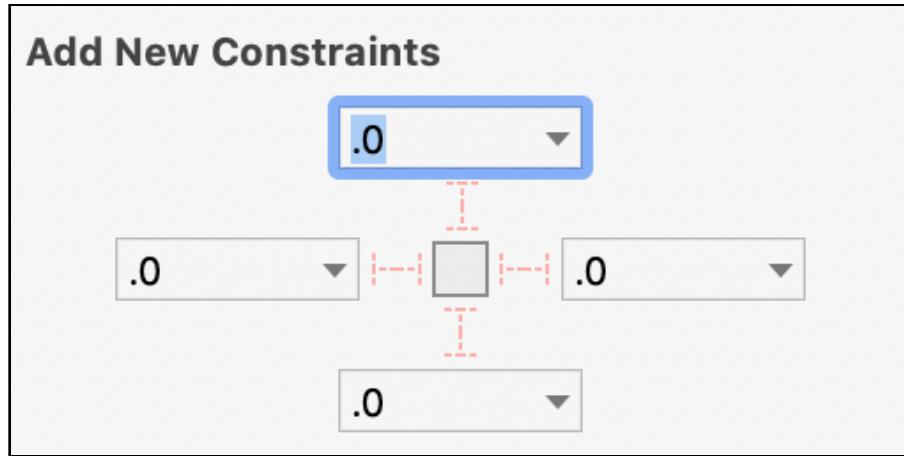
Since you no longer need the preview now, click **Adjust Editor Options**. From the drop-down menu, select **Show Editor Only**.

In Interface Builder, you can use the **Add New Constraints** menu to add new constraints to a view.

In the editor, select **background image view**. On the bottom-right corner of the editor, click **Add New Constraints**.

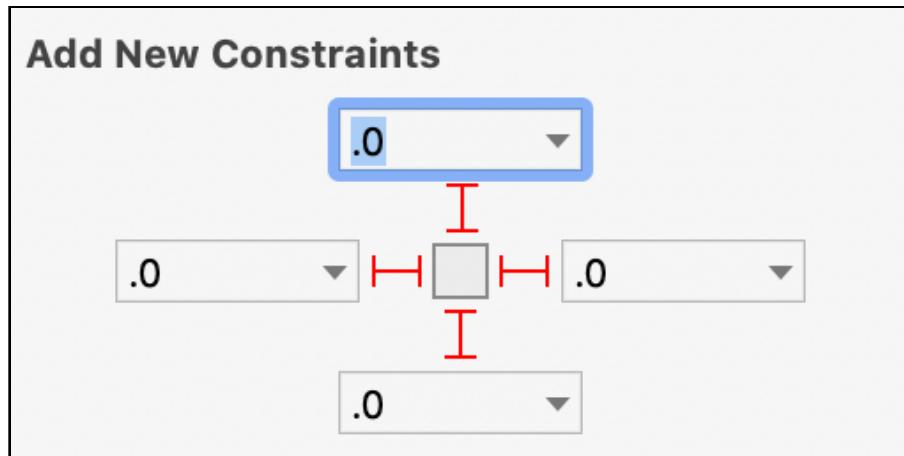


At this point, the Add New Constraints menu appears.

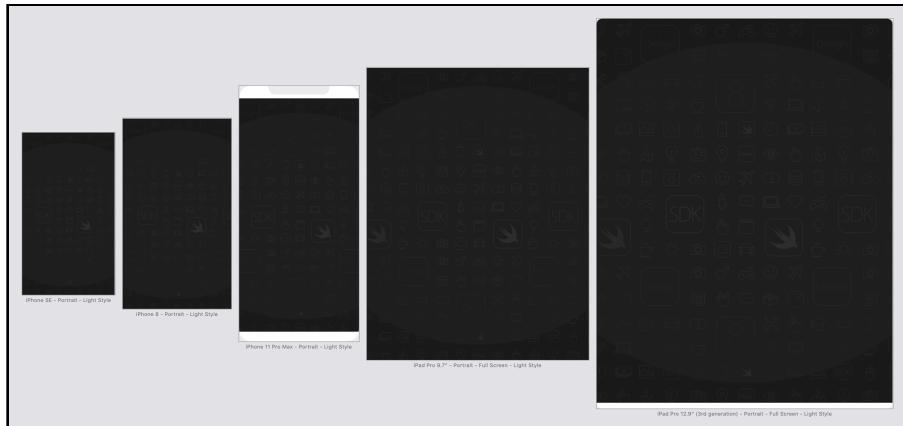


The square in the middle represents the selected view. The four red “beams” represent constraints relative to the view’s top, bottom, left and right edges. The text field next to the beam specifies the constraint’s constant value.

Click all **four red beams** for top, bottom, leading and trailing edge constraints. Make sure the constraint constants are all set to **0**. Click **Add 4 Constraints** to add the constraints.

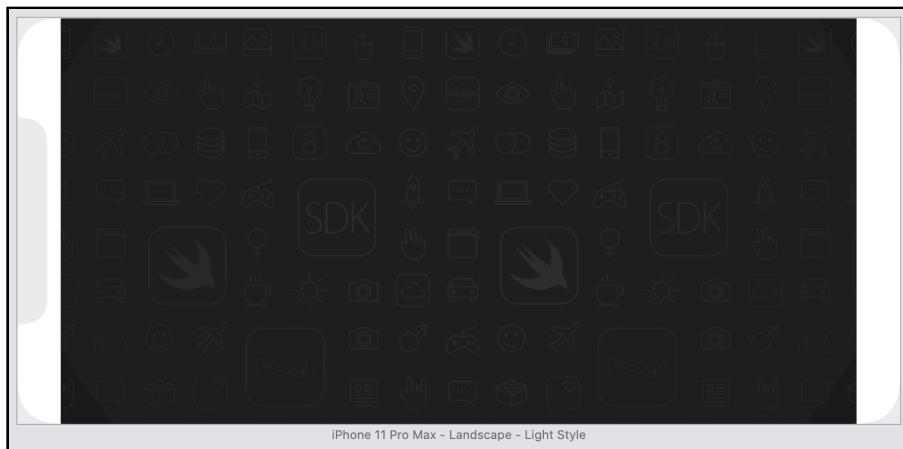


Open **Preview**. With the same devices on preview, you'll see this:



The background image view fills the view controller's view on all devices except for the iPhone 11 Pro Max and the iPad Pro 12.9" (3rd generation). On the iPhone, you can see white areas above and below the background view. On the iPad Pro 12.9", you see a white area below the background view.

On the Preview panel, hover your cursor over the **iPhone 11 Pro Max**. Click the **rotation button** to rotate the device's orientation from portrait to landscape. You'll see that the iPhone 11 Pro Max in landscape orientation has a noticeable margin around the background image view.



The background image view doesn't go all the way to the edges of the view controller's view on the iPhone 11 Pro Max. This is due to the Safe Area layout guide. The Safe Area layout guide makes it easier to create constraints in a way where content doesn't get obstructed on a device with a sensor notch or a rounded screen.

The iPhone 11 Pro Max is one of those devices.

## Editing constraints

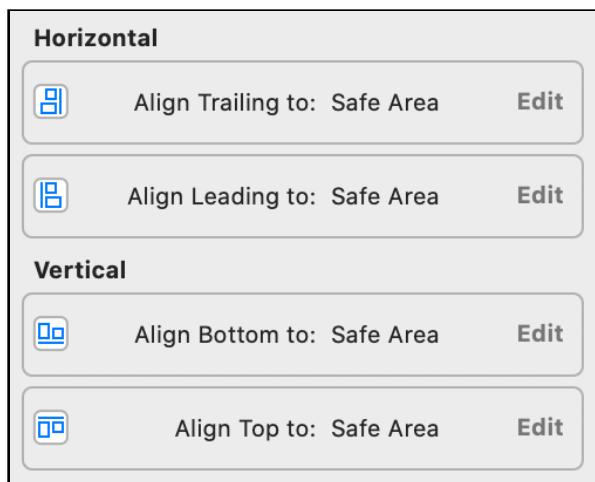
The Safe Area layout guide helps prevent obscured content. However, with this background image view, you want to have it fill the screen edges on every device.

In the editor, click **View as: iPhone 8** to open the preview device selection panel. Select **iPhone 11 Pro Max**.



This switches the storyboard's preview device to **iPhone 11 Pro Max**. This isn't necessary to edit the constraint. However, this does help you see the changes around the edges of the device in the storyboard.

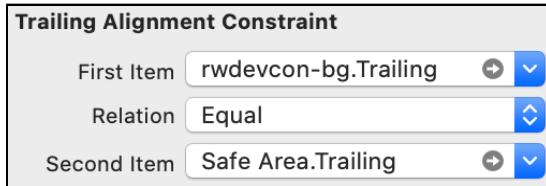
Select **background image view** in the storyboard. Open the Inspectors panel, and select **Size inspector**. Look at the constraints section.



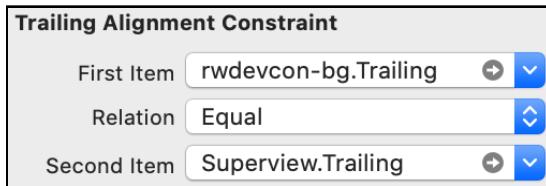
The background image view has four constraints. All of the constraints are relative to the Safe Area. You're going to update each constraint's relative item to the superview.

For each of the constraints, do the following:

1. Double-click the **constraint**. This shows the constraint's details in the Size inspector.

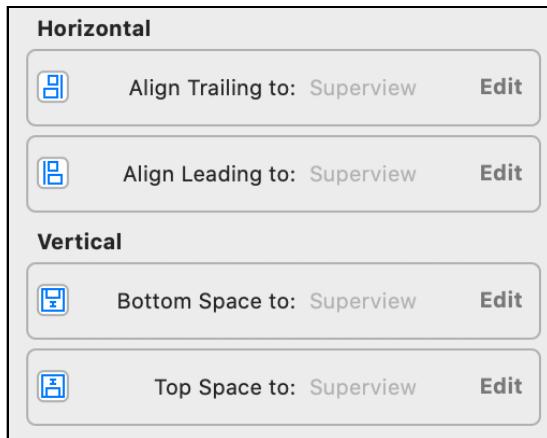


2. Click the **Second Item** drop-down menu. Select **Superview** to replace Safe Area.



3. Set the constant to **0**.

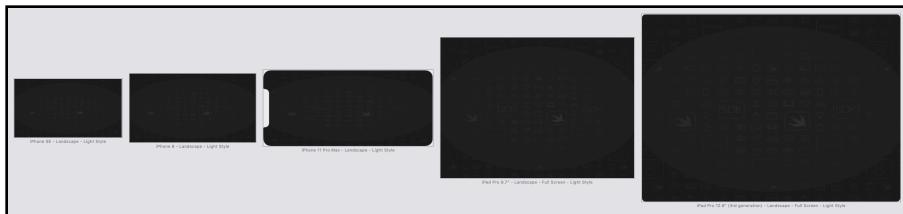
Afterward, the Constraints section will look like this:



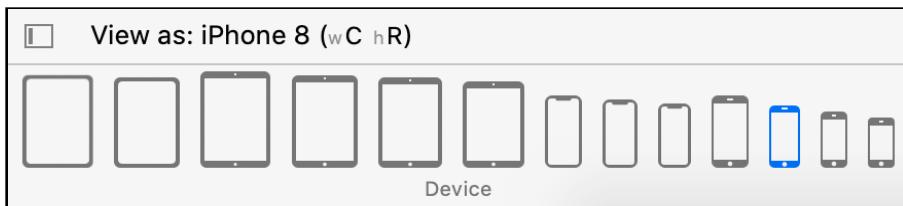
In Preview mode, you'll see the background view fill all of the edges on every device in portrait orientation:



You'll also see the background view fill all of the edges in landscape orientation:



Great! Now, open the editor. Switch the storyboard's preview device back to the **iPhone 8**.



## Constraint inequalities

The background image view uses equality constraints. In addition to equality constraints, you can also create inequality constraints using Auto Layout.

There are instances where you want to offer your constraints more flexibility than equality constraints do. Inequality constraints allow a view's attribute to be a range of numbers.

For example, a view's leading edge can space between 0 and 20 from the superview. The final constraint leading space value will be determined by different variables, which you'll see throughout different parts of the book. The idea behind inequality constraints is to let Auto Layout position or size within a value range.

It's time to create some inequality constraints.

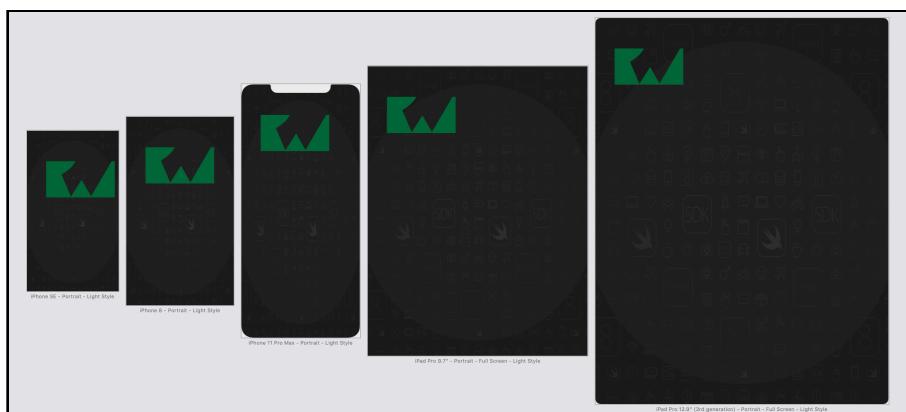
## Implementation

Are you ready to spice up the launch screen? Fantastic!

You'll add a logo image view with the Ray Wenderlich logo to the launch screen. Using constraints inequality, you'll make the logo image view size and position according to screen width while simultaneously maintaining a width-to-height aspect ratio.

In the editor, drag a **UIImageView** onto the launch screen view controller's view. Using the Attributes inspector, set the image view's image to **rw-logo**. Set the image's content mode to **Scale to Fill**.

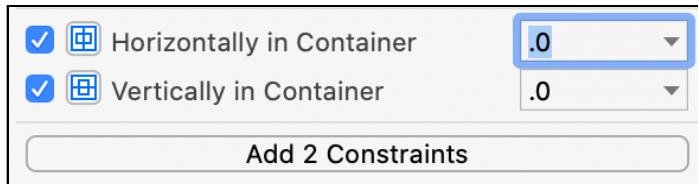
Preview the launch screen. It will look something like this:



In the editor, select the **logo image view**. In the bottom-right of the editor, click the **Align** menu.



Align the logo image view **vertically** and **horizontally** inside the container and click the button to add the two constraints.



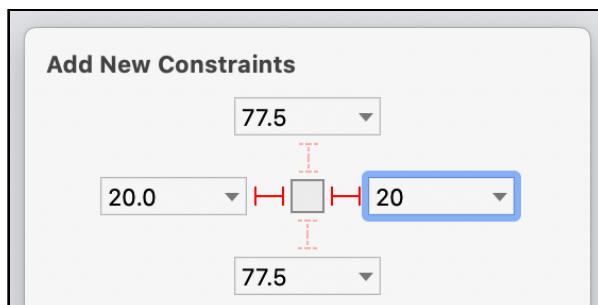
Once you add the constraints, look at the preview; it'll look like this:



This is a great example of intrinsic size. The size of the rw-logo image causes the UIImageView to have an intrinsic size. Without any other size constraints, the image view's intrinsic size defines the view's size. Although this image view looks decent on iPads, there's a lot of room for improvement on iPhones.

Add the following constraints to the logo image view:

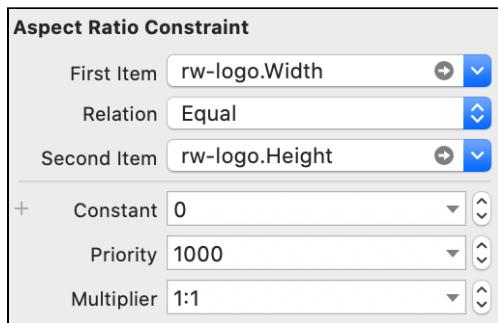
- A **leading constraint** with a **20** constant.
- A **trailing constraint** with a **20** constant.



Also, add an aspect ratio constraint:



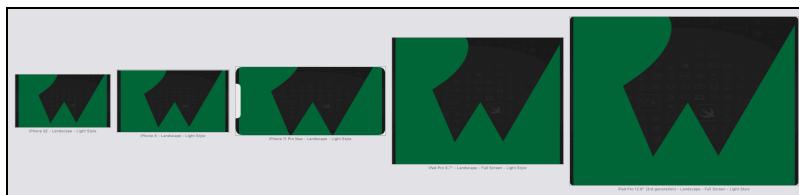
Ensure that the aspect ratio constraint's multiplier value is 1:1; this keeps the image view's width equal to its height. To set the aspect ratio constraint's multiplier, open the logo image view's Size inspector. Double-click the aspect ratio constraint and enter **1:1** in the Multiplier text field.



In preview, the launch screen looks like this in portrait:

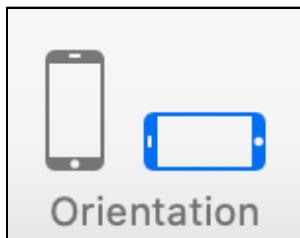


And like this in landscape:



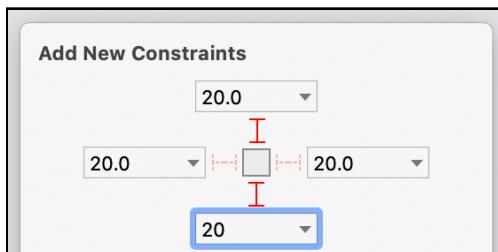
In portrait view, the logo image view is oversized on the larger devices. In landscape, the image is obscured on all devices. You're going to fix the layout for landscape orientation first.

Click the **View as** button. Then, select **landscape** in the orientation menu to change the storyboard's preview device orientation.



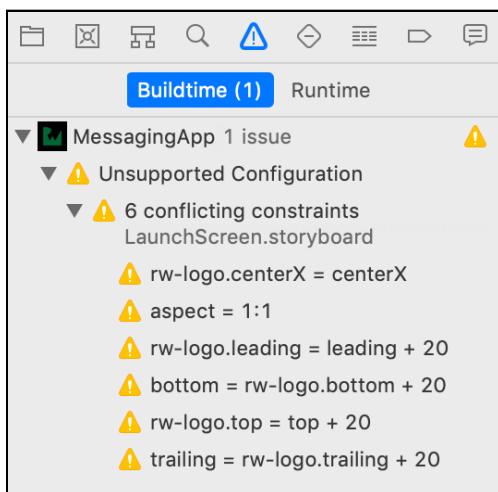
Next, add the following constraints to the logo image view:

- A **top constraint** with a **20** constant.
- A **bottom constraint** with a **20** constant.

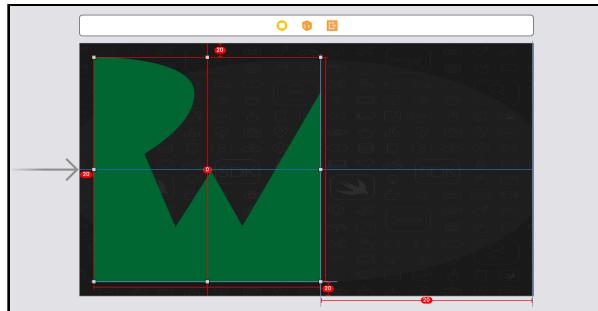


This keeps the logo image view from expanding over the vertical edges of the screen.

At this time, with the Issue navigator selected, you'll see a build time warning with six conflicting constraints.



The layout engine is unable to simultaneously satisfy all of the logo image view's constraints on the preview device size. Your editor will look something like this:



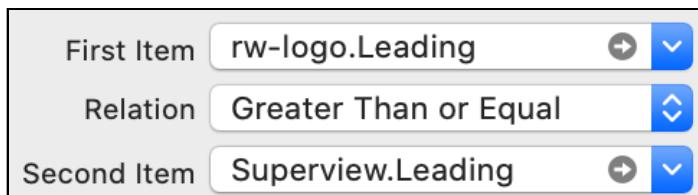
The issue is that you told Auto Layout that this image should be exactly 20 points from the left and the top. You also told Auto Layout that the image should have a 1:1 aspect ratio and center vertically and horizontally in the container. To simultaneously satisfy these constraints, the logo image view's leading edge spacing has to exceed 20 on the preview device.

You could simply resolve this issue by removing the aspect ratio constraint. However, the aspect ratio of the image view is a requirement. The aspect ratio constraint keeps the logo image's width and height equal to each other.

This is where inequality constraints can come into play. You don't need the image's leading edge to space exactly 20 points from the superview's leading edge. You just need it to be *at least* 20 points from the edge and having the constraint take on a value greater than 20 is fine too.

Now, you'll set the leading and trailing constraints' relations to **greater than or equal to**. Here's one way to update the constraint relation.

Select the **logo image view**. Open the **Size inspector**. Double-click the **leading constraint**. Once you see the constraint details panel, set the constraint relation to **Greater Than or Equal** from the relation drop-down menu.

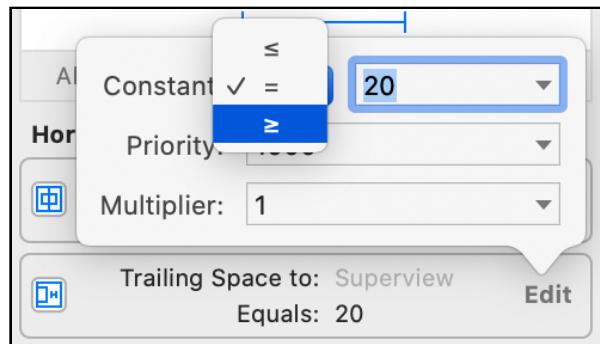


After updating the constraint relation, the constraint will look like this:



Here's another way to update the constraint relation. Select the **logo image view**. Open the **Size inspector**.

Now, click the **Edit button** on the trailing constraint. Then, click **relation dropdown menu** and select  $\geq$  to set the inequality.



Afterward, your launch screen preview will look something like this in landscape:



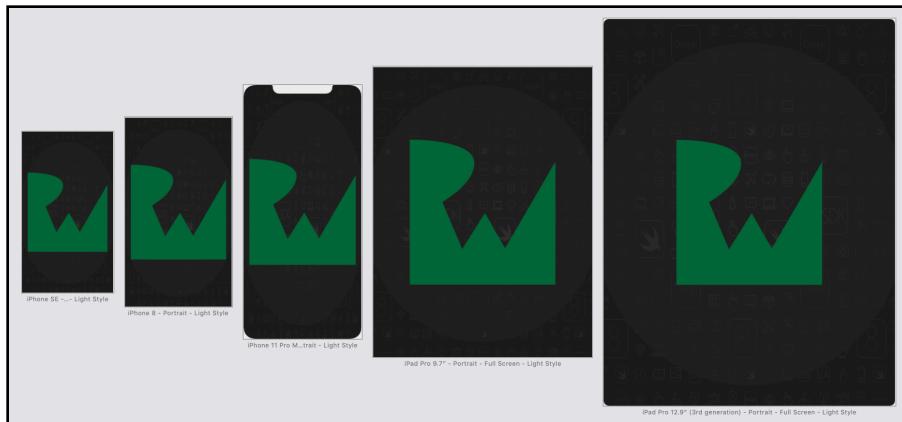
Flip the devices to portrait in preview, and you'll see this:



The logo image view is unable to satisfy all constraints with current device layout conditions. Similar to the scenario above, you can resolve the conflicting constraints buildtime warning by applying inequality constraints. This enables the logo image view to be flexible with the top and bottom spacing while conforming to the aspect ratio constraint on the vertical axis.

Set the top and bottom constraint relations to **greater than or equal**.

After the updated constraint relations, your launch screen preview looks like this in portrait:



As for the landscape orientation, that looks like this:



The launch screen is looking great and ready for different device sizes. Great job!

## Pros/cons and who is it for?

Apple has taken something complex and turned it into something comparatively elegant. However, as with any solution, there are benefits and drawbacks.

Here are five pros of using storyboard:

- As the name implies, storyboards become a flow diagram. A storyboard makes visualization of the app's flow vivid and obvious. You can tell by looking at a storyboard that the first view controller transitions to the second view controller with the segue indicator arrow. You can see the UI elements inside of a view controller. Also, available to you with a storyboard is the real-time update of how UI objects appear after constraint updates. These are three examples of the many visual benefits from storyboards.
- Almost every iOS developer learns to construct layouts in storyboards at some point today. Under the circumstance where the storyboard complexity is low, on-boarding a developer into your project is easier in the sense that more developers are well-versed with storyboards than other methods of layout construction. Plus, most developers start out learning to construct Auto Layout constraints using storyboards.
- Apple pushes developers to construct layouts in storyboards with Interface Builder. Historically, Apple has continued to add features, improve upon existing features and provide developer support materials on constructing layouts in Interface Builder. You can expect to continue support from Apple with the development of Interface Builder. Apple has emitted its aura of pushing everyone to learn to code. Using Interface Builder is often dramatically less intimidating than building layouts in other methods like code.
- Building out your layout in the Interface Builder gives you the benefits of Apple's behind the scene optimization of your layouts.

- Layout changes to your UI objects are reflected in the storyboard. This omits the need to build and run your app to see how your UI objects have changed. Consequently, this saves time.

Here are five cons of using storyboard:

- The infamous merge conflicts resolution problem. Merge conflicts are easy to come by when two or more developers make changes onto a single storyboard. Because storyboard files are stored as XML, the files are not exactly the most reader-friendly. When there are multiple changes from different developers using the same storyboard file, you may find yourself spending some time resolving merge conflicts. This effect can compound over time. Apple has taken measures to make merge conflicts less of a problem with more readable XML files and by making it easier to split up large storyboard files. However, the storyboard merge conflict problem persists.
- Do you want everything that has to do with the app UI in one place? Interface Builder can be problematic as your app complexity grows. You can set your UI to different values to do different things in Interface Builder with storyboards. However, at the same time, you can also make changes to your UI objects in code. You may wonder why your UI looks completely different on your device than the storyboard. In addition, you may find your app requiring both storyboard and code maintenance instead of one or the other. This can lead to presentation logic ambiguities.
- Re-usability is vital for maintainability. In the scenarios of where you want to build on top of the existing view controller or reuse certain UI elements from a view controller you storyboard, then you are out of luck. The storyboard does not support this.
- Interface Builder is a graphical user interface built as an additional layer for generating XML codes. The additional layer makes Interface Builder prone to UI bugs on the editor and at runtime. The reliability of this layer is dependent on the Xcode team.
- Using Interface builder increases compile time compared to code implementation. As complexity increases in your storyboard files, compile-time can take up a chunk of development time. Consider the compounding effect of development time.

When deciding on your approach, take consideration of your variabilities and the pros and cons of each layout construction approach. Generally, apps that are simple in design and presentation logic make storyboard a great partner in crime.

# Using Xibs

Have you ever heard about .xib files? Even if you say no, I bet you've probably used them. Those screens that are part of a scene in your storyboard are .xib files. The only difference is that, thanks to the storyboard, you have some benefits, like segues and a nice look at the workflow of your app. Before carrying on and to avoid ambiguity, it's important to understand the differences between .xib and .nib files.

As found in the official Apple documentation at <https://apple.co/2YVj8CO>:

*"A .nib file describes the visual elements of your application's user interface. This includes windows, views, controls, and many others. It can also describe non-visual elements, such as the objects in your application that manage your windows and views."*

Usually, people refer to .nib and .xib as the same thing; the difference relies on the fact that .xibs are only flat files — XML files — meanwhile, .nibs are deployable files. To use a .xib file in the final compiled app, it must be compiled into a .nib.

## How to achieve modularity with .xib

If you're working on a team, it can be pretty rough to deal with merge conflicts when everybody is using a single storyboard. That's why you need a way to work on screens in a more decoupled way.

Sometimes, you need to create parts of the user interface that should be reusable and independent. For those cases, .xibs are a really good option. It's similar to the way you create storyboards, but with no attachment. So you can have graphics interfaces that don't depend on other views or transitions.

### The .xib object life cycle

It's important to understand how .nibs files are loaded, so you can create better interfaces that behave as expected and have good performance. The following diagram shows the chain of steps executed to display a view using a nib:

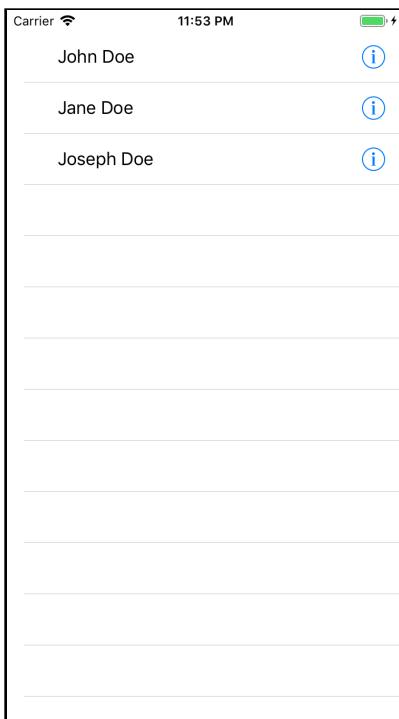


1. Load the contents of the file and any referenced resource files into memory.
2. Instantiate all of the objects inside the .xib file; by default, every object receives an `init(coder:)` message.

3. Reestablish all Outlet and Action Connections.
4. Call `awakeFromNib()`.
5. Display the view.

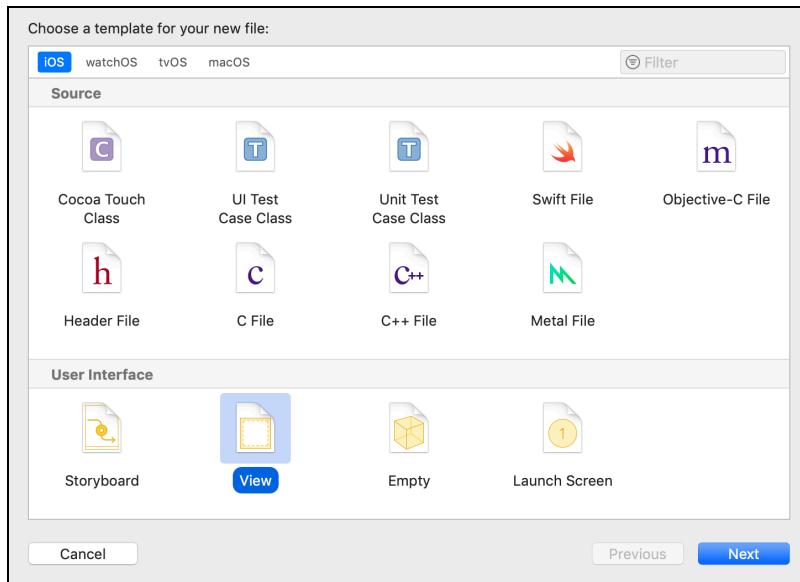
## Practical use case

Open **MessagingApp.xcodeproj** in the **starter** folder. Build and run. You can see a contact list; it's a simple table view that shows the name of the contact and has an accessory info button on the right. But if you press the info button, nothing happens. Well, it's time to do something about that!

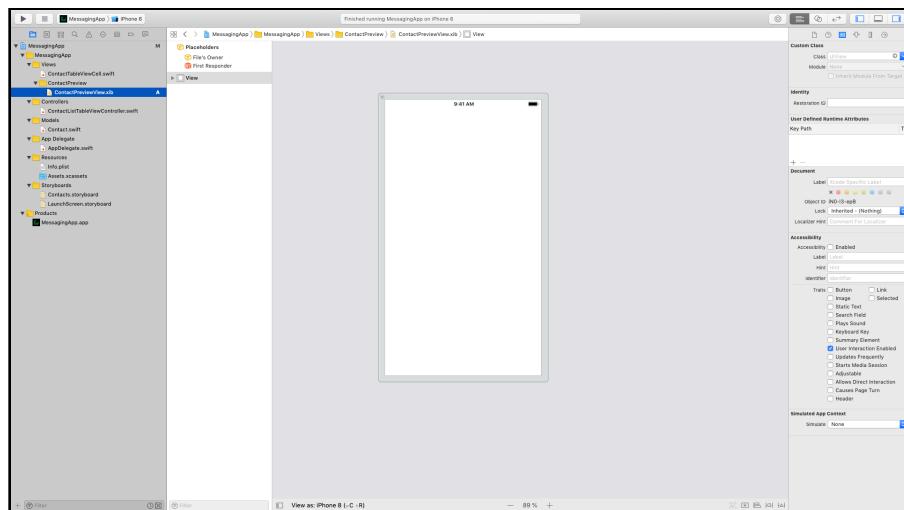


Stop the project and expand to Views folder on the Project navigator. **Right-click** over the **ContactPreview** folder and select **New File....**

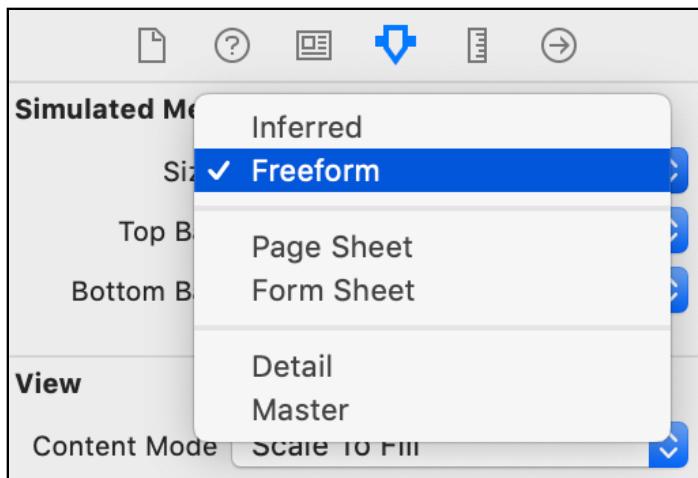
Under the User Interface section, select **View** and click **Next**.



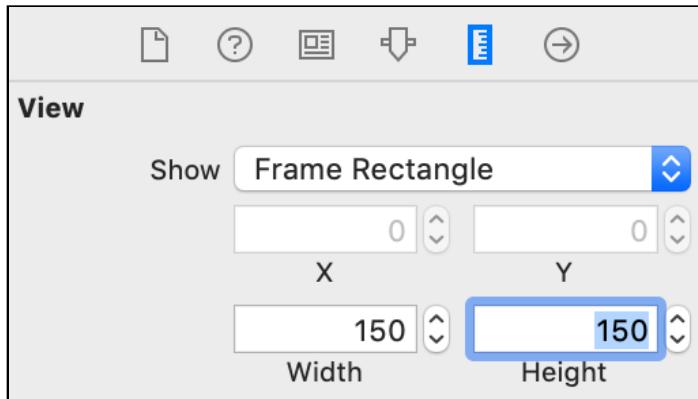
In the Save As text entry, enter **ContactPreviewView** and press **Create**. This creates a **ContactPreviewView.xib**, which is just an empty view, for now.



Select the View and go to the Attributes inspector. On the simulated metrics section, select **Freeform** for the size attribute.



Now, go to the Size inspector, and set the width and height equal to **150**.



Finally, go to the Attributes inspector, and for background, select **black color**. The view will look smaller now — just remember these are simulated metrics to help give you a better idea of how the view will look at runtime. You still need to set the necessary dimensions and position for the view, also known as constraints.



You need to assign a file owner to the new .xib file, so you can instantiate it and use it in your app.

In the same group, add another new file. Select **Cocoa Touch Class** with **UIView** subclass and name it **ContactPreviewView**.

Open **ContactPreviewView.xib**. Select **File's Owner** in the document outline, and go to the Identity inspector. Select **ContactPreviewView** for the class attribute.

Add the following code, starting at the beginning, inside the **ContactPreviewView** class in the **ContactPreviewView.swift** file:

```
// 1
override init(frame: CGRect) {
    super.init(frame: frame)
    loadView()
}
// 2
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    loadView()
}

func loadView() {
    // 3
    let bundle = Bundle(for: ContactPreviewView.self)
    // 4
    let nib =
        UINib(nibName: "ContactPreviewView", bundle: bundle)
    // 5
    let view = nib.instantiate(withOwner: self).first as! UIView
    // 6
    view.frame = bounds
    // 7
    addSubview(view)
}
```

Here's what you did:

1. Override the `init(frame:)` constructor so that you can call `loadView()`.
2. Override the required `init(coder:)` constructor so that you can call `loadView()`. Since it's a required constructor, not implementing it will cause an error.

3. Get a reference to the bundle that contains the **ContactPreviewView.xib** file.
4. Create an instance of the **ContactPreviewView.xib**, indicating its containing bundle.
5. Instantiate the view and assign the owner, which is the current class.
6. Set `view.frame` equal to `bounds`. This gives `view` the same dimensions as its parent.
7. Add the instantiated view to the current view.

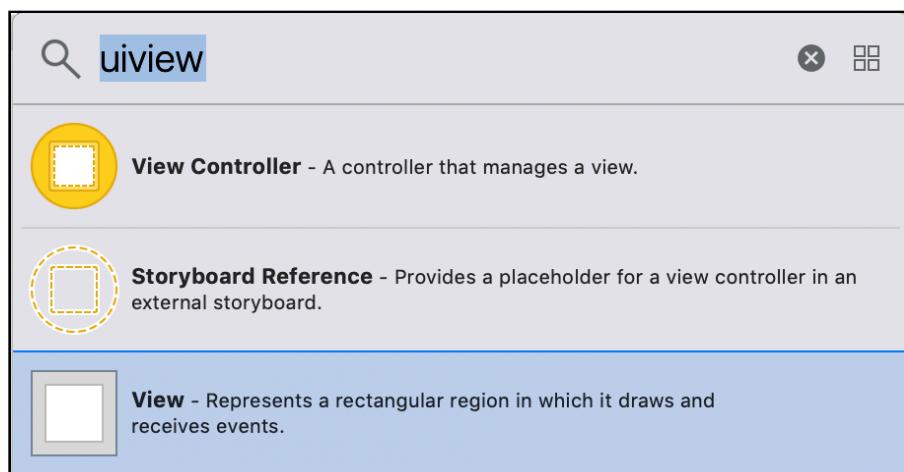
You can now use `ContactPreviewView` anywhere in your project, such as in `ContactListTableViewController`.

Open **ContactListTableViewController.swift** (in the Controllers group) and add this code immediately after the `cellIdentifier` declaration:

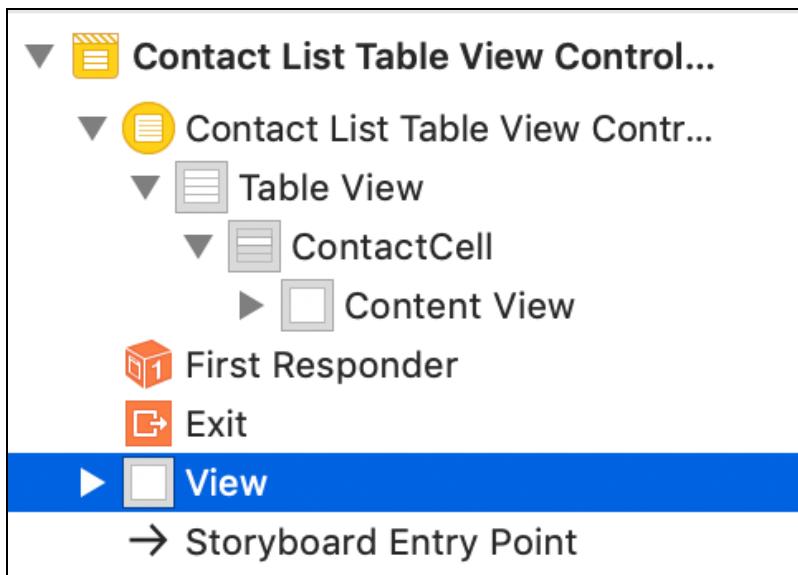
```
@IBOutlet var contactPreviewView: ContactPreviewView!
```

This code creates an outlet to reference the custom view. Now, add the custom view to the Contacts storyboard.

Open **Contacts.storyboard**. Press **Command-Shift-L** and look for **uiview**.

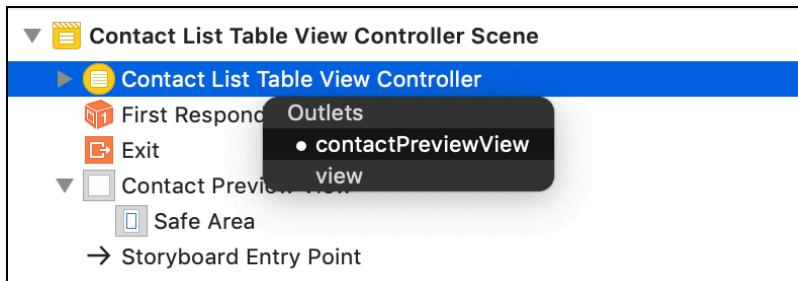


Drag the view object below the Exit item in the document outline.



Select the view. In the Identity inspector, select `ContactPreviewView` for the class attribute.

**Control-drag** from `ContactListViewController` to the `ContactPreviewView`. An outlets popup will appear; select `contactPreviewView`.



It's time to set up the info accessory button on the Contact List.

Go to **ContactListTableViewController.swift** and type the following code at the end of the class:

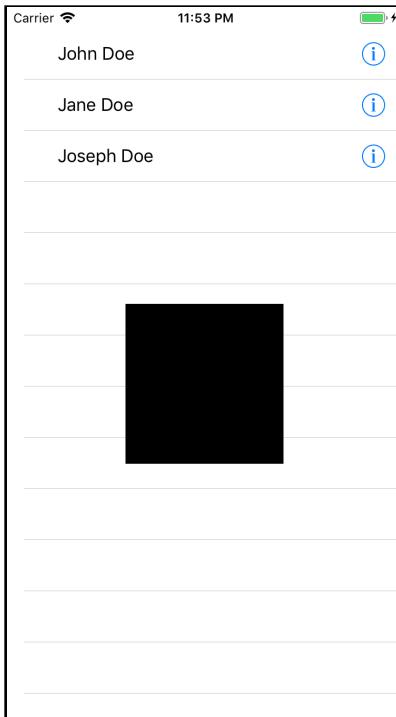
```
// MARK: - Setup Contact Preview
override func tableView(_ tableView: UITableView,
    accessoryButtonTappedForRowWith indexPath: IndexPath) {
    // 1
    let contact = contacts[indexPath.row]
    // 2
    view.addSubview(contactPreviewView)
    // 3
    contactPreviewView
        .translatesAutoresizingMaskIntoConstraints = false
    NSLayoutConstraint.activate([
        contactPreviewView.widthAnchor.constraint(
            equalToConstant: 150),
        contactPreviewView.heightAnchor.constraint(
            equalToConstant: 150),
        contactPreviewView.centerXAnchor.constraint(
            equalTo: view.centerXAnchor),
        contactPreviewView.centerYAnchor.constraint(
            equalTo: view.centerYAnchor)
    ])
    // 4
    contactPreviewView.transform =
        CGAffineTransform(scaleX: 1.25, y: 1.25)
    contactPreviewView.alpha = 0
    // 5
    UIView.animate(withDuration: 0.3) { [weak self] in
        guard let self = self else { return }
        self.contactPreviewView.alpha = 1
        self.contactPreviewView.transform =
            CGAffineTransform.identity
    }
}
```

Here's what you did:

1. Get the corresponding contact using `indexPath.row` as the index.
2. Add the new view to the current main view.
3. Set the constraints to make sure the view has a size of  $150 \times 150$  and is centered vertically and horizontally.

4. Start with the view 1.25 times larger than normal and fully transparent.
5. Create an animation to shrink the view down to normal and fade in to fully opaque.

Build and run. Tap any of the accessory info buttons; you'll see a black square showing at the center of the screen.



The view is showing as expected. It's time to add some content.

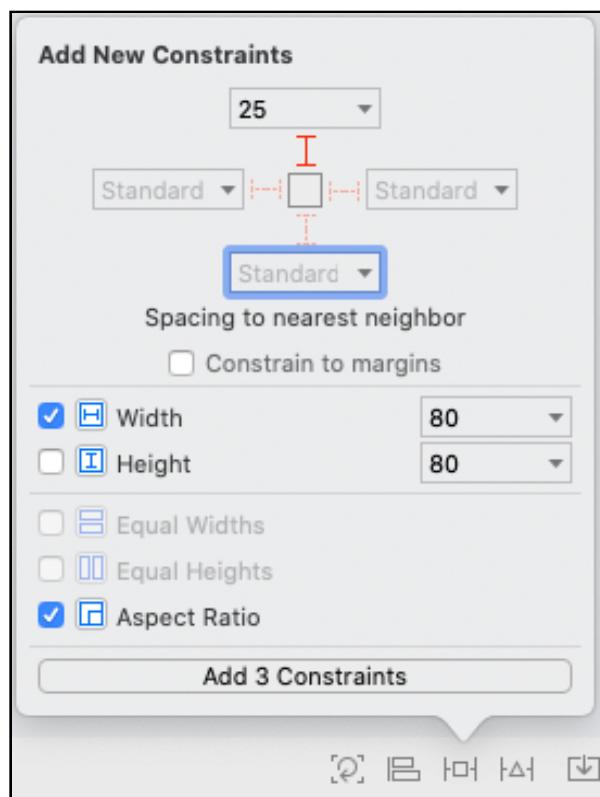
Open **ContactPreviewView.xib**. Select the **View** element in the document outline and change the background in the Attributes inspector to **rw-light**.

Press **Command-Shift-L** and type **image view**. Drag and drop the image view inside the blank view.

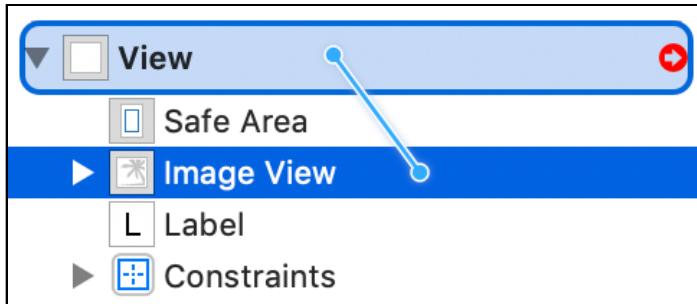
Press **Command-Shift-L** again, and this time look for a **label**. Once you find it, drag the label and drop it where you dropped the image view.



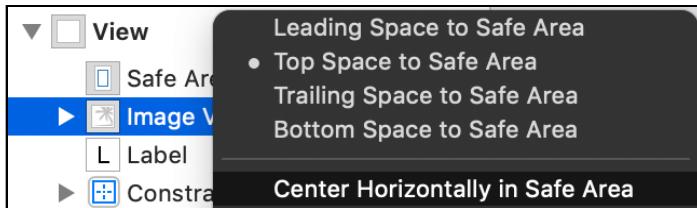
Select the **Image View** and click **Add New Constraints** on the bottom right corner. Set the top constraint to **25**, width to **80** and select the **Aspect Ratio** checkbox. Click **Add 3 Constraints**:



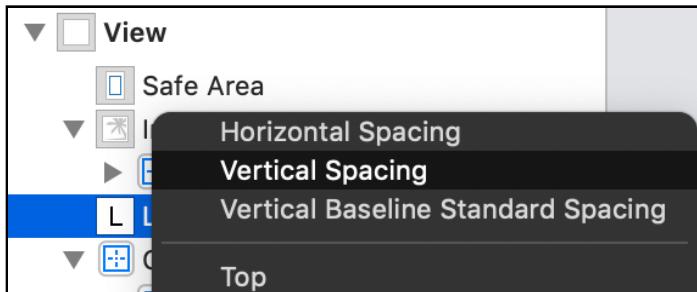
In the document outline, **Control-drag** from the Image View to the **View** object.



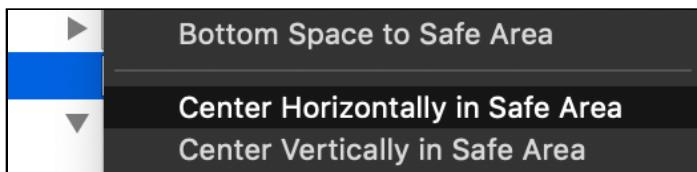
From the popup that just appeared, select **Center Horizontally in Safe Area**.



Now, you'll add some constraints to the label. Select **Label** and then **Control-drag** until you select the **Image View**. From the popup that just appeared, select **Vertical Spacing**.



Select **Label** and **Control-drag** until you select the **View**, and select **Center Horizontally in Safe Area**.

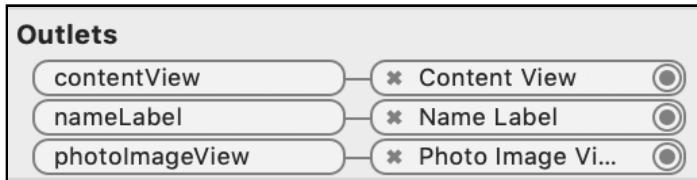


You now have the view all set up. Don't worry if it doesn't look good yet; you'll fix that in a moment.

Go to **ContactPreviewView.swift**, and immediately after the first open brace, add the following code:

```
@IBOutlet var photoImageView: UIImageView!
@IBOutlet var nameLabel: UILabel!
@IBOutlet var contentView: UIView!
```

Here, you create three outlets that will connect to the label, image view and the content view of the .xib file. To make those connections, go back to **ContactPreviewView.xib**. Select the **File's Owner** item and go to the Connections inspector. Connect each outlet with its corresponding element. Do that by dragging from the circle at the right side of each outlet to the element.



Time to set up the data to display. Go to **ContactListTableViewController.swift** and, immediately before `view.addSubview(contactPreviewView)`, add the following code:

```
contactPreviewView.nameLabel.text = contact.name
contactPreviewView.photoImageView.image =
    UIImage(named: contact.photo)
```

This piece of code fills the name label and the image view with its corresponding data.

Build and run. Tap any of the accessory info buttons, and you'll see the contact preview. Currently, there's no way to dismiss this modal. You need to fix that.

First, set up an animation to hide the Contact Preview view. Add this block of code at the end of **ContactListTableViewController**.

```
@objc private func hideContactPreview() {
    // 1
    UIView.animate(withDuration: 0.3, animations: { [weak self] in
        guard let self = self else { return }
        self.contactPreviewView.transform =
            CGAffineTransform(scaleX: 1.25, y: 1.25)
        self.contactPreviewView.alpha = 0
    })
}
```

```
    }) { (success) in
        // 2
        self.contactPreviewView.removeFromSuperview()
    }
}
```

Here's what you did:

1. Create an animation to scale the view up and fade it out.
2. Remove the view when the animation ends.

Next, you need to create a function to set up the gesture recognizer. Add this code after `hideContactPreview`:

```
private func configureTapGesture() {
    // 1
    let tapGesture = UITapGestureRecognizer(
        target: self,
        action: #selector(hideContactPreview))
    // 2
    contactPreviewView.addGestureRecognizer(tapGesture)
    view.addGestureRecognizer(tapGesture)
}
```

Here's what you did:

1. Create a `UITapGestureRecognizer` that will trigger `hideContactPreview` when tapped.
2. Add the gesture to `contactPreviewView` and the view. This code allows the method to be called when the user taps the view or the popup view.

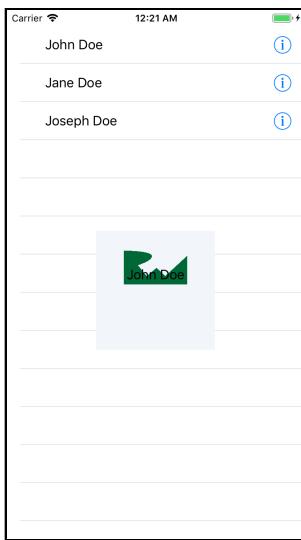
Finally, add the following code in `viewDidLoad()`:

```
configureGestures()
configureTapGesture()
```

After the view controller's view has loaded, you'll invoke registering a tap gesture onto the view.



Build and run.

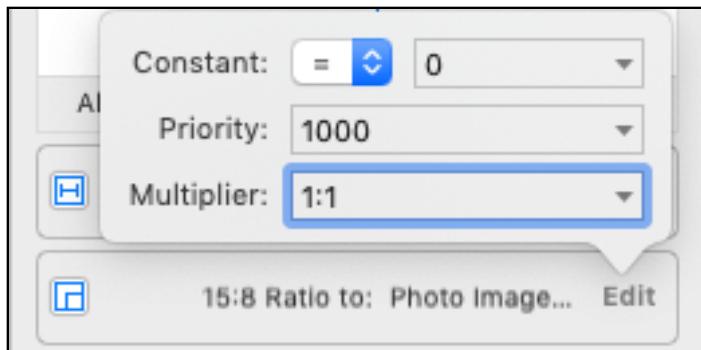


At this point, the contact preview doesn't display as expected. Therefore, it's necessary to edit the constraints to give the correct values.

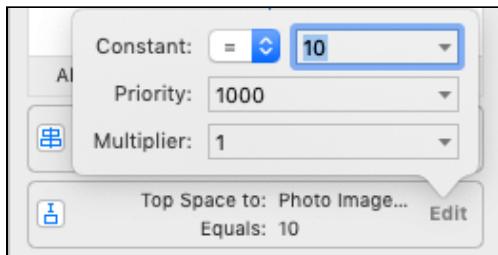
## Editing constraints using Interface Builder

Open **ContactPreviewView.xib**. Select the **Photo Image View**, and on the right side of the Interface Builder, select the Size inspector.

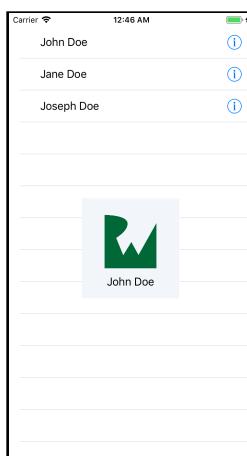
Click **Edit** on the constraint that contains the text **Ratio to...**, and set the multiplier to be equal to **1:1**, and then press **Enter**.



Select the **Name Label** in the document outline and go to the Size inspector. Click **Edit** on the constraint that starts with **Top Space to....**. Make the constant equal to **10**, and then press **Enter**.



Build and run.



You now have a nice looking and reusable contact preview modal.

## Pros/cons and who is it for?

.xib files can offer many advantages, especially when compared to storyboards. But remember, you can use both tools in your project.

### Pros

- Since .xibs have less UI described in one file, it can be easier to manage or avoid conflicts, compared to storyboards.
- It's easy to reuse a .xib file throughout a project, to avoid creating duplicated interfaces, or even use them in other projects.

- It can help to have part of the system more encapsulated.
- It shines when you need to create custom controls. If you're going to use Interface Builder to create custom controls, .xib files are the way to go.

### Cons

- You can't have relationships between screens, something that's simple and useful while using storyboards.
- No visualization of the workflow of your app. That's something you can only have with storyboards.
- If the view is dynamic, it can be difficult to create using .xibs.

When deciding on using .xib files, take consideration of the problem you and your team are facing. Then, account for the pros and cons of utilizing .xib files. Generally, Interface Builder users see .xib files offer significant modularity and reusability benefits.

## Challenges

Great job making it this far. Here are a few challenges to help solidify what you learned.

Create a custom label and position and shape the custom label according to the following specifications:

- Add a label to the view controller in **Profile.storyboard**. This will be referred to as the profile name label.
- Center the profile name label horizontally with the profile image view.
- Make the profile name label and profile image view have equal widths.
- Position the profile name label 16 points from the bottom of the profile image view.
- Make sure the bottom of the profile name label is at least 20 points from the bottom of the safe area.
- Under the project's **General/Deployment Info** settings, set the main interface to **Profile.storyboard**. Afterward, run your project to see how your user interface layout looks on the simulator.

After attempting the challenge, you can use the final project file to compare your implementation.

## Key points

- The preview feature helps you see your UI on different screen sizes without running the project on a simulator or device.
- You can use the Add New Constraints button to apply Auto Layout constraints in Interface Builder.
- You can use the Attributes inspector to set the object's properties.
- Remember to account for the Safe Area layout guide to ensure devices with a notch display as intended at runtime.
- You can set equality and inequality constraint relations between items.
- The Issue navigator can show you the conflicting constraints in a storyboard.
- Using .xib files is a great way to achieve modularity.
- There are benefits and drawbacks to every tool. Building Auto Layout constraints in storyboards or .xib files with Interface builder is no exception. Consider the pros and cons of each to conclude an optimized solution for what you need.

# Chapter 3: Stack View

By Jayven Nhan

`UIStackView` is a smart view container that intelligently arranges its subviews. Using stack views, you can create adaptive layouts with fewer constraints because most of the heavy layout work is done for you. In fact, Apple recommends that developers use stack views over manual constraints where possible.

To effectively use stack views, you need to familiarize yourself with their behavior. You need to understand how a stack view decides to align, distribute, space, size and position its subviews. The configurable stack view properties determine these factors.

In this chapter, you'll learn about the following stack view topics:

- Embedding views inside a stack view.
- Adding constraints to a stack view.
- Aligning and distributing views within a stack view.
- Nesting stack views.
- Deciding when and when not to use stack views.

These topics will help you understand how stack views operate. By the end of this chapter, you'll have gained a solid foundation of stack views and be able to implement adaptive layouts with stack views in your projects.



## Implementing a vertical stack view

Adding a stack view onto a view controller's view using Interface Builder is similar to adding any other standard view object: You drag and drop a stack view object from the Object Library onto a view controller's view. However, instead of doing that here, you'll learn how to embed existing views into a vertical stack view.

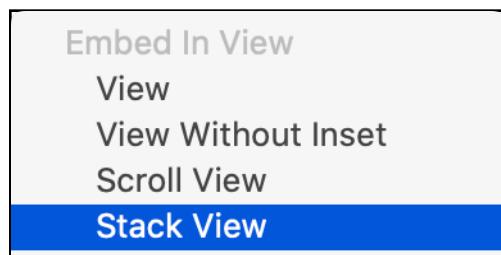
First, open the **starter project**. Then, in the **Storyboards** group, open **Profile.storyboard**.

In the editor's document outline, use **Command-click** to select the **Profile Image View** and the **Full Name Label**.

At the bottom of the editor, click **Embed In**.



From the dialog, select **Stack View**.



In the document outline, you'll see a stack view with the profile image view and full name label embedded. You may notice that some constraints are missing in the embedded views. For instance, the constraint that spaces the profile image view's bottom edge and the full name label's top edge.

Your next task is to add constraints to the stack view.

## Adding constraints to a stack view

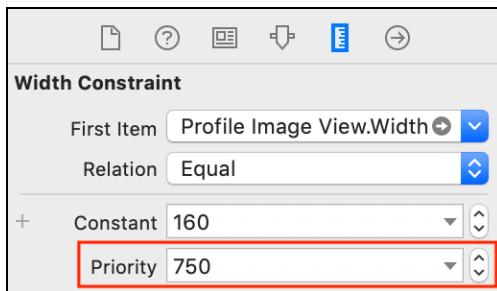
Select the **Profile Image View** and open the **Size inspector**. Use **Shift-click** to select all of the view's constraints, then click **delete** to delete the constraints.

Now, add the following constraints to the **Profile Image View**:

- 160 width with a 750 priority.

- 1:1 multiplier for a width to height aspect ratio.

You set the width constraint's priority in the constraint details panel.



The 160 @ 750 width constraint allows the profile image view to get as close to 160 as possible. Instead of using a required constraint priority, you set a high constraint priority, which enables Auto Layout to ignore this constraint and satisfy a different constraint if necessary.

For example, suppose the full name label's intrinsic content width is greater than 160 points due to an increase in font size or longer text; the full name label's width will take precedence over the profile image view width. If the full name label's width is 184, the layout engine is permitted to break the profile image view width.

The aspect ratio constraint keeps the width and height of the profile image view equal to each other.

In the document outline, select **Stack View** and add the following constraints:

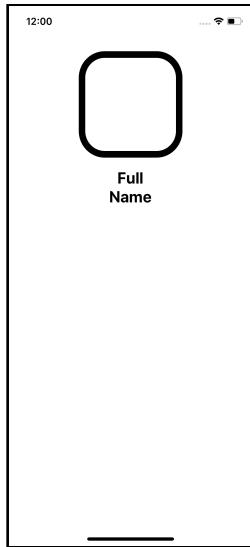
- 24 points spacing between stack view's top edge and Safe Area's top edge.
- 20 points spacing between stack view's leading edge and its superview's leading edge. Set the constraint relation to greater than or equal to.
- Align horizontally in container's center.

The top constraint sets the y position of the stack view. The leading constraints add limitations to how far the stack view can expand horizontally. The center horizontally alignment constraint ensures the stack view always centers in the view.

The stack view's subviews determine the stack view's width. Between the label width and the image view width, the view with the larger width determines the stack view's width. The stack view's width also determines the stack view's x position.

The stack view's subviews and spacings between subviews determine the stack view's height. The stack view now fulfills the Auto Layout position and size requirements.

Build and run, and you'll see something like this:



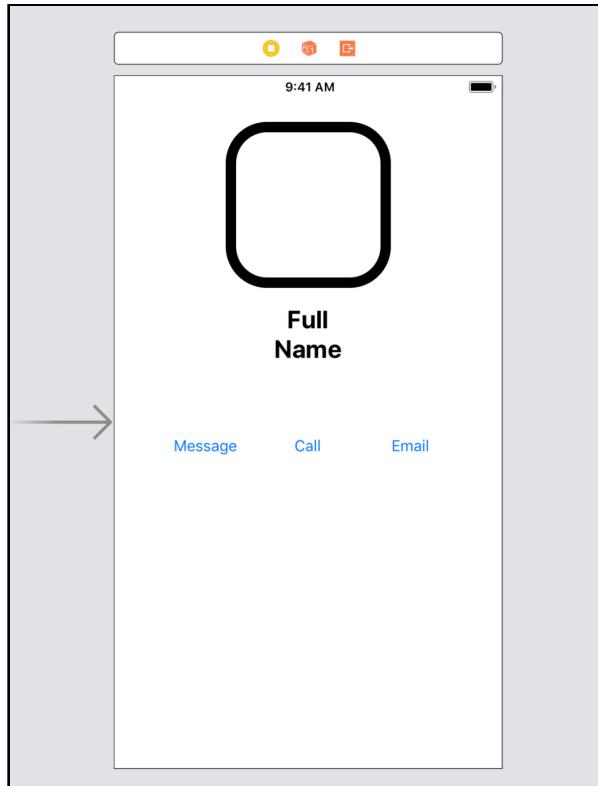
## Embedding views into a horizontal stack view

A stack view distributes its views in one of two axes: horizontal or vertical. In this section, you'll implement a stack view that distributes its subviews on the horizontal axis. For this exercise, you'll create a horizontal stack view with three embedded buttons.

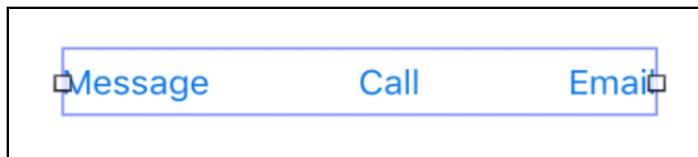
In the main storyboard, drag three buttons onto the profile view controller's view with the following titles:

- Message
- Call
- Email

Your view will look like this:



Select all **three buttons** and embed them into a **stack view**. Once you embed the buttons, you'll see the following in the editor:



Based on the initial positioning of the buttons, you'll notice that the stack view distributes its subviews horizontally within Interface Builder. This is one of the smart adjustments and added benefits of using Interface Builder.

With stack views, you can typically add or remove subviews without adding additional constraint configurations. For example, imagine one day that you want to add a new button between the message and call buttons. Using a stack view, you simply drag and drop a new button between the message and call buttons. From there, the stack view automatically handles the layout.

With manual constraints, you'd typically need to remove and reconfigure the old view constraints and configure the new view's constraints. This is extraneous work, especially when stack views are available at your disposal.

There are still layout edge cases you may need to handle, but for the most part, your layout is good to go.

## Alignment and distribution

You're ready to learn about stack view's alignment and distribution properties.

Select the horizontal stack view. In the Attributes inspector, you'll see the alignment and distribution properties. These are two of stack view's magical properties for automatic layout of the subviews, and in this section, you'll take a deeper dive into how each one works.

## Alignment

Alignment defines the stack view's subviews layout arrangement perpendicular to the stack view's axis. The alignment property values are different for a horizontal stack view versus a vertical stack view. You won't implement all of the possible property values in this book; however, you'll get to see the effect each property has on a stack view.

First up, you'll look at a stack view's alignment properties on the horizontal axis. When a stack view's axis is horizontal, there are six possible values: fill, top, bottom, center, first baseline and last baseline.

### Horizontal axis fill alignment

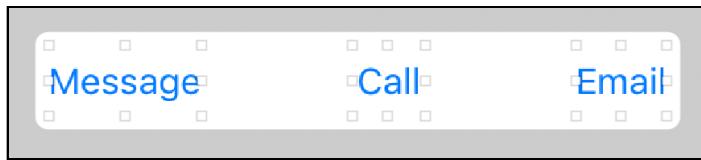
In a horizontal stack view, a **fill** alignment makes the views take up all of the vertical space inside the stack view.

Whenever you're setting up a superview/subview relationship, you can take the approach to define external constraints to set the size of the superview and then cascade those down to the subviews. Alternatively, you can let the size of the subviews help dictate the size of the superview. The same is true of stack views.

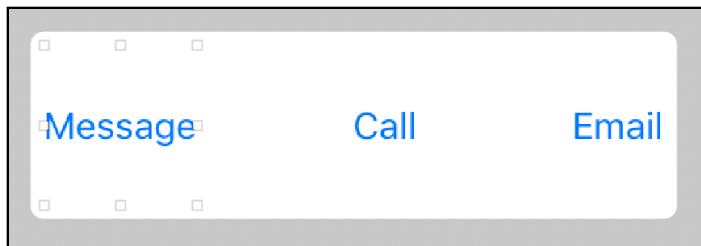
In a horizontal stack view, if no external constraints are setting the height, the view with the greatest height determines the stack view's height.



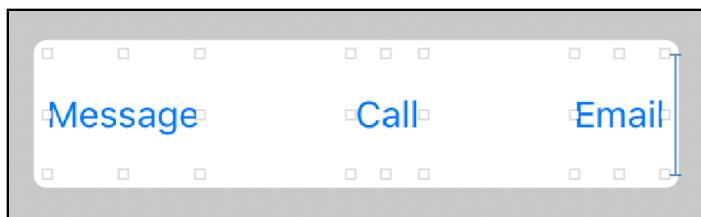
With the buttons embedded into the stack view, the stack view looks like this:



In this case, the intrinsic size of the buttons set the height of the stack view. For more on intrinsic size, see Chapter 8, “Content Hugging and Compression Resistance Priorities.” You can change the intrinsic height of the message button by adding two additional lines to its title. Doing so causes the stack view looks like this:



Without the additional line spacings, but with a height constraint of 50 on the email button, the stack view looks like this:

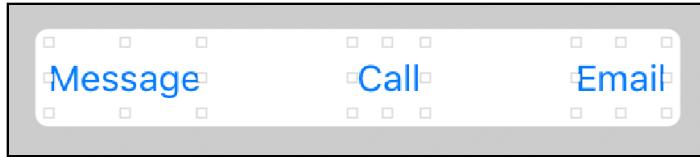


In each case, the height of the tallest view determines the height of the stack view; and because the alignment is **fill**, each other view in the stack view matches that height.

### Horizontal axis top, bottom and center alignments

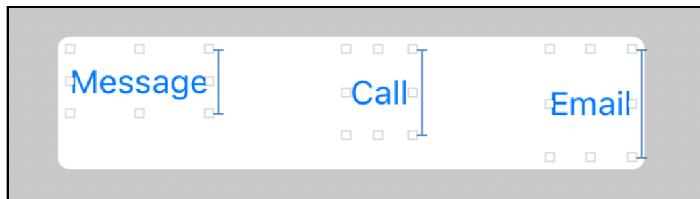
The top and bottom alignment properties arrange the subviews toward the top and bottom edge of the stack view, respectively. The center alignment property arranges the subviews directly in the middle of the stack view’s axis.

If each of the subviews is the same height, a stack view with top, bottom or center alignments looks the same as a stack view with fill alignment:

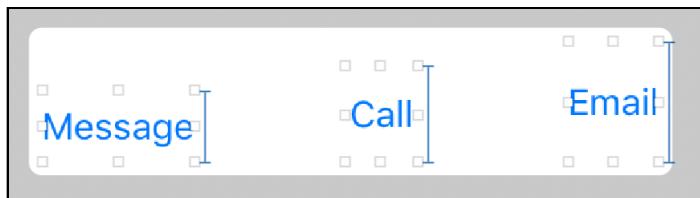


However, when the buttons are different heights, you can see the difference each of these settings makes.

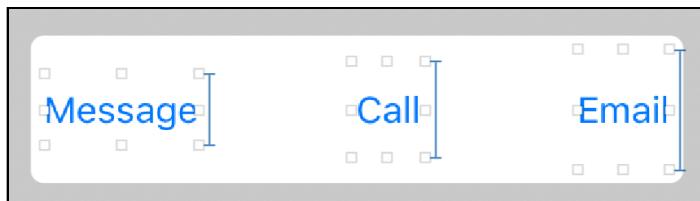
Here, the stack view alignment is set to top:



With the alignment set to bottom, the stack view subviews are lined up at the bottom:



With the alignment set to center, each subview is centered vertically in the stack view:

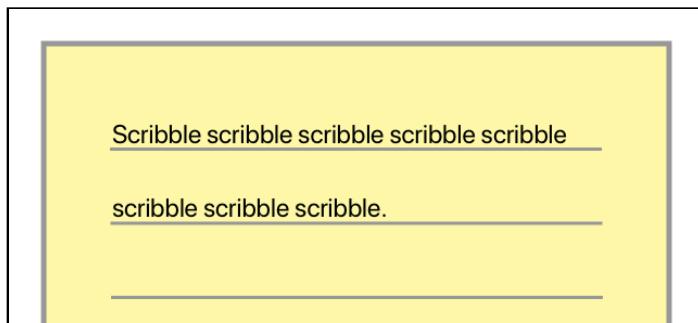


The last two options are the baseline alignments.

## Horizontal axis first baseline and last baseline alignments

Imagine a yellow notepad. When you write on the notepad, you have lines on which to write your letters. These lines help you write in a straight line and mitigate the chances of letters moving in all kinds of directions.

Here's an example to help visualize a baseline:



With UI objects such as a label, you can have multiple lines of text. When you have two labels in a horizontal stack view, you can decide whether you want the labels to align their first or last line of texts.

The following is an example of a stack view with two labels. The left-hand side label is three lines of text. The right-hand side label is one line of text.

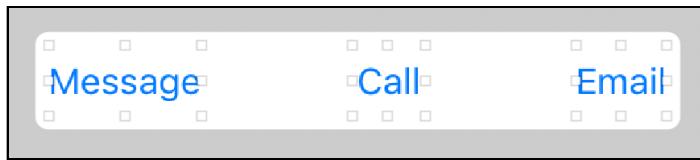
Here, the stack view alignment is set to first baseline:



Here, the stack view alignment is set to last baseline:

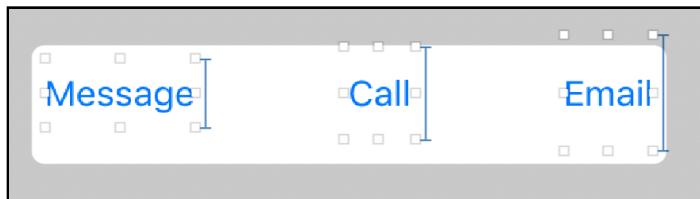


Here's the stack view with the action buttons and its alignment set to first baseline:

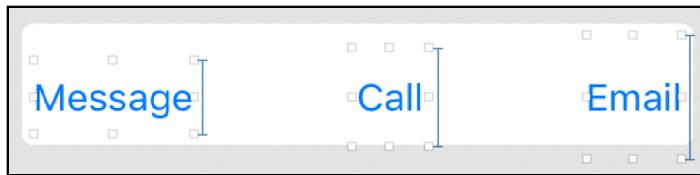


The stack view looks the same as the fill alignment configuration.

To see how the baseline alignment works, give the buttons different heights. This is how the stack view subviews look with the alignment set to the first baseline:



With the alignment set to the last baseline, the stack view subviews arrange like so:



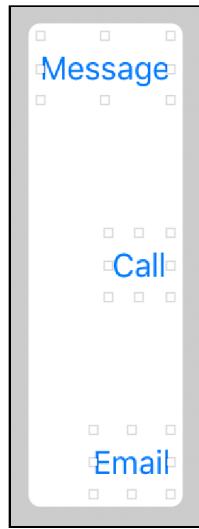
## Vertical axis alignments

The alignment options for a vertical stack view are: leading, trailing, fill and center.

If you reconfigure the stack view with the action buttons as a vertical stack view, leading alignment looks like this:



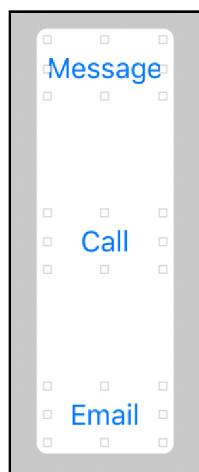
Trailing alignment, like this:



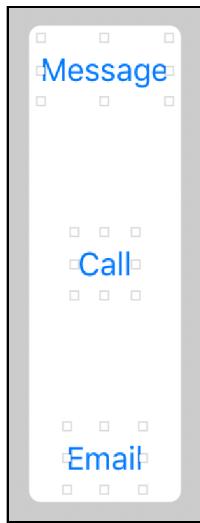
The vertical axis alignments do not have the first baseline and last baseline options. The assumption, here, is that this is partly because writings aren't commonly written with words read in 90 degrees from top to bottom. Words are generally written and read on the horizontal axis forward or backward, commonly dependent on the language in use.

The remaining vertical axis alignments are fill and center.

Fill looks like this:



Center looks like this:



Those are the four vertical axis alignment properties. Next up: the distribution property.

## Distribution property

You've seen that alignment controls the views' layout perpendicular to the axis. Distribution controls the views' layout parallel to the axis.

There are five distribution properties in both the horizontal and vertical axes. They are as follows:

- **Fill:** Size the subviews to fill up all of the space along the axis, but one view may size differently than the others.
- **Fill Equally:** Size the subviews to fill up all of the space along the axis, but make each subview have the same size.
- **Fill Proportionally:** Size the subviews to fill up all of the space along the axis, but resize each subview proportional to its intrinsic size.
- **Equal Spacing:** Size the subviews according to their intrinsic size or other constraints, but position them so that the space between each of them is equal and at least the stack view's **Spacing** property.

- **Equal Centering:** Size the subviews according to their intrinsic size or other constraints, but position them so that the distance between each of the centers is equal.

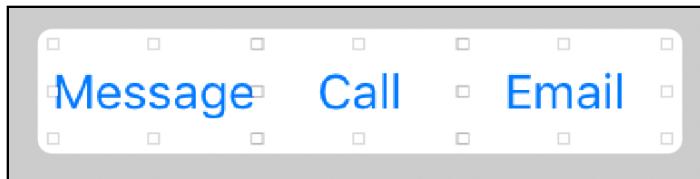
Which distribution do you think is fitting for having three buttons of equal size?

You're correct if you picked the fill equally distribution. The fill equally distribution will help you achieve buttons of equal sizes within a stack view.

In **Profile.storyboard**, select the horizontal stack view.

Set its distribution to **Fill Equally** in the **Attributes inspector**, and set the spacing to **0**.

Your horizontal stack view will now look like this:



## Nesting stack views

Just when you think stack views couldn't get more powerful — BAM! — nested stack views. Yes, you read that right. It's possible to have a stack view of stack views.

Embed the **vertical stack view** and the **horizontal stack view** into a vertical stack view.

Set the **spacing** of the container stack view to **16**. Set the alignment to **Fill**.

Next, add the following constraints to the stack view:

- 24 points spacing between stack view's top edge and Safe Area's top edge.
- 20 points spacing between the stack view's leading edge and superview's leading edge.
- Align horizontally in container's center.

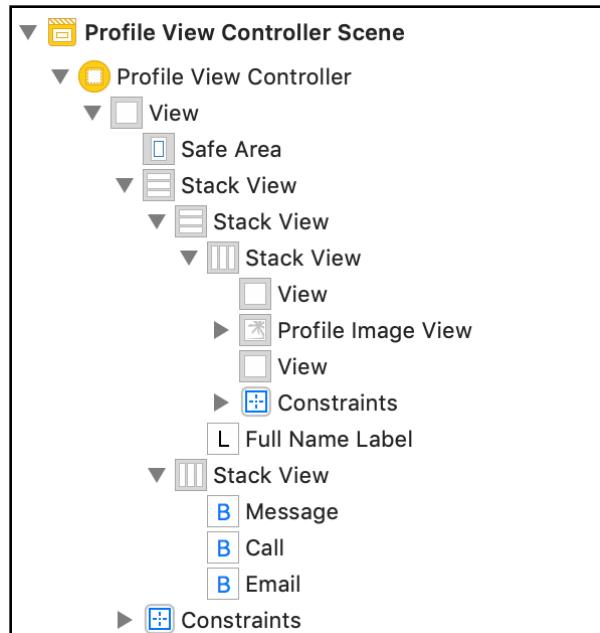
At the moment, the label's text attributes determine the image view's width. But what if you don't want the image view to expand or shrink depending on other views? What if you want the image view to have a fixed width without affecting the label's ability to expand or shrink according to its text attributes?

You can achieve this type of arrangement without adding many additional constraints using spacer views.

Create a stack view with left spacer view, the profile image view and right spacer view arrangements using the following steps:

1. Embed the **Profile Image View** into a stack view.
2. Set the stack view's axis to **horizontal**.
3. Drag two **UIView** objects into the stack view.
4. Position a **UIView** on the left, the **profile image view** in the middle, and a **UIView** on the right in the stack view.
5. Add an equal width constraint between the left spacer view and the right spacer view.
6. Set both the left and right spacer views' **background color** to **clear**.
7. Change the profile image view's width constraint priority to **1000**.

In the document outline, your Profile View Controller scene will look something like this:



## When not to use stack views

When you layout the UI, the first tool that comes to mind should be the stack view. A stack view reduces the number of constraints, makes adding and removing views trivial, supports intuitive animations and more.

That said, there are times when a stack view may not be your best option. For example, when your views need to behave differently than a stack view's default behaviors. In that case, you may want to work with manual constraints instead.

Imagine you have a stack view with a left view and a right view. The stack view has a fixed height, and its subviews have equal width. When you animate the right view's `isHidden` property from `false` to `true`, the stack view animates your views a certain way. The left view will take up the space of the right view as the left view expands, and the right view shrinks. If the animation behavior is not what you want, then you may find manual constraints to be a more suitable solution for the UI you want to achieve.

In addition to the stack view's behavior expectations, you may be working on a legacy codebase. The legacy codebase may have a pre-iOS 9.0 deployment target. This means that if you use stack views, you would need to support both pre-iOS 9.0 and iOS 9.0 and newer. This is a maintenance consideration that may affect crucial business decisions. This is another example of when stack views can be a less viable solution in comparison to creating additional constraints.

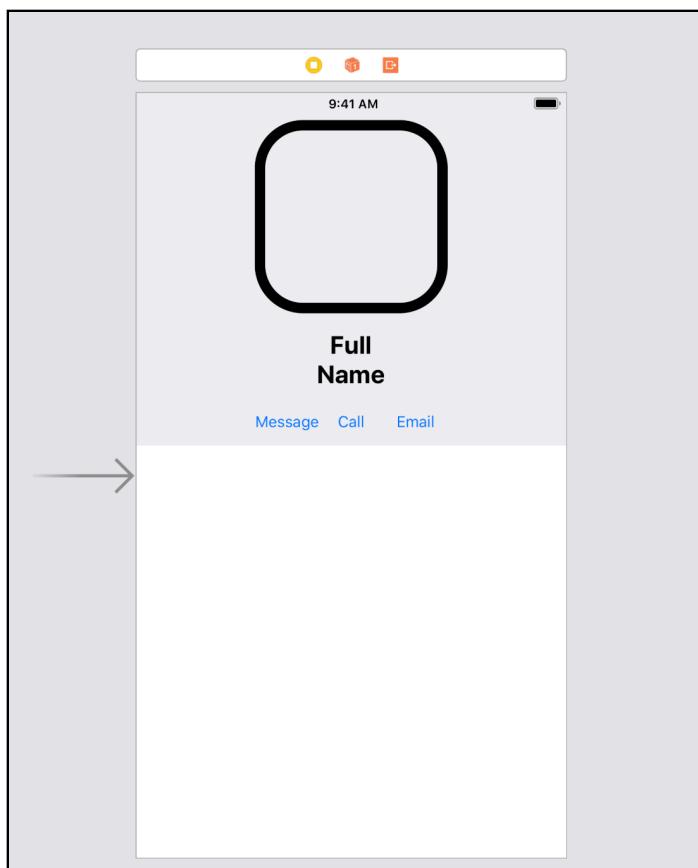
Overall, the stack view is a great tool that provides flexibility and adaptability to your user interface.

## Challenges

The profile view controller could use a facelift. Add a background view with the following specifications:

1. Embed the **container stack view** into a background view.
2. Align the **container stack view** bottom edge equal to the background view's bottom edge with 8 points spacing.
3. Align the **container stack view** top edge equal to the background view's superview's top edge with 26 points spacing.

4. Set the **container stack view leading edge greater than or equal** to the superview's leading edge with 20 points spacing.
5. Set the background view's background color to **Group Table View Background Color**.
6. Set the background view's **leading, top and trailing edges equal** to the superview.
7. Add a constraint that aligns the background view's **bottom edge greater than or equal** to the superview's Safe Area bottom edge with a zero constant.



## Key points

- The stack view is a smart container that positions and sizes the views contained within itself.
- Stack views empower developers to create adaptive layouts using fewer Auto Layout constraints.
- Use stack view properties to modify the positions and sizes of the views within itself.
- You can nest stack views within another stack view.
- When creating layouts, a stack view should be your go-to tool. However, there are occasions where you may not want to use stack views.

# Section II: Intermediate Auto Layout

Build on what you learned in Section I to begin using Auto Layout in more complex ways. Specifically, you'll cover:

- **Chapter 4: Construct Auto Layout with Code:** Learn how to build your user interface without using Interface Builder. Explore the Visual Format Language (VFL), which you can use to describe a set of constraints. Learn to refactor UIs built with Interface Builder into code.
- **Chapter 5: Scroll View:** Use scroll views to create user interfaces that go beyond the size of a physical screen. Learn about the special challenges they present when using Auto Layout and discover how to configure them.
- **Chapter 6: Self-Sizing Views:** Learn how to configure views that change size to account for dynamic content. See how to use Auto Layout to configure dynamically-sized cells in table views and collection views.
- **Chapter 7: Layout Guides:** Explore using Layout Guides to create space in your layout without using empty views. Learn about the system-provided guides and how to create your own custom guides.
- **Chapter 8: Content-Hugging and Compression-Resistance Priorities:** Discover what happens when the Auto Layout engine must choose between conflicting constraints. Learn how to use priorities to communicate how the system should resolve ambiguities to create the layout you want.
- **Chapter 9: Animating Auto Layout Constraints:** Learn to animate constraints to create unique and engaging user experiences in your apps. See how animation can provide feedback, focus user attention and improve navigation.
- **Chapter 10: Adaptive Layout:** See Auto Layout's real power come to life as you learn how to build adaptive user interfaces that adjust to screen size and orientation. Learn about traits such as layout direction, dynamic type size and size classes. Discover how trait collections allow you to build an adaptive layout without writing device-specific code.

- **Chapter 11: Dynamic Type:** Learn to make your app more accessible by supporting Dynamic Type. Understand how Auto Layout and Dynamic Type interact so you can manage layout changes in your app based on user type size preferences.
- **Chapter 12: Internationalization and Localization:** Learn how Auto Layout can assist you in internationalizing your app. See how to test if your app is ready for localization. Discover the things you must consider when creating your constraints to allow your app to handle other languages seamlessly.
- **Chapter 13: Common Auto Layout Issues:** Learn how to investigate when Auto Layout doesn't give you the desired result. See how to read Auto Layout's log messages and how to use other Xcode tools, such as symbolic breakpoints and the view debugger, to resolve Auto Layout conflicts.

# Chapter 4: Construct Auto Layout with Code

By Jayven Nhan

There are two ways to implement Auto Layout into your projects. You already learned how to implement Auto Layout using Interface Builder. Now, it's time to learn the second approach: using code.

Almost every iOS developer eventually raises the question: Should you construct your UI using storyboards or code? There is no silver bullet, and no one-size-fits-all solutions; however, there are solutions that fit much better based on your specific needs and requirements.

To help you achieve fluency in constructing UIs using code, you'll learn about the following topics in this chapter:

- Launching a storyboard view controller using code.
- Launching a non-storyboard view controller using code.
- Refactoring Interface Builder UIs into code.
- Using visual format language to construct Auto Layout.
- Benefits and drawbacks of constructing Auto Layout using code.



As a developer, you'll see projects implement their UIs using Interface Builder, code, and in some cases, both approaches within the same project. To build optimized solutions for new projects, and to help maintain existing projects, it's vitally important to understand both methods of building an app's UI.

By the end of this chapter, you'll know how to use code interchangeably with Interface Builder. You'll also gain the knowledge to make more decisive presentation logic decisions to achieve more optimal solutions.

## Launching a view controller from the storyboard in code

Open **MessagingApp.xcodeproj** in the **starter** folder, and then open the project target's general settings. Set the **Main Interface** text field to empty.



Main Interface

Build and run, and you'll see a black screen.

With the Interface Builder implementation, the app launches the initial view controller of the storyboard set in the target's Main Interface. To do something similar in code, you need to take a different approach.

Open **AppDelegate.swift** and replace the code inside `application(_:didFinishLaunchingWithOptions:)` with the following:

```
// 1
let storyboard = UIStoryboard(name: "TabBar", bundle: nil)
// 2
let viewController =
    storyboard.instantiateInitialViewController()
// 3
window = UIWindow(frame: UIScreen.main.bounds)
// 4
window?.rootViewController = viewController
// 5
window?.makeKeyAndVisible()
return true
```

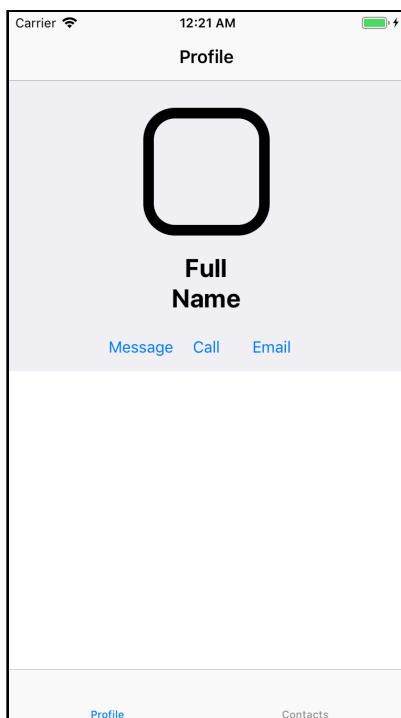
Here's what you've done:

1. Initialize the storyboard in code using the storyboard name.
2. Create a reference to the storyboard's initial view controller.

3. Set the app delegate's window using the device's screen size as the frame.
4. Set the window's root view controller to the storyboard's initial view controller.
5. By calling `makeKeyAndVisible()` on your window, window is shown and positioned in front of every window in your app. For the most part, you'll only need to work with one window. There are instances where you'd want to create new windows to display your app's content. For example, you'll work with multiple windows when you want to support an external display in your app. Chapter 17, "Auto Layout for External Displays", covers supporting external displays.

When you use storyboards, the app delegate's window property is automatically configured. In contrast, when you use code, you need to do more manual work. This is generally true when using code over Interface Builder.

Build and run, and you'll see the following:



That's only a taste of what it's like using more code and less Interface Builder. Are you ready for some more?

## Launching a view controller without initializing storyboard

You now know how to launch a view controller in code from a storyboard. But no set rule dictates that a project can't mix storyboards/.xibs and code. For example, there may come a time where your team's objective is to refactor an existing codebase that uses storyboards/.xibs into one that uses code. You're going to do that now.

First, **delete** the following Interface Builder files:

- **Profile.storyboard**
- **TabBar.storyboard**

Then, remove all of the code inside **ProfileViewController**'s body except for **viewDidLoad()**.

When you're done, **ProfileViewController.swift** will look like this:

```
import UIKit

final class ProfileViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

Up next, you'll create your UI properties and layout in code rather than using Interface Builder.

Open **AppDelegate.swift**.

**Tip:** Press **Command-Shift-O** to open quickly. Type **AppDelegate**. Xcode will suggest **AppDelegate.swift**. Press **Return** to open the file.

Replace the existing code inside `application(_:didFinishLaunchingWithOptions:)` with the following:

```
let viewController = TabBarController()  
window = UIWindow(frame: UIScreen.main.bounds)  
window?.rootViewController = viewController  
window?.makeKeyAndVisible()  
return true
```

Here, you command the app to initialize `TabBarController` using its initializer method.

Build and run, and you'll see a tab bar controller with a black background.



When you add a view controller onto a storyboard, the background view is set to white by default. In code, the view controller's view has a `nil` background color, which shows up as black. Once again, this is one of the automated steps as a result of using Interface Builder.

One of the benefits of initializing your view controller with its initializer method is that the view controller's type is explicit. Whereas when you initialize a view controller from a storyboard's initial view controller, you need additional code to ensure that the view controller returned is `TabBarController`.

Next, you'll rebuild the profile view controller user interface in code.

## Building out a view controller's user interface in code

When you built the UI in Interface Builder, the profile view controller had a header view with a gray background. The header view encapsulated the main stack view, and the main stack view encapsulated two stack views. One stack view contained the profile image view and the full name label. The other stack view contained the action buttons. It's time to recreate those user interfaces in code.

Create a new Swift file named `ProfileHeaderView.swift` inside the **User Profile/Views** group.

Replace the existing template code with the following:

```
import UIKit

final class ProfileHeaderView: UIView {
    override init(frame: CGRect) {
        super.init(frame: frame)
        backgroundColor = .groupTableViewBackground
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }
}
```

With the code above, you override `init(frame:)`. The method you've overridden is an initializer method. As the method name implies, this is where you add your initialization code, such as setting the view's background color.

Afterward, `init(coder:)` takes care of the object's archiving and unarchiving processes for Interface Builder. This method is handy when you launch an object from storyboard/.xib and want to configure the object at the initialization phase. Generally, when a view is created in code, `init(frame:)` is the initializer used, and when a view is created from a storyboard or .xib, `init(coder:)` is used instead.



## Layout anchors

Before the introduction of layout anchors, developers created constraints in code using `NSLayoutConstraint` initializers. `NSLayoutConstraint` describes the relationship between two user interface objects, and that relationship has to satisfy the Auto Layout engine. Although this approach still works, there's room for improvements in the code readability and cleanliness departments. Consequently, Apple introduced layout anchors for this purpose.

`NSLayoutAnchor` is a generic class built to create layout constraints using a fluent interface.

Are you ready for a comparison between constraints created using `NSLayoutConstraint` initializers and constraints created using layout anchors? Sure you are!

Say you'd like to create a square view centered vertically and horizontally inside of a view controller's view. To create this layout in code using `NSLayoutConstraint` initializers, it would look like this:

```
NSLayoutConstraint(  
    item: squareView,  
    attribute: .centerX,  
    relatedBy: .equal,  
    toItem: view,  
    attribute: .centerX,  
    multiplier: 1,  
    constant: 0).isActive = true  
  
NSLayoutConstraint(  
    item: squareView,  
    attribute: .centerY,  
    relatedBy: .equal,  
    toItem: view,  
    attribute: .centerY,  
    multiplier: 1,  
    constant: 0).isActive = true  
  
NSLayoutConstraint(  
    item: squareView,  
    attribute: .width,  
    relatedBy: .equal,  
    toItem: nil,  
    attribute: .notAnAttribute,  
    multiplier: 0,  
    constant: 100).isActive = true  
  
NSLayoutConstraint(  
    item: squareView,  
    attribute: .height,  
    relatedBy: .equal,  
    toItem: nil,  
    attribute: .notAnAttribute,  
    multiplier: 0,  
    constant: 100).isActive = true
```

```
item: squareView,  
attribute: .width,  
relatedBy: .equal,  
toItem: squareView,  
attribute: .height,  
multiplier: 1,  
constant: 0).isActive = true
```

Whereas, creating constraints in code using layout anchors would look like this:

```
squareView.centerXAnchor.constraint(  
    equalTo: view.centerXAnchor).isActive = true  
squareView.centerYAnchor.constraint(  
    equalTo: view.centerYAnchor).isActive = true  
squareView.widthAnchor.constraint(  
    equalToConstant: 100).isActive = true  
squareView.heightAnchor.constraint(  
    equalTo: squareView.heightAnchor).isActive = true
```

Between the two approaches, using layout anchors is comparatively cleaner, more succinct and more readable. The benefits of using layout constraints don't just stop at the fluent interface.

Another benefit you get for using layout anchors is type checking. Type checking mitigates the chance of creating invalid constraints in code. Type checking helps validate horizontal axis, vertical axis, height and width constraints.

If you try to compile the following code:

```
view.leadingAnchor.constraint(equalTo: squareView.topAnchor)
```

The compiler won't allow it because this code tries to create a constraint between an `NSLayoutXAxisAnchor` and an `NSLayoutYAxisAnchor`. The compiler recognizes that the constraint is invalid, and invalid constraints are reported as build-time errors thanks to layout anchor's type checking.

Using layout anchors won't prevent you from creating invalid constraints entirely. Despite the type checking efforts, layout anchors are still susceptible to invalid constraints. This is because the compiler allows you to create constraints between one view's leading and trailing layout anchor and another view's left/right layout anchor. This constraint is allowed because both anchors are x-axis layout anchors. Therefore, the constraint compiles fine in Xcode.

As it turns out, however, the Auto Layout engine restricts the relationship between trailing and leading layout anchors with left or right layout anchors. Such constraints will cause a runtime crash.



Now that you understand the essence of layout anchors and their benefits, it's a great time to put them to work in your project.

## Setting up profile header view

Open **ProfileViewController.swift** and add the following property to **ProfileViewController**:

```
private let profileHeaderView = ProfileHeaderView()
```

Next, add the following method to **ProfileViewController**:

```
private func setupProfileHeaderView() {
    // 1
    view.addSubview(profileHeaderView)
    // 2
    profileHeaderView.translatesAutoresizingMaskIntoConstraints =
        false
    // 3
    profileHeaderView.leadingAnchor.constraint(
        equalTo: view.leadingAnchor).isActive = true
    profileHeaderView.trailingAnchor.constraint(
        equalTo: view.trailingAnchor).isActive = true
    profileHeaderView.topAnchor.constraint(
        equalTo: view.safeAreaLayoutGuide.topAnchor).isActive = true
    profileHeaderView.bottomAnchor.constraint(
        lessThanOrEqualTo: view.safeAreaLayoutGuide.bottomAnchor)
        .isActive = true
}
```

With this code, you:

1. Add **profileHeaderView** as a subview of the view controller's view.
2. Set **profileHeaderView.translatesAutoresizingMaskIntoConstraints** to **false**. Before Auto Layout behaves as you'd expect from Interface Builder, this is a property that you need to remember to always set to **false**. It's set to **true** by default. Autoresizing mask is Auto Layout's predecessor. It's a layout system that's a lot less comprehensive when compared to Auto Layout.
3. Set and activate the profile header view's leading, trailing, top and bottom anchors.

You can now refactor the code you added earlier.

Replace the code in `setupProfileHeaderView()` with this:

```
view.addSubview(profileHeaderView)
profileHeaderView.translatesAutoresizingMaskIntoConstraints = false
NSLayoutConstraint.activate(
    [profileHeaderView.leadingAnchor.constraint(
        equalTo: view.leadingAnchor),
     profileHeaderView.trailingAnchor.constraint(
        equalTo: view.trailingAnchor),
     profileHeaderView.topAnchor.constraint(
        equalTo: view.safeAreaLayoutGuide.topAnchor),
     profileHeaderView.bottomAnchor.constraint(
        lessThanOrEqualTo: view.safeAreaLayoutGuide.bottomAnchor)])
```

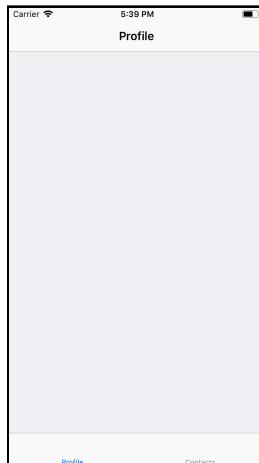
This code looks cleaner and is relatively more performant because of the Auto Layout engine's life cycle. Instead of bringing up the Auto Layout engine three additional times, you bring on the Auto Layout engine once and activate a list of constraints at one time versus individually. Although Interface Builder internal configurations aren't completely explicit, it's likely that this is also an automatic optimization implemented for developers who work with Interface Builder.

Now, add the following code to the end of `viewDidLoad()`:

```
view.backgroundColor = .white
setupProfileHeaderView()
```

This code sets the view's background color to white. It then calls the method you implemented to set up the profile header view.

Build and run, and you'll see the following screen:



## Refactoring profile image view

It's time to create the profile image view in code.

Open **ProfileImageView.swift** and remove **@IBDesignable** from the class declaration.

After that, remove everything inside **ProfileImageView**, and replace the body of **ProfileImageView** with the following:

```
// 1
enum BorderShape: String {
    case circle
    case squircle
    case none
}

let boldBorder: Bool

var hasBorder: Bool = false {
    didSet {
        guard hasBorder else { return layer.borderWidth = 0 }
        layer.borderWidth = boldBorder ? 10 : 2
    }
}

// 2
private let borderShape: BorderShape

// 3
init(borderShape: BorderShape, boldBorder: Bool = true) {
    self.borderShape = borderShape
    self.boldBorder = boldBorder
    super.init(frame: CGRect.zero)
    backgroundColor = .lightGray
}

// 4
convenience init() {
    self.init(borderShape: .none)
}

// 5
required init?(coder aDecoder: NSCoder) {
    self.borderShape = .none
    self.boldBorder = false
    super.init(coder: aDecoder)
}
```

Here's how this code works:

1. First, you declare a `BorderShape` enum. This contains an additional `none` case for when you want the image view to have no particular border shape. Notice that the access modifier for `BorderShape` is no longer `private`. Instead, it's `internal` since there is no other explicit access modifier declared on the property. This is so that `BorderShape` becomes accessible to other objects when they initialize `ProfileImageView`. `boldBorder` is used to determine `ProfileImageView`'s layer border width for when `hasBorder` is `true`. When `hasBorder` is `false`, the view's layer border width is simply set to zero.
2. You also have `borderShape`, which has changed from a string type to use the enum. One of the great benefits of building your UI in code is that you can make use of a lot more great Swift features. With Interface Builder, configuring a view's property using an enum isn't really an option.
3. Then, there's `init(borderShape:boldBorder:)`. This method initializes `ProfileImageView` by taking in the `borderShape` and `boldBorder` parameters and setting the class properties appropriately. Then, the method calls the superclass initializer method and passes in `.zero` for the frame. The frame size isn't a concern since the `ProfileImageView` will use Auto Layout to determine the view's size and position. Next, you set the background color to light gray.
4. Here, you have a convenience initializer which mitigates the need to pass in a `BorderShape` into the first initializer. This convenience initializer allows you initialize a `ProfileImageView` as `ProfileImageView()`. When you use this convenience initializer, `borderShape` is set to `none`.
5. Finally, you have `init(coder:)`, which is used when you create `ProfileImageView` in Interface Builder. In this case, you initialize `borderShape` to `none` and `boldBorder` to `false`.

You've added the value type, properties and initializer methods. It's time to configure the border.

Add the following code to `ProfileImageView`:

```
// 1
override func layoutSubviews() {
    super.layoutSubviews()
    setupBorderShape()
}

private func setupBorderShape() {
    hasBorder = borderShape != .none
```

```
// 2
let width = bounds.size.width
let divisor: CGFloat
switch borderShape {
case .circle:
    divisor = 2
case .squircle:
    divisor = 4
case .none:
    divisor = width
}
let cornerRadius = width / divisor
layer.cornerRadius = cornerRadius
}
```

Here's what you added:

1. `layoutSubviews()` is called when the constraint-based layout has finished its configuration. This is the time when `ProfileImageView` sets up the border shape of the view by calling `setupBorderShape()`.
2. Inside of `setupBorderShape()`, `borderShape` determines the corner radius. When `borderShape` is a `circle`, the layer's corner radius will be half of the view's width. When `borderShape` is a `squircle`, the layer's corner radius will be a quarter of the view's width. When `borderShape` is set to `none`, the layer's corner radius will be 1.

You're well on your way to becoming an Auto Layout code warrior. But there's still more to do.

## Refactoring profile name label

Open `ProfileNameLabel.swift` and remove `@IBDesignable` from the class declaration.

Next, replace everything inside of `ProfileNameLabel` with the following code:

```
// 1
override var text: String? {
    didSet {
        guard let words = text?
            .components(separatedBy: .whitespaces)
            .else { return }
        let joinedWords = words.joined(separator: "\n")
        guard text != joinedWords else { return }
        DispatchQueue.main.async { [weak self] in
            self?.text = joinedWords
    }
}
```

```
    }

// 2
init(fullName: String? = "Full Name") {
    super.init(frame: .zero)
    setTextAttributes()
    text = fullName
}

// 3
required init?(coder: NSCoder) {
    super.init(coder: coder)
}

// 4
private func setTextAttributes() {
    numberOfLines = 0
    textAlignment = .center
    font = UIFont.boldSystemFont(ofSize: 24)
}
```

With this code, you:

1. Override `text` to add a `didSet` observer. Every time the `text` value changes, `didSet` will get called. Safely unwrap using the guard statement to make sure that there's at least one word to extract from `text`. If `text` is `nil`, simply return and do nothing. If `text` is not `nil`, then add `\n` between every word. The `\n` separator creates a line spacing between each word. Afterward, `ProfileImageView`'s `text` is set to the new string joined by a separator that you create.
2. The initializer method takes parameters to set its properties and calls `init(frame:)` on the super class. Next, you call `setTextAttributes()` to set up some `UILabel` properties.
3. Implement the required initializer for when Interface Builder initializes `ProfileNameLabel`.
4. Finally, `setTextAttributes` sets the `numberOfLines`, `textAlignment` and `font` properties.

So, that's how you create the profile name label in code. You're ready to look at refactoring the stack views from Interface Builder.

## Refactoring stack views

To begin rebuilding the stack views in code, add the following extension to the bottom of **ProfileHeaderView.swift** (outside of the class):

```
private extension UIButton {
    static func createSystemButton(withTitle title: String)
        -> UIButton {
        let button = UIButton(type: .system)
        button.setTitle(title, for: .normal)
        return button
    }
}
```

This is a convenience method you can use to create a system button with the given title.

Next, add the following properties to **ProfileHeaderView**:

```
// 1
private let profileImageView =
    ProfileImageView(borderShape: .squircle)
private let leftSpacerView = UIView()
private let rightSpacerView = UIView()

private let fullNameLabel = ProfileNameLabel()

// 2
private let messageButton =
    UIButton.createSystemButton(withTitle: "Message")
private let callButton =
    UIButton.createSystemButton(withTitle: "Call")
private let emailButton =
    UIButton.createSystemButton(withTitle: "Email")

// 3
private lazy var profileImageStackView =
    UIStackView(arrangedSubviews:
        [leftSpacerView, profileImageView, rightSpacerView])

private lazy var profileStackView: UIStackView = {
    let stackView = UIStackView(arrangedSubviews:
        [profileImageStackView, fullNameLabel])
    stackView.distribution = .fill
    stackView.axis = .vertical
    stackView.spacing = 16
    return stackView
}()

private lazy var actionStackView: UIStackView = {
```

```
let stackView = UIStackView(arrangedSubviews:
    [messageButton, callButton, emailButton])
stackView.distribution = .fillEqually
return stackView
}()

private lazy var stackView: UIStackView = {
    let stackView = UIStackView(arrangedSubviews:
        [profileStackView, actionStackView])
    stackView.axis = .vertical
    stackView.spacing = 16
    return stackView
}()
```

With this code, you:

1. Initialize the profile image view, left spacer view, right spacer view and full name label.
2. Initialize the three action buttons using `createSystemButton(withTitle:)`, which you added earlier.
3. Initialize four stack views. The first stack view contains the profile image view and the spacer views. The second stack view encapsulates the profile image stack view and the full name label. The third stack view includes the action buttons. The fourth stack view stacks the second and third stack views together.

You're ready to set up the layout of the stack view. Add the following method to `ProfileHeaderView`:

```
private func setupStackView() {
    // 1
    addSubview(stackView)
    stackView.translatesAutoresizingMaskIntoConstraints = false

    // 2
    NSLayoutConstraint.activate(
        [stackView.centerXAnchor.constraint(equalTo: centerXAnchor),
         stackView.leadingAnchor.constraint(
             greaterThanOrEqualTo: leadingAnchor, constant: 20),
         stackView.leadingAnchor.constraint(
             lessThanOrEqualTo: leadingAnchor, constant: 500),
         stackView.bottomAnchor.constraint(
             equalTo: bottomAnchor, constant: -8),
         stackView.topAnchor.constraint(
             equalTo: topAnchor, constant: 26),

         profileImageView.widthAnchor.constraint(
             equalToConstant: 120),
         profileImageView.heightAnchor.constraint(
             equalToConstant: 120)])
```



```
        equalTo: profileImageView.heightAnchor),  
  
    leftSpacerView.widthAnchor.constraint(  
        equalTo: rightSpacerView.widthAnchor)  
])  
  
// 3  
profileImageView.setContentHuggingPriority(  
    UILayoutPriority(251),  
    for: NSLayoutConstraint.Axis.horizontal)  
profileImageView.setContentHuggingPriority(  
    UILayoutPriority(251),  
    for: NSLayoutConstraint.Axis.vertical)  
  
fullNameLabel.setContentHuggingPriority(  
    UILayoutPriority(251),  
    for: NSLayoutConstraint.Axis.horizontal)  
fullNameLabel.setContentHuggingPriority(  
    UILayoutPriority(251),  
    for: NSLayoutConstraint.Axis.vertical)  
fullNameLabel.setContentCompressionResistancePriority(  
    UILayoutPriority(751),  
    for: NSLayoutConstraint.Axis.vertical)  
  
messageButton.setContentCompressionResistancePriority(  
    UILayoutPriority(751),  
    for: NSLayoutConstraint.Axis.horizontal)  
}
```

Here, you:

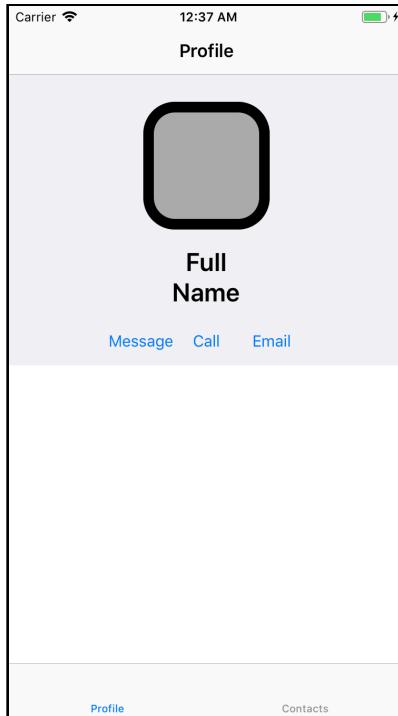
1. Add `stackView` as the view's subview. Then, set `translatesAutoresizingMaskIntoConstraints` to `false` so that your Auto Layout constraints can behave correctly without the interference of the autoresizing masks.
2. Set up the Auto Layout constraints on `stackView` and its subviews to match the layout behavior from the Interface Builder implementation.
3. Set the content hugging priority and compression resistance priority on `profileImageView`, `fullNameLabel` and `messageButton` to match the layout behavior from the Interface Builder implementation.

Finally, add the following code to `init(frame:)`:

```
setupStackView()
```

And voilà! You have yourself a `ProfileViewController` created entirely in code.

Build and run, and you'll see something like this:



There's more to learn about creating constraints in code.

## Auto Layout with visual format language

Using visual format language is another way of constructing your Auto Layout in code. Building Auto layout in code has come a long way in terms of code readability since the debut of visual format language. As you may have guessed, visual format language isn't exactly the most user-friendly tool available. So you may be wondering: Why should anyone learn visual format language to construct Auto Layout then?

Here are two reasons to get familiar with it:

1. Refactor or maintain legacy code which constructs Auto Layout constraints using visual format language.
2. Read and comprehend Auto Layout runtime errors.

The second reason is particularly significant for anyone who works with Auto Layout using Interface Builder or code. Because you'll see a lot of symbols with visual format language, it's a good idea to get familiar with them.

## Symbols

For reference, here are the symbols to describe your layout in visual format language:

- | superview
- – standard spacing (usually 8 points; value can be changed if it is the spacing to the edge of a superview)
- == equal widths
- -20- non-standard spacing (20 points)
- <= less than or equal to
- >= greater than or equal to
- @250 priority of the constraint; can have any value between 0 and 1000
- 250 - low priority
- 750 - high priority
- 1000 - required priority

## Visual format string example

```
H: |-[label(labelHeight)]-16-[imageView(>=250,<=300)]-16-
[button(88@250)]-|
```

Here's what the string above does to create constraints:

- H: indicates that the constraints are for the horizontal arrangement.
- |-[label creates a constraint between the superview's leading edge and the label's leading edge. label should be found in the views dictionary. More on views dictionary in the following sections.
- label(labelHeight) sets the label's height to labelHeight. This key should be found in the metrics dictionary. More on the metrics dictionary in the following sections.

- ]-16-[`imageView` sets a constraint with 16 spacings between `label`'s trailing edge and `imageView`'s leading edge.
- [`imageView(>=250,<=300)`] sets a width greater than or equal to 250 constraint and less than or equal to 300 constraint on `imageView`.
- ]-16-[`button` sets a 16 spacings constraint between `imageView`'s trailing edge and `button`'s leading edge.
- [`button(88@250)`] gives `button` a width constraint equal to 88 with a low priority. This allows the Auto Layout engine to break this constraint when needed.
- ]-| sets a constraint with standard spacing between `button`'s trailing edge and the superview's trailing edge.

## Thinking visual format language

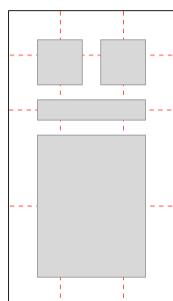
Despite having the word visual in visual format language, it isn't precisely the most visual-friendly. It's important to have the right strategy to think about constraints to effectively create or maintain constraints created using visual format language.

From reading visual format language string inputs, although not visual-friendly code, it can express a lot to the Auto Layout engine. In this section, you'll learn to express visual format language string inputs.

### Horizontal and vertical axes

When you think about visual format language, imagine either drawing a line from top-to-bottom or left-to-right on your device. Then, looking at the line you drew, track down the property name of every UI object the line passes through.

Look at the following diagram:



Count the number of red lines on the diagram. There are five. In your constraints code, you'll create five sets of constraints using visual format language: three horizontal arrangements and two vertical arrangements. Notice the connection? Five lines, five sets of constraints. This is generally true unless you prefer to split the constraints up or utilize the layout options parameter for certain scenarios when creating your constraints.

You use either an H: or V: to specify horizontal or vertical arrangements, respectively, in a visual format string.

Great, you know the sets of constraints to create when you see a layout. Now, you'll learn to inform the constraints about view position and spacing, and you'll learn how metrics and views in a visual format string come together to create constraints.

## Metrics dictionary

You can define metrics string with a dictionary using visual format language. You can then use a metrics key-value pair to define a constraint's constant or multiplier. You may want to use the metrics dictionary to pass in values for the visual format string to reference.

A metrics dictionary can look like this:

```
["topSpacing": topSpace,  
 "textFieldsSpacing": 8]
```

The dictionary you just saw consists of a key-value pair with a `topSpacing` key. The value corresponding to the key is a constant declared as `topSpace`. There's another key-value pair with a `textFieldsSpacing` key. It has a value of 8.

When the visual format string makes use of `topSpacing` or `textFieldsSpacing`, the value is referenced from the metrics dictionary.

## Views dictionary

So, how are you going to tell the constraints about the views using visual format language? This is where the views dictionary comes into play. Similar to the metrics dictionary, your key-value pairs consist of a string and the view object for the key and value, respectively.

For example, look at the following dictionary:

```
["textField": textField,  
 "imageView": imageView]
```

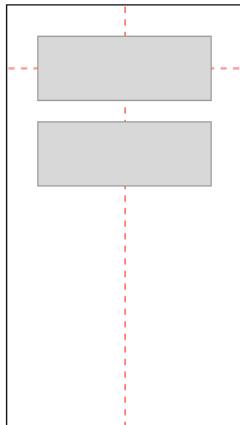
Your visual format string can use the `textField` string to reference the `textField` user interface object. You can also use the `imageView` string to reference the `imageView` user interface object.

The metrics and views dictionaries will make more sense when you build out your user interface using visual language format.

## Layout options

When creating your constraints using visual format language, you'll have the option to use the `options` parameter. `options` lets you describe the views' constraints perpendicular to the current layout orientation.

Look at the following diagram:



Imagine the top view has its leading, top, trailing and height constraints. The bottom view has its top and height constraints. The bottom view is missing the horizontal constraints arrangement. This is where layout options are handy.

You can use the following layout options:

```
[.alignAllLeading, .alignAllTrailing]
```

This helps you achieve the diagram's bottom view layout. Also, this tells the Auto Layout engine that you want all of your views in a visual format string to share the same leading and trailing constraints. The top view has its leading and trailing constraints. Thus, the bottom view simply infers these constraints and shares the top view's leading and trailing constraints.

That's enough theory for now; you're ready to get your hands dirty with visual format language code implementation.

## Setting up constraints

Open **NewContactViewController.swift**.

You can see that there's some existing code within `NewContactViewController`. The focus of this section is on the Auto Layout constraint construction for `profileImageView`, `firstNameTextField` and `lastNameTextField` using visual format language. You'll do this in `setupViewLayout()`, which is called from `viewSafeAreaInsetsDidChange()`.

When the root view's safe area insets change, iOS calls `viewSafeAreaInsetsDidChange()` to inform your app of the latest safe area insets. Upon calling the method, constraints are deactivated and emptied in preparation for handing the newest constraints.

Add the following code to `setupViewLayout()`:

```
// 1
let safeAreaInsets = view.safeAreaInsets

let marginSpacing: CGFloat = 16
let topSpace = safeAreaInsets.top + marginSpacing
let leadingSpace = safeAreaInsets.left + marginSpacing
let trailingSpace = safeAreaInsets.right + marginSpacing

// 2
var constraints: [NSLayoutConstraint] = []

// 3
view.addSubview(profileImageView)
profileImageView.translatesAutoresizingMaskIntoConstraints =
    false
view.addSubview(firstNameTextField)
firstNameTextField.translatesAutoresizingMaskIntoConstraints =
    false
view.addSubview(lastNameTextField)
lastNameTextField.translatesAutoresizingMaskIntoConstraints =
    false
```

With this code, you:

1. Define top, leading and trailing margin spacing constants that are used to create Auto Layout constraints.
2. Initialize an empty array of type `NSLayoutConstraint` collection.

3. Add the profile image view and text fields to the view hierarchy. You also disable the autosizing mask, which you need to do for views created in code that use Auto Layout.

Add the following code to the end of `setupViewLayout()`:

```
// 1
let profileImageViewVerticalConstraints =
    NSLayoutConstraint.constraints(withVisualFormat:
        "V:|-topSpacing-[profileImageView(profileImageViewHeight)]",
        options: [],
        metrics: [
            "topSpacing": topSpace, "profileImageViewHeight": 40],
        views: ["profileImageView": profileImageView])
constraints += profileImageViewVerticalConstraints

// 2
let textFieldsVerticalConstraints =
    NSLayoutConstraint.constraints(withVisualFormat:
        "V:|-topSpacing-[firstNameTextField(profileImageView)]-"
        "textFieldsSpacing-[lastNameTextField(firstNameTextField)]",
        options: [.alignAllCenterX],
        metrics: [
            "topSpacing": topSpace,
            "textFieldsSpacing": 8],
        views: [
            "firstNameTextField": firstNameTextField,
            "lastNameTextField": lastNameTextField,
            "profileImageView": profileImageView])
constraints += textFieldsVerticalConstraints

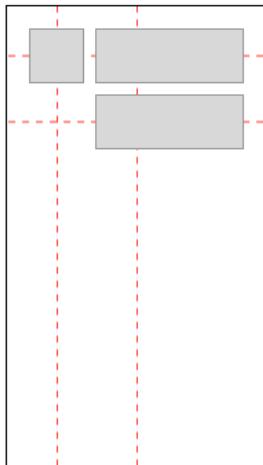
// 3
let profileImageViewToFirstNameTextFieldHorizontalConstraints =
    NSLayoutConstraint.constraints(withVisualFormat:
        "H:|-leadingSpace-[profileImageView(profileImageViewWidth)]-"
        "[firstNameTextField(>=200@1000)]-trailingSpace-|",
        options: [],
        metrics: [
            "leadingSpace": leadingSpace,
            "trailingSpace": trailingSpace,
            "profileImageViewWidth": 40],
        views: [
            "profileImageView": profileImageView,
            "firstNameTextField": firstNameTextField])
constraints +=
    profileImageViewToFirstNameTextFieldHorizontalConstraints

// 4
let lastNameTextFieldHorizontalConstraints =
    NSLayoutConstraint.constraints(
        withVisualFormat:
            "H:[lastNameTextField(firstNameTextField)]",
```

```
options: [],
metrics: nil,
views: [
    "firstNameTextField": firstNameTextField,
    "lastNameTextField": lastNameTextField])
constraints += lastNameTextFieldHorizontalConstraints

// 5
NSLayoutConstraint.activate(constraints)
self.constraints = constraints
```

In this example, the constraints look something like this on a diagram:

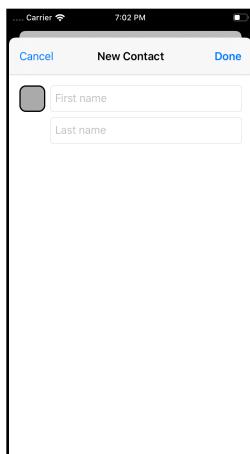


So, with this code, you:

1. You create a constraint with `topSpacing` spacings between the superview's top edge and `profileImageView`'s top edge. `profileImageView`'s height constraint is set to equal to `profileImageViewHeight`. Finally, you add the vertical constraints you created into the `constraints` array.
2. You create a constraint with `topSpacing` spacings between the superview's top edge and `firstNameTextField`'s top edge. `firstNameTextField`'s height is set to equal to `profileImageView`. Next, you create a constraint with `textFieldsSpacing` spacings between `firstNameTextField`'s bottom edge and `lastNameTextField`'s top edge. `lastNameTextField`'s height constraint is set to equal to `firstNameTextField`. Finally, you add the vertical constraints you created into the `constraints` array.

3. You create a constraint with `leadingSpace` spacings between the superview's leading edge and `profileImageView`'s leading edge. The `profileImageView`'s width constraint is set to equal to `profileImageViewWidth`. You add a constraint with standard spacing between `profileImageView`'s trailing edge and `firstNameTextField`'s leading edge. The `firstNameTextField`'s width constraint is set with a required constraint priority and a minimum width value of 200. Before wrapping up the visual format string, you set a spacing equal to `trailingSpace` between `firstNameTextField`'s trailing edge and the superview's trailing edge. Finally, you add the horizontal constraint you created into the `constraints` array.
4. The next set of constraints comes from the second horizontal line coming down from the diagram. You set up `lastNameTextField`'s width equal to `firstNameTextField`. It's unnecessary to set additional spacing thanks to the centering of the x-axis from step #2. Finally, you add the horizontal constraint you created into the `constraints` array
5. Activate all of the constraints inside of `constraints`. Then, set the recently created constraints to `NewContactViewController`'s `constraints`. This is so you can deactivate the constraints for when the safe area insets change. For example, when the device's orientation changes.

Build and run. Tap the **Contacts** tab. Tap the + tab bar button. You'll see:



There you have it: Your layout built in code using visual language format. However, it's usually more work for the same design. Plus, visual language format is not the most user-friendly. Having said that, understanding visual format language can still assist you in debugging constraint problems, maintaining legacy codebases, refactoring legacy codebases and more.

# Benefits and drawbacks from choosing the code approach

Whether you choose Interface Builder or code to layout your user interface, it is an unquestionably subjective matter. Before you come to a decisive conclusion on your approach, have a look at the benefits and drawbacks when using code to construct Auto Layout.

Here are five benefits of using code to construct Auto Layout constraints:

- Everything technical that storyboard can do, code can do too. But, not vice versa.
- Readable merge conflict(s) means less time spent fixing merge conflicts.
- Code compilation time reduction.
- All the user interface logic lives in code. This benefit mitigates events such as having to find whether a property is changed in Interface Builder or code.
- Easy UI maintenance with the right coding infrastructure. Manage UI constants such as fonts, colors and constraint values with ease.

Here are five drawbacks of using code:

- Higher learning curve compared to using Interface Builder.
- Naming conventions and code cleanliness are vital for code maintenance when working on a team.
- Inability to add user interface objects and create constraints for the user interface objects visually like in Interface Builder.
- More developers are familiar with using Interface Builder than developers who are familiar with using code to build an app's layout. If you work with developers who are less familiar with using code, more time may be needed to properly onboard the developers.

- Opportunity cost of missing out some of the automation Interface Builder provides when you build your layouts in Interface Builder.

Because there's no one-size-fits-all solution, one of the more important steps to coming to the optimal solution is clearly defining the problem. What exactly are you or your team trying to solve?

There are benefits and drawbacks to using Interface Builder and code. You need to consider the tradeoffs before deciding which to use. Questions such as:

- Does it make sense for a project to sacrifice Interface Builder automation for ease of source control when merge conflicts arise?
- Is it preferable to keep Interface Builder visualizations in the sacrifice of reduced compile time?

Personalization is a big consideration. For example:

- Which of the tradeoffs apply to the scenario at hand?
- What about personal preferences? Everyone has their personal preferences. Some people enjoy using Interface Builder over code, and others do not.

When choosing an approach to build your app's UI, take time to consider the problems, tradeoffs and personalizations fully. The optimal solution will arise by building a solution tailored to your specific team and project requirements.

Ultimately, it's up to you and your team to decide.

## Challenges

You've reached the end of the chapter. To help solidify your understanding, try these challenges:

- Recreate the `ContactListTableViewController`'s UI entirely in code.
- Recreate the `ContactTableViewCell`'s UI entirely in code.
- Recreate the `ContactPreviewView`'s UI entirely in code.

## Key points

- Working with code requires more upfront manual work than working with Interface Builder.
- You can refactor UI layouts built in Interface Builder into code format.
- There are various methods to create Auto Layout constraints using code.
- Learning visual format language, although rarely seen on new projects, can assist you in debugging constraint conflicts and maintaining legacy codebases.
- Consider the pros and cons when choosing between Interface Builder and code approach to creating your UI layout.

# Chapter 5: ScrollView

By Libranner Santos

By now, you understand the power of stack views. But what options do you have when you need to create user interfaces that go beyond the screen size? Scroll views.

Scroll views allow you to expand your interfaces beyond the limits of the screen. As written in the official Apple documentation at <https://apple.co/2WoN2Be>:

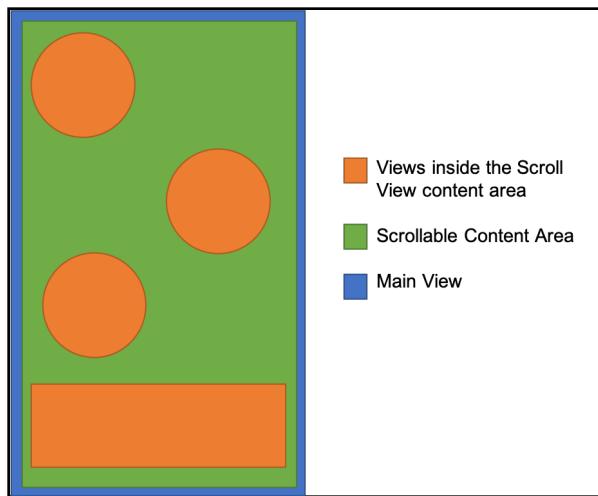
*UIScrollView: A view that allows the scrolling and zooming of its contained views.*

A scroll view acts as a parent view and can contain as many views as your interface needs; they are a key component for any app that requires more space. For example, you might need more space to display parts of a form or when dealing with a significant amount of text. Or maybe your app includes a large image that users need to zoom into and some way to navigate the UI that mimics a carousel. These are all great reasons to use a scroll view, and in this chapter, you'll learn how to use them in your app.



## Working with scroll view and Auto Layout

Scroll views are different than other views when it comes to Auto Layout because you need to make two types of constraints: One that sets the x, y, width and height of the scroll view, and one that sets the x, y, width and height of the subviews in the content area. Generally, when you make constraints between the scroll view and views outside of its view hierarchy, you set the scroll view's frame. But, when you make constraints between the scroll view and views inside of its view hierarchy, you set the frame of the subviews in the scrollable content area.

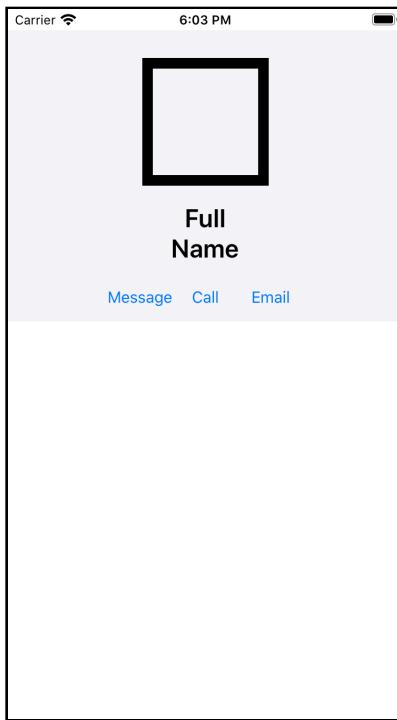


When working with scroll views, keep the following in mind:

- You need to define both the size and position of the scroll view's frame within its superview and the size of the scroll view's content area.
- To define the content area, it's best to add a content view that's anchored to the edges of the scroll view.
- If you don't want to have horizontal scrolling, make the content view's width equal to the scroll view's width. The same is true for vertical scrolling but using the height instead.
- When adding content, add constraints related to the content view. If you want to create a floating effect for a view inside of the scroll view, add constraints between the target view and objects outside of the scroll view.

# Adding the Options Menu to the Profile Screen

Go to the starter project, open the MessagingApp project, and build and run.



The app already contains part of the UI you need; however, the bottom part is missing. You'll need to add some buttons so that the user can see all of the options.

Go to **ProfileViewController.swift** and add the following code below `viewDidAppear(_ :)`:

```
private func setupMainStackView() {
    mainStackView.axis = .vertical
    mainStackView.distribution = .equalSpacing
    mainStackView.translatesAutoresizingMaskIntoConstraints = false

    view.addSubview(mainStackView)

    let contentLayoutGuide = view.safeAreaLayoutGuide
    NSLayoutConstraint.activate([
        mainStackView.topAnchor.constraint(equalTo: contentLayoutGuide.topAnchor),
        mainStackView.bottomAnchor.constraint(equalTo: contentLayoutGuide.bottomAnchor),
        mainStackView.leadingAnchor.constraint(equalTo: contentLayoutGuide.leadingAnchor),
        mainStackView.trailingAnchor.constraint(equalTo: contentLayoutGuide.trailingAnchor)
    ])
}
```

```
    mainStackView.leadingAnchor.constraint(equalTo:  
        contentLayoutGuide.leadingAnchor),  
    mainStackView.trailingAnchor.constraint(equalTo:  
        contentLayoutGuide.trailingAnchor),  
    mainStackView.topAnchor.constraint(equalTo:  
        contentLayoutGuide.topAnchor),  
    mainStackView.bottomAnchor.constraint(equalTo:  
        contentLayoutGuide.bottomAnchor),  
)  
  
    setupProfileHeaderView()  
    setupButtons()  
}
```

With this new method, you:

1. Set up `mainStackView`, indicating its alignment, axis and distribution. You also set its `translatesAutoresizingMaskIntoConstraints` to `false`. The `mainStackView` is already declared at the top of the class.
2. Add `mainStackView` to `view` so that you can add it to the view hierarchy.
3. Create the constant `contentLayoutGuide` to get a reference to the `view.safeAreaLayoutGuide`. You'll create constraints related to this layout guide instead of the view itself. In this case, you'll use the `safeAreaLayoutGuide`, which allows creating UIs respecting the margins on the devices, so things like the iPhone X notch do not interfere with the views.
4. Set the leading, trailing and top constraints for `mainStackView` so that you can properly position it on the screen. Since the Stack View will grow vertically, it's not necessary to indicate the bottom constraint.

Go to `setupProfileHeaderView()` and replace its code with the following:

```
profileHeaderView.translatesAutoresizingMaskIntoConstraints =  
    false  
profileHeaderView.heightAnchor.constraint(  
    equalToConstant: 360).isActive = true  
mainStackView.addSubview(profileHeaderView)
```

This code:

1. Removes all of the previous constraints and adds one constraint for the height.
2. Adds `profileHeaderView` to `mainStackView`.

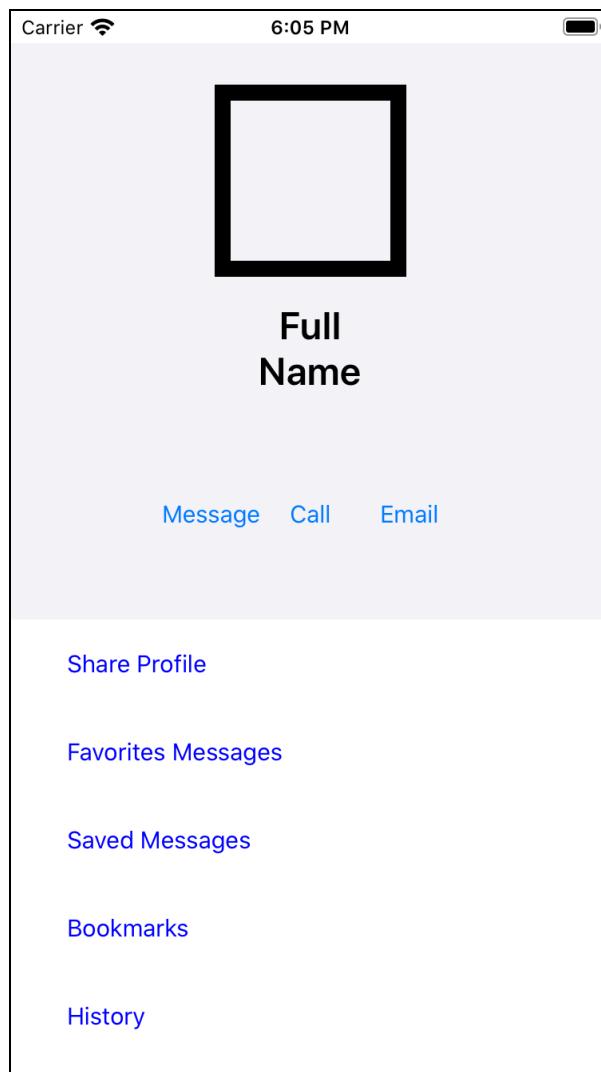
Because the view is now inside of a stack view, you don't have to add the same constraints you added before; the alignment and distribution properties will cause the contained views to resize accordingly. That's part of the magic behind working with stack views.



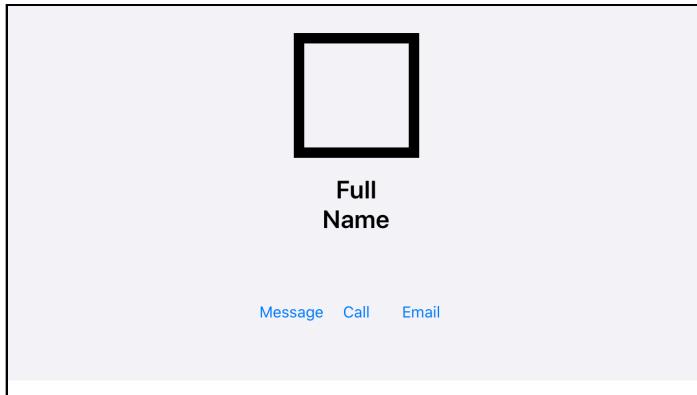
Go to `viewDidLoad()` and replace the call to `setupProfileHeaderView()` with the following:

```
setupMainStackView()
```

Build and run.



The app now displays *some* options in the bottom section; however, there are additional options you can't see. More importantly, if you rotate the simulator using **Command-Right-Arrow**, you'll see even fewer options or not at all depending on the device:



Stop the project, and in **ProfileViewController.swift**, go to `setupButtons()`. Look for the following block of code:

```
func setupButtons() {
    let buttonTitles = [
        "Share Profile", "Favorites Messages", "Saved Messages",
        "Bookmarks", "History", "Notifications", "Find Friends",
        "Security", "Help", "Logout"]

    let buttonStack = UIStackView()
    buttonStack.translatesAutoresizingMaskIntoConstraints = false
    buttonStack.alignment = .fill
    buttonStack.axis = .vertical
    buttonStack.distribution = .equalSpacing

    buttonTitles.forEach { (buttonTitle) in
        buttonStack.addArrangedSubview(
            createButton(text: buttonTitle))
    }

    mainStackView.addArrangedSubview(buttonStack)
    NSLayoutConstraint.activate([
        buttonStack.widthAnchor.constraint(equalTo:
            mainStackView.widthAnchor),
        buttonStack.centerXAnchor.constraint(equalTo:
            mainStackView.centerXAnchor)
    ])
}
```

This code takes an array named `buttonTitles` and creates a button for each title. In this case, there are ten titles, but the app doesn't show all of them because the size of the stack view after — adding all the buttons — is greater than the space available on the screen. To fix this problem, you'll use a scroll view.

## Setting up the scroll view

Go to the top of `ProfileViewController` and add a new property after the `mainStackView` declaration. This new property will contain the Scroll View:

```
private let scrollView = UIScrollView()
```

Next, add this method below `setupMainStackView()`:

```
private func setupScrollView() {
    //1
    scrollView.translatesAutoresizingMaskIntoConstraints = false
    view.addSubview(scrollView)

    //2
    NSLayoutConstraint.activate([
        scrollView.leadingAnchor.constraint(equalTo:
            view.leadingAnchor),
        scrollView.trailingAnchor.constraint(equalTo:
            view.trailingAnchor),
        scrollView.topAnchor.constraint(equalTo:
            view.safeAreaLayoutGuide.topAnchor),
        scrollView.bottomAnchor.constraint(equalTo:
            view.safeAreaLayoutGuide.bottomAnchor)
    ])
}
```

With this new method, you:

1. Set `translatesAutoresizingMaskIntoConstraints` to `false` so that `scrollView` won't translate the autoresizing mask into constraints. Because you're creating all of the constraints using Auto Layout, this translation is not necessary.
2. Create leading, trailing, top and bottom constraints for `scrollView` anchored to its superview. Since these are constraints to a view outside of the scroll view's hierarchy, they'll set the scroll view's frame.

You now need to put `mainStackView` inside of the newly created scroll view.

Replace `setupMainStackView()` with the following:

```
private func setupMainStackView() {
    mainStackView.axis = .vertical
    mainStackView.distribution = .equalSpacing
    mainStackView.translatesAutoresizingMaskIntoConstraints
        = false

    //1
    scrollView.addSubview(mainStackView)

    //2
    let contentLayoutGuide = scrollView.contentLayoutGuide

    NSLayoutConstraint.activate([
        //3
        mainStackView.widthAnchor.constraint(equalTo:
            view.widthAnchor),
        mainStackView.leadingAnchor.constraint(equalTo:
            contentLayoutGuide.leadingAnchor),
        mainStackView.trailingAnchor.constraint(equalTo:
            contentLayoutGuide.trailingAnchor),
        mainStackView.topAnchor.constraint(equalTo:
            contentLayoutGuide.topAnchor),
        //4
        mainStackView.bottomAnchor.constraint(equalTo:
            contentLayoutGuide.bottomAnchor)
    ])
}

//5
setupProfileHeaderView()
setupButtons()
}
```

This code looks similar to the previous implementation; however, there are some important modifications:

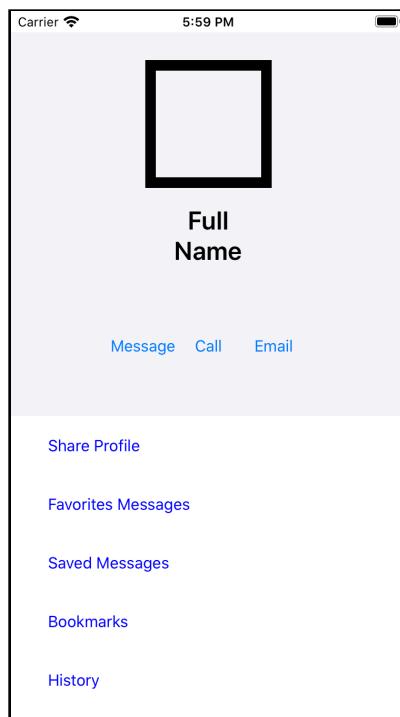
1. You made `mainStackView` a subview of `scrollView`.
2. The constant `contentLayoutGuide` now contains a reference to the scroll view's **Content Layout Guide**. This layout guide represents the content area of the `scrollView`. You could have used the `scrollView` itself to create the constraint, and it would look the same, but doing it as you did here brings more clarity to your code.

3. There's a new constraint for the width of the `mainStackView`. Notice that `widthAnchor` is created in relation to the view. That's because a scroll view takes on the size of its content, and not doing it this way will cause the scroll view to shrink. Also, by doing this, you're indicating that horizontal scrolling is disabled. All the other constraints are between `mainStackView` and the `contentLayoutGuide` reference created above.
4. There's another new constraint between the bottom of the `mainStackView` and the `contentLayoutGuide`. This is what makes the `scrollView` grow so that it can fit all of the views.

The only thing left to do is to call your set up methods. In `viewDidLoad()`, add the following immediately before the call to `setupMainStackView()`:

```
setupScrollView()
```

Build and run.



## Using the Frame Layout Guide

Before you move on, there's one more refactor you can do. Go to `setupScrollView()` and replace the implementation with this one:

```
private func setupScrollView() {
    scrollView.translatesAutoresizingMaskIntoConstraints = false
    view.addSubview(scrollView)

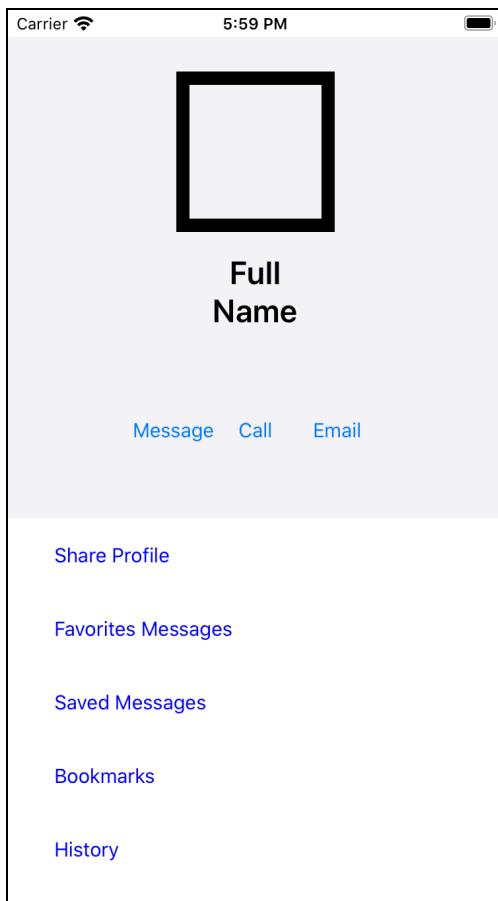
    //1
    let frameLayoutGuide = scrollView.frameLayoutGuide

    //2
    NSLayoutConstraint.activate([
        frameLayoutGuide.leadingAnchor.constraint(equalTo:
            view.leadingAnchor),
        frameLayoutGuide.trailingAnchor.constraint(equalTo:
            view.trailingAnchor),
        frameLayoutGuide.topAnchor.constraint(equalTo:
            view.safeAreaLayoutGuide.topAnchor),
        frameLayoutGuide.bottomAnchor.constraint(equalTo:
            view.safeAreaLayoutGuide.bottomAnchor)
    ])
}
```

Here's what you did:

1. Create the constant `frameLayoutGuide` to get a reference to the `scrollView.frameLayoutGuide`. You'll create constraints related to this layout guide instead of the scroll view itself. This layout guide refers to the frame of the scroll view, not the content area.
2. Create the leading, trailing, top and bottom constraints between the Frame Layout Guide of `scrollView` and the `view`. Notice how the leading and trailing constraints are created using the `view`, not the Safe Area Layout Guide; in this case, `scrollView` will occupy the entire width of the screen.

Build and run.

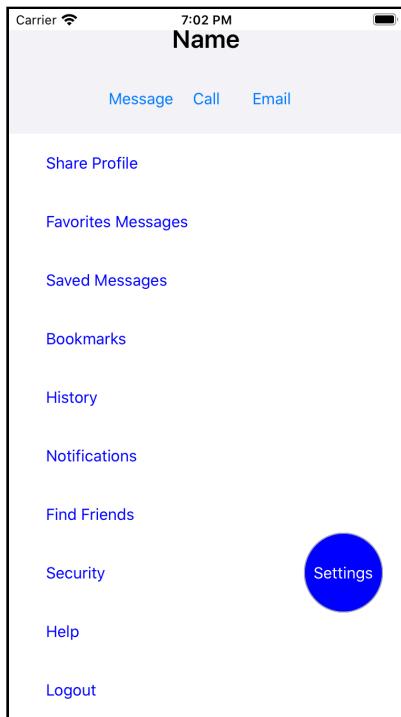


It looks like nothing changed, but with the use of the Frame Layout Guide, your code is now more precise and easier to understand. If you want to learn more about Layout Guide, read Chapter 7, “Layout Guide.”

That wasn't too bad, was it? With the new options menu you added, users will have a better experience with your app. Before wrapping things up, try running the app on different simulators so you can see how it works on any screen size.

## Challenge

For this challenge, you'll need to create the **Settings** button and add it to the scroll view. This button should appear near the bottom of the scroll view and close to the right side, but it should not be part of the stack view. For reference, here's how it should look:



## Key points

- Constraints between a scroll view and the views outside of its view hierarchy act on the scroll view's frame.
- Constraints between a scroll view and views inside of its view hierarchy act on the scroll view's content area.
- Be extra careful when setting the size and position of your scroll view. Remember that apart from setting its size and position, you'll have to specify a content area that will affect the way the scroll view behaves.

- It's strongly recommended to add a content view that acts as a container for all of the views inside of the scroll view. This makes it easier to work with a scroll view that can grow in size.
- While working with stack views (or any views) inside of a scroll view that acts as the content area, the width and height determine if the scroll view will have vertical and horizontal scrolling.
- Remember to use `frameLayoutGuide` and `contentLayoutGuide` when creating the constraints for scroll views.

# 6 Chapter 6: Self-Sizing Views

By Jayven Nhan and Libranner Santos

Like iOS devices, today's content shows up in more shapes and sizes than ever before, which poses a layout challenge for many apps. To account for dynamic content, developers need views with self-sizing capabilities. Whether you're reading your messages, browsing through your photo albums, or choosing your preferred font size, the content you receive can vary significantly in size.

An app that uses static-sized views for dynamic content can face many drawbacks, for example:

- Content truncation.
- Inefficiency in screen space utilization.
- Inability to support user interface preferences.

Drawbacks like the ones above can drive countless users away from your app due to a poor user experience. With self-sizing views, you can address these problems, but first, you need to understand how they work. Often we think of Auto Layout from the top down. A full-screen view controller dictates the size of its view. Then, if you start creating constraints to it, you build the Auto Layout system from the top down. But there's another option: You can create constraints on the child views and then from those views to their superviews. There's no real difference in the type of constraints, it's just a different way of thinking about them.



In this chapter, you'll learn about the following topics:

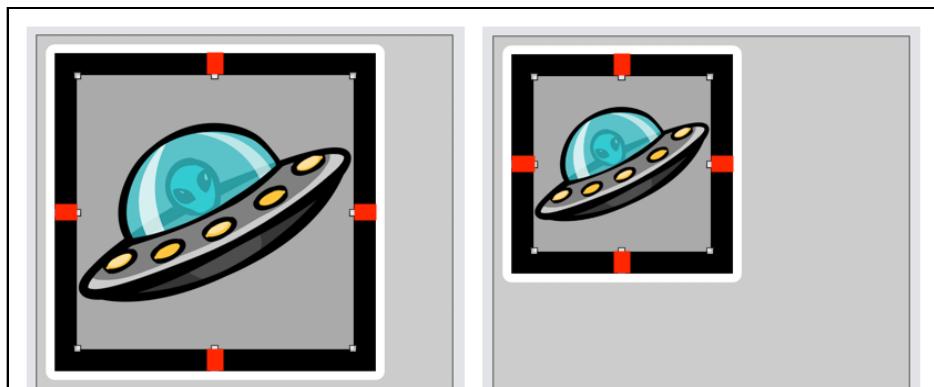
- Strategies to accomplish self-sizing views.
- Sizing views with the bottom-to-top in a view hierarchy approach.
- Dynamic sizing of table view cells.
- Sizing views with the top-to-bottom in a view hierarchy approach.
- Manually sizing collection view cells.

By the end of this chapter, you'll know how to prepare your app's user interface to consume and display virtually any content.

## Accomplishing self-sizing views

Usually, a self-sizing view has a position determined by outside constraints or its parent view. This type of setup leaves two metrics for the view to determine: width and height. In some cases, like with a table view cell, the width is also determined by the parent, leaving only the height. Essentially, a self-sizing view acts as a container view for the views within itself. As long as the container view can figure out its size definitively, it can self-size.

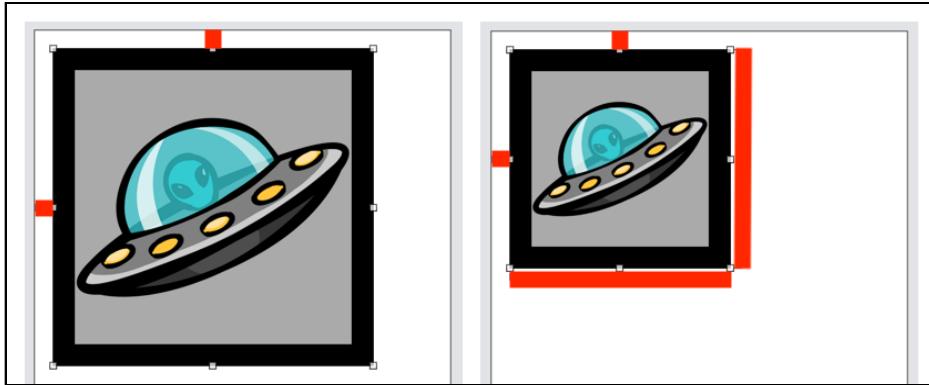
A view's size derives in one of two ways: bottom-to-top or top-to-bottom in a view hierarchy. A view either gets its size from the container view, or the view is the container view and gets its size from its child views. Look at the following diagram:



Both container views (black background) encapsulate an image view. Both image views contain the same image and have identical standard spacing Auto Layout constraints (indicated in red lines) around the edges.

However, the left container view is bigger than the right container view.

Look at the container view's constraints:



Both container views have leading and top edge constraints. However, only the right container view has a set of width and height constraints.

If a container view contains child views with intrinsic size, it'll grow and shrink to accommodate those views while taking Auto Layout constraints, such as the padding around the child views, into account. In this case, an inside-out or bottom-to-top approach to the view hierarchy gives the container view its size.

On the other hand, if a container view has a fixed width and/or height, then the child views will grow and shrink to accommodate the container view's size constraints. In this case, an outside-in or top-to-bottom approach to the view hierarchy gives the child views their size.

For example, think of a table view. If the table view sets a fixed row height, like 50 points, then the row size is going to be 50 points high, no matter what the size of its children. If there are less than 50 points of content, there will be extra whitespace in that row. And if another row has more than 50 points of content, it will be clipped or shrunk to fit within 50 points. But, if you don't set an explicit row height and let the cells self-size, the rows that have less content will be smaller, and the rows with more content will grow to show all the content.

You'll want to use the inside-out or bottom-to-top in the view hierarchy approach to give the container view its size. Let the container view derive its width and height from its children and Auto Layout. The child views must interconnect in a way with Auto Layout that pushes and pulls the container view outward or inward and grows or shrinks the container view. Consequently, the views within give shape to the container view.

What you've seen here with the child views giving shape to the container view is analogous to self-sizing `UITableViewCell` and `UICollectionViewCell`. In this chapter, you'll learn about the bottom-to-top approach to size a `UITableViewCell`. In contrast, you'll also learn about the top-to-bottom approach to size a `UICollectionViewCell`.

If self-sizing `UICollectionViewCell` using the bottom-to-top approach interests you, read Chapter 11, "Dynamic Type."

## Table views

In the previous chapter, you saw how you could use a scroll view to create an interface that goes beyond the physical size of the screen. But scroll views aren't the only views at your disposal in iOS. There's another convenient tool available for when you need to display lists of elements in an organized and user-friendly way. These views are known as table views.

A table view is a view that displays information using a single column. It's one of the most common objects you'll see in iOS because it's an intuitive and useful way for users to view and interact with data. Usually, when you need to display a list of data, table views are the first types of views to come to mind. They're relatively easy to implement, and they come with a set of methods and properties out-of-the-box, making them a great choice.

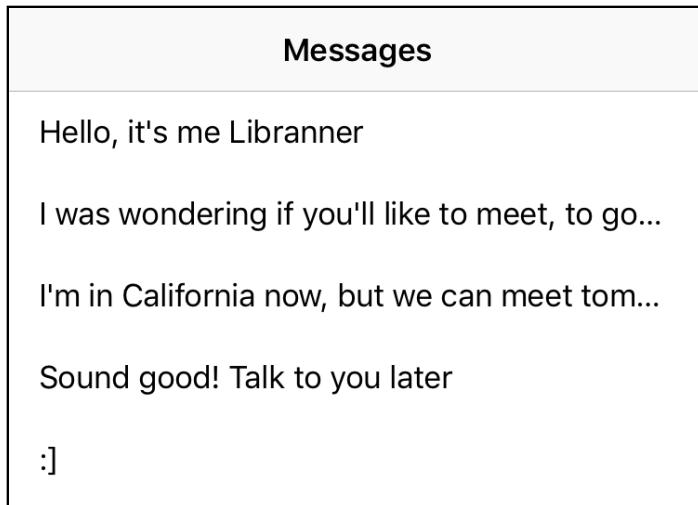
To use table views, you first need to know about two protocols: `UITableViewDelegate` and `UITableViewDataSource`. The first protocol, `UITableViewDelegate`, handles things like what to do when a user taps a specific cell. `UITableViewDataSource`, on the other hand, contains the methods you can use to show the data for each of a table view's cells.

In this section, you'll modify the UI for a messaging app using `UITableView`; however, it's not that simple because the content of the cells can vary in size, so they need to grow accordingly. But don't worry, with the power of Auto Layout and self-sizing table view cells, you'll be able to get things working in no time. This section will demonstrate sizing views using the bottom-to-top in a view hierarchy approach.

## Self-sizing table view cells

Open the table view's **starter project**. Build and run.

The app consists of a simple table view controller containing a set of messages.



At first glance, the app doesn't look that great because:

1. The text is getting cut off because the cells don't resize based on their content.
2. The app doesn't resemble a typical chat app.
3. There's no way to differentiate the participants.

To solve the first issue, you need cells that adapt their size based on their content. Before Auto Layout, you may have made some manual calculations and used `tableView(_:heightForRowAt:)` to provide the height for each cell. But with Auto Layout and self-sizing table view cells, you don't need to do that anymore.

Stop the app so you can get to work.

To use self-sizing cells, you need to follow three simple rules:

1. Set the table view's `rowHeight` to `UITableView.automaticDimension`.
2. Set the table view's `estimatedRowHeight` to the value you want or do the same by implementing the height estimation delegate method.

**Note:** The default value for `estimatedRowHeight` is `automaticDimension`, meaning the table view will set an estimated value for you. That makes setting the estimated row height optional, but it's good to implement it when you have an idea of what the cell height should be. Try to be as accurate as possible, because it's used by the system to calculate things like the scroll bar height.

3. Use Auto Layout for the UI elements inside of the cells. Keep in mind that you have to create the constraints within the content view of the table view cell. It's also mandatory to have an unbroken chain of constraints for this to work.

Open `MessagesTableViewController.swift`, and inside `configureTableView()`, add the following code at the bottom:

```
tableView.rowHeight = UITableView.automaticDimension
```

This satisfies Step #1 above.

On `tableView(_:cellForRowAt:)`, look at how the cell is instantiated:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let message = messages[indexPath.row]

    //1
    var cell: MessageBubbleTableViewCell
    if message.sentByMe {
        cell = tableView.dequeueReusableCell(withIdentifier: MessageBubbleCellType.rightText.rawValue,
                                            for: indexPath) as! RightMessageBubbleTableViewCell
    } else {
        cell = tableView.dequeueReusableCell(withIdentifier: MessageBubbleCellType.leftText.rawValue,
                                            for: indexPath) as! LeftMessageBubbleTableViewCell
    }

    //2
    cell.textLabel?.text = message.text
    return cell
}
```

With this code, you:

1. Use an if statement to decide whether the cell should be an instance of `RightMessageBubbleTableViewCell` or `LeftMessageBubbleTableViewCell` based on the value of `message.sentByMe`.
2. Set the text property of `textLabel` with the value of `message.text`.

To customize the cell, you can use `tableView(_:cellForRowAt:)`.

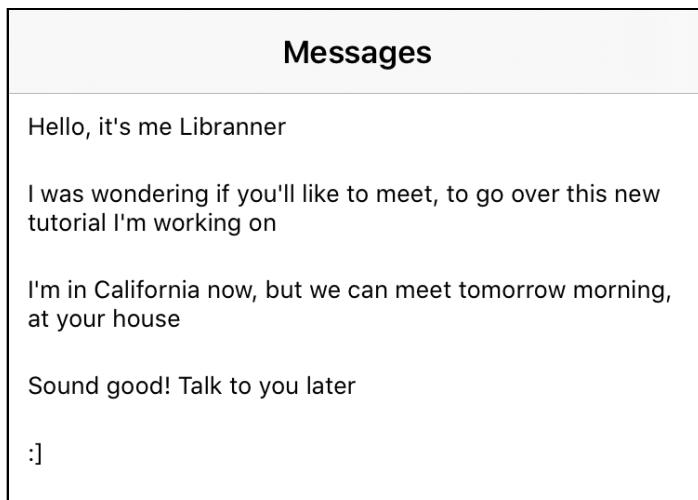
First, in `tableView(_:cellForRowAt:)`, replace `cell.textLabel?.text = message.text` with `cell.messageLabel.text = message.text`. This property is part of the custom implementation based on `MessageBubbleTableViewCell`.

Next, you need to create the constraints between the content and the container view. Go to `MessageBubbleTableViewCell.swift` and uncomment the commented code inside of `configureLayout()`:

```
func configureLayout() {  
    contentView.addSubview(messageLabel)  
  
    NSLayoutConstraint.activate([  
        messageLabel.topAnchor.constraint(  
            equalTo: contentView.topAnchor,  
            constant: 10),  
  
        messageLabel.rightAnchor.constraint(  
            equalTo: contentView.rightAnchor,  
            constant: -10),  
  
        messageLabel.bottomAnchor.constraint(  
            equalTo: contentView.bottomAnchor,  
            constant: -10),  
  
        messageLabel.leftAnchor.constraint(  
            equalTo: contentView.leftAnchor,  
            constant: 10)  
    ])  
}
```

This code adds the `messageLabel` to the content view and sets up the constraints so that the label gets displayed correctly. One important thing to notice here is that you add elements to `contentView` rather than directly to the view itself. You should always do this when the content view is the default superview displayed by the cell. For example, if you want to implement edit mode, using the content view will allow the cell to transition gracefully from one state to the other.

Build and run.



You've sized the content view using the bottom-to-top in a view hierarchy approach. The views within the content view determines the height of the content view.

The app still doesn't look quite right, but at least now all of the text is visible. In the next section, you'll improve the UI further to make it look more like a typical chat app.

## Implementing self-sizing cells

To give this app a standard-looking chat app appearance, you'll place the messages inside chat bubbles and make them look different for each user.

Go to **MessageBubbleTableViewCell.swift**, which is inside **Views**, and remove all of the code in `configureLayout()` except for the first line. After that, add the following above the first line:

```
contentView.addSubview(bubbleImageView)
```

This line of code adds a previously customized image view that will contain the bubble image.

When you're done, the function will look like this:

```
func configureLayout() {
    contentView.addSubview(bubbleImageView)
    contentView.addSubview(messageLabel)
}
```

In the `messageLabel` property initializer, change the `textColor` so that it contrasts better with the background:

```
lazy var messageLabel: UILabel = {
    let messageLabel = UILabel(frame: .zero)
    messageLabel.textColor = .white
    ...
}()
```

Open **LeftMessageBubbleTableViewCell.swift**, and inside `configureLayout()`, after `super.configureLayout()`, add the following block of code:

```
NSLayoutConstraint.activate([
    //1
    contentView.topAnchor.constraint(
        equalTo: bubbleImageView.topAnchor,
        constant: -10),
    contentView.trailingAnchor.constraint(
        greaterThanOrEqualTo: bubbleImageView.trailingAnchor,
        constant: 20),
    contentView.bottomAnchor.constraint(
        equalTo: bubbleImageView.bottomAnchor,
        constant: 10),
    contentView.leadingAnchor.constraint(
        equalTo: bubbleImageView.leadingAnchor,
        constant: -20),
    //2
    bubbleImageView.topAnchor.constraint(
        equalTo: messageLabel.topAnchor,
        constant: -5),
    bubbleImageView.trailingAnchor.constraint(
        equalTo: messageLabel.trailingAnchor,
        constant: 10),
    bubbleImageView.bottomAnchor.constraint(
        equalTo: messageLabel.bottomAnchor,
        constant: 5),
    bubbleImageView.leadingAnchor.constraint(
        equalTo: messageLabel.leadingAnchor,
        constant: -20)
])
//3
let insets = UIEdgeInsets(
    top: 0,
```



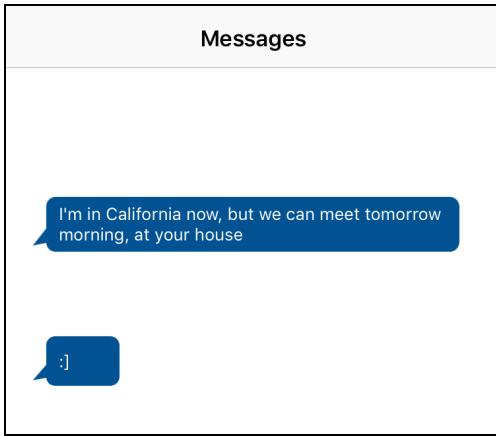
```
    left: 20,
    bottom: 0,
    right: 10)
//4
let image = UIImage(named: blueBubbleImageName)!
    .imageFlippedForRightToLeftLayoutDirection()
//5
bubbleImageView.image = image.resizableImage(
    withCapInsets: insets,
    resizingMode: .stretch)
```

With this code, you:

1. Create top, trailing, bottom and leading constraints for `contentView`. This will guarantee that the content changes its size depending on the dimensions of `bubbleImageView`.
2. Create top, trailing, bottom and leading constraints for `bubbleImageView`. This will apply some padding between `bubbleImageView` and `messageLabel`. It will also make `bubbleImageView` adapt its size according to the amount of text.
3. Create an `UIEdgeInsets`. The insets are necessary so that the chat bubble has the proper padding.
4. Get the corresponding image and call `imageFlippedForRightToLeftLayoutDirection()` to make sure the image is correct in both left-to-right and right-to-left languages.
5. Apply the insets to the image, set the resizing mode to stretch and assign the result to the `image` property of `bubbleImageView`. This code is what makes it possible to have bubbles with different sizes without affecting the quality of the image.

One important thing to notice is the trailing constraint `contentView.trailingAnchor.constraint(greaterThanOrEqualTo: bubbleImageView.trailingAnchor, constant: 20)`. Since this constraint uses the `greaterThanOrEqualTo` relation, the view can grow horizontally, while always keeping at least a 20 point margin from its container.

Build and run.



All done for the left bubble, which will contain the message sent by the other user. You're ready to set up the right bubble, which will be the messages sent by you.

Open **RightMessageBubbleTableViewCell.swift**, and in `configureLayout()`, after `super.configureLayout()`, add the following block of code:

```
NSLayoutConstraint.activate([
    contentView.topAnchor.constraint(
        equalTo: bubbleImageView.topAnchor,
        constant: -10),
    contentView.trailingAnchor.constraint(
        equalTo: bubbleImageView.trailingAnchor,
        constant: 20),
    contentView.bottomAnchor.constraint(
        equalTo: bubbleImageView.bottomAnchor,
        constant: 10),
    //1
    contentView.leadingAnchor.constraint(
        lessThanOrEqualTo: bubbleImageView.leadingAnchor,
        constant: -20),
    //2
    bubbleImageView.topAnchor.constraint(
        equalTo: messageLabel.topAnchor,
        constant: -5),
    bubbleImageView.trailingAnchor.constraint(
        equalTo: messageLabel.trailingAnchor,
        constant: 20),
    bubbleImageView.bottomAnchor.constraint(
        equalTo: messageLabel.bottomAnchor, constant: 5),
    bubbleImageView.leadingAnchor.constraint(
        equalTo: messageLabel.leadingAnchor,
        constant: -10)
])
```

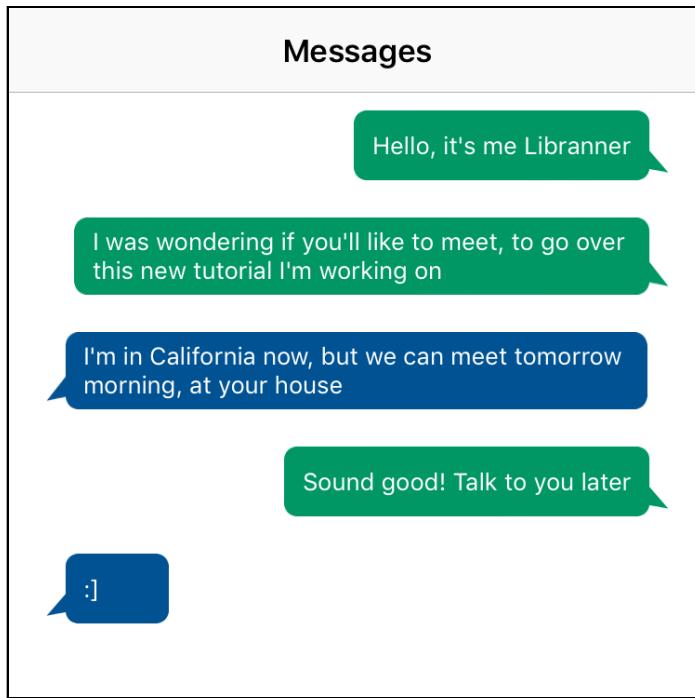


```
//3
let insets = UIEdgeInsets(
    top: 0,
    left: 10,
    bottom: 0,
    right: 20)
//4
let image = UIImage(named: greenBubbleImageName)!
    .imageFlippedForRightToLeftLayoutDirection()
//5
bubbleImageView.image = image.resizableImage(
    withCapInsets: insets,
    resizingMode: .stretch)
```

This method looks similar to the one you wrote for the left bubble with some important changes:

1. The relation used for the leading anchor is `lessThanOrEqualTo`, as this view should align to the right. It grows horizontally from right-to-left, adding a margin of 20 on the leading side.
2. All of the other constraints are similar; however, the bubble needs to align to the right, so some of the values regarding leading and trailing anchors are switched.
3. Create an `UIEdgeInsets`. The insets are necessary so that the chat bubble has the proper padding.
4. Get the corresponding image and call `imageFlippedForRightToLeftLayoutDirection()` to make sure the image is correct in both left-to-right and right-to-left languages.
5. Apply the insets to the image, set the resizing mode to stretch and assign the result to the `image` property of `bubbleImageView`. This code is what makes it possible to have bubbles with different sizes without affecting the quality of the image.

Build and run.



Everything looks great! You were able to create a nice-looking chat app, and the cells adapt to their content. With self-sizing table view cells, you can build interfaces that respond well to different types of content.

**Exercise: Modify the cell so that it can display images and text.**

For this exercise, modify the cell so that it can display images and text. The `imageName` property on the `Message` model contains the image name; to make it easier, the images are part of the bundle.

Give it a try, and afterward, check the challenge project to see how you did.

## CollectionView

`UITableView` presents a list of rows in a single column. You'd typically want to choose table views for row layouts. Table views can benefit you with an easy to setup layout process. You can get them up and running in almost no time. As amazing as table views are, they do fall short when you want to support layouts beyond a single-column layout.

Whenever there's a list of items you want to present, it's a good idea to start and see if table views can do the job. If not, then look into `UICollectionView`. When it comes to layout possibilities, the sky is the limit with collection views.

Here are some layouts, which table views will have a hard time achieving, but not for collection views:

Say you want any of these layouts:

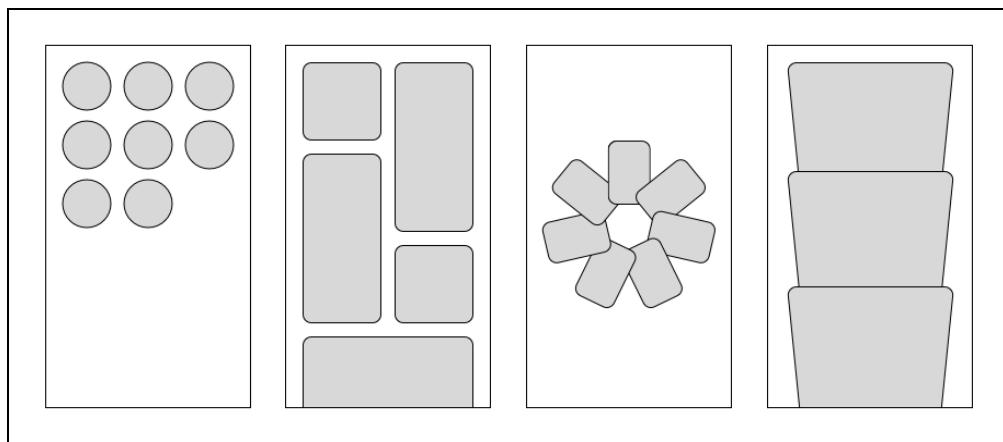


Table views will have a hard time or not be able to achieve the layouts above. `UICollectionView`, on the other hand, is more fit to handle the layouts above with greater ease.

With `UICollectionView`, you can present a list with more layout possibilities.

In this section, you'll learn about the following `UICollectionView` topics:

- Why collection views are useful.
- Collection view anatomy.
- Setting up a collection view with flow layout.

- Working with UICollectionViewDataSource.
- Manually sizing collection view cells.
- Working with UICollectionViewDelegate.
- Working with UICollectionViewDelegateFlowLayout.
- Working with reusable supplementary views.
- Handling an app's change in orientation and cell layout.

To help focus on the layout code that makes up a collection view, you'll implement one programmatically. After doing so, you'll have a solid foundation on how to set up a collection view. More importantly, you'll be able to better understand the layout possibilities collection views have to offer. Contrasting to the table view section, this section will demonstrate sizing views using the top-to-bottom in a view hierarchy approach.

## Why collection view?

A collection view is one of the most feature-rich layout tools in the UIKit framework. A collection view can display a list of items in almost any layout imaginable. Collection views come to mind usually when a table view doesn't offer the more complex layout features. In other words, whenever you intend your list to present layout in ways different than rows, a collection view is often the hero to save the day.

Some features available in a collection view but not in a table view are:

- Presenting a list in a grid layout.
- Scrolling a list in the horizontal direction.
- Sizing a cell's width and height.
- Built-in interactive animations (insert, delete or reorder items).

UICollectionView opens you up to a vast world of layout possibilities. You have greater layout flexibility than UITableView with UICollectionView. For this reason alone, it's well worth the time investment for UICollectionView mastery.

Collection views excel at grid layout. You can find apps with grid layout throughout the App Store nowadays. You have apps such as *App Store*, *Photos*, *Instagram*, *Pinterest*, *Netflix*, *Airbnb* and many more that implement a grid layout.

## Collection view anatomy

To build out a collection view, there are four basic components you need to know about to get started:

- **UICollectionViewLayout:** Every collection view requires a layout object. `UICollectionViewLayout` decides how the items inside a collection view position and look. You can find yourself dealing with items such as collection view cell, supplementary view, and decoration view.
- **UICollectionViewCell:** This is similar to `UITableViewCell`; however, it has greater layout flexibility. In addition to a configurable cell height, you can configure a collection view cell's width.
- **Supplementary View:** You can use supplementary views to present headers and footer views in a collection view.
- **Decoration View:** Unlike a collection view cell and supplementary view, a decoration view's presentation is independent of the collection view's data source. Decoration views act solely as visual adornments for a section or the entire collection view. You typically won't need to make use of decoration views.

In a collection view, you'll find that `UICollectionViewLayout` and `UICollectionViewCell` are required to populate the collection view. Supplementary and decoration views are optional items.

To organize items into a grid, you'd typically use a concrete layout object that comes with `UIKit`, **`UICollectionViewFlowLayout`**, so you won't need to implement the details for positioning each item within a collection view.

`UICollectionViewFlowLayout` is a subclass of `UICollectionViewLayout` made to organize items into a grid.

Subclass your own `UICollectionViewLayout` when you want a layout that's different than a grid layout like a circular layout or when you want finer control over the layout details. In those scenarios, you may want to create a layout object that subclasses `UICollectionViewLayout`. Otherwise, make use of `UICollectionViewFlowLayout` to avoid reinventing the wheel.

This chapter makes use of `UICollectionViewFlowLayout` instead of subclassing `UICollectionViewLayout` and, because you won't be creating a custom layout class, the chapter will omit the use of decoration views.

All right, time to get your hands dirty with collection views.

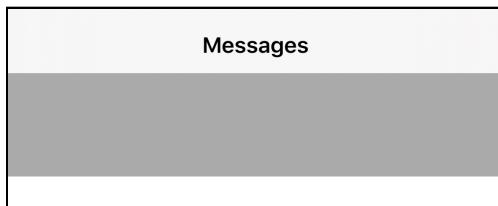
## Building mini-story view

In many social media apps, you'll find story sharing features. The stories contain events shared within the last 24 hours. You can find the story sharing features in apps such as *Snapchat*, *Messenger* and *Instagram*.

The mini-story view you build will contain user stories. Each story will come in the shape of a circle. The mini-story view will also be scrollable in the horizontal direction.

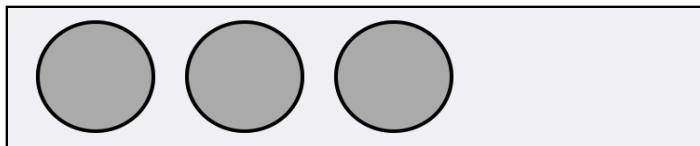
### Setting up collection view properties

Open the collection view's **starter project**. Build and run.



You have a view controller with a `MiniStoryView`. The `MiniStoryView` contains an empty collection view at the moment.

Your goal is to build out the mini-story view with a collection view inside, and achieve the following layout:



To begin, in **User Story/Views**, open `MiniStoryView.swift`. Add the following properties:

```
// 1
private let verticalInset: CGFloat = 8
private let horizontalInset: CGFloat = 16
// 2
private lazy var flowLayout: UICollectionViewFlowLayout = {
    let flowLayout = UICollectionViewFlowLayout()
    flowLayout.minimumLineSpacing = 16
    flowLayout.scrollDirection = .horizontal
    flowLayout.sectionInset = UIEdgeInsets(
        top: verticalInset,
        left: horizontalInset,
```

```
        bottom: verticalInset,
        right: horizontalInset)
    return flowLayout
}()
// 3
private lazy var collectionView: UICollectionView = {
    let collectionView = UICollectionView(
        frame: .zero,
        collectionViewLayout: flowLayout)
    collectionView.register(
        MiniStoryCollectionViewCell.self,
        forCellWithReuseIdentifier: cellIdentifier)
    collectionView.showsHorizontalScrollIndicator = false
    collectionView.alwaysBounceHorizontal = true
    collectionView.backgroundColor = .systemGroupedBackground
    collectionView.dataSource = self
    collectionView.delegate = self
    return collectionView
}()
```

The code above defines some properties to set up a collection view. Here's a closer look:

1. Create two constants. The vertical inset provides the collection view with a top and bottom content insets of 8 points spacing. Similarly, the horizontal inset provides the collection view with a left and right content insets of 16 points spacing.
2. Lazily load the collection view flow layout to utilize `verticalInset` and `horizontalInset`. You specify the line spacing between cells and the scroll direction to horizontal. Also, you set the section inset with `inset` properties you've created earlier and some user interface properties.
3. Initialize a collection view with `flowLayout`, configure user interface properties, set the data source and set the delegate.

You're off to a great start, but there are more steps involved to get your collection view looking better.

## Creating Auto Layout constraint extensions

In your project, you can abstract commonly used code into a helper method. Also, extensions allow you to add additional functionalities to existing code.

Under **App/Extensions**, create a Swift file named **UIView+Constraints.swift**. Then, add the following code to the file:

```
import UIKit

extension UIView {
    func fillSuperview(withConstant constant: CGFloat = 0) {
        guard let superview = superview else { return }
        translatesAutoresizingMaskIntoConstraints = false
        NSLayoutConstraint.activate(
            [leadingAnchor.constraint(
                equalTo: superview.leadingAnchor,
                constant: constant),
             topAnchor.constraint(
                equalTo: superview.topAnchor,
                constant: constant),
             trailingAnchor.constraint(
                equalTo: superview.trailingAnchor,
                constant: -constant),
             bottomAnchor.constraint(
                equalTo: superview.bottomAnchor,
                constant: -constant)])
    }
}
```

`fillSuperview(withConstant:)` is accessible to every `UIView` in the project. In many scenarios, you'd want a view to fill out the superview. Instead of having to write almost identical code over and over again, you can create an extension like you did. Then, create a helper method to reduce the number of lines of code. The `constant` parameter is there for when you want to add spacing to leading, trailing, top, and bottom anchors. Its default value is `0`.

## Adding collection view constraints

Now, add the following code to `setupCollectionView()` in `MiniStoryView`:

```
addSubview(collectionView)
collectionView.fillSuperview()
```

With this code, you add a collection view to the view's subview and then, set up the Auto Layout constraints.

## Manually sizing collection view cells

There are two ways to determine a collection view cell size. Although you can size collection view cells manually or automatically, this section focuses on ways to manually size collection view cells.

You can define the collection view cell size using `flowLayout` and setting its `itemSize`.

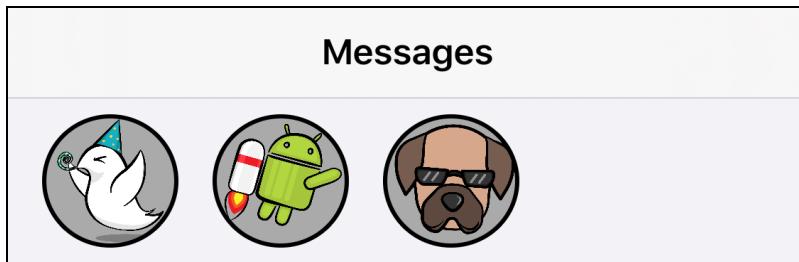
Add the following code to `MiniStoryView`:

```
override func layoutSubviews() {
    super.layoutSubviews()
    let height = collectionView.frame.height - verticalInset * 2
    let width = height
    let itemSize = CGSize(width: width, height: height)
    flowLayout.itemSize = itemSize
}
```

The logic here is straightforward. The collection view cell's height equals the collection view frame's height minus the top and bottom insets. The cell width equals to height. You create a `CGSize` object with the width and height values you declared earlier and pass the size as the flow layout's item size. You set the `flowLayout` item size here because you want to use the collection view frame height only after the collection view lays out.

With the implementation from above, you have cells with equal width and height.

Build and run, and you'll see your collection view and collection view cells.



You've sized the content view's child views using the top-to-bottom in a view hierarchy approach. The content view determines the size of the child views.

You've covered quite a number of collection view components. Yet, it's still the tip of the iceberg for what's possible. The fundamentals are what you'll use to create complex systems. Helping you see different ways in which collection views are utilized can give you ideas on how to create different layouts.

Earlier, the chapter mentions that you can have almost any layout you can imagine. Time to give collection view cells the Super Mario mega-mushroom and go bigger with collection view cells. Along the way, you'll pick up on skills on building out collection views.

# Building user story view controller's collection view

Now, you'll build out the collection view to display the user stories. You can see a user's stories by tapping on a `MiniStoryCollectionViewCell` in the `MessagesViewController`. You'll work with the collection view's `UICollectionViewDelegateFlowLayout`, reusable supplementary views and handle app orientation transitions.

## Working with `UICollectionViewDelegateFlowLayout`

Earlier, you've learned to set the collection view item size, content insets, and line spacings with the collection view flow layout properties. Now, you'll learn to implement the collection view item size and content insets by implementing `UICollectionViewDelegateFlowLayout`'s methods.

Add the following code to the end of `UserStoryViewController.swift`:

```
// MARK: - UICollectionViewDelegateFlowLayout
extension UserStoryViewController: UICollectionViewDelegateFlowLayout {
    // 1
    func collectionView(
        _ collectionView: UICollectionView,
        layout collectionViewLayout: UICollectionViewLayout,
        sizeForItemAt indexPath: IndexPath
    ) -> CGSize {
        return collectionView.frame.size
    }
    // 2
    func collectionView(
        _ collectionView: UICollectionView,
        layout collectionViewLayout: UICollectionViewLayout,
        insetForSectionAt section: Int
    ) -> UIEdgeInsets {
        return .zero
    }
    // 3
    func collectionView(
        _ collectionView: UICollectionView,
        layout collectionViewLayout: UICollectionViewLayout,
        minimumLineSpacingForSectionAt section: Int
    ) -> CGFloat {
        return 0
    }
}
```

Here's what's happening with this code:

1. The method asks for the item size. You return the collection view frame size.
2. The method asks for the content inset. You return zero for left, right, top, and bottom content insets with `.zero`.
3. The method asks for the minimum line spacing. You return zero to have no spacing between items.

Add the following code to `collectionView`'s initializer right before returning `collectionView`:

```
collectionView.delegate = self
```

This sets the collection view's delegate to `UserStoryViewController`, and the `UICollectionViewDelegateFlowLayout` methods will take part in shaping the cells.

If you take a look inside `collectionView`'s closure, you'll see that `isPagingEnabled` is set to `true`. When this property is `true`, the collection view scrolls and stops on multiples of the scroll view's bounds.

Build and run, then tap a story. `UserStoryViewController` will have collection view cells filling the screen.



When you end scrolling on the collection view, you'll see the cells always have its edges filling the screen.

## Working with reusable supplementary views

Now, your goal is to implement a header view in the collection view. Open **HeaderCollectionReusableView.swift** inside of **User Story/Views**. Add the following code to the `setupStackView()`:

```
addSubview(stackView)
stackView.fillSuperview()
NSLayoutConstraint.activate(
    [topSpacerView.heightAnchor.constraint(
        equalTo: bottomSpacerView.heightAnchor)])
)
```

Here, you make `stackView` fill the superview with the Auto Layout constructor helper method. And, you set the top and bottom spacer views to have equal heights.

Open **UserStoryViewController.swift**. Add the following code to `collectionView's` closure right before returning `collectionView`:

```
collectionView.register(
    HeaderCollectionReusableView.self,
    forSupplementaryViewOfKind:
        UICollectionView.elementKindSectionHeader,
    withReuseIdentifier: headerViewIdentifier)
```

Here's what you've done with the code above. Collection view registers `HeaderCollectionReusableView` as a section header. The process of registering and dequeuing `UICollectionViewReusableView` is similar to that of a `UICollectionViewCell`.

Inside the `UICollectionViewDataSource` extension section, add the following code:

```
// 1
func collectionView(
    _ collectionView: UICollectionView,
    viewForSupplementaryElementOfKind kind: String,
    at indexPath: IndexPath
) -> UICollectionViewReusableView {
// 2
guard let headerView = collectionView
    .dequeueReusableSupplementaryView(
        ofKind: UICollectionView.elementKindSectionHeader,
        withReuseIdentifier: headerViewIdentifier,
        for: indexPath) as? HeaderCollectionReusableView
    else { fatalError("Dequeued unregistered reusable view") }
```

```
// 3
headerView.configureCell(username: userStory.username)
return headerView
}
```

With the code above, you:

1. Implement a `UICollectionViewDataSource`'s method that determines what supplementary views are displayed.
2. Dequeue a `HeaderCollectionReusableView` as a header view using a reusable identifier.
3. Pass in the user story's username to the cell for label and image configurations.

Finally, time to set the size of the header view. In the `UICollectionViewDelegateFlowLayout` extension section, implement the following method:

```
func collectionView(
    collectionView: UICollectionView,
    layout collectionViewLayout: UICollectionViewLayout,
    referenceSizeForHeaderInSection section: Int
) -> CGSize {
    return collectionView.frame.size
}
```

The code above sets the header view size equal to the collection view frame.

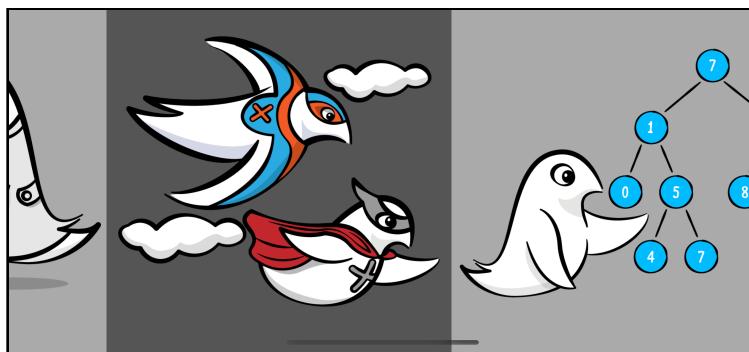
Build and run, and you'll see the following screen when you tap the first collection view cell in `MiniStoryView`.



Next, you'll learn to adapt your collection view layout for when your device changes its orientation.

## Handling app orientation transitions

Currently, there's a visual conflict when `UserStoryViewController` is present and the device rotates. To see the visual conflict, build and run. Tap a user story. With a `UserStoryViewController` in view, scroll to the second collection view cell. Rotate the device. You'll see something like this:



Instead of the screen above, you want to center the cells with no sign of the previous/next cells.

Time to fix this. First, implement the following `UIScrollViewDelegate` method in `UserStoryViewController`:

```
// 1
func scrollViewWillEndDragging(
    scrollView: UIScrollView,
    withVelocity velocity: CGPoint,
    targetContentOffset: UnsafeMutablePointer<CGPoint>
) {
    let contentOffsetX = targetContentOffset.pointee.x
    let scrollViewWidth = scrollView.frame.width
    currentItemIndex = Int(contentOffsetX / scrollViewWidth)
}
```

The code above does the following:

1. `scrollViewWillEndDragging(_:withVelocity:targetContentOffset:)`, as the method name suggests, is called when the scroll view goes into dragging motion and then ends the dragging motion. For example, a user is scrolling a collection view. When the user stops scrolling, the method gets called. Hence, when you end dragging on a collection view, the method gets called.

2. Calculate the `currentItemIndex` using `targetContentOffset` and the view's frame. Once the user ends dragging on the collection view, the collection view stops at a specific position. You also know the x offset value from the original position. You have defined that the scroll view frame width is a multiple of the collection view cell's frame width. Using this information, you can calculate the current collection view item index by simply dividing the x content offset by the scroll view frame width.

You use `currentItemIndex` so that the collection view can know the content offset position you want after the device rotates. This helps center the collection view cells after device rotation.

Add the following method to `UserStoryViewController`:

```
private func centerCollectionViewContent() {
    DispatchQueue.main.async { [weak self] in
        guard let self = self else { return }
        // 1
        let x = self.collectionView.frame.width
            * CGFloat(self.currentItemIndex)
        let y: CGFloat = 0
        let contentOffset = CGPoint(x: x, y: y)

        // 2
        self.collectionView.setContentOffset(
            contentOffset, animated: false)
    }
}
```

You did the following with the code above:

1. Calculate the collection view content offset. The content offset for x takes the collection view's frame width and multiplies it by the `currentItemIndex`. The content offset for y is zero, as there's no need for vertical spacing adjustment. Afterward, you initialize a `CGPoint` named `contentOffset` with x and y.
2. Set the `collectionView`'s content offset equal to `contentOffset` with animation set to `false`.

To finish off, add the following code to `UserStoryViewController`:

```
override func viewWillTransition(
    to size: CGSize,
    with coordinator: UIViewControllerTransitionCoordinator
) {
    super.viewWillTransition(to: size, with: coordinator)
    centerCollectionViewContent()
}
```

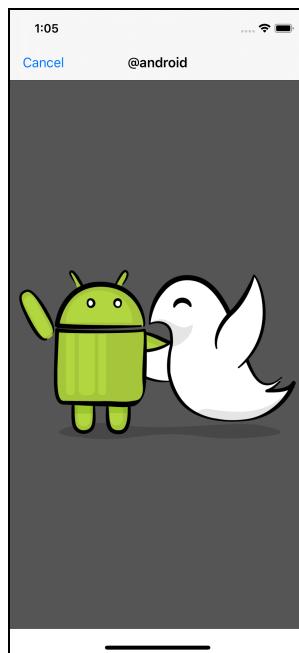


When the device rotates, you call `centerCollectionViewItem()`. This centers the appropriate collection view cell.

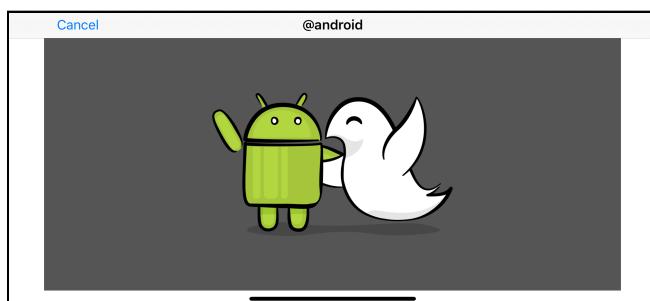
Now, upon device rotation, your collection view cell will center itself using the visible collection view index prior to rotation.

Build and run, then tap a user story. With `UserStoryViewController` in view, scroll to the second collection view cell. Rotate the device.

A device that starts in the portrait orientation looks like this:



And, looks like this after rotating to the landscape orientation:



## Challenges

Using what you've learned so far, implement a collection view will the following instructions:

1. Create and implement `StoryProgressView` in `UserStoryViewController`. This view contains a collection view. The number of cells equals to the number of **user story events**.
2. `StoryProgressView` should only be visible after scrolling past the header view. As an additional challenge, use `scrollViewDidScroll(_:)` to fade `StoryProgressView`'s based on the scroll position. For example, when you scroll, and half of the header view is visible, you should set `StoryProgressView`'s alpha to `0.5`.
3. `StoryProgressView`'s cell highlight should match `UserStoryViewController`'s `currentIndexIndex`. Use `scrollViewDidEndDecelerating(_:)`. For example, with the dog's stories, if you end scrolling at the last collection view cell item, the story progress view should look like this:



4. When you tap on a cell in `StoryProgressView`, you set the cell to highlight and scroll `UserStoryViewController`'s collection view to the cell with the matching index.
5. Implement `StoryProgressView.layoutSubviews()` to handle changes in orientation. You can do this by invalidating the flow layout. When you invalidate a layout, you ask the app to re-query the layout information.

That's it. Go ahead and jump on the challenge. Afterward, review the final project to compare solutions.

## Key points

- Use self-sizing views for dynamic content.
- When working with self-sizing table view cells, make sure to set `rowHeight` of the table view to `automaticDimension`.
- Be careful while setting the constraints inside of the table view cell. You should create these constraints with the content view, not the cells.
- `UICollectionView` offers more layout flexibilities than `UITableView`.
- A collection view requires a `UICollectionViewLayout` and `UICollectionViewCell` to populate itself. Supplementary and decoration views are optional items.
- Typically, use `UICollectionViewFlowLayout` in the collection view for grid layout.
- Subclass `UICollectionViewLayout` for layout that couldn't be easily achieved with `UICollectionViewFlowLayout`.
- You can manually or dynamically size a collection view cell or reusable view.
- Set the collection view data source with `UICollectionViewDataSource`.
- You can size a collection view cell or reusable view by adopting and implementing methods from `UICollectionViewDelegateFlowLayout`.
- Get notified of collection view events with `UICollectionViewDelegate`.
- To make your layout look great, you may need to adjust it for the event of device orientation.

# 7

# Chapter 7: Layout Guides

By Libranner Santos

Layout guides are rectangular areas that you can use to create spaces between your views. In the past, you had to create “dummy views” that acted as spacers. These spacers allowed you to have the desired layout for the UI. Well, with Auto Layout and layout guides, dummy views are no longer necessary.

So, what’s the difference between creating dummy views and using layout guides? To answer this question, you’ll look at some of the key benefits of using layout guides, and how they compare to dummy views:

- Dummy views can interfere with messages that are meant for one of their children. For example, gestures can easily get captured by the wrong view.
- Creating dummy views results in a higher performance cost since these views are part of the view hierarchy and have to be maintained. On the other hand, layout guides are not a visual element since their job is simply to define a rectangular area in the view hierarchy.
- Layout guides are created using code; this allows for more customization, especially when you’re creating all of your constraints within code.



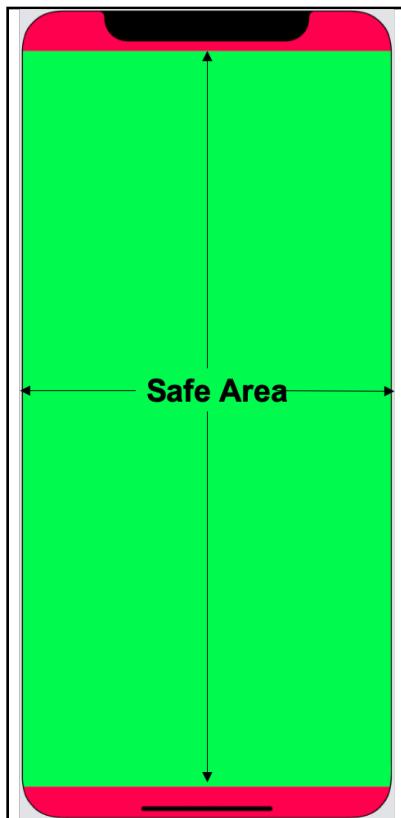
# Available Layout Guides

UIKit comes with some layout guides out of the box; they will help you create adaptable interfaces. Their main focus is to make it easy for developers to know the available space at any time, since this can vary from one device to another. These layout guides are **Safe Area** and **Layout Margin** and **Readable Content**.

## Safe Area layout guide

The iOS SDK comes with several layout guides, but the most used is the **Safe Area layout guide**. The Safe Area layout guide represents the portion of the screen that is not covered by the navigation bars, tab bars, and toolbars.

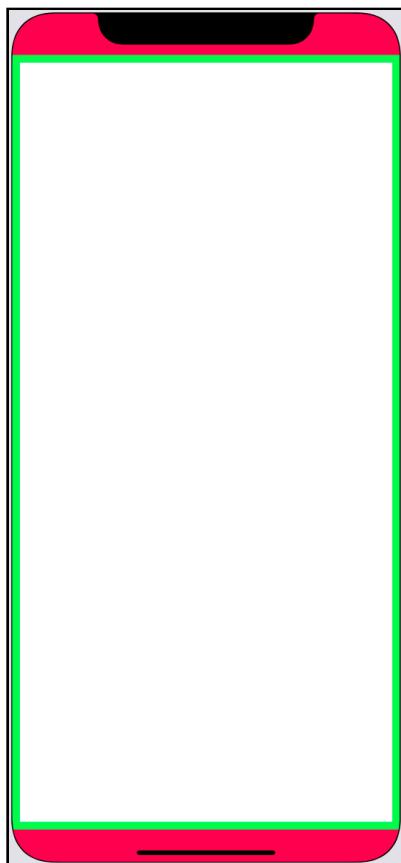
The green rectangle represents the safe area. Notice how it respects the margin so the views don't get covered up by the notch:



## Layout Margin guides and Readable Content guides

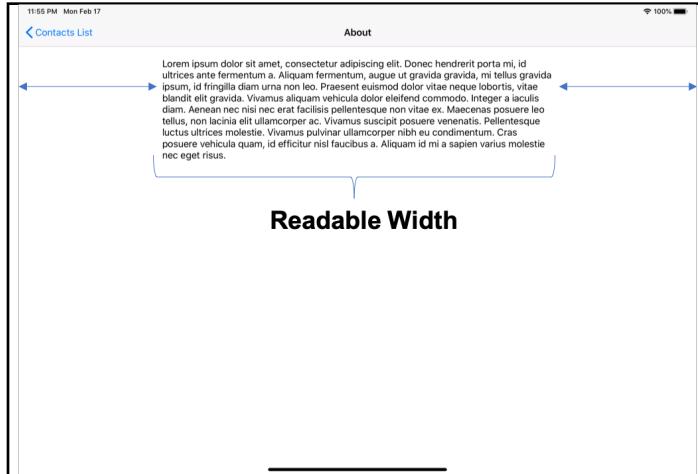
The Safe Area layout guide isn't the only popular guide available in the SDK. The two most notable ones are the **Layout Margin** and the **Readable Content** guides.

The Layout Margin represents the margins of a view. You can use it to create constraints to the view's margins instead of the view itself.



Here, the white rectangle represents a view whose leading, top, bottom, and trailing constraints are attached to their counterpart in the layout margin guide of its parent. That's why there is some spacing between the parent view (the green one) and its child (the white one).

The Readable Content guide represents the recommended area you can use while working with text views within your app. It's a good idea to use this guide because it helps users read without the need to move their heads to track the lines.



The Readable Content guide is also useful because it specifies the total width you can use to display text. So, for example, suppose you want to organize your text into columns. You can use the total width value, provided by this layout guide, to know how many columns you can have. Furthermore, the Readable Content Guide will never extend beyond the Layout Margin guide.

In the next section, you'll create your first layout guide.

## Creating layout guides

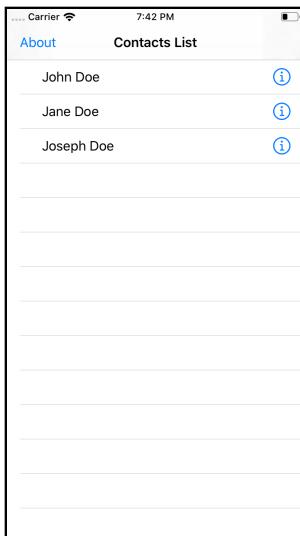
You create layout guides in code using the following these steps:

1. Instantiate a new layout guide.
2. Add to its container view by calling `addLayoutGuide(_:)`.
3. Define the constraints for the layout guide.

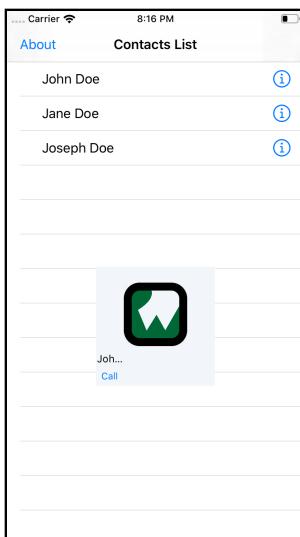
Apart from creating spaces between the views, you can also use layout guides to contain other views. This is useful when you want to create a container that centers on a set of views.

## Creating custom layout guides

Load the starter project and open the MessagingApp project. Build and run.



The apps display a list of constants. Now, pay attention to the left margin. And, on the right side, on the first contact, tap the info accessory button.



Oh, that's not good! The name of the contact and the text for the call button is cut off. But don't worry, you can fix this problem using layout guides.

Open **ContactPreviewView.swift** and add the following code at the end of the class:

```
private func setupLayoutInView(_ view: UIView) {
    //1
    let layoutGuide = UILayoutGuide()

    //2
    view.addLayoutGuide(layoutGuide)

    //3
    nameLabel.translatesAutoresizingMaskIntoConstraints = false
    callButton.translatesAutoresizingMaskIntoConstraints = false

    //4
    NSLayoutConstraint.activate([
        nameLabel.topAnchor.constraint(
            equalTo: layoutGuide.topAnchor),
        nameLabel.leadingAnchor.constraint(
            equalTo: layoutGuide.leadingAnchor),
        nameLabel.trailingAnchor.constraint(
            equalTo: layoutGuide.trailingAnchor),
        callButton.bottomAnchor.constraint(
            equalTo: layoutGuide.bottomAnchor),
        callButton.centerXAnchor.constraint(
            equalTo: nameLabel.centerXAnchor)
    ])

    //5
    let margins = view.layoutMarginsGuide

    //6
    NSLayoutConstraint.activate([
        layoutGuide.topAnchor.constraint(
            equalTo: photoImageView.bottomAnchor, constant: 5),
        layoutGuide.leadingAnchor.constraint(
            equalTo: margins.leadingAnchor),
        layoutGuide.trailingAnchor.constraint(
            equalTo: margins.trailingAnchor),
        layoutGuide.bottomAnchor.constraint(
            equalTo: margins.bottomAnchor)
    ])
}
```

With this code, you:

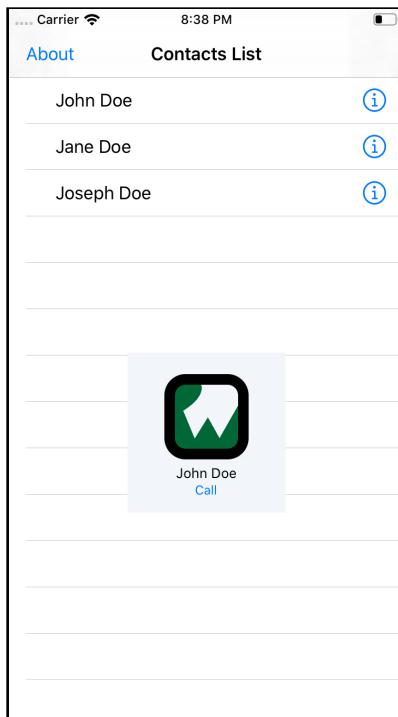
1. Instantiate a new layout guide.
2. Add the new layout guide to the current view, which is **ContactPreviewView**.
3. Set `translatesAutoresizingMaskIntoConstraints` to `false` for `nameLabel` and `callButton`.

4. Create top, leading and trailing constraints for `nameLabel` anchored to `layoutGuide`.
5. Create a constant to hold the Layout Margin guide of the view. This allows you to create constraints anchored to the margins of the view.
6. Create top, leading, trailing and bottom constraints for `layoutGuide`.

`layoutGuide` acts as a container for `nameLabel` and `callButton`. So the constraints that affect `layoutGuide` directly affect the positioning of the views.

Before you check your work, in `loadView()`, add a call to `setupLayoutInView(view)` immediately before `addSubview(view)`. By calling this method, you ensure everything is properly set up when the view is created.

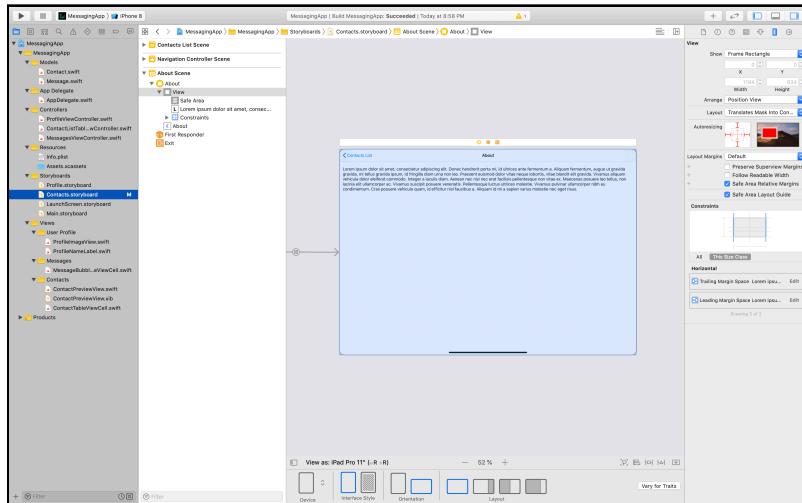
Build and run. Tap any contact within the list.



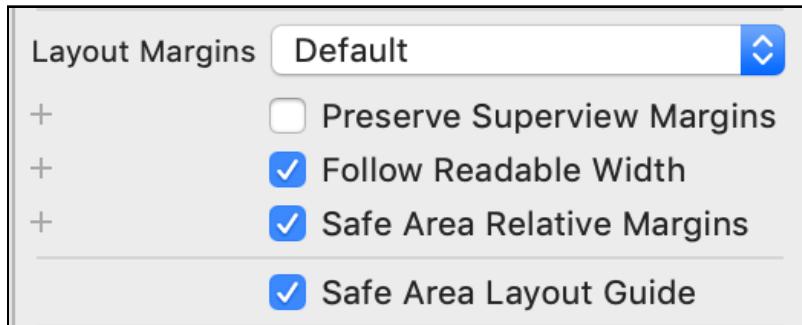
Excellent, it's already looking better!

While still running the app, tap the **About** button in the top left corner. This action displays a screen with a significant amount of text.

Stop the project and open **Contacts.storyboard**. On the bottom center of the Interface Builder, select **View As: iPad Pro 11"** and set the orientation to **Landscape**. The preview is now set to use an 11 inch iPad to display the screen.

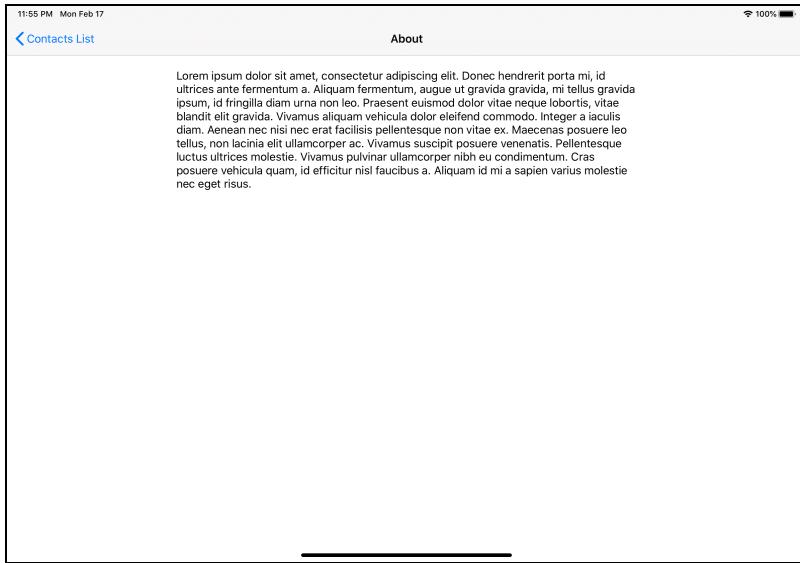


On the About Scene, select the **View element**, and on the Size Inspector, mark the **Follow Readable Width** option.



At this point, you'll notice that the text on the screen is center aligned, and the margin is added so that the user can better read the longer text; this is all possible thanks to the Readable Content guide.

Now, run the app on an iPad to see how it looks.



Without a doubt, layout guides are a convenient tool. They allow you to better organize elements inside of the UI, they serve as a reference for the child elements, and they eliminate the need for dummy views when all you need is some spacing between the views.

## Key points

- Rather than using dummy views, create container layout guides and use them as spacing views.
- Use Readable Content guides when you need to display text within your app.
- Use Layout Margin Guide to restraint all the content inside a view to respect margins.

# Chapter 8: Content-Hugging & Compression-Resistance Priorities

By Libranner Santos

The **content-hugging and compression-resistance priorities (CHCR priorities)** define how a view behaves when constraints that are anchored to it attempt to make the view smaller or larger than its intrinsic size. The content-hugging priority represents the resistance a view has to grow larger than its intrinsic content size. Conversely, the compression-resistance priority represents the resistance a view has to shrink beyond its intrinsic content size. To better understand these concepts, look at the following image:



Although the image only shows the priorities for the horizontal axis, you can set content-hugging and compression-resistance priorities for both the horizontal and vertical axes. In other words, there are four priorities for every view.



## Intrinsic Size and Priorities

Some views have a natural size, which is usually determined by their content and margin. For example, when you use a `UITextField`, it includes a default height. This height is determined by the font size plus the margin between the text and its borders; this is all part of the intrinsic content size of the `UITextField`. Other examples of views that size automatically are images, labels and buttons, where their size is based on their content.

When a view has an intrinsic size, the Auto Layout system creates constraints on your behalf, making the size equal to its intrinsic size. This is great when the default size works for your design. It even saves you time because you don't have to set the width and height constraints for these views manually. However, if you don't want the view's size to equal the intrinsic size, you can set the content-hugging and compression-resistance priorities.

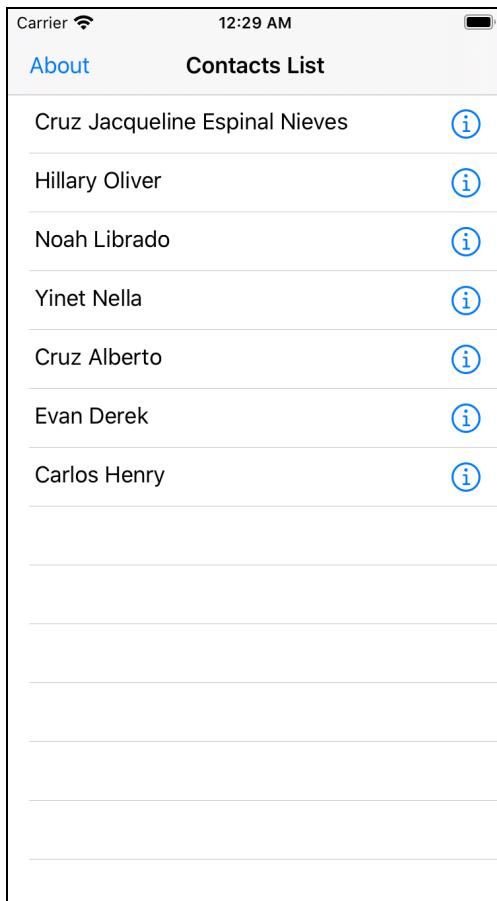
Try to use intrinsic content size when possible, but be sure to set the priorities accordingly. These are some of the benefits of doing this:

- You'll save time because you need fewer constraints since some of them are automatically given by the intrinsic content size.
- By using the right priorities, you can make any view grow dynamically depending on the content it holds. This can help you create better and more adaptable user interfaces.
- You'll avoid letting the Auto Layout system decide how to manage conflicts. When the Auto Layout system tries to solve constraint issues, it does so randomly, which could result in unexpected problems.
- By setting lower priorities, you can increase your app's performance. This is possible because the Auto Layout algorithm finds which constraints can be broken faster. In short, not all constraints added are required.

Are you ready to get into some code? Good, because it's time to get your hands dirty.

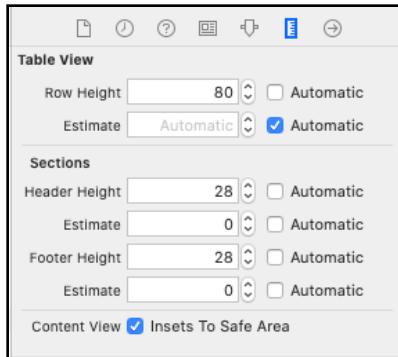
## Practical use case

In this section, you'll see a practical use case for applying content-hugging and compression-resistance priorities. But first, build and run the app, so you know what to expect.

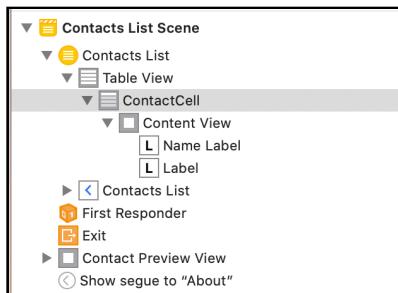


Currently, the app displays a list of contacts. For this exercise, you'll add the ability to see the last message received from a contact.

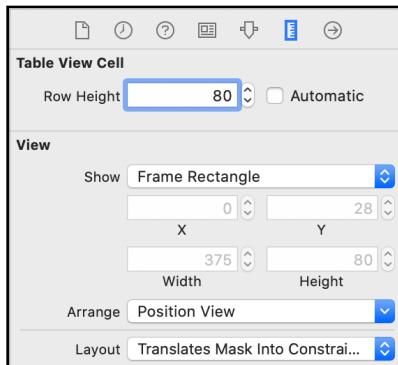
Open **Contacts.storyboard**, and select the **Table View** in the **Contacts List Scene**. Once selected, go to the Size inspector and set Row Height to **80** and make sure Automatic is **unchecked**.



Now, select **ContactCell** in the document outline.

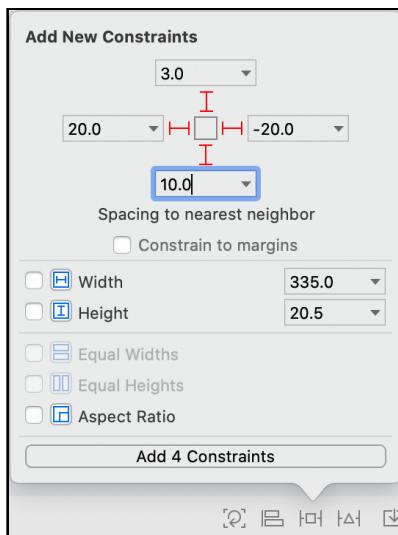


In the Size inspector, set Row Height to **80**. This will make the cell larger so that you can display the last message. Once again, make sure Automatic is **unchecked**.

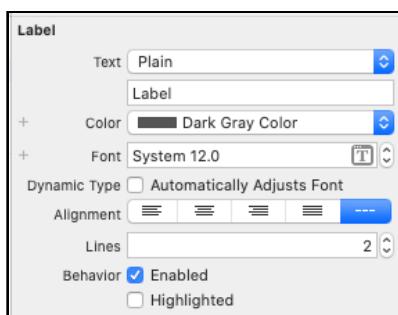


Next, add a new label below the current one with the following constraints:

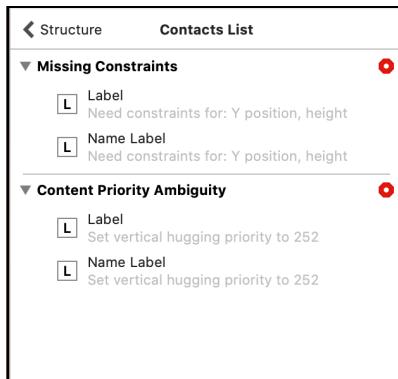
- A value of **3** from the Name Label for the Top.
- A value of **20** from its superview for the Leading.
- A value of **20** from its superview for the Trailing.
- A value of **10** from its superview for the Bottom.



With the label still selected, go to the Attributes inspector and set Font Size to **12**. Also, set Color to **Dark Gray** and Lines to **2**.



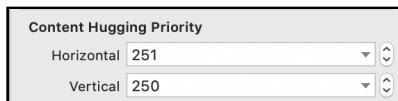
The new label now has all of its constraints, but there's still something wrong: In the top right corner of the document outline, there's a red icon. This red icon indicates there are conflicts that Auto Layout can't resolve on its own. Click the red icon to get more information.



First, there's a message about missing constraints for both of the labels. It appears as if Auto Layout doesn't know the height or y position for either label — that's weird since you created the top constraints already, right?

You'll also see there's a **Content Priority Ambiguity**. What this means is that since both labels have the same level of priority, Auto Layout doesn't know which one to stretch.

To solve these issues, select the label you just created, and in the Size inspector, navigate to the Content Hugging Priority section, which is at the bottom. In this section, you'll see the value assigned for the horizontal and vertical priorities. By default, all elements of the UI are given a priority value of 251; change the value for the vertical priority to **250**.

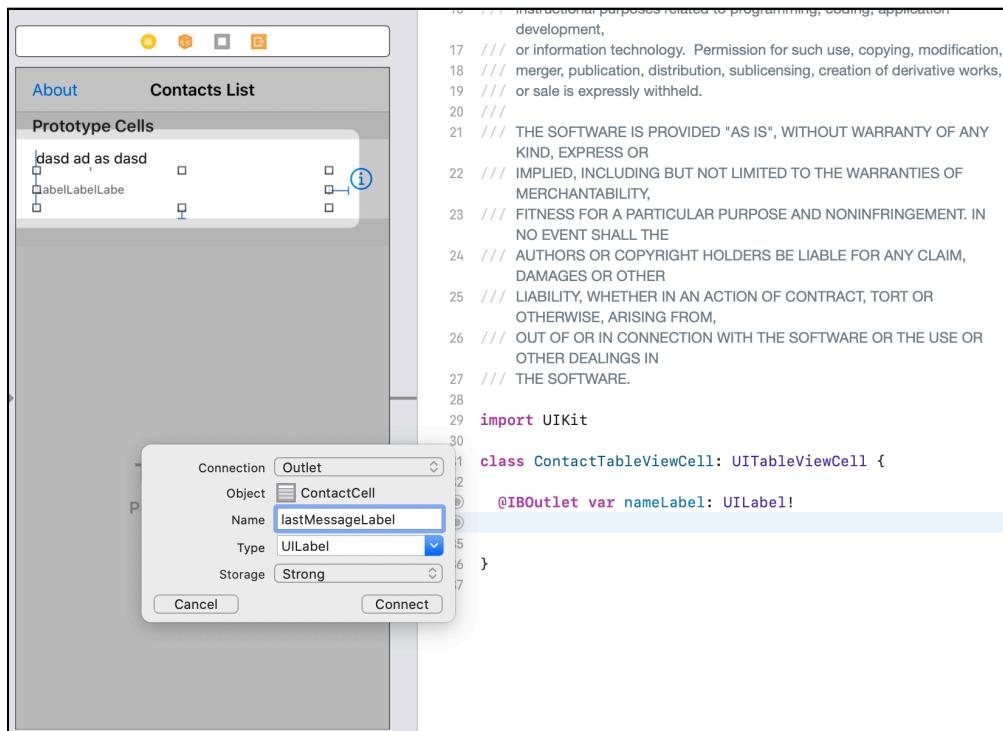


The conflicts are now gone.

By decreasing the priority for one label, you gave Auto Layout all it needs to handle the cases where there's more room than needed for the content. In this case, it'll stretch the message label.

Next, you need to create a connection to the label so that you can show the last message sent by the contact.

Go to the Editor menu and click **Assistant**. Verify the code showing is for `ContactTableViewCell`. Now, drag a line for the Outlet to the corresponding label.



After the outlet is set up, `ContactTableViewCell` will look like this:

```

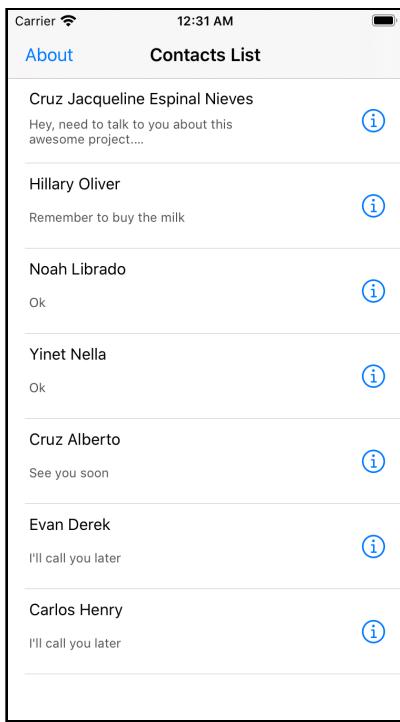
29 import UIKit
30
31 class ContactTableViewCell: UITableViewCell {
32     @IBOutlet var nameLabel: UILabel!
33     @IBOutlet var lastMessageLabel: UILabel!
34 }

```

Open `ContactListTableViewController.swift`, and in `tableView(_:cellForRowAt:)`, add the following line after `cell.nameLabel.text = contact.name`:

```
cell.lastMessageLabel.text = contact.lastMessage
```

Build and run.

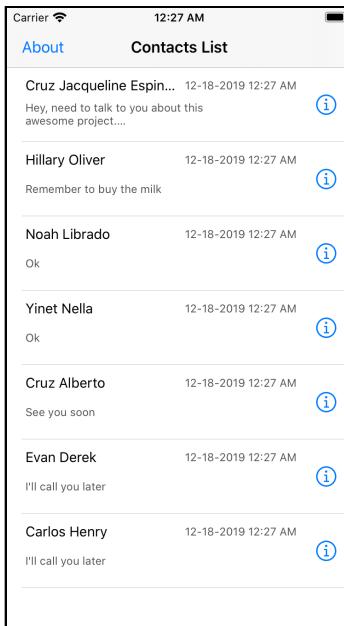


Excellent! Thanks to you, the users can see the last message sent by their contacts on the Contact List screen.

This is a simple example of the things you can accomplish when you manipulate the content-hugging and compression-resistance priorities in your favor. It can certainly save you some headaches when creating flexible user interfaces.

## Challenge

Your challenge for this chapter is to add a label to display the date of the last message on the top right corner of the contact. For this challenge, you'll need to add a `UILabel` to the Contact cell. You'll also need to add the corresponding constraints and make sure to set the proper values for the content compression-resistance priority. If all goes well, your updated view will look like this:



Notice how the label with the date and time has priority over the name of the contact.

## Key points

- The content-hugging priority represents the resistance to grow larger than the view intrinsic content size.
- The compression-resistance priority represents the resistance to shrink beyond the view intrinsic content size.
- Use intrinsic content size in combination with priorities to achieve adaptable user interfaces.

# Chapter 9: Animating Auto Layout Constraints

By Libranner Santos

So far, you've been creating static constraints and learning how to structure your app's UI using Auto Layout. But did you know that you can also animate your constraints? It's true — and in this chapter, you'll learn how to do it.

With animations, you can:

- Give feedback to the users.
- Direct the attention to a specific part of the app.
- Help the user to identify connections between parts of the app.
- Create a better look and feel.
- Improve navigation.

When done properly, animations can increase user engagement and are often what makes (or breaks) your app's success. When creating animations, you need to consider three things: start value, end value and time.

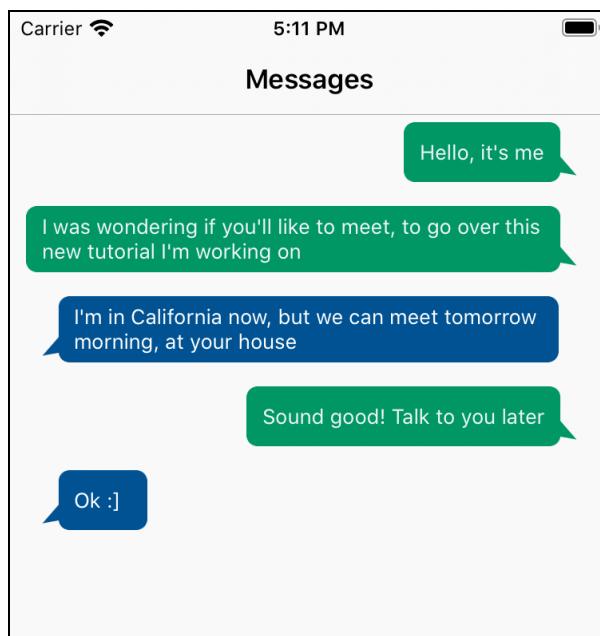


# Animate Auto Layout using Core Animations

You can animate constraints either by changing the constant of a constraint or by activating and deactivating constraints. The method you choose depends largely on what you're trying to accomplish.

In this section, you'll play with constraint animations by updating the MessagingApp project. Specifically, you'll add new functionality to the app that allows users to like messages and mark them as favorite. Here's how it'll work: When a user double-taps a message, the app will show a toolbar with buttons to like the messages and to mark the message as favorite. This functionality will only work for messages not sent by the user.

To begin, open the MessagingApp project and build and run it.



The app displays a chat between two users; the messages on the right are the ones sent by you. If you double-tap one of the blue bubble messages now, nothing happens. But that's about to change!

## Setting up the delegate for MessageBubbleTableViewCell

Open **MessageBubbleTableViewCell.swift** and add the following code above the class declaration:

```
protocol MessageBubbleTableViewCellDelegate {
    func doubleTapForCell(_ cell: MessageBubbleTableViewCell)
```

This creates a new protocol that lets you define what happens when the user double-taps a cell.

Add this property at the top of the class:

```
var delegate: MessageBubbleTableViewCellDelegate?
```

This property allows the cell to remember its delegate.

Next, add the following new method and place it anywhere within the class:

```
@objc func doubleTapped() {
    delegate?.doubleTapForCell(self)
```

In a moment, you'll connect the double-tap gesture to this method. When this method is called, it notifies the delegate of the double tap.

You're almost done with the delegate set up. In `awakeFromNib()`, below `super.awakeFromNib()`, add the following code:

```
let gesture = UITapGestureRecognizer(
    target: self,
    action: #selector(doubleTapped))
gesture.numberOfTapsRequired = 2
gesture.cancelsTouchesInView = true
contentView.addGestureRecognizer(gesture)
```

This code adds a new gesture recognizer to the cell, specifically to the content view. Every time the user double-taps the content view, `doubleTapped()` gets called.

Go to **MessagesViewController.swift** and declare the following two properties at the top:

```
private let toolbarView = ToolbarView()
private var toolbarViewTopConstraint: NSLayoutConstraint!
```

This instantiates a `ToolbarView` and provides a way for you to keep track of its top constraint. This is the constraint you're going to animate, so you need to be able to access it later.

The project already includes a custom view named `ToolbarView`; you'll use this view to show the like and favorite buttons to the user. The specific details of the implementation are not important for this exercise; all you need to know is that it's a simple view containing a stack view and two buttons: one to like the message and the other to mark it as a favorite.

Below `loadMessages()`, add the following code:

```
private func setupToolbarView() {
    //1
    view.addSubview(toolbarView)

    //2
    toolbarViewTopConstraint =
        toolbarView.topAnchor.constraint(
            equalTo: view.safeAreaLayoutGuide.topAnchor,
            constant: -100)

    toolbarViewTopConstraint.isActive = true

    //3
    toolbarView.leadingAnchor.constraint(
        equalTo: view.safeAreaLayoutGuide.leadingAnchor,
        constant: 30).isActive = true
}
```

With this code, you:

1. Add `toolbarView` to `view`, which makes it part of the view hierarchy.
2. Set up `toolbarViewTopConstraint`, which positions `toolbarView` 100 points from the top. This position essentially hides the toolbar.

- Set up the leading constraints for toolbarView. Inside the initialization of ToolbarView, the width and height constraints are already set up, so you only need to indicate the horizontal and vertical constraints.

You now need to call `setupToolbarView()` when the view loads. In `viewDidLoad()`, below the call to `loadMessages()`, add the following:

```
setupToolbarView()
```

The next step is to have the toolbar appear when the user double-taps a message bubble. To accomplish this, you need to set up `MessageBubbleTableViewCellDelegate`.

First, add the following line before the `return cell` statement in `tableView(_:cellForRowAt:)`:

```
cell.delegate = self
```

Next, add the following new extension to `MessagesViewController`:

```
extension MessagesViewController:  
MessageBubbleTableViewCellDelegate {  
    func doubleTapForCell(_ cell: MessageBubbleTableViewCell) {  
        //1  
        guard let indexPath = self.tableView.indexPath(for: cell)  
            else { return }  
        let message = messages[indexPath.row]  
        guard message.sentByMe == false else { return }  
  
        //2  
        toolbarViewTopConstraint.constant = cell.frame.midY  
  
        //3  
        toolbarView.alpha = 0.95  
  
        //4  
        toolbarView.update(  
            isLiked: message.isLiked,  
            isFavorited: message.isFavorited)  
  
        //5  
        toolbarView.tag = indexPath.row  
  
        //6  
        UIView.animate(  
            withDuration: 1.0,  
            delay: 0.0,  
            usingSpringWithDamping: 0.6,  
            initialSpringVelocity: 1,
```

```
options: [],
animations: {
    self.view.layoutIfNeeded()
},
completion: nil
}
```

With this code, you:

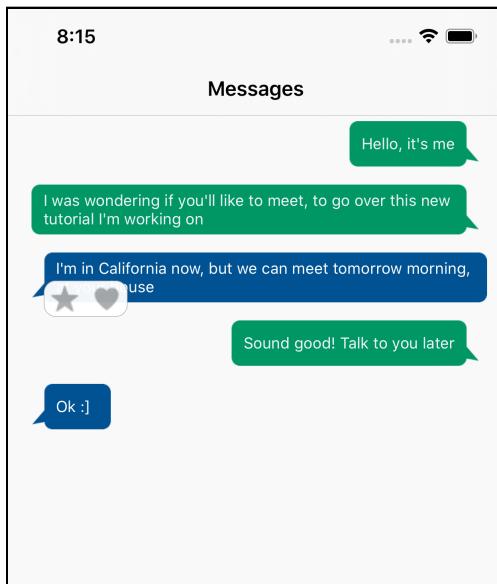
1. Ensure the message isn't one sent by the local user.
2. Change the value of `toolbarViewTopConstraint.constant` to `cell.frame.midY`.
3. Set an alpha of `0.95` to `toolbarView`.
4. Update the buttons of `toolbarView`. For example, if the message was already liked, the button is filled in.
5. Set `toolbarView.tag` to equal the `indexPath.row`. This allows you to identify the message.
6. Reproduce an animation. Don't focus too much on all of the parameters of the call to `UIView.animate(withDuration:delay:usingSpringWithDamping:initialSpringVelocity:options:animations:completion:)`. Later, you can change the duration, delay and other properties, giving you a different visual experience of the animation.

The call to `self.view.layoutIfNeeded()` forces an **Update Pass**, so Auto Layout has to satisfy the new constraints. The Update Pass is part of the Render Loop — the process responsible for rendering — and keeps the UI up to date using the constraints. If you want to know more about this process, refer to Chapter 15, “Optimizing Auto Layout Performance.”

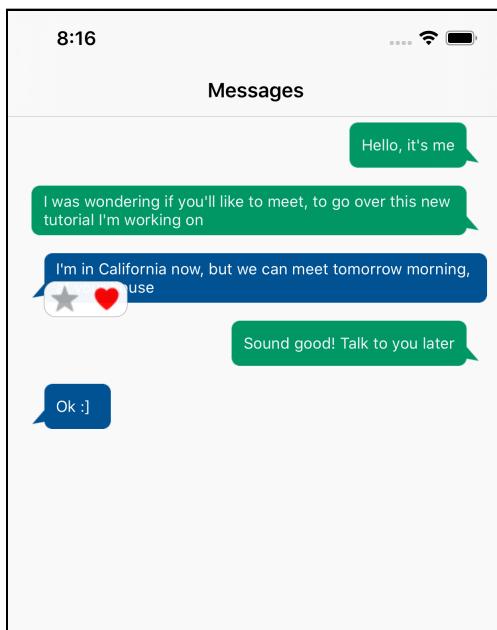
Build and run.



Double-tap one of the blue bubbles and the toolbar appears over the corresponding message.



Tap any of the buttons on the toolbar, and you'll see its icon gets filled in with a color.



At the moment, there's no way to hide the toolbar after it appears. To implement the hide functionality, add the following code after `setupToolbarView()`:

```
@objc func hideToolbarView() {
    //1
    self.toolbarViewTopConstraint.constant = -100

    //2
    UIView.animate(
        withDuration: 1.0,
        delay: 0.0,
        usingSpringWithDamping: 0.6,
        initialSpringVelocity: 1,
        options: [],
        animations: {
            self.toolbarView.alpha = 0
            self.view.layoutIfNeeded()
        },
        completion: nil
    )
}
```

With this code, you:

1. Change the value of `constant` to `-100` on `toolbarViewTopConstraint`, causing `toolbarView` to move out of sight as it will now be `-100` points from the top.
2. Set up a call to `UIView.animate(withDuration:delay:usingSpringWithDamping:initialSpringVelocity:options:animations:completion:)`.
3. Set the `alpha` of `toolbarView` to `0`, causing the view to slowly disappear as part of the animation.
4. Call `self.view.layoutIfNeeded()` so that a layout pass is scheduled.

Next, add the following code at the end of `viewDidLoad()`:

```
let gesture = UITapGestureRecognizer(
    target: self,
    action: #selector(hideToolbarView))
gesture.numberOfTapsRequired = 1;
gesture.delegate = self
tableView.addGestureRecognizer(gesture)
```

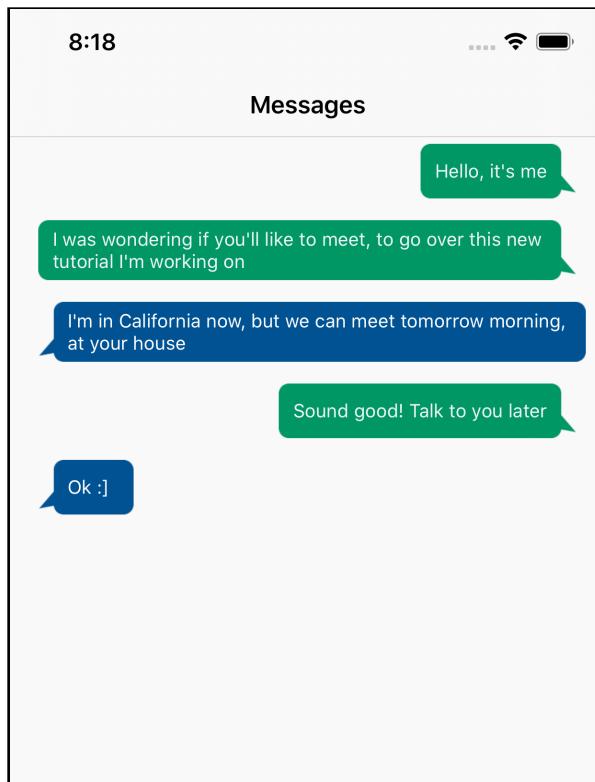
This code adds a gesture recognizer to `tableview`. When the user taps on the table view itself, `hideToolbarView()` is called. It also sets the delegate of the gesture to `self`.

To conform to this protocol, create a new extension at the bottom of the file, by adding the following code:

```
extension MessagesViewController: UIGestureRecognizerDelegate {  
    func gestureRecognizer(  
        _ gestureRecognizer: UIGestureRecognizer,  
        shouldReceive touch: UITouch  
    ) -> Bool {  
        return touch.view == tableView  
    }  
}
```

Here, you implement `gestureRecognizer(_:shouldReceive:)`. Notice that this method returns the result of comparing `touch.view` to `tableView`. This guarantees that `hideToolbarView()` gets called only when the user taps on the `tableView`, not any of its children.

Build and run. Tap any of the blue message bubbles and then tap something else. Notice how the toolbar gracefully disappears.



Keep the application running and try this:

1. Double-tap any of the blue bubbles, and then tap either the like or favorite button.
2. Tap outside the bubble.
3. Double-tap again over the previous bubble.

Sure enough, the state isn't updating. In other words, the previously selected option isn't getting saved.

Go to `MessagesViewController` and add the following line at the end of `setupToolbarView()`:

```
toolbarView.delegate = self
```

If you look at `ToolbarView.swift`, you'll see that it defines a delegate protocol which allows it to notify a delegate when the user taps either button in the toolbar. With the line above, you declare that `MessagesViewController` is the toolbar's delegate.

Next, add the following new extension at the bottom of `MessagesViewController.swift`:

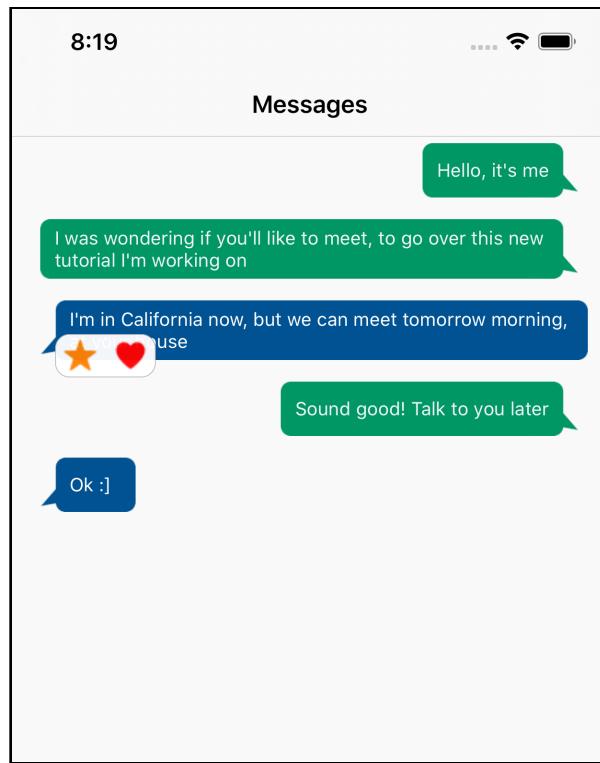
```
extension MessagesViewController: ToolbarViewDelegate {
    func toolbarView(
        _ toolbarView: ToolbarView,
        didFavoritedWith tag: Int
    ) {
        messages[tag].isFavorited.toggle()
    }

    func toolbarView(
        _ toolbarView: ToolbarView,
        didLikedWith tag: Int
    ) {
        messages[tag].isLiked.toggle()
    }
}
```

With this code, you implement the delegate methods for `toolbarView`. Now, when the user taps one of the buttons on the toolbar, the value of `isFavorite` and `isLiked` gets updated for that message. This happens thanks to the `tag` parameter, which represents the position of the specific message in `messages` array.

Build and run. Tap one of the blue bubble messages and tap either the Favorite or Like button. Now, tap outside of that message, and double-tap the same message again.





Notice, you're now saving the state.

Congratulations, you just created a better user experience by using constraint animations.

## Key points

- Remember, you can activate and deactivate constraints to create animations.
- Use animations to create a more engaging user experience.
- To force Auto Layout to satisfy the new constraints, call `layoutIfNeeded()` on the affected view.

# Chapter 10: Adaptive Layout

By Libranner Santos

Adaptability is all about guaranteeing a good user experience across all iOS devices and screen sizes. With so many options, it can be challenging to develop apps that look good on everything. Unfortunately, creating storyboards and views for each screen size and orientation doesn't scale well, so it's critical to build your apps with **adaptive user interfaces** that use **adaptive layouts**.

Adaptive apps rely on the trait system and trait collections. A **trait collection** is a set of traits and their respective values. A **trait** describes the current environment for your app. For example, traits can include layout direction, dynamic type size and size classes.

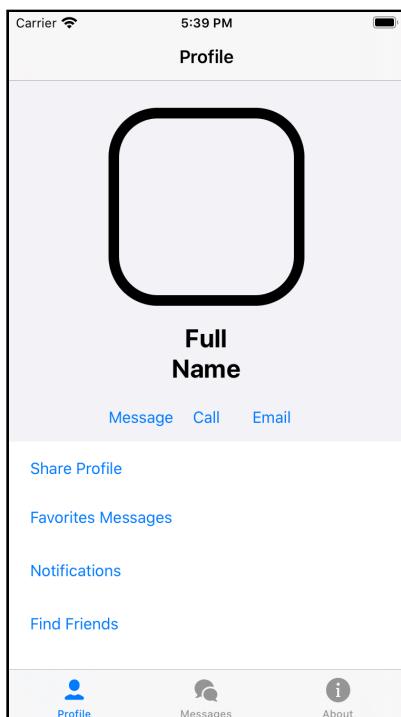
The main goal of adaptive layout is to allow you to create apps for all iOS devices without the need for device-specific code. In this chapter, you'll learn how to create adaptable apps by using size classes and adaptive images. Throughout this chapter, you'll use the tools that UIKit already provides. For more adaptive layout content, read Chapter 15, "Dynamic Type," and Chapter 16, "Internationalization and Localization."



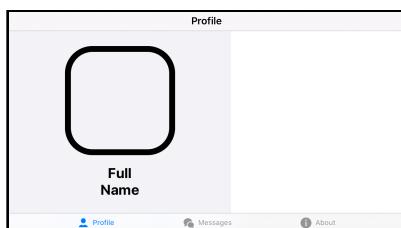
# One storyboard to run them all

Depending on the complexity of your app, you can use different strategies to accomplish adaptability. By using the right constraints, your screens can adapt gracefully to different screen sizes and orientations.

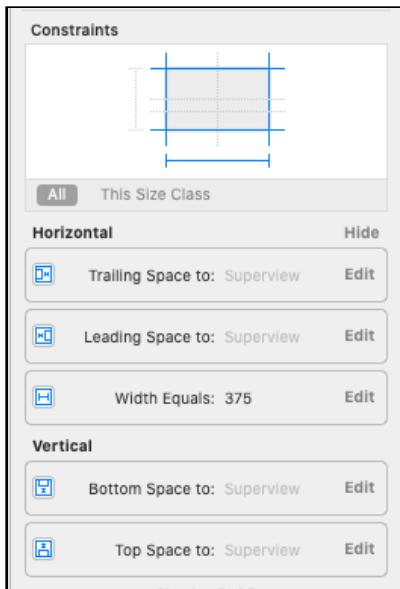
Go to the starter project and open the `MessagingApp` project. Select the **iPhone 8** simulator, then build and run.



The Profile screen looks good in portrait mode. Now, press **Command-Right Arrow** to switch the simulator orientation to landscape.



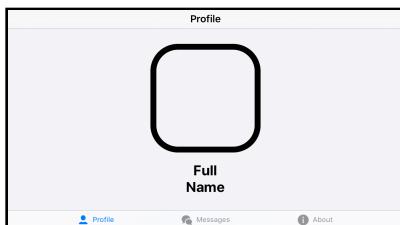
Notice the views aren't using all of the available width. Go back to Xcode, and open **Profile.storyboard**. In the document outline, look for **Profile Scene** and select **Main Stack View**. Open the Size inspector.



Look at the constraints; there's one for the width to make sure it's equal to **375**. Select that constraint and press **delete** to remove it. Since the available screen size isn't always going to be 375, it doesn't make sense to keep this constraint.

Select **Main Stack View** in the document outline, and **Control-drag to View**, which is located at the top of the document outline. On the modal window that pops up, select **Equal Widths**.

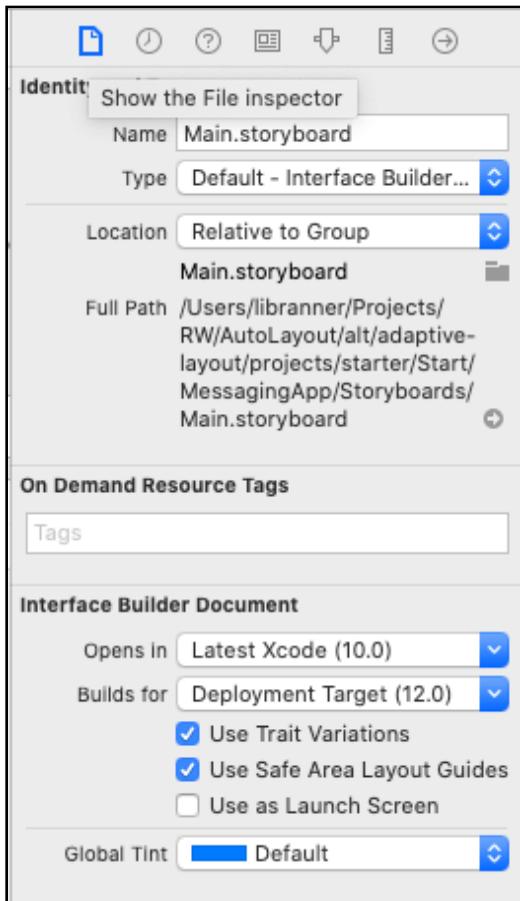
Now, build and run.



Rotate the device from portrait to landscape and then back to portrait. Notice how the screen adapts to the available width after deleting the width constraint.

## Setting up the storyboard

Go to **Main.storyboard**, and press **Command-Option-1** to show the File inspector. On the Interface Builder Document section, make sure that **Use Trait Variations** and **Use Safe Area Layout Guides** are both checked. Note that these options are selected by default in the latest versions of Xcode.

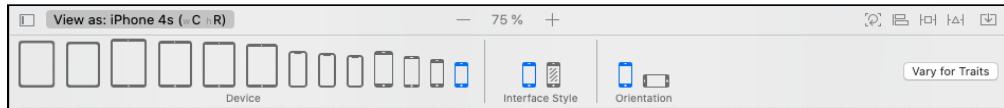


Here's what these options do:

- **Use Trait Variations** allows the storyboard to store data for more than one device family. You need this option enabled to create adaptive layouts.
- **Use Safe Area Layout Guides** makes the apps use the available space to draw layouts, respecting things like the notch on the latest iPhone devices.

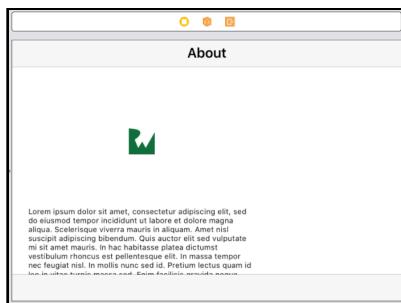
## Previewing layouts

On the bottom bar in Interface Builder, click **View as: <iOS Device>**. This action expands the Device Configuration Bar.



The **Device Configuration Bar** allows you to see how the user interface will look on different devices and conditions. In the Devices section, select **iPhone 4S**, which is the right-most icon shown in the Device area. Almost immediately, all of the screens in the storyboard will resize to represent how the interface will look on the selected device.

In the Orientation area, select **Landscape**. Choose the **About Scene**, and the layout will look like this:

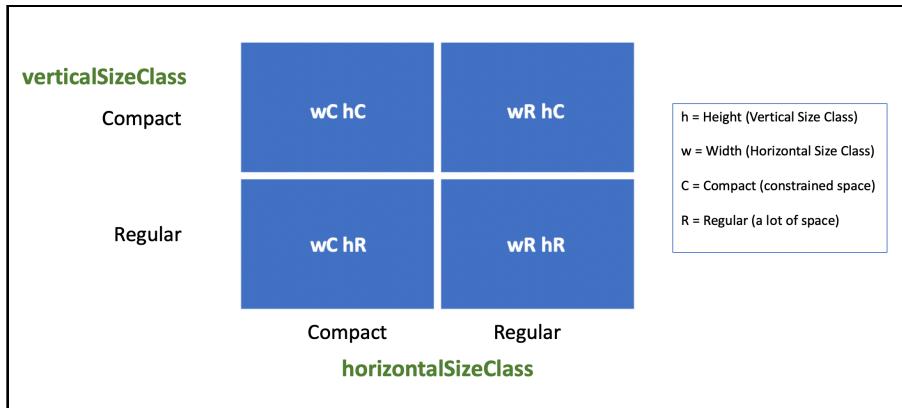


Notice how the text is getting cut off, and the layout isn't using all of the available space. To fix this problem and to make sure the interface looks good on multiple devices in either orientation, you'll use **size classes**.

## Size classes

When creating layouts, you should always think in terms of available space. Size classes make it possible to know how much available space there is by taking into account the device and the environment in which the app is running. For example, apps running on an iPhone 6 won't have the same available space as apps running on an iPad.

Size classes are attributes that represent the content area available using two traits: **horizontalSizeClass** and **verticalSizeClass**. These two traits can be either **regular** or **compact**. The regular value represents expansive space, meaning there's a fair amount of space available. The compact value represents constrained space, meaning there's not much space available. Using these two traits, you can have four possible combinations.



The use of size classes allows you to have more flexibility and awareness while creating user interfaces, which significantly reduces the need for device-specific code.

Here's a list of the values of size classes on different devices:

Device	Portrait Orientation	Landscape Orientation
12.9" iPad Pro	Regular width, regular height	Regular width, regular height
11" iPad Pro	Regular width, regular height	Regular width, regular height
10.5" iPad Pro	Regular width, regular height	Regular width, regular height
9.7" iPad	Regular width, regular height	Regular width, regular height
7.9" iPad mini 4	Regular width, regular height	Regular width, regular height
iPhone XS Max	Compact width, regular height	Regular width, compact height
iPhone XS	Compact width, regular height	Compact width, compact height
iPhone XR	Compact width, regular height	Regular width, compact height
iPhone X	Compact width, regular height	Compact width, compact height
iPhone 8 Plus	Compact width, regular height	Regular width, compact height
iPhone 8	Compact width, regular height	Compact width, compact height
iPhone 7 Plus	Compact width, regular height	Regular width, compact height
iPhone 7	Compact width, regular height	Compact width, compact height
iPhone 6s Plus	Compact width, regular height	Regular width, compact height
iPhone 6s	Compact width, regular height	Compact width, compact height
iPhone SE	Compact width, regular height	Compact width, compact height

## Multitasking and size classes

As you can see in the previous table, for the iPad, you usually have regular width and height. This changes when the system is using a split view since there's less space available.

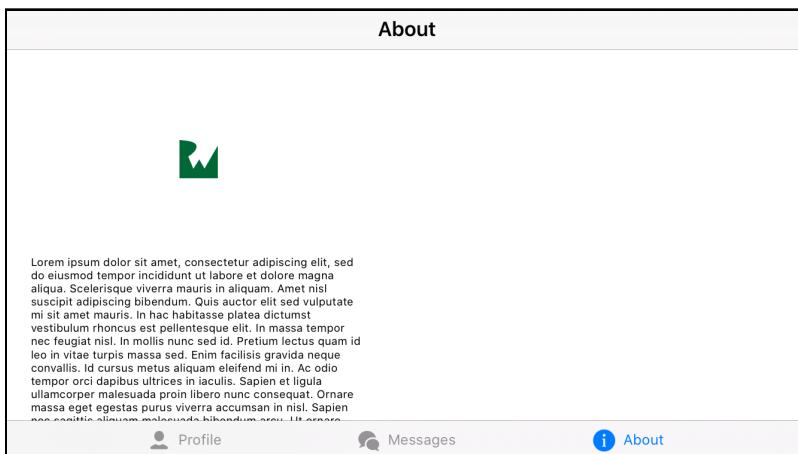
The following example shows how a split view can change the size classes:

Device	Mode	Portrait orientation	Landscape orientation
12.9" iPad Pro	2/3 split view	Compact width, regular height	Regular width, regular height
	1/2 split view	N/A	Regular width, regular height
	1/3 split view	Compact width, regular height	Compact width, regular height
7.9" iPad mini 4	2/3 split view	Compact width, regular height	Regular width, regular height
	1/2 split view	N/A	Compact width, regular height
	1/3 split view	Compact width, regular height	Compact width, regular height

## Working with size classes

Go back to Xcode. Open **Main.storyboard** and select the **About Scene**. The view contains two elements: an image view and a label — neither have constraints. Build and run to see the About Scene.

Go to the About tab, and rotate the device using **Command-Right Arrow**.



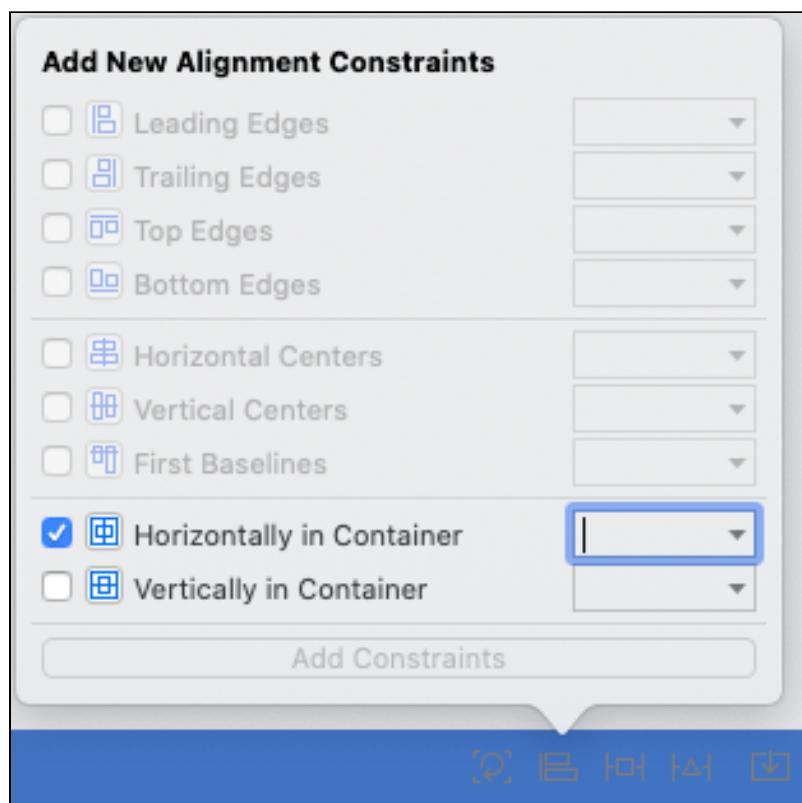
Notice the user interface isn't correct. As it turns out, in this specific case, adding constraints isn't enough because the final product should adapt depending on the orientation of the device.

On the Device Configuration Bar, set the device back to **iPhone 8** in portrait. Click **Vary for Traits** from the popup that appears and select both options: width and height. The bar will turn blue, like this:

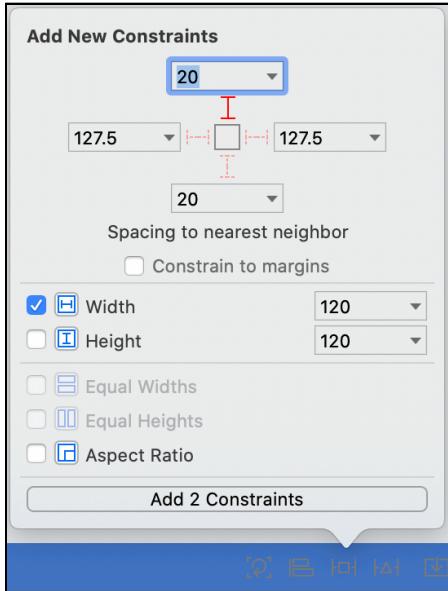


On the left side, you can see an icon for the device you've selected on the **View as**. Click this icon to see the list of devices that will be affected by the constraints you're about to create.

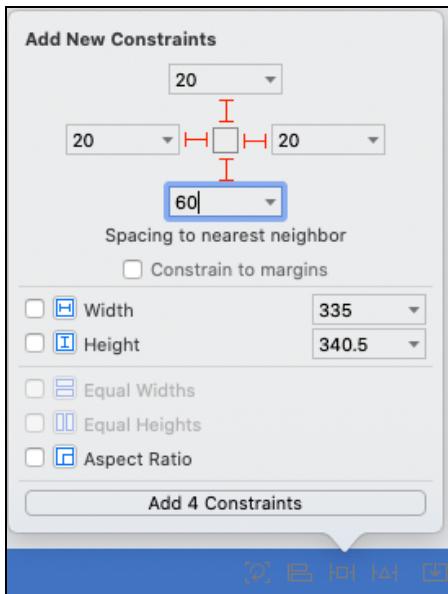
Now you can add constraints to the image. Select the image and then use **Align** to select **Horizontally in Container**:



Next, set the top constraint to **20**, and the width to **120**. Make sure **Constrain to margin** is unchecked, and click **Add 2 Constraints**.

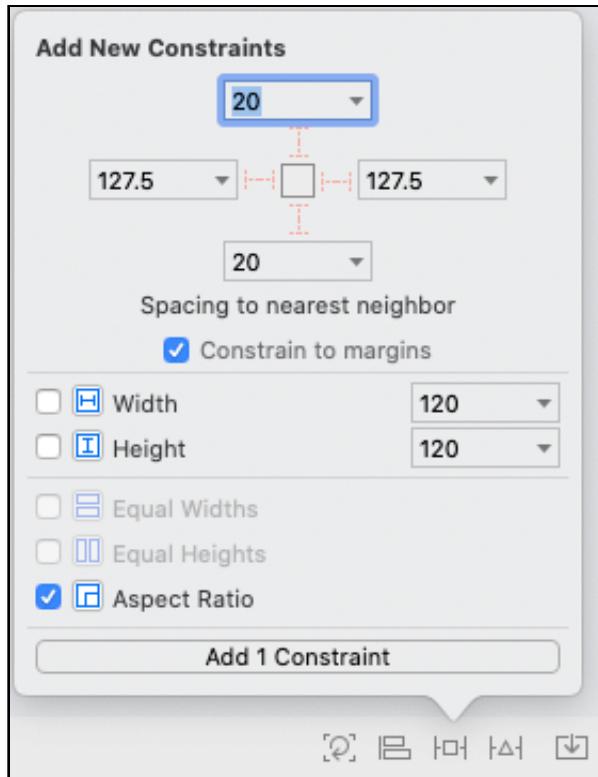


Now, select the label. Click the **Pin** menu. Make the top, leading and trailing constraints equal to **20**, and set the bottom to **60**. Make sure **Constrain to margin** is unchecked.



Add those four constraints, and then click **Done Varying** on the bottom bar.

Select the image view, and using the **Pin** menu, create a constraint for the **aspect ratio**, and click **Add 1 Constraint**.



Since this constraint was created without any traits specified, it will affect all size classes.

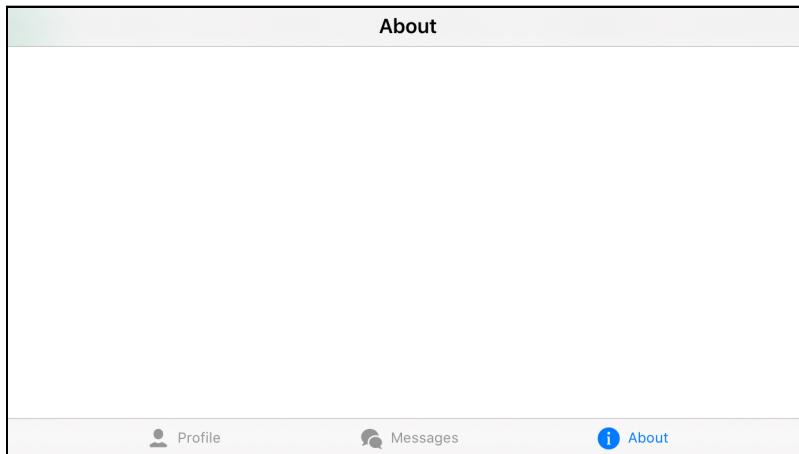
Verify that the value for the aspect ratio is **1:1**. You can do this by selecting the constraint in the document outline and modifying the multiplier value in the Size inspector.



Build and run.

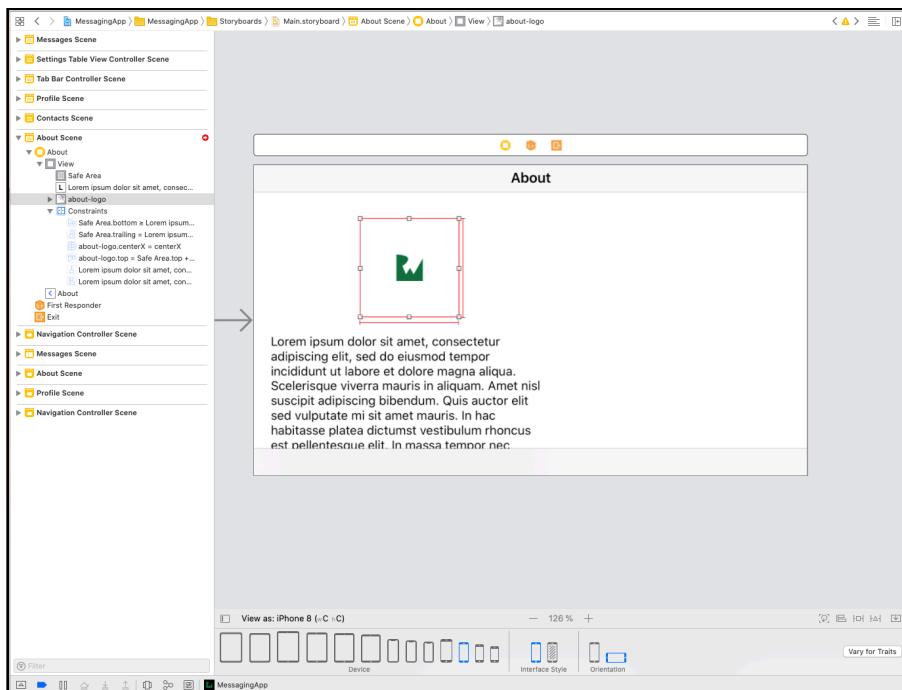


Excellent, the app now looks good in portrait mode; however, try switching to landscape using **Command-Right Arrow**, and you'll see that the views disappear.

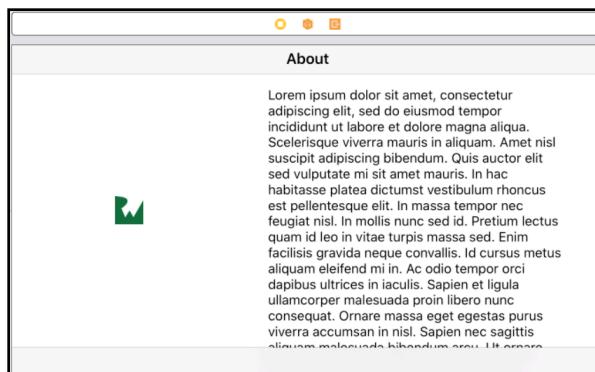


The constraints you created are **installed** only for Compact Width and Regular Height size classes. The iPhone 8 in portrait mode has that size class, but in landscape, the device screen changes to Compact Width and Compact Height.

Open **Main.storyboard**, and on the Device Configuration Bar, select landscape orientation.



The constraints in the document outline look faded; this means they're not installed for the current size class. Move the image view and the label, so they're side-by-side horizontally.

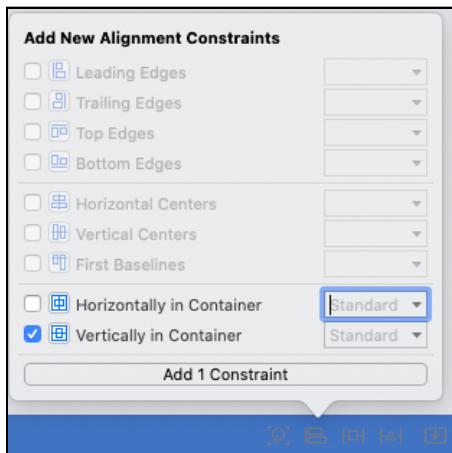


On the right side of the Device Configuration Bar, click **Vary for Traits**. From the popup that appears, select both options: width and height.

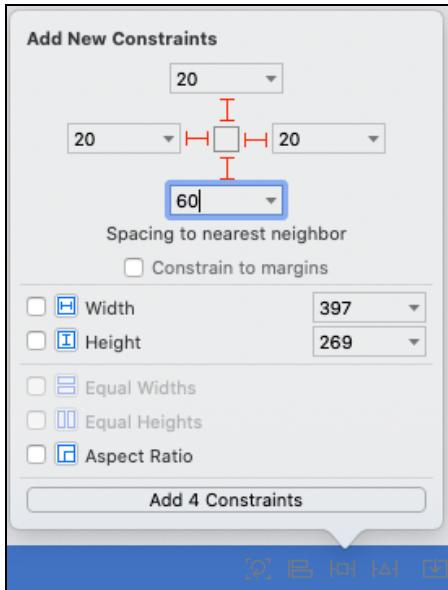
Select the image view and create the constraints using the **Pin** menu. Make sure **Constrain to margin** is unchecked, and then create the following constraints for the **leading edge** and **width**. When you're done, click **Add 2 Constraints**:



Using the **Align** menu, select **Vertically in Container** for the image view.

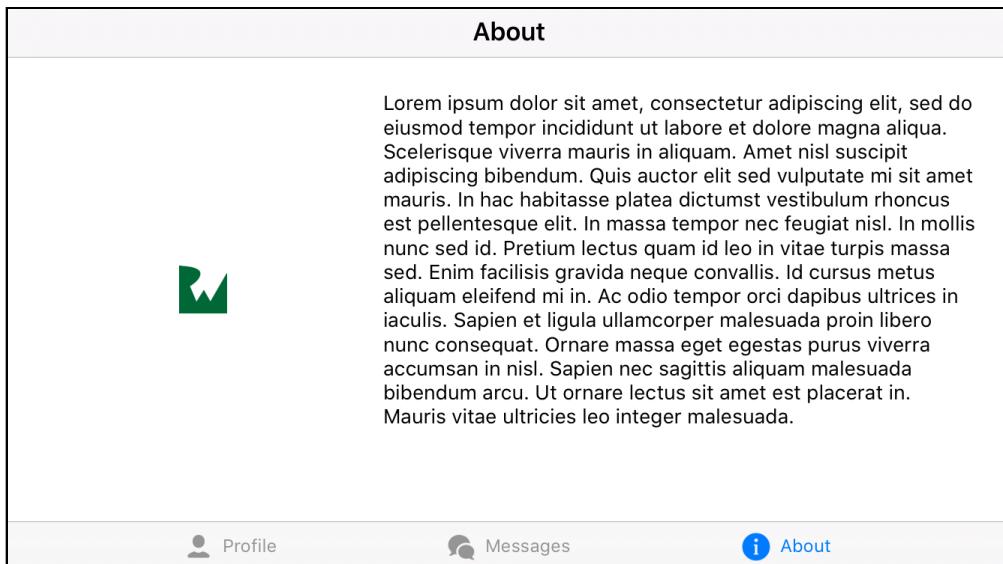


Now, select the label and create and set the top, leading, and trailing constraints, so they're equal to **20**. Set the bottom constraint equal to **60** using the **Pin** menu.



Click **Done Varying** on the bottom bar.

Build and run.

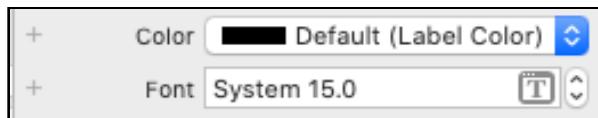


The app now adapts perfectly no matter the orientation.

## Changing properties

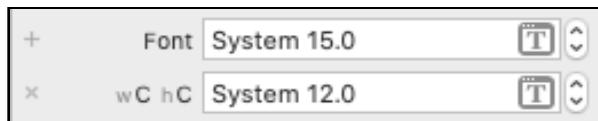
Apart from constraints, you can also change the value of some properties using size classes.

On **Main.storyboard**, go to the **About Scene** and select the label. Now, look at the Attributes inspector.



The **Color** and **Font** properties both have the plus sign on the left side, which means they can have variations depending on the size classes.

Click the **plus button** on the left side of the **Font** property. Select compact for width and height variations, and then **Add Variation**. This creates a new element containing the value for the font property for the specified variation. Change the font size to **12**. The Attributes inspector will look like this:



Build and run. Rotate the device and see how the font size for the label changes depending on the orientation.

## Trait environment and trait collections

Every time a device changes its orientation, traits containing the new configuration are propagated throughout the app from the screen to the presented view.

You can respond to these changes in code using `traitCollectionDidChange(_ :)`; this is called on any object that conforms to `UITraitEnvironment`.

Open **AboutViewController.swift** and add the following code inside of the class:

```
private func setupContactUsButton(  
    verticalSizeClass: UIUserInterfaceSizeClass  
) {  
    //1  
    NSLayoutConstraint.deactivate(contactButtonConstraints)  
  
    //2  
    if verticalSizeClass == .compact {
```

```
//3
contactUsButton.setTitle("Contact Us", for: .normal)
//4
contactButtonConstraints = [
    contactUsButton.widthAnchor.constraint(
        equalToConstant: 160),
    contactUsButton.heightAnchor.constraint(
        equalToConstant: 40),
    contactUsButton.trailingAnchor.constraint(equalTo:
        view.safeAreaLayoutGuide.trailingAnchor,
        constant: -20),
    contactUsButton.bottomAnchor.constraint(equalTo:
        view.safeAreaLayoutGuide.bottomAnchor,
        constant: -10),
]
} else {
//5
contactUsButton.setTitle("", for: .normal)
//6
contactButtonConstraints = [
    contactUsButton.widthAnchor.constraint(
        equalToConstant: 40),
    contactUsButton.heightAnchor.constraint(
        equalToConstant: 40),
    contactUsButton.centerXAnchor.constraint(
        equalTo: view.centerXAnchor),
    contactUsButton.bottomAnchor.constraint(equalTo:
        view.safeAreaLayoutGuide.bottomAnchor,
        constant: -10),
]
}
//7
NSLayoutConstraint.activate(contactButtonConstraints)
}
```

With this code, you:

1. Deactivate all the constraints in `contactButtonConstraints`.
2. Check if `verticalSizeClass` is equal to `.compact`.
3. Change the `contactUsButton` button title to Contact Us.
4. Make `contactButtonConstraints` equal to the set of constraints determined for this configuration. In this case, width equal to 160, height equal to 40, trailing equal to -20 from `safeAreaLayoutGuide.trailingAnchor`, and bottom equal to -10 from `safeAreaLayoutGuide.bottomAnchor`.

5. In case `verticalSizeClass` is not equal to `.compact`. Set the title to an empty string.
6. Make `contactButtonConstraints` equal to the set of constraints determined for this configuration.
7. Activate the constraints in `contactButtonConstraints`.

Below `setupContactUsButton(verticalSizeClass:)`, add the following code:

```
override func traitCollectionDidChange(_  
    previousTraitCollection: UITraitCollection?  
) {  
    //1  
    super.traitCollectionDidChange(previousTraitCollection)  
    //2  
    if traitCollection.verticalSizeClass !=  
        previousTraitCollection?.verticalSizeClass {  
        //3  
        setupContactUsButton(  
            verticalSizeClass: traitCollection.verticalSizeClass)  
    }  
}
```

Here's what you did:

1. Call `super.traitCollectionDidChange(previousTraitCollection)` so that elements in the higher hierarchy stay up to date with the changes.
2. Check if the `verticalSizeClass` for the current `traitCollection` is different from the previous one.
3. Call `setupContactUsButton(verticalSizeClass:)`.

Go to `viewDidLoad()`, and add these two lines at the end:

```
//1  
view.addSubview(contactUsButton)  
  
//2  
setupContactUsButton(  
    verticalSizeClass: traitCollection.verticalSizeClass)
```

With this code, you:

1. Add `contactUsButton` to the view so that it can be part of the view hierarchy.
2. Call `setupContactUsButton(verticalSizeClass:)` so that `contactUsButton` is properly set up the first time the view appears.

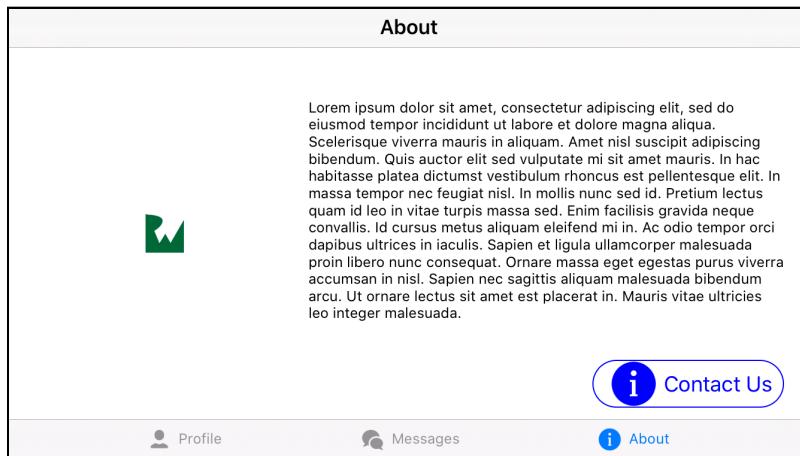


Build and run. Rotate the device, and check how the contact us button at the bottom is displayed differently depending on the device configuration.

Here it is in portrait mode:



Here it is in landscape mode:

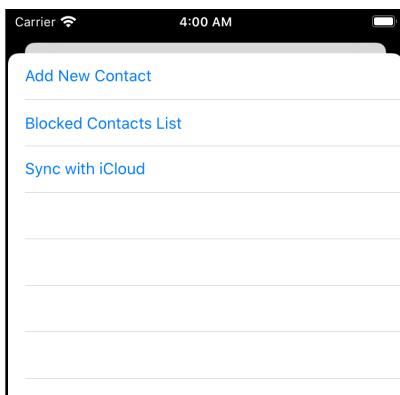


# Adaptive presentation

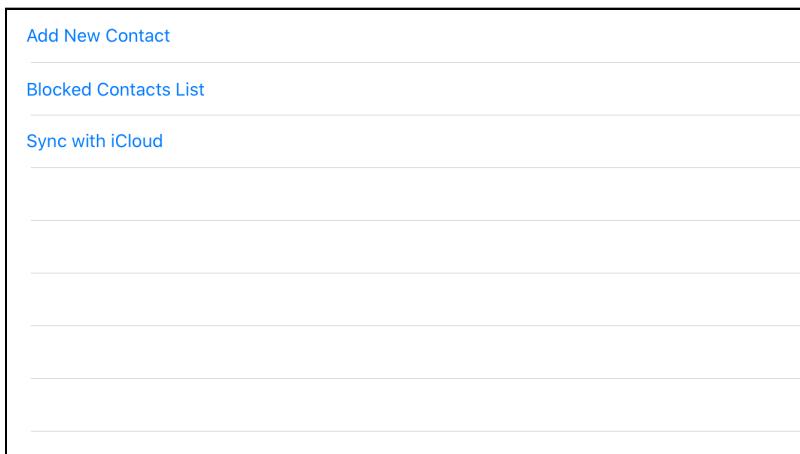
A view controller can be presented in different ways, depending on the environment. By default, the system will try to accommodate the view controller, but you can decide how you want your view controller to adapt.

Build and run.

Go to the **Messages** tab and tap **Options**.

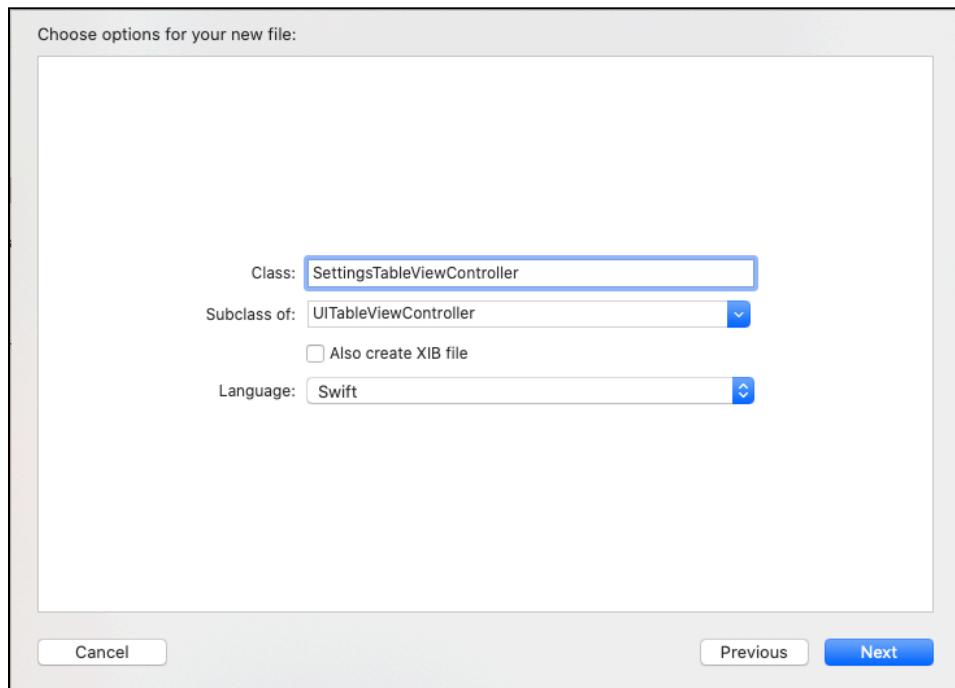


A sheet with some options appears; you can hide it by dragging the view to the bottom. Now, tap **Options** again and put the device in landscape using **Command-Right Arrow**.



There's no way for you to dismiss the view controller while the device is in landscape — time to fix that.

Right-click over the **Controllers** folder and select **New file....** Choose **Cocoa Touch Class** and click **Next**. Set **Class** to **SettingsTableViewController**. Set **Subclass of** to **UITableViewController**. Set the **Language** to **Swift**. Also, verify that **create XIB files** is unchecked, then click **Next**. Finally, click **Create**.



Open **SettingsTableViewController.swift** and remove everything except the class declaration. When you're done, your code will look like this:

```
import UIKit

class SettingsTableViewController: UITableViewController {
```

Type this inside the class:

```
//1
override func awakeFromNib() {
    super.awakeFromNib()

//2
navigationItem.leftBarButtonItem = UIBarButtonItem(
    barButtonSystemItem: .done,
    target: self,
    action: #selector(dismissModal))
```

```
//3  
modalPresentationStyle = .popover  
//4  
popoverPresentationController!.delegate = self  
}
```

Here's what you did:

1. Override `awakeFromNib`. You want to execute code before the view controller loads.
2. Set `leftBarButtonItem` of `navigationItem` equal to an instance of `UIBarButtonItem` with an action that calls `dismissModal()`.
3. Set the value of `modalPresentationStyle` to `.popover` so that you have a default presentation style.
4. Set the view controller as the `popoverPresentationController` delegate.

Below the code you just added, add this:

```
@objc private func dismissModal() {  
    dismiss(animated: true)  
}
```

This new function gets called when the button on the navigation bar is tapped. It calls `dismiss(animated: true)` on the view controller.

Next, add the following new extension outside of the class:

```
extension SettingsTableViewController:  
UIPopoverPresentationControllerDelegate {  
//1  
func adaptivePresentationStyle(  
    for controller: UIPresentationController  
) -> UIModalPresentationStyle {  
//2  
switch (  
    traitCollection.horizontalSizeClass,  
    traitCollection.verticalSizeClass) {  
//3  
case (.compact, .compact):  
    return .fullScreen  
default:  
    return .none  
}  
}
```



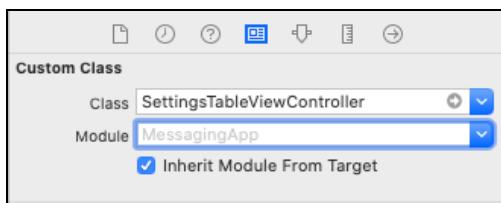
```
//4
func presentationController(
    controller: UIPresentationController,
    viewControllerForAdaptivePresentationStyle
    style: UIModalPresentationStyle
) -> UIViewController? {
//5
    return UINavigationController(
        rootViewController: controller.presentedViewController)
}
```

Here's what you did:

1. Implement `adaptivePresentationStyle(for:)` to choose the modal presentation style depending on the size classes.
2. Create a switch statement using the horizontal and vertical size classes of `traitCollection`.
3. When both size classes are `compact`, return `fullscreen` as the presentation style; otherwise, return `none` so that the default presentation style is used, which in this case, is `popover` as indicated in `awakeFromNib`.
4. Implement  
`presentationController(_:viewControllerForAdaptivePresentationStyle:)`. This allows you to return a different view controller than the one being presented.
5. Return a `UINavigationController` whose root view controller is the `presentedViewController`. Thanks to this, the controller will have a navigator bar, where the `UIBarButtonItem` you created in `awakeFromNib` will appear. Note that this navigation view controller is ignored when the presentation mode is `popover`.

Open **Main.storyboard** and look for the table view controller containing the options — it's connected through a segue to the Contacts Scene.

Select **Table View Controller**, and in the Identity inspector, set **Class** to **SettingsTableViewController**. Now the view controller uses the class you set up with the `UIPopoverPresentationControllerDelegate`.



Build and run.

Go to the **Messages** tab. Tap **Options** and the sheet appears just like before. Tap outside of the popover to dismiss it, then put the simulator in landscape mode using **Command-Right Arrow**. Tap **Options** again.



There's now a navigation bar at the top, and you can close the view controller by tapping **Done**.

## UIKit and adaptive interfaces

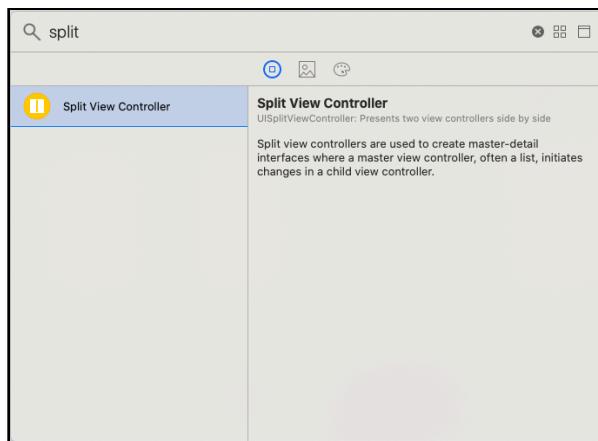
UIKit provides tools to make adaptable user interfaces. Some of these tools include:

- Split view controller
- Layout guides
- UIAppearance proxy

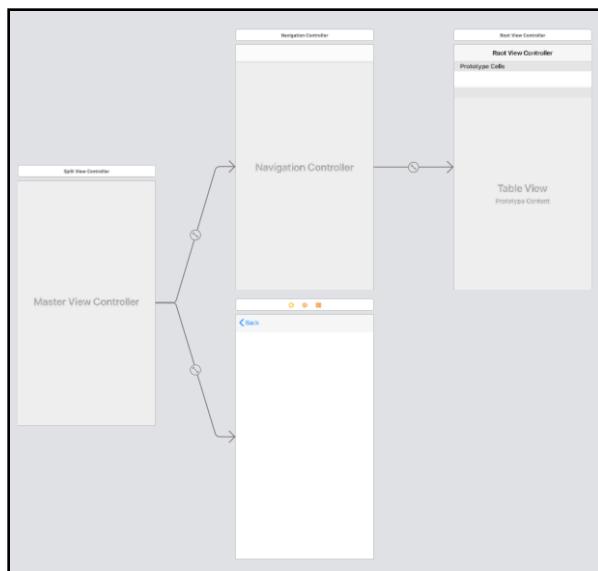
## The split view controller

The **split view controller** acts as a container view controller that manages two child view controllers. If you've used the Settings app on an iPad, you may have noticed that the experience is different from what you have on an iPhone 6, for example. The app changes to display the information in a master-detail configuration. This happens thanks to the split view controller.

Go to **Main.storyboard**. Press **Command-Shift-L**, and type **split** into the search bar.



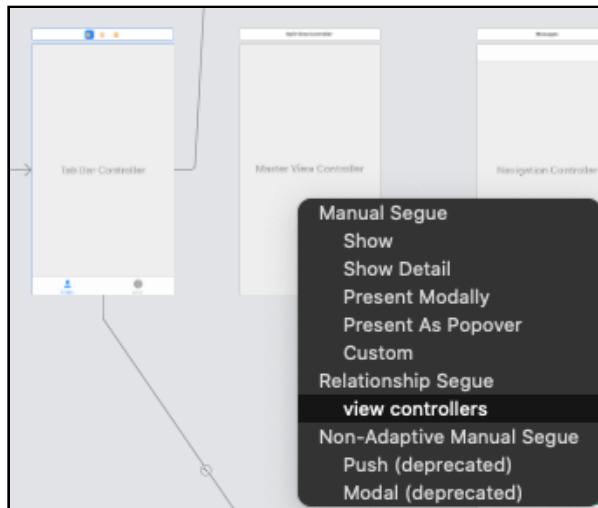
Drag a **Split View Controller** into the storyboard.



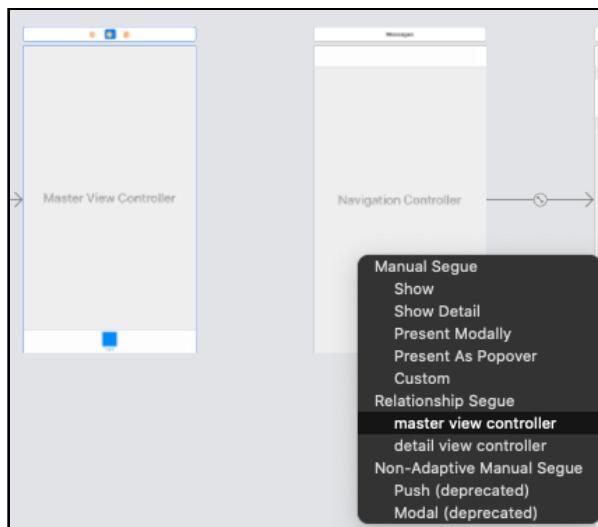
Remove all of the *newly added* view controllers except the **Master View Controller**.

Remove the connection between the **Tab Bar Controller** and the **Navigation Controller** connected to the **Contacts Scene**.

**Control-drag** from the **Tab Bar Controller** to the **Master View Controller**, and select **view controllers**.



**Control-drag** from the **Master View Controller** to the **Navigation Controller** connected to the **Contacts Scene**, and select **master view controller**.



Look for the **Messages Scene** and remove the segue going to it. Embed the scene in a navigation view controller using **Editor** ▶ **Embed In** ▶ **Navigation Controller**.

**Control-drag** from the **Master View Controller** to the navigation controller you just created, and select **detail view controller** on the popup that appears.

In the Project navigator, **right-click** over the **Controllers** folder and click **New file...**. Select **Cocoa Touch Class**, and click **Next**. Set **Class** to **SplitViewController**. Set **Subclass of** to **UISplitViewController**, and the **Language** to **Swift**. Also, ensure that **create XIB files** is unchecked, then click **Next**. Finally, click **Create**.

Open **SplitViewController.swift**, and replace everything in the class with the following code:

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

    //1
    guard
        let leftNavController =
            viewControllers.first as? UINavigationController,
        let masterViewController =
            leftNavController.viewControllers.first
            as? ContactListViewController,
        let detailViewController = (viewControllers.last
            as? UINavigationController)?.topViewController
            as? MessagesViewController
    else { fatalError() }

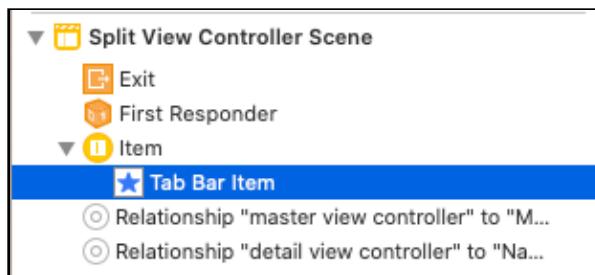
    //2
    let firstContact = masterViewController.contacts.first
    detailViewController.contact = firstContact
    //3
    masterViewController.delegate = detailViewController
    //4
    detailViewController.navigationItem
        .leftItemsSupplementBackButton = true
    detailViewController.navigationItem
        .leftBarButtonItem = displayModeButtonItem
}
```

With this code, you:

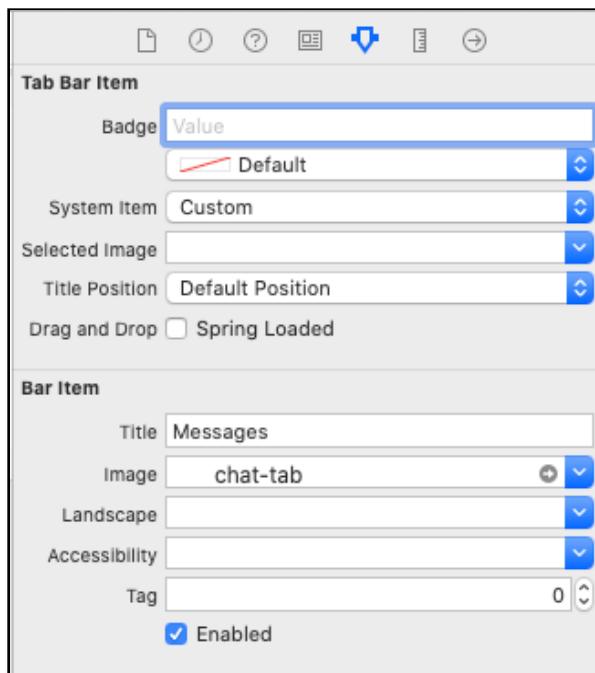
1. Get the master view controller and detail view controller from the `viewControllers` property of the split view controller.
2. By default, the split view controller will show the first contact. You obtain it by calling `first` on the `contacts` array of the `masterViewController`.

3. Set `masterViewController` delegate to `detailViewController`.
4. Make `detailViewController` replace its left navigation item with a button that will toggle the display mode of the split view controller. This button will be visible only on iPad; you'll get a button in the top left to toggle the table view display.

Go to **Main.storyboard**. Select **Master View Controller** and press **Command-Option-4** to show the Identity inspector. Set **Class** to **SplitViewController**. In the document outline, select the tab bar item.



In the Attributes inspector, set **Title** to **Messages**, and set **Image** to **chat-tab**.



In the Tab Bar Controller, **click-drag** the Messages tab bar item to the middle.

There's only one thing missing: You need to wire the master and detail view so that when a user taps a contact, the corresponding messages appear.

Open **ContactListTableViewController.swift**, and add this code before the class declaration:

```
protocol ContactListTableViewControllerDelegate: class {
    func contactListTableViewController(
        _ contactListTableViewController:
        ContactListTableViewController,
        didSelectContact selectedContact: Contact
    )
}
```

You'll implement this protocol on the detail view controller so that it can display the proper messages when the selected contact changes.

Add the following code to the top of the class:

```
weak var delegate: ContactListTableViewControllerDelegate?
```

This code declares the `delegate` property.

Now, replace the `tableView(_:didSelectRowAt:)` implementation with the following:

```
override func tableView(
    _ tableView: UITableView,
    didSelectRowAt indexPath: IndexPath
) {
    //1
    guard
        let messagesViewController =
            delegate as? MessagesViewController,
        let messagesNavigationController =
            messagesViewController.navigationController
    else {
        return
    }

    //2
    let selectedContact = contacts[indexPath.row]
    messagesViewController.contactListTableViewController(
        self,
        didSelectContact: selectedContact)

    //3
    splitViewController?.showDetailViewController(

```



```
    messagesINavigationController,
    sender: nil)
}
```

This code:

1. Checks that the delegate is not `null` and gets a reference to the navigation controller containing `messagesViewController`.
2. Calls `contactListTableViewController(_:didSelectContact:)` on the delegate passing the selected contact.
3. Calls `showDetailViewController` on `splitViewController`, passing `messagesINavigationController`. This makes the split view controller show the detail view.

You can remove `prepare(for:sender:)` since you won't need it anymore.

Open **MessagesViewController.swift**, and at the bottom, add the following extension:

```
extension MessagesViewController: ContactListTableViewControllerDelegate {
    func contactListTableViewController(
        _ contactListTableViewController:
        ContactListTableViewController,
        didSelectContact selectedContact: Contact
    ) {
        contact = selectedContact
    }
}
```

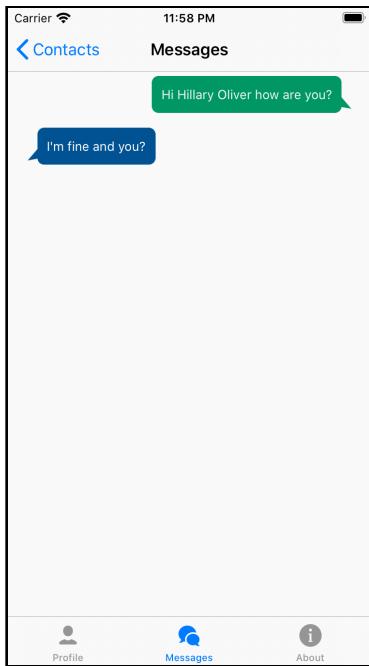
Here, you implement `ContactListTableViewControllerDelegate`. This implementation ensures the `contact` property gets the selected contact. The property `contact` has a `didSet` observer that will reload the table view with the corresponding data.

Build and run.

Select the messages tab on the tab bar. By default, the split view controller shows the first detail item. Tap the back button so that you can see the master view with the contacts list.

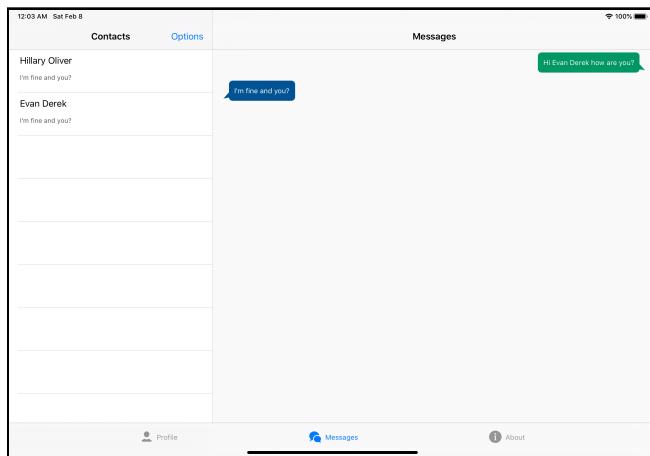


You can tap any of the contacts and see how it works.



Stop the simulator, and select the **iPad Pro (11-inch)** as the simulator. Build and run.

Go to the messages tab. Tap the back button and see how you get a different user interface. Rotate the device using **Command-Right Arrow**. Now you get a master-detail view similar to the one used in the Settings app or the Mail app.



This functionality is all possible thanks to the **split view controller** — and you got it all almost for free. Pretty cool! :]

## Use your layout guides

The system comes with predefined layout guides that can make apps adapt better to different devices. One clear example is the **Safe Area Layout Guide** that helps prevent content from getting behind the iPhone X notch. You can learn more about this in Chapter 7, “Layout Guides”.

## UIAppearance

UIAppearance serves as a proxy to have access to the mutable appearance of some classes, like UINavigationBar, UIButton and UIBarButtonItem. By changing the attributes for these classes, you can create consistent themes that you can use throughout the app.

Open `AppDelegate.swift`, and in `application(_:didFinishLaunchingWithOptions:)`, before `return true`, add this:

```
//1
let verticalRegularTrait =
    UITraitCollection(verticalSizeClass: .regular)
//2
let regularAppearance =
    UINavigationBar.appearance(for: verticalRegularTrait)
let regularFont = UIFont.systemFont(ofSize: 20)
//3
regularAppearance.titleTextAttributes =
    [NSAttributedString.Key.font: regularFont]

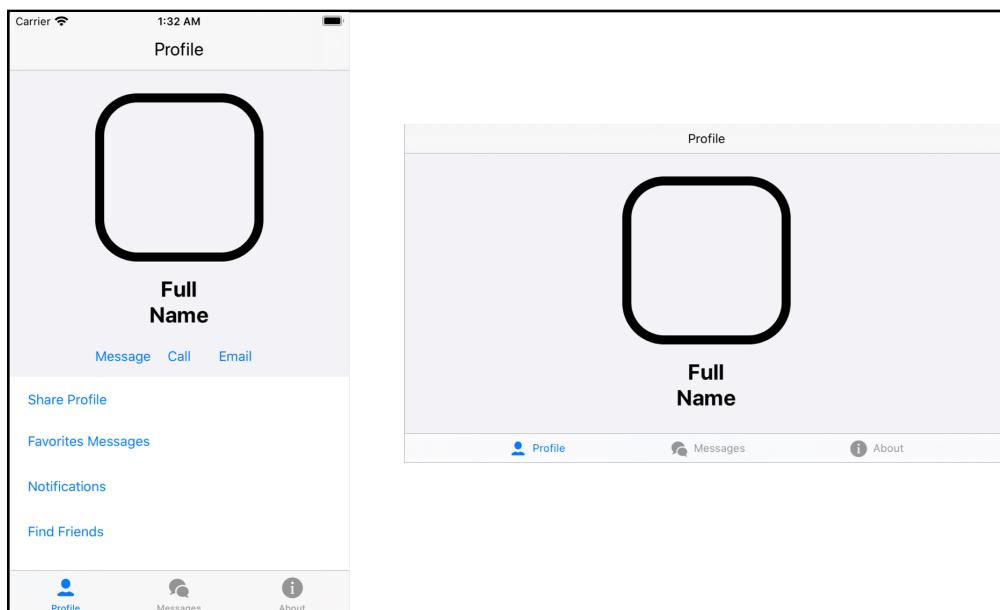
//4
let verticalCompactTrait =
    UITraitCollection(verticalSizeClass: .compact)
//5
let compactAppearance =
    UINavigationBar.appearance(for: verticalCompactTrait)
let compactFont = UIFont.systemFont(ofSize: 14)
//6
compactAppearance.titleTextAttributes =
    [NSAttributedString.Key.font: compactFont]
```

This code:

1. Creates a new trait collection with a regular vertical size class.
2. Grabs a reference to the appearance for the `NavigationBar` for the previously declared trait collection `verticalRegularTrait`.
3. Sets `titleTextAttributes` so that it uses the `regularFont`. This will make the font size 20, when the screen has a lot of space available vertically — for example, an iPhone 8 in portrait mode.
4. Creates a new trait collection with a compact vertical size class.
5. Grabs a reference to the appearance for the `NavigationBar` for `verticalCompactTrait`.
6. Sets `titleTextAttributes` so that it used the `compactFont`. This will make the font size 14 when the screen has little space available vertically — for example, an iPhone 8 in landscape mode.

Build a run. Make sure to select the **iPhone 8** as the simulator.

Rotate the device to see how the navigation bar title looks bigger in portrait mode and smaller in landscape mode.



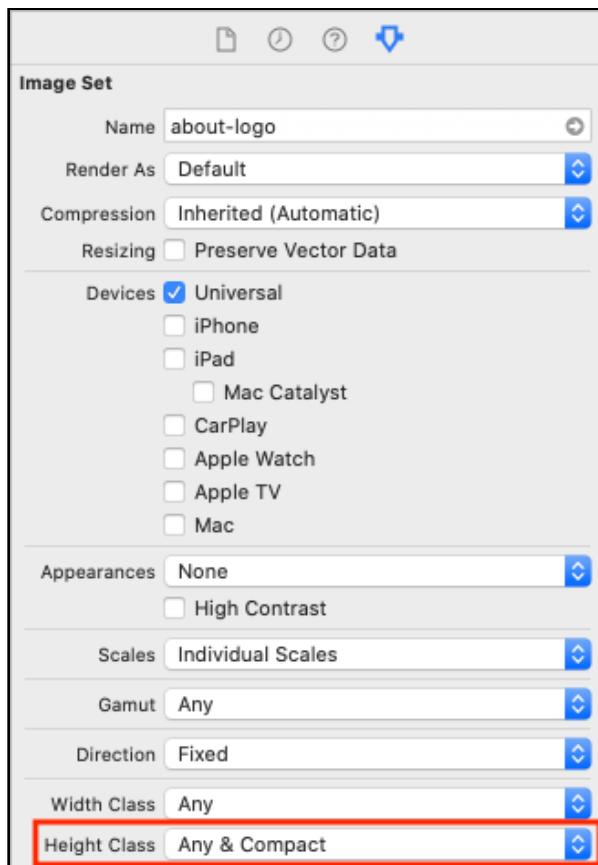
## Adaptive images

Image assets should be adaptive, too. In this section, you'll use Asset Catalogs to manage images and provide different versions of them depending on the size class. Also, you'll explore how the alignment and slicing tool can help you select parts of an image and indicate how an image should resize when necessary.

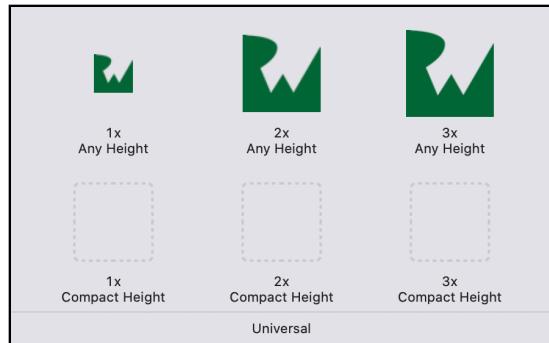
## Images and traits

Asset catalogs give you the possibility of having multiple images depending on the trait environment. You can have different image assets for different size classes.

Back in Xcode, open **Assets.xcassets** and select **about-logo**. In the Attribute inspector, set **Height Class** to **Any & Compact**.



This will add three slots on your image set. These new slots allow you to add images for when the height trait is compact.



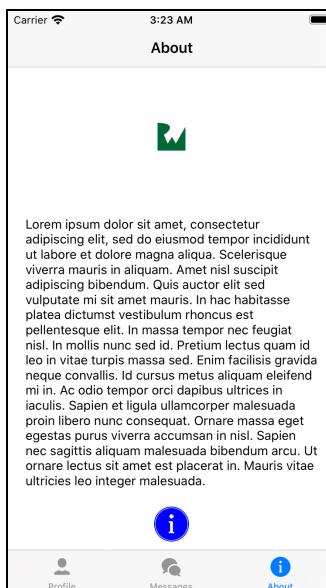
The already existing images have the label **Any Height**; this means they'll be used for any other configurations.

Look for the **assets** folder — it's in the same directory as the **starter** and **final** folders. Drag all of the images, individually, to their designated slots.

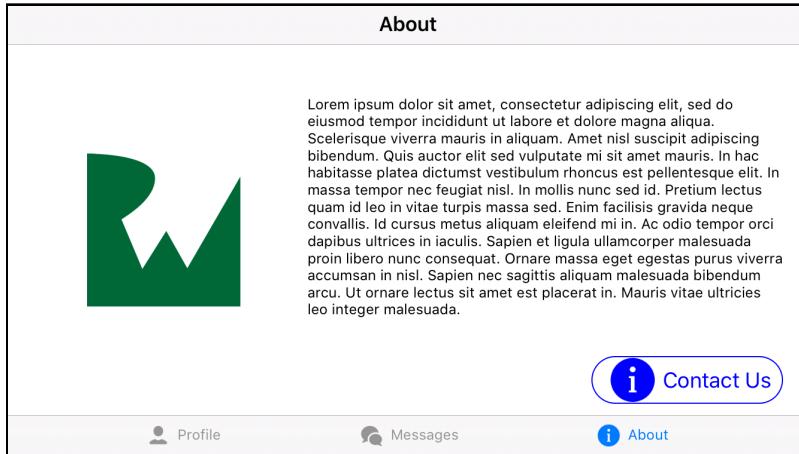
Build and run.

Go to the **About** tab and rotate the device. When the device is in landscape, the images are bigger; when in portrait, the images are smaller.

Here's how the app looks in portrait:



Here's how the app looks in landscape:



Great! You've used the power of asset catalogs and size classes to create a better user interface.

## Alignment insets and slicing

Using the Asset Catalog, you can indicate which parts of an image you want to use so that your app looks good in different scenarios. For this, you need to use **alignment insets** and **slicing**.

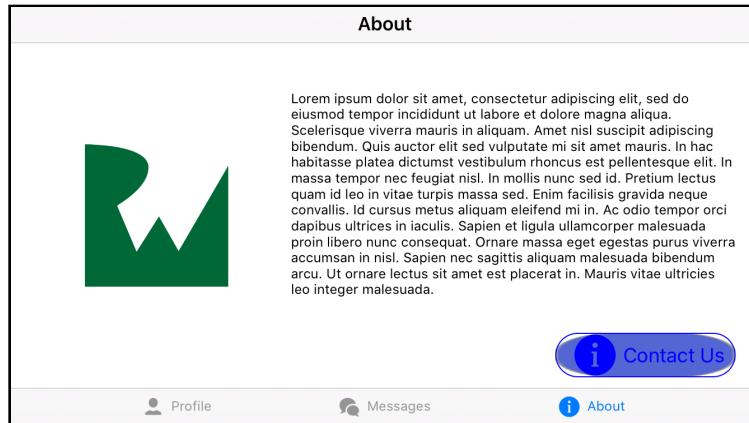
Alignment allows you to take part of an image by specifying margins. Meanwhile, **slicing** gives you the ability to have images that resize nicely, stretching just the parts you want. This is commonly used when you want to give a view a background, but the view can have different sizes.

Go to **AboutViewController.swift**, and in the `lazy var contactUsButton`, and change the title color to **white**. Add the following before the `return` statement.

```
button.setBackgroundImage(  
    UIImage(named: "button-background"),  
    for: .normal)
```

Build and run.

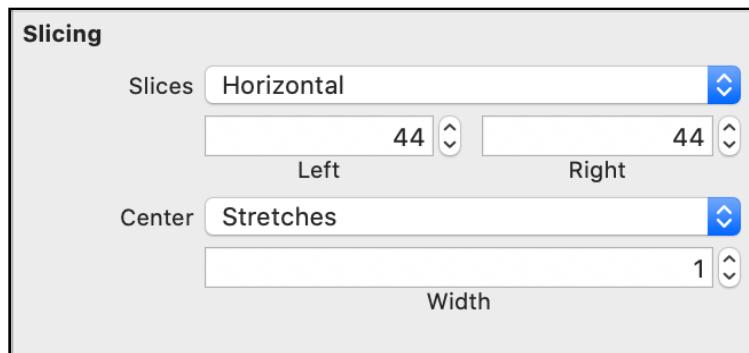
Go to the **About** tab and put the device in landscape. Notice the background looks distorted.



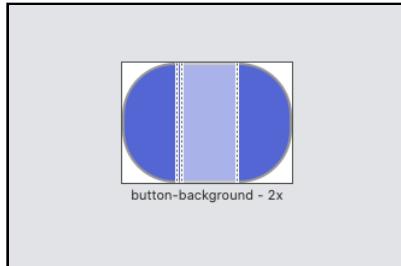
In **Assets.xcassets**, select **button-background**. Choose the image in the **2x** slot, and in the Attribute inspector, go to the bottom and you'll reach the **Slicing** section.



Set **Slices** to **Horizontal**. Set a value of **44** for **Left** and **Right**. And, set **Center** to **Stretches**.



Click **Show Slicing** on the bottom bar, and you'll see a visual representation of what you did.



You can also use this to change the values by dragging the dotted lines.

Build and run, and go to the **About** tab. Now, rotate the device so that you can see how the button background adapts for different orientations.

When in portrait:



When in landscape:



Excellent! You now have a new set of tools for creating layouts and making them adaptive. Using these tools, you can make your apps look great on any device and orientation.

## Challenge

The About screen currently has constraints for Compact Width/Regular Height and Compact Width/Compact Height traits. Your challenge is to add constraints using the Regular Width and Regular Height traits, so the user interfaces look good on more devices.

## Key points

- Size Classes determine how the user interface is laid out.
- You can modify how view controllers are presented using `UIPopoverPresentationControllerDelegate`.
- UIKit provides tools that help you create adaptive interfaces such as `UISplitViewController`, `UIAppeareance` proxy and Layout Guides.
- You can have different images for specific size classes.
- Split View Controller is a handy tool that comes with UIKit; you can use it to display master-detail like layouts.
- Use the `UIAppeareance` proxy when you want to have a consistent user interface attributes across the entire app.
- You can use Alignment and Slicing to control how your images stretch and to show specific portions when desired.

# Chapter 11: Dynamic Type

By Jayven Nhan

Dynamic Type is an iOS feature that enables app content to scale according to the user's font size preference. For the millions of people without perfect vision, having support for Dynamic Type makes all the difference. Without it, your app's user experience will likely suffer.

Visual impairment is one of the world's leading disabilities. Yet, most apps on the App Store fail to support Dynamic Type. But there's no reason your apps have to fall into that category.

In this chapter, you'll learn about:

- Reasons for supporting Dynamic Type.
- Supporting Dynamic Type on an existing app.
- Preferred font sizes.
- Growing and shrinking text.
- Supporting Dynamic Type using custom fonts.
- Growing and shrinking non-text UI elements.
- Managing layout changes based on font preferences.

By the end of this chapter, you'll know how to add support for Dynamic Type in your iOS apps.

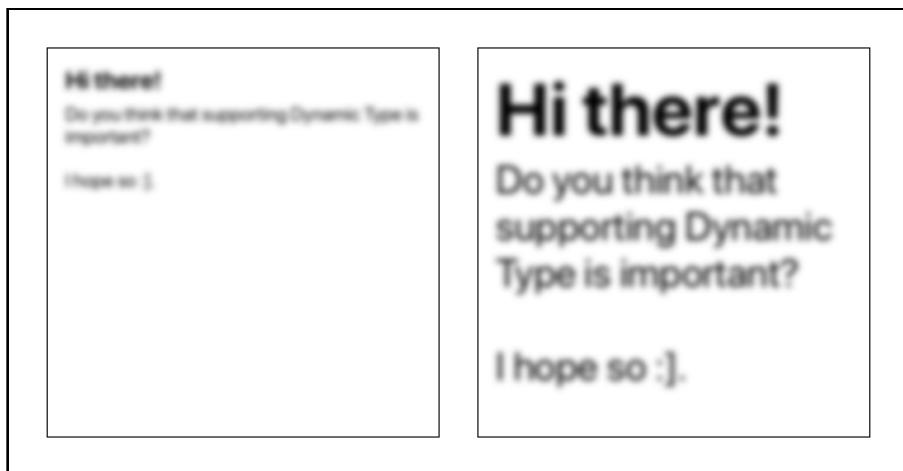
## Why Dynamic Type?



Dynamic Type makes your app usable by a broader audience, regardless of age or eyesight. In this section, you'll learn about five reasons to support Dynamic Type.

## Readability

Having a readable app is important. Supporting larger font sizes may not make a huge difference to someone with near-perfect vision, but many do not fall into that category. Look at the following image:



The blur effect represents how someone with vision problems may see the text. Although the same amount of blur exists on both images, the image on the left is almost unreadable and asks for a tremendous cognitive load to work out the letters and words. If your app is difficult to use, or even unusable, because your users can read your interface, you'll lose sales.

## Temporary or chronic injuries

You may think that users seldom need to change their text size preferences once they find a size that works. However, sometimes a person's eyesight deteriorates. Whether the deterioration is temporary or chronic, Dynamic Type can help ease the burden with large font size.

## Competition differentiation

Building apps accessible for everyone isn't always easy, but an accessible app differentiates itself from non-accessible apps. A comprehensive feature-packed app isn't worth much to users if they can't see what they're doing. When it comes to downloading apps, users will almost always choose the one that brings more value and is easier to use, so make sure your app is beautiful at all text sizes.

## High user retention rate over time

If your app accommodates all users regardless of how well they see, users won't need to go looking for an alternate solution should they ever need to increase the app's font size. Consequently, the app retention rate goes up over time.

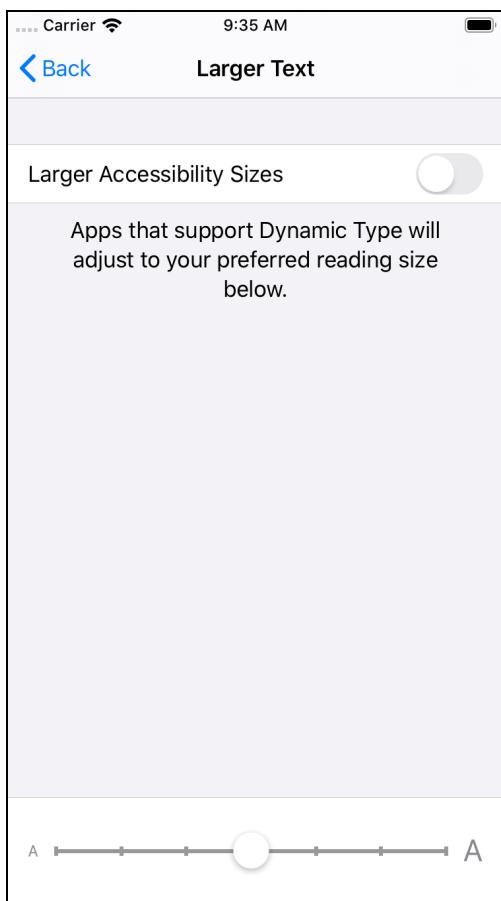
## Monetary gains

You can gain more users by ensuring that your app works equally great for all who use it. So, don't regularly tax your user's brainpower, which can make using your app exhausting. Instead, let users access their font preferences so they can use your app with relative ease. This makes for happy users, and happy users can't wait to come back for more.

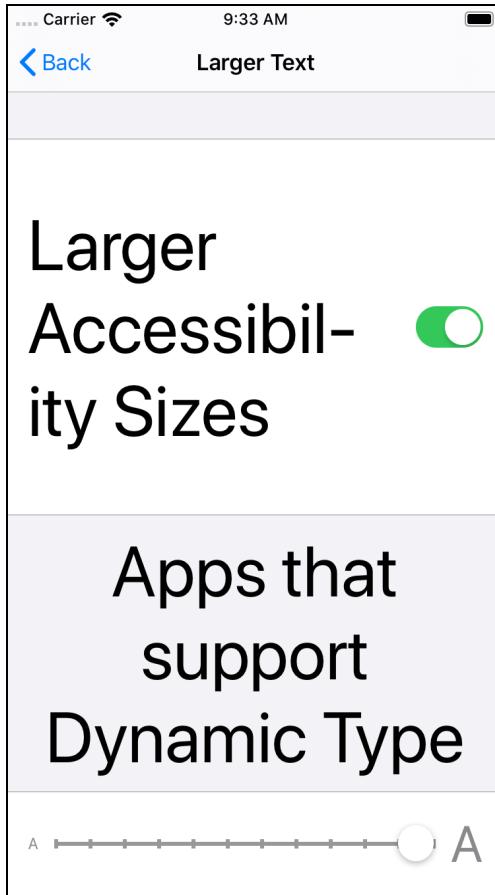
## Setting the preferred font size

Before you begin, the first step is to change your device's preferred font size. On iOS 13, follow these steps:

1. Open the Settings app.
2. Tap **Accessibility** ▷ **Display & Text Size** ▷ **Larger Text**.
3. Adjust the slider left or right to increase or decrease the preferred font size.



You can enable larger text sizes by toggling **Larger Accessibility Sizes**.



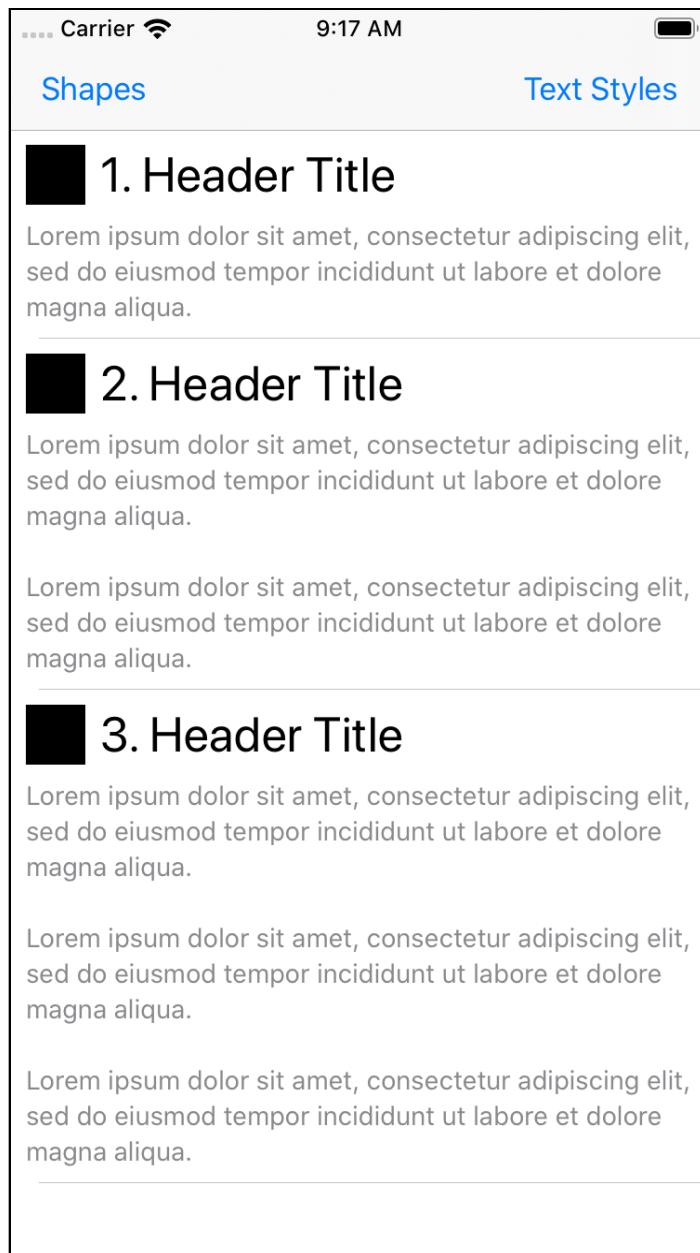
Enabling Larger Accessibility Sizes gives you five additional larger text font size options.

Now that you've set your preferred font size, you're ready to implement Dynamic Type.

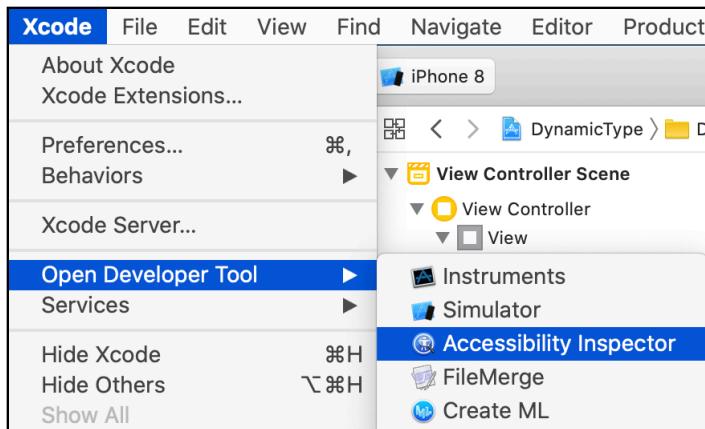
## Making labels support Dynamic Type

Dynamic Type almost works right out of the box. With Storyboards and text styles, you can size a label's text.

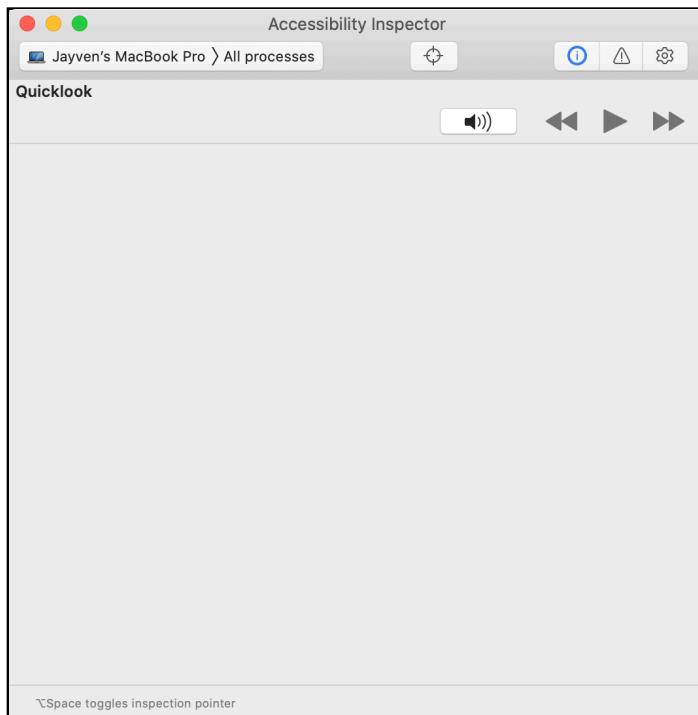
Open **DynamicType.xcodeproj** in the **starter** folder. Build and run.



In the Xcode menu bar, select **Xcode** ▶ **Open Developer Tool** ▶ **Accessibility Inspector**.

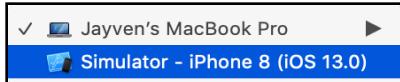


This shows the Accessibility Inspector.



There are a lot of benefits to using the Accessibility Inspector, such as setting the preferred font size. Setting the system's preferred font size here means you don't have to switch back and forth between the current app and the Settings app.

First, in the top-left corner, select the iOS simulator.



To set the system's preferred font size, click the **Settings** button.



You can now use the Font size slider to scale the font size up or down.

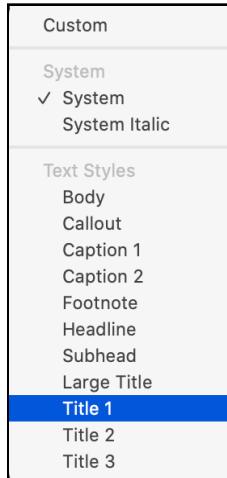
Drag the Font size slider all the way to the right.



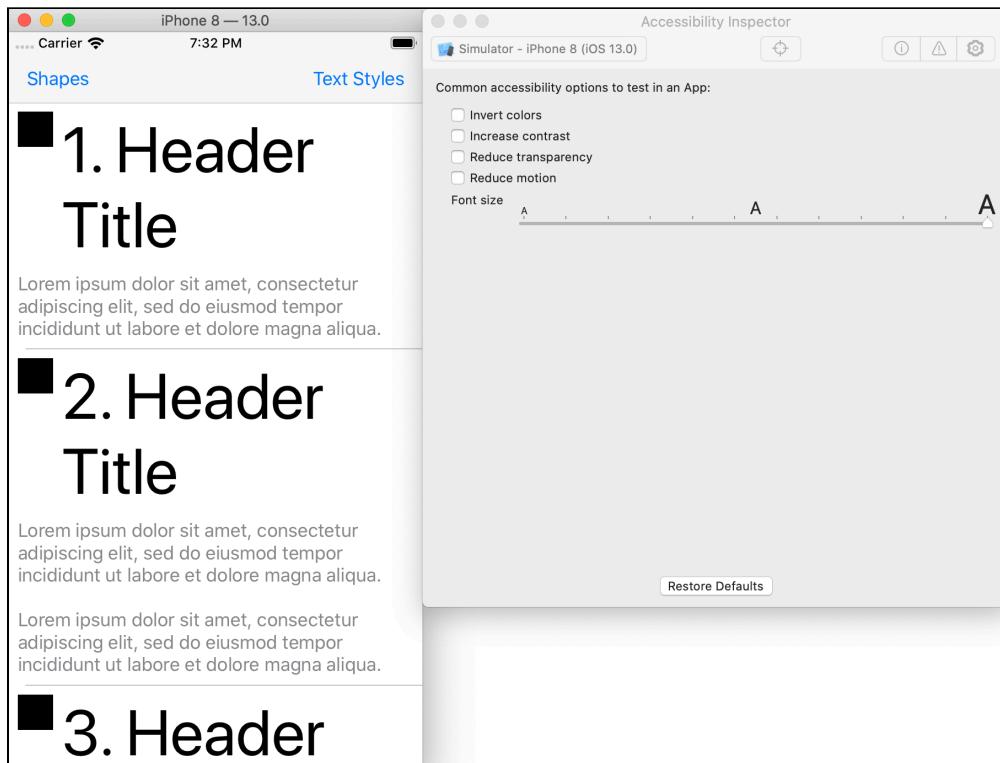
Notice how the app's labels look the same as they did before. Currently, the text sizes don't scale up or down when you change the values on the Font size slider, so you'll need to fix that.

Open **Main.storyboard**. In the View Controller Scene, select the table view cell's **header label** and open the Attributes inspector.

Change the label's font to **Title 1**.



Build and run.



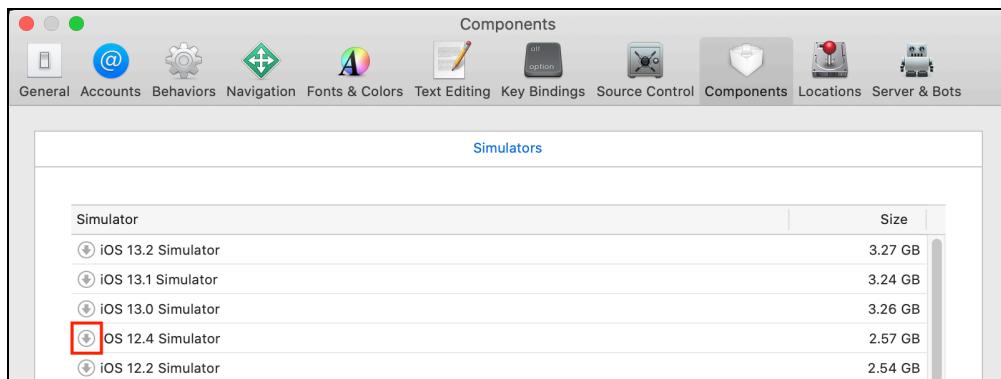
You'll see that the title label's font gets larger and smaller as you increase and decrease the font size preference. Your labels utilize the system's default font: Apple's SF fonts. Labels using Apple's SF fonts can easily support Dynamic Type.

On an iOS 13 device, a label with text style font increases and decreases dynamically inside a UITableViewCell at runtime without any further configuration. On a pre-iOS 13 device, you need to configure the label further to achieve the same effect. If you don't already have an iOS 12 simulator, you need to download one in Xcode since you'll need to test this feature to make sure it works.

Next to the scheme menu in the toolbar, click the **run destination** menu. Select **Download Simulators**.

Add Additional Simulators...  
**Download Simulators...**

Click the **download** button next to iOS 12.4 Simulator.



Grant Xcode **permission** to install the simulator on your Mac when prompted. You'll see a checkmark when the simulator is installed.



Close the **Components** window. Click the **run destination** menu. You'll see simulators denoted with different iOS versions.



Now, you should be able to test your project against iOS 12 and 13 simulators.

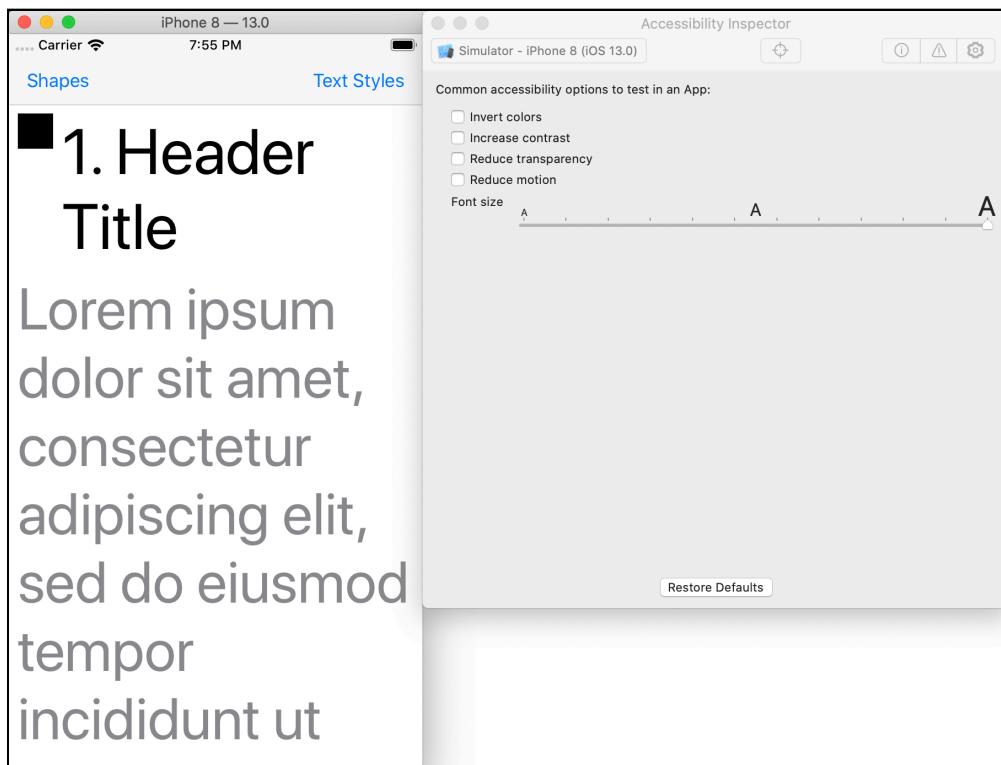
Open **Main.storyboard**. In the header label's Attributes inspector, enable the **Dynamic Type** option by checking the **Automatically Adjusts Font** checkbox.

**Dynamic Type  Automatically Adjusts Font**

Right below the header label is the description label. You also want the description label's font to scale automatically, so do the following:

1. Set description label's font to **Body** text style.
2. Enable the description label's **Dynamic Type** option.

Build and run.



Both the header and description labels now scale according to the user's font size preference. Plus, they're able to update at runtime dynamically. In other words, every time your users change their font size preference, they don't need to kill and reopen your app to see label texts scale up/down.

If you place a `UILabel` inside a `UITableViewCell` on iOS 13 devices, fonts with a system text style will scale at runtime without needing to enable **Automatically Adjusts Font**. Pre-iOS 13 devices, on the other hand, will need to enable **Automatically Adjusts Font** for runtime text-scaling regardless. This is an exception to take note of.

Different iOS versions can handle **Automatically Adjusts Font** differently. The rule of thumb is to enable **Automatically Adjusts Font** for runtime text-scaling with backward compatibility support.

## Making custom fonts support Dynamic Type

Making custom fonts dynamic requires a few more steps. At the time of writing this chapter, Interface Builder does not support setting up dynamic custom fonts. Consequently, you need to set a dynamic custom font using code.

Open **TableViewCell.swift**, and add the following value types above the **TableViewCell** declaration:

```
fileprivate let customFontSizeDictionary:  
[UIFont.TextStyle: CGFloat] =  
[.largeTitle: 34,  
.title1: 28,  
.title2: 22,  
.title3: 20,  
.headline: 17,  
.body: 17,  
.callout: 16,  
.subheadline: 15,  
.footnote: 13,  
.caption1: 12,  
.caption2: 11]  
  
enum FontWeight: String {  
    case regular = "Regular"  
    case bold = "Bold"  
}  
  
enum CustomFont: String {  
    case avenirNext = "AvenirNext"  
    case openDyslexic = "OpenDyslexic"  
}
```

Here, you create enums to minimize the chances of mistyping a font name or weight.

Now, add the following extension to the end of **TableViewCell.swift**:

```
extension UILabel {  
    func set(  
        customFont: CustomFont,  
        fontWeight: FontWeight,
```

```
        textStyle: UIFont.TextStyle = .body
    ) {
    // 1
    let name = "\(customFont.rawValue)-\(fontWeight.rawValue)"
    guard let size = customFontSizeDictionary[textStyle]
        else { return }
    // 2
    guard let font = UIFont(name: name, size: size)
        else { fatalError("Retrieve \(name) with error.") }
    // 3
    let fontMetrics = UIFontMetrics(forTextStyle: textStyle)
    self.font = fontMetrics.scaledFont(for: font)
    // 4
    adjustsFontForContentSizeCategory = true
}
}
```

With this code, you:

1. Generate the font name. Then, decide the font size multiplier based on `textStyle`.
2. Safely unwrap the custom font. This ensures that the custom font is accessible to the app. If it isn't, the app throws a fatal error describing the missing custom font.
3. Set the label's font to an automatic scaling font from the custom font. The automatic scaling font generates from `UIFontMetrics`.
4. Have the label automatically update the font according to the device's content size category, instead of handling font changes manually with `UIContentSizeCategoryDidChangeNotification`.

Add the following code to `TableViewCell`:

```
private func setupDynamicCustomFont(_ customFont: CustomFont) {
    headerLabel.set(
        customFont: customFont,
        fontWeight: .regular,
        textStyle: .title1)
    descriptionLabel.set(
        customFont: customFont,
        fontWeight: .regular)
}
```

Here, you set the header and description label fonts to scale using the extension method you created earlier. You also pass in font customization parameters.



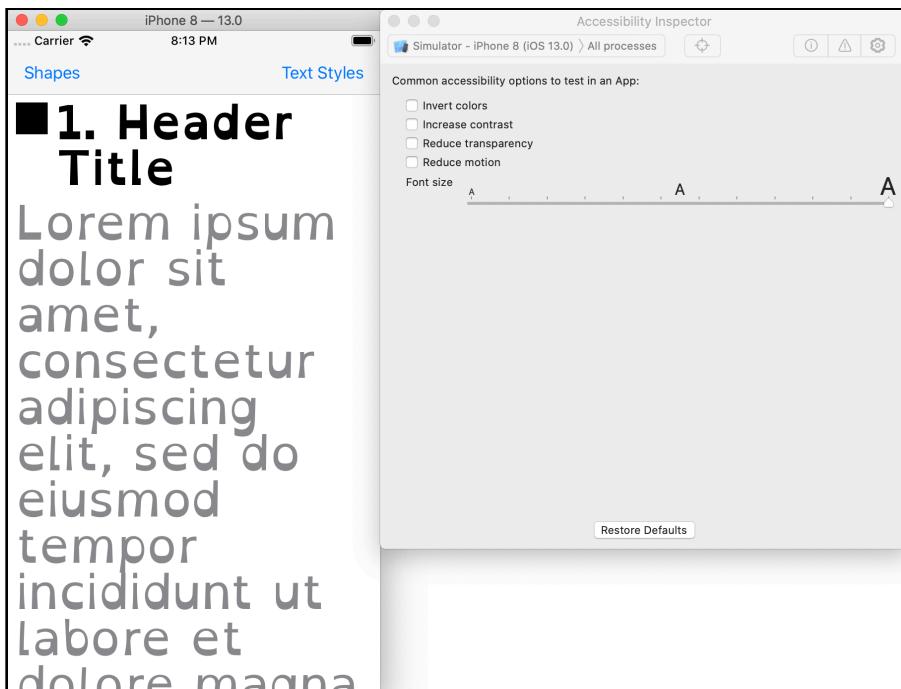
Finally, add the following code to the end of `configureCell(text:index:setCustomFont):`:

```
setupDynamicCustomFont(.openDyslexic)
```

This code puts dynamic custom fonts into effect for `TableViewCell`'s labels.

Build and run.

At the largest accessibility font preference, you'll see the following:



Nice, you've created a dynamically scaling custom font.

## Managing layouts based on the text size

In addition to having readable text, you need your app to look good when users change their text size. In code, you'll need to identify the user's preferred font size to make that happen. The first way to identify the user's preferred font size is to get the view's trait collection.

Typically, you'd override `traitCollectionDidChange(_:)` to make further layout changes based on the user's text size preference — even on a `UITableViewCell`. However with iOS 13, `traitCollectionDidChange(_:)` doesn't get called in a `UITableViewCell`. For this reason, and to ensure backward compatibility, you need to take a different approach.

Add the following code to `TableViewCell`:

```
func updateTraitCollectionLayout() {
    // 1
    let preferredContentSizeCategory =
        traitCollection.preferredContentSizeCategory
    // 2
    let scaledValue = UIFontMetrics.default
        .scaledValue(for: profileImageViewWidth)
    profileImageViewWidthConstraint.constant = scaledValue
    // 3
    topStackView.axis =
        preferredContentSizeCategory.isAccessibilityCategory
            ? .vertical : .horizontal
    // 4
    profileImageStackView.isHidden = preferredContentSizeCategory
        == .accessibilityExtraExtraLarge
}
```

And, add a call to your new method in `configureCell(text:index:setCustomFont:)` before the guard statement:

```
updateTraitCollectionLayout()
```

So far, you've done the following:

1. Retrieved the current font size preference.
2. Scaled the profile image view width using `UIFontMetrics`'s default font, body text style.
3. When the user's preferred font size is in the accessibility sizes range, set the axis orientation of `topStackView` to vertical. Otherwise, made the axis orientation horizontal.
4. For the extra, extra, extra-large font size preference, you hid the profile image view.

In `ViewController.swift`, add the following code after `viewDidAppear(_:)`:

```
override func traitCollectionDidChange(
    _ previousTraitCollection: UITraitCollection?
```

```
) {  
    super.traitCollectionDidChange(previousTraitCollection)  
    guard traitCollection.preferredContentSizeCategory !=  
        previousTraitCollection?.preferredContentSizeCategory  
    else { return }  
    let indexPaths = (0..        .map { IndexPath(row: $0, section: 0) }  
    DispatchQueue.main.async { [weak self] in  
        self?.tableView.reloadRows(at: indexPaths, with: .none)  
    }  
}
```

When the user changes the preferred font size, you use `ViewController.traitCollectionDidChange(_:)` to update the table view cells.

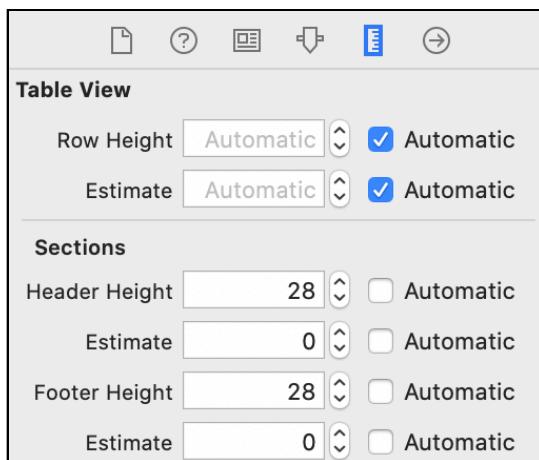
Build and run.

Now, you'll see the app's layout changes based on the user's preferred font size.

## Supporting Dynamic Type with UITableView

Dynamic Type works right out of the box with the standard `UITableView`. By default, the header and cell height dynamically scale to fit the header and cell content.

After dragging a `UITableView` into the Interface Builder, the default **Size inspector** settings are as follows:



The standard UITableView enables self-sizing table view cells. To enable self-sizing headers and footers, you need to set their respective height and height estimate to automatic. As long as you pin the views to the edges of your UITableViewCell's content view, your table view cell height will size accordingly. You can read about the details behind self-sizing UITableViewCell in Chapter 6, "Self-Sizing Views".

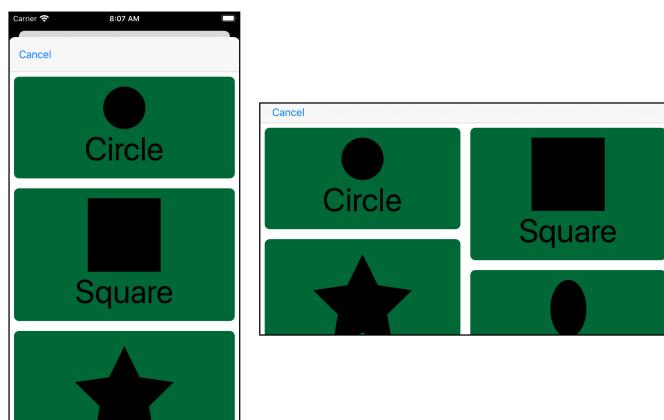
## Supporting Dynamic Type with UICollectionView

The standard UICollectionView requires more work to support self-sizing cells. Specifically, when you use a collection view for non-line based layouts, you'll need to set up a custom UICollectionViewLayout.

To achieve self-sizing collection view cells with a custom UICollectionViewLayout, here's what you'll do:

1. Handle the collection view cell layout attributes caching.
2. Set the collection view content size.
3. Implement layout invalidation logic.
4. Return the layout attributes at each cell item's index path.
5. Load the layout attributes for cells that are within the collection view

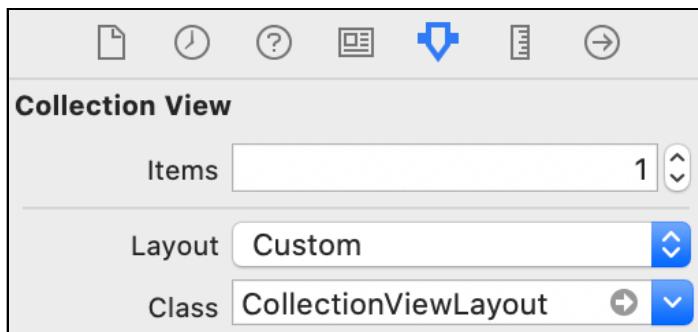
The final result will look like this:



Open **Main.storyboard**. CollectionViewController's collection view prototype cell contains an image view and a label inside a stack view. The stack view's edges are pinned to the edges of the cell's content view. This is the Auto Layout setup for CollectionViewCell.

To set a collection view's custom layout, do the followings:

1. Select **CollectionViewController**.
2. Open the **Document outline**.
3. Select **Collection View**.
4. In the Attributes inspector, set **Layout** to **Custom** and **Class** to **CollectionViewLayout**.



Next, you'll prepare the presentation of the custom collection view layout.

## Preparing custom collection view layouts

Open **CollectionViewLayout.swift**. Add the following properties to **CollectionViewLayout**:

```
// 1
weak var delegate: CollectionViewLayoutDelegate?
// 2
private var columns: Int {
    return UIDevice.current.orientation ==
        .portrait ? 1 : 2
}
// 3
private let cellPadding: CGFloat = 8
// 4
private var contentWidth: CGFloat = 0
private var contentHeight: CGFloat = 0
// 5
```

```
private var contentBounds: CGRect {
    let origin: CGPoint = .zero
    let size = CGSize(
        width: contentWidth,
        height: contentHeight)
    return CGRect(origin: origin, size: size)
}
// 6
private var cachedLayoutAttributes:
    [UICollectionViewLayoutAttributes] = []
```

Here's what each property handles:

1. The delegate feeds the custom view layout with cell sizing information. This includes the image view height and the label's text at each collection view cell's index path.
2. Returns the columns per row count based on the device's orientation: one column for portrait, and two columns for landscape.
3. Declares a cell padding value.
4. The content width and height will make the collection view's content size.
5. This computed property defines the collection view's content bounds made up of its content position and size.
6. The layout information for populating your collection view cells.

Next, override the following method in `CollectionViewLayout`:

```
override func prepare() {
    super.prepare()
    // 1
    guard let collectionView = collectionView
        else { return }
    cachedLayoutAttributes.removeAll()
    // 2
    let size = collectionView.bounds.size
    let safeAreaContentInset = collectionView.safeAreaInsets
    collectionView.contentInsetAdjustmentBehavior = .always
    contentWidth = size.width -
        safeAreaContentInset.horizontalInsets
    contentHeight = size.height -
        safeAreaContentInset.verticalInsets
    // 3
    makeAttributes(for: collectionView)
}
```

Here's what you did with the code:

1. Safely unwrap the collection view property for the collection view customization. Remove all of the existing cached layout attributes.
2. Set the collection view's content width and initial height to the collection view's bounds size. During the bounds size calculation, you'll account for the safe area content inset. As `makeAttributes(for:)` generates layout attributes, it'll determine the final collection view content height.
3. Call `makeAttributes(for:)` to create the layout attributes that will position and size the collection view cells.

`prepare()` is the place to do all of the overhead work for your collection view. The overhead work includes handling layout attributes caches, defining the collection view's content size and populating the collection view cells.

## Creating collection view layout attributes

Add the following code to `makeAttributes(for:)`:

```
// 1
guard let delegate = delegate else { return }
let itemWidth = contentView.bounds.size.width / CGFloat(columns)
// 2
var xOffsets: [CGFloat] = []
(0..
```

Here's what you did:

1. Safely unwrap `delegate`. You'll need this delegate to get the cell's content for sizing.
2. Each cell needs an x position within the collection view. You'll position your cell closer or further from the content bounds origin based on the column index.
3. Initialize a column index property for reference.

4. Initialize an array of zeroes with elements count equal to `columns`.
5. Initialize a `Range<Int>` from zero to the number of items in the collection view at section 0.

Add the following code to the end of `makeAttributes(for:)`:

```
for item in items {  
    // 1  
    let indexPath = IndexPath(item: item, section: 0)  
    // 2  
    let itemHeight = makeItemHeight(  
        atIndexPath: indexPath,  
        itemWidth: itemWidth,  
        withCollectionView: collectionView,  
        delegate: delegate)  
    // 3  
    let frame = CGRect(  
        x: xOffsets[column],  
        y: yOffsets[column],  
        width: itemWidth,  
        height: itemHeight)  
    // 4  
    let insetFrame = frame.insetBy(  
        dx: cellPadding,  
        dy: cellPadding)  
    // 5  
    let layoutAttributes =  
        UICollectionViewLayoutAttributes(  
            forCellWith: indexPath)  
    layoutAttributes.frame = insetFrame  
    cachedLayoutAttributes.append(layoutAttributes)  
    // 6  
    contentHeight = max(contentHeight, frame.maxY)  
    // 7  
    yOffsets[column] += itemHeight  
    // 8  
    column = column < columns - 1 ? column + 1 : 0  
}
```

Looping through the collection view `items`, here's what you do at each iteration:

1. Initialize the cell's index path using `item`.
2. Get the item's height using `makeItemHeight(atIndexPath:itemWidth:withCollectionView:delegate:)`.
3. Generate the cell's initial frame using x and y offsets at the current column index.
4. Using `frame`, create another `CGRect` accounting for the cell's padding.

5. Initialize the collection view layout attributes with the cell's index path. Set the layout attributes frame. Append the layout attributes into the cached layout attributes array.
6. Set the collection view's content height to the greater value between the current content height and the frame's max-y.
7. Advance the y-offset value by the item height.
8. When the column index reaches the last column, reset the column index to zero. Otherwise, increment the column index.

The cell's width depends on the device's orientation; however, the cell's height depends on the cell's content combined height, which you'll work on next.

Replace the body of

`makeItemHeight(atIndexPath:itemWidth:withCollectionView:delegate:)`  
with the following code:

```
// 1
let imageHeight = delegate.collectionView(
    collectionView,
    heightForImageAtIndexPath: indexPath)
// 2
let labelText = delegate.collectionView(
    collectionView,
    labelTextAtIndexPath: indexPath)
let maxLabelHeightSize = CGSize(
    width: itemWidth,
    height: CGFloat.greatestFiniteMagnitude)
let boundingRect = labelText.boundingRect(
    with: maxLabelHeightSize,
    options: [.usesLineFragmentOrigin],
    attributes:
        [NSAttributedString.Key.font:
            UIFont.preferredFont(forTextStyle: .headline)],
    context: nil)
let labelHeight = ceil(boundingRect.height)
// 3
let itemHeight = cellPadding * 2 + imageHeight + labelHeight
return itemHeight
```

With this code, you:

1. Get the image height.
2. Using the label's text and font attribute, calculate the label's height.

3. Combine the cell paddings, label height and image height. Return the item's height.

## Overriding layout attributes methods

Add the following method overrides to `CollectionViewLayout`:

```
// 1
override func layoutAttributesForItem(
    at indexPath: IndexPath)
-> UICollectionViewLayoutAttributes? {
    return cachedLayoutAttributes[indexPath.item]
}
// 2
override func layoutAttributesForElements(
    in rect: CGRect)
-> [UICollectionViewLayoutAttributes]? {
    return cachedLayoutAttributes.filter {
        rect.intersects($0.frame)
    }
}
```

Here's what you did:

1. When the collection view asks for the cell layout attributes at an index path, return the cached layout attributes using the index path.
2. When scrolling, the cells for display on a screen can change as you scroll. The collection view layout will calculate the layout attributes for cells that are within the collection view frame.

## Setting the collection view content size

Add the following property and method overrides to `CollectionViewLayout`:

```
// 1
override var collectionViewContentSize: CGSize {
    return contentBounds.size
}
// 2
override func shouldInvalidateLayout(
    forBoundsChange newBounds: CGRect) -> Bool {
    guard let collectionView = collectionView
        else { return false }
    return newBounds.size
        != collectionView.bounds.size
}
```

With the code added, here's what you did:

1. Set the collection view content size to the content bounds size. The size is made up of the content width and height in `prepare()`.
2. Re-query for the layout geometry information only when the collection view's new bounds size differs from the current collection view bounds size.

## Setting up the collection view layout delegate

Open `CollectionViewController.swift`.

To adopt and conform to `CollectionViewLayoutDelegate`, add the following code to the end of the file:

```
extension CollectionViewController: CollectionViewLayoutDelegate {
    func collectionView(
        _ collectionView: UICollectionView,
        heightForImageAtIndexPath indexPath: IndexPath
    ) -> CGFloat {
        return shapes[indexPath.item].image.size.height
    }

    func collectionView(
        _ collectionView: UICollectionView,
        labelTextAtIndexPath indexPath: IndexPath
    ) -> String {
        return shapes[indexPath.item].shapeName.rawValue
    }
}
```

With this code, you return the respective shape's image height and label text at the cell's index path.

Add the following code to `setupCollectionViewLayout()`:

```
guard let collectionViewLayout =
    collectionView.collectionViewLayout
    as? CollectionViewLayout else { return }
collectionViewLayout.delegate = self
```

Here, you set the `CollectionViewController` as the object to provide the layout geometry information.

## Invalidating collection view layout for trait collection

Finally, add the following code to `CollectionViewController`:

```
override func traitCollectionDidChange(  
    previousTraitCollection: UITraitCollection?  
) {  
    super.traitCollectionDidChange(previousTraitCollection)  
    guard previousTraitCollection?.preferredContentSizeCategory  
        != traitCollection.preferredContentSizeCategory  
        else { return }  
    collectionView.collectionViewLayout.invalidateLayout()  
}
```

With the code added, the collection view re-queries its layout information when the text size preference changes.

Build and run, and tap the **Shapes** button.

With the largest preferred text size, you'll see the following in the portrait orientation:

And here's what you'll see in landscape orientation:

Now, the collection view cells dynamically adjust their size based on the cell content at runtime.

## Challenges

Build and run. Tap the **Text Styles** bar button item, and you'll see `InfoViewController`. Your challenge is to make `InfoViewController` support Dynamic Type.

You'll need to fulfill the following requirements with `InfoViewController`:

- In portrait orientation, set each label's font to the corresponding font in `labelTextStyleTuples`.
- In landscape orientation, set each label's font to Avenir Next.
- Make all labels support Dynamic Type fonts.
- Make labels scale when a user changes the font size preferences at runtime.

While working through this challenge, make use of `labelFontTextStyleTuples` in `InfoViewController`.

## Key points

- The Dynamic Type feature allows users to scale content sizes based on the device's font size preference.
- Supporting Dynamic Type makes your app usable for a broader audience, which comes with benefits, including being different from the competition and increasing app retention rate.
- Apps using Apple's SF font can easily support Dynamic Type.
- You can implement custom fonts with Dynamic Type.
- You can scale non-text user interface elements according to the font size preferences.
- Stack view makes it easy to make layout changes according to the font preferences.
- When a cell uses Auto Layout, the standard `UITableView` supports Dynamic Type out of the box.
- For collection views using non-line based layouts, you can support Dynamic Type using custom collection view layouts.

# Chapter 12:

# Internationalization & Localization

By Libranner Santos

Apps should look and feel local to the user's region, so if you're aiming to create apps that users can use globally, you need to keep internationalization and localization in mind.

As it turns out, Interfaces created with Auto Layout can support internationalization, and Xcode provides the tools to help you localize all of your resources.

In this chapter, you'll learn how to test if your app is internationalization-ready. You'll also learn about the things you need to consider while creating your constraints, so they can properly handle different languages.

## What's internationalization and localization?

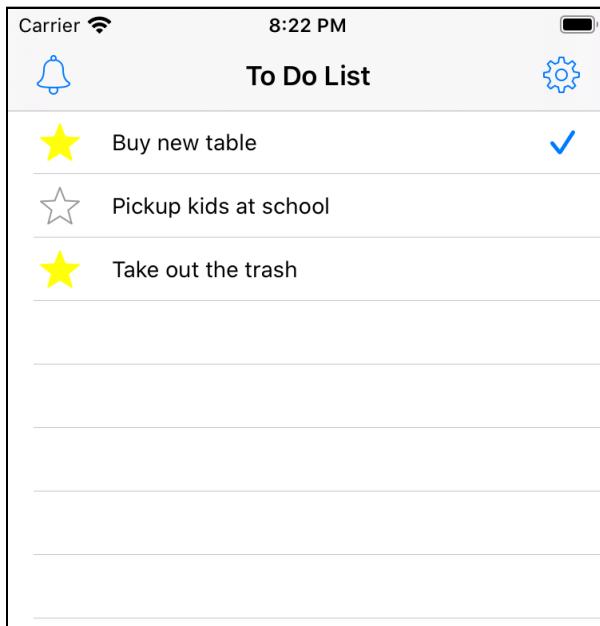
**Internationalization** refers to creating apps that are agnostic to the language, which means they can handle changes in the content size — for example, displaying text in different languages. **Localization**, on the other hand, means translating the resources into different languages.

These two concepts are crucial to providing a good user experience, especially when users expect their apps to use their preferred language. As a developer, you must prepare your layouts to support internationalization and localization. Thankfully, with Auto Layout, developing your app for internationalization and localization support is reasonably easy to achieve.



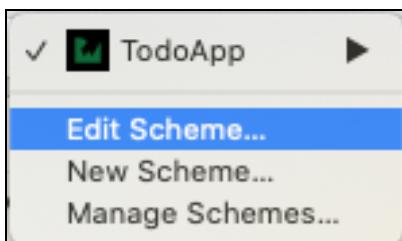
# Auto Layout and internationalization

Go to the starter project and open the TodoApp project. Now, build and run.

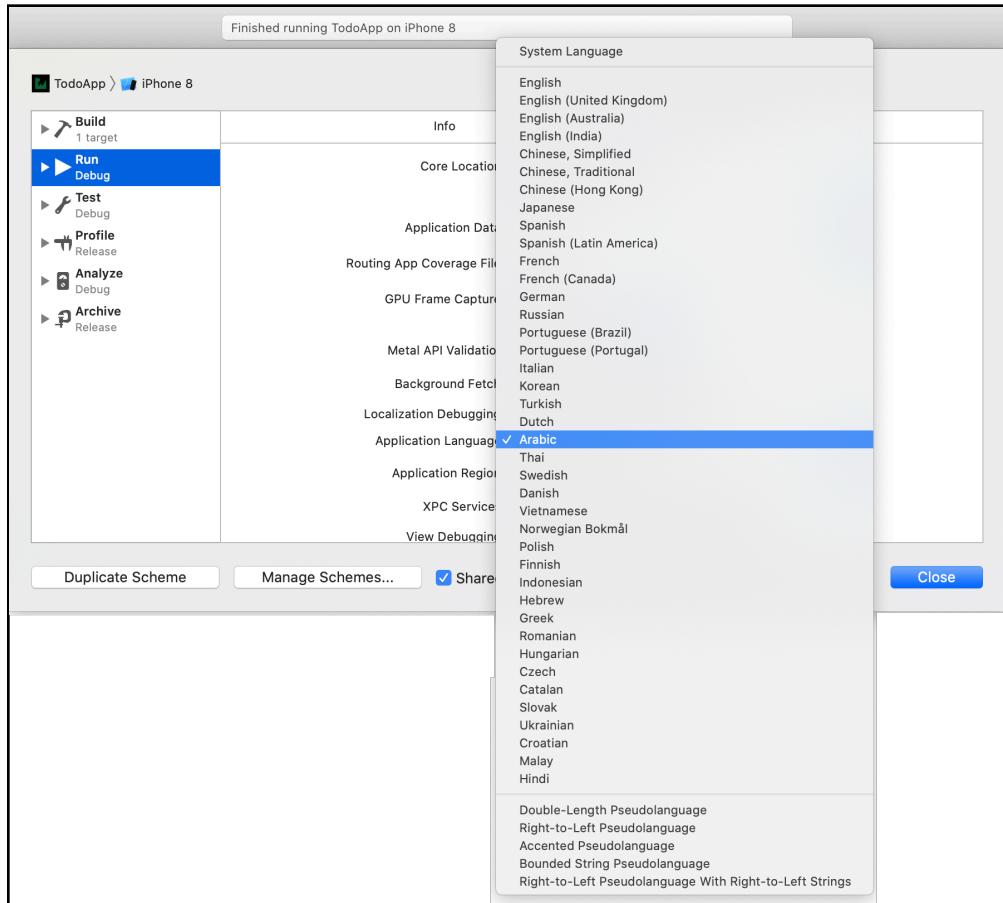


The app you'll work with in this chapter is a to-do list app. It shows a list of tasks and marks the ones that are complete. On the left side, you'll see an icon that indicates favorited to-do items.

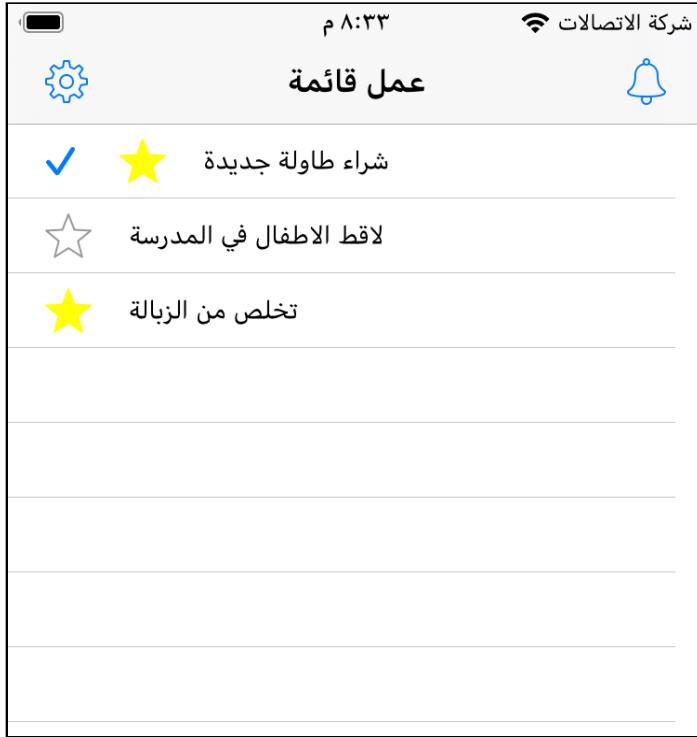
Click the active scheme in the toolbar, and select **Edit Scheme....**



A modal window appears. Select the **Options** tab, and click the drop-down menu for **Application Language**. Here, you can select from a variety of languages. For this exercise, choose **Arabic**.



Open the app again, and you'll see that the tasks are displayed incorrectly. Because you read Arabic from right-to-left, your layout is no longer correct: The checkmark icon and the star icon need to switch places, and the text needs to align to the right.



Go to the project, and inside the **Views** folder, open **TaskTableViewCell.swift**. In **setupConstraints()**, change all uses of **leftAnchor** to **leadingAnchor**, and all uses of **rightAnchor** to **trailingAnchor**. The updated method looks like this:

```
private func setupConstraints() {
    contentView.addSubview(taskNameLabel)
    contentView.addSubview(favoriteButton)

    taskNameLabel.translatesAutoresizingMaskIntoConstraints = false
    favoriteButton.translatesAutoresizingMaskIntoConstraints = false

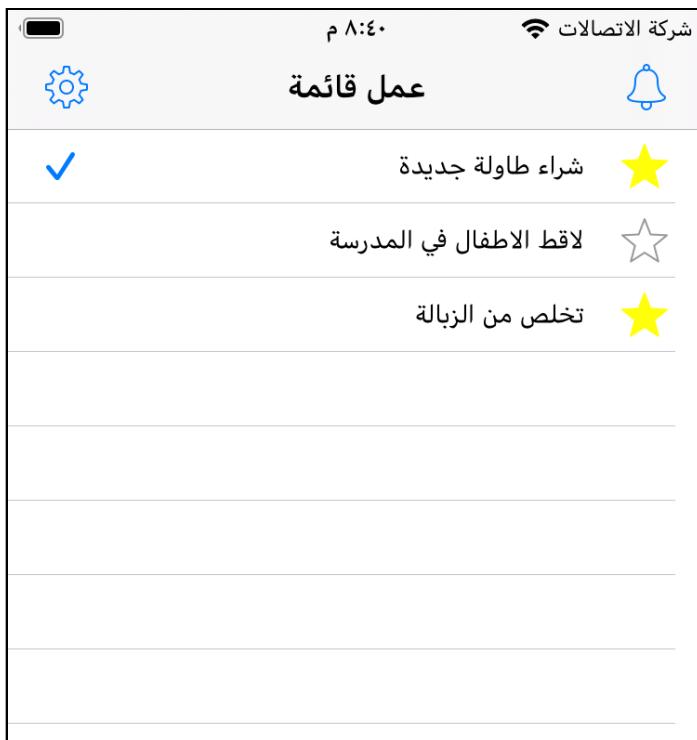
    let constraints = [
        favoriteButton.leadingAnchor.constraint(
            equalTo: contentView.leadingAnchor,
            constant: 20),
        favoriteButton.centerYAnchor.constraint(
            equalTo: contentView.centerYAnchor),
```

```
taskNameLabel.leadingAnchor.constraint(
    equalTo: favoriteButton.trailingAnchor,
    constant: 20),
taskNameLabel.trailingAnchor.constraint(
    lessThanOrEqualTo: contentView.trailingAnchor,
    constant: -20),
taskNameLabel.centerYAnchor.constraint(
    equalTo: contentView.centerYAnchor)
]

NSLayoutConstraint.activate(constraints)
}
```

By using leading and trailing anchors, you let the system deal with accommodating the user interface depending on the orientation of the selected language.

Build and run.

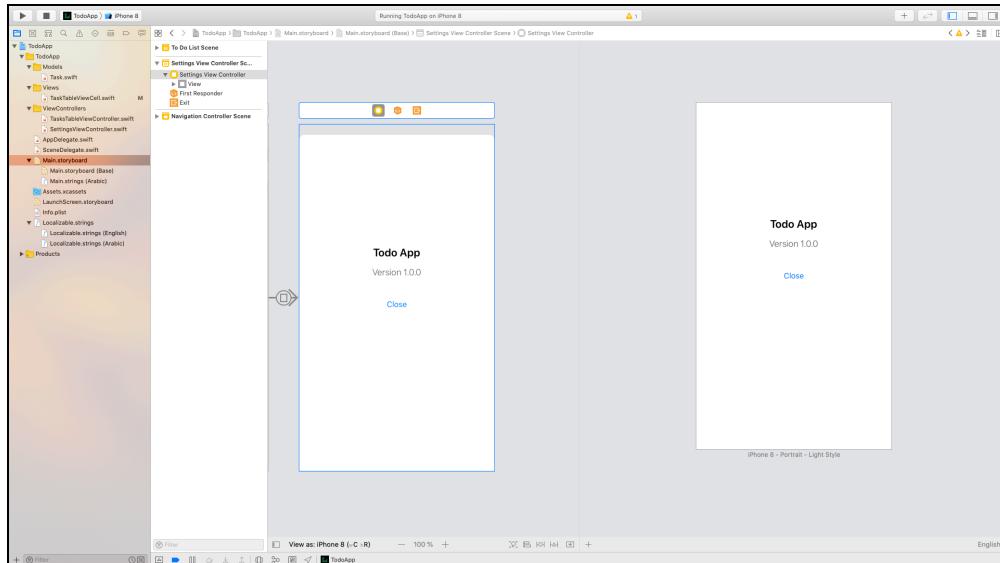


Excellent, now the tasks are correctly aligned.

# Previewing languages in Interface Builder

Interface Builder helps you preview screens in different languages. For this project, the localization files already exist.

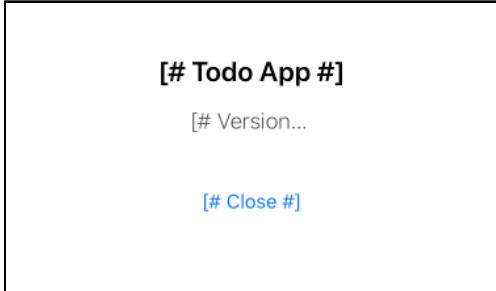
Go to **Main.storyboard** and select **Settings View Controller**. Press **Command-Option-Enter** to show the preview for the selected controller.



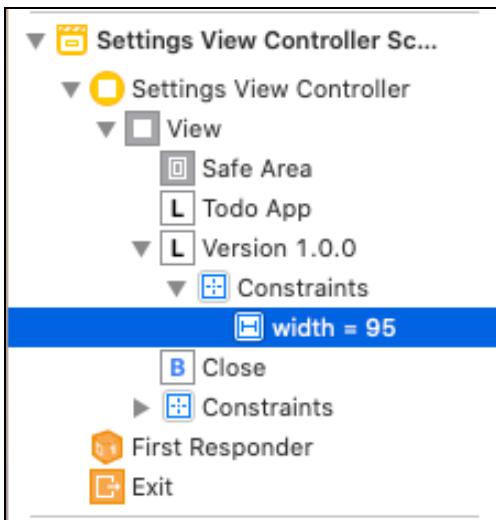
On the right side, you can see how the screen should look at runtime. On the bottom right, you can change the language for this preview. Click **English** and select **Arabic**.



The screen now shows all of the text in Arabic. Click the language button again, and select **Bounded String Pseudolanguage**. This setting adds some characters to all of the strings, which helps you visualize how the interface will handle larger content.

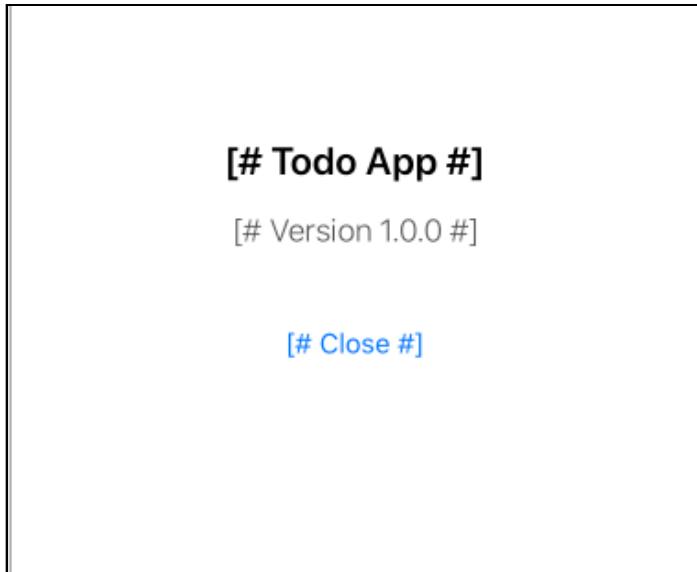


Notice that the version label is getting cut off. This display issue happens when there are constraints that force the label to have a fixed size. In the Document Outline, expand the **Constraints** under the version label:



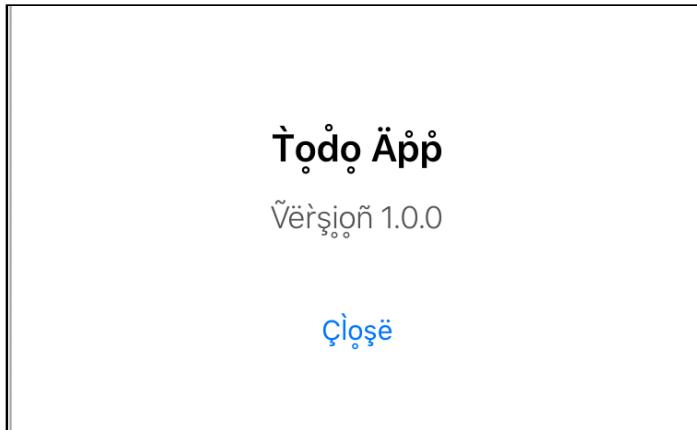
Press **delete** to remove the width constraint.

Look at the preview again, and notice the label is now displaying properly:

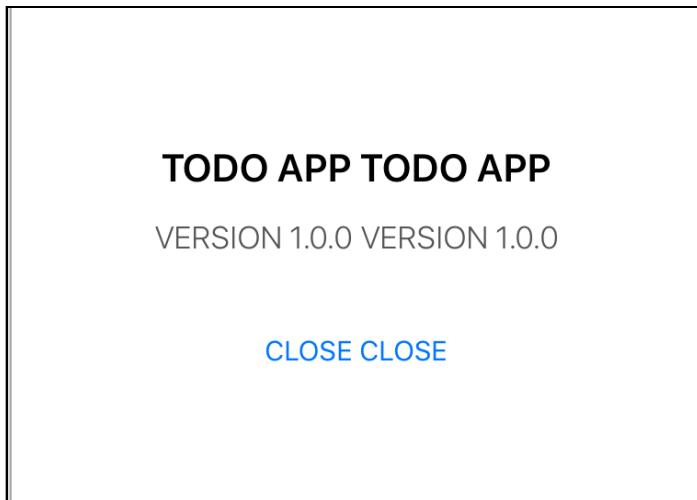


Try the other options, **Accented Pseudolanguage** and **Double-Length Pseudolanguage**, to test how the app handles taller content:

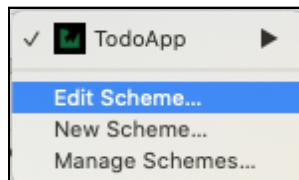
Here's the preview for Accented Pseudolanguage:



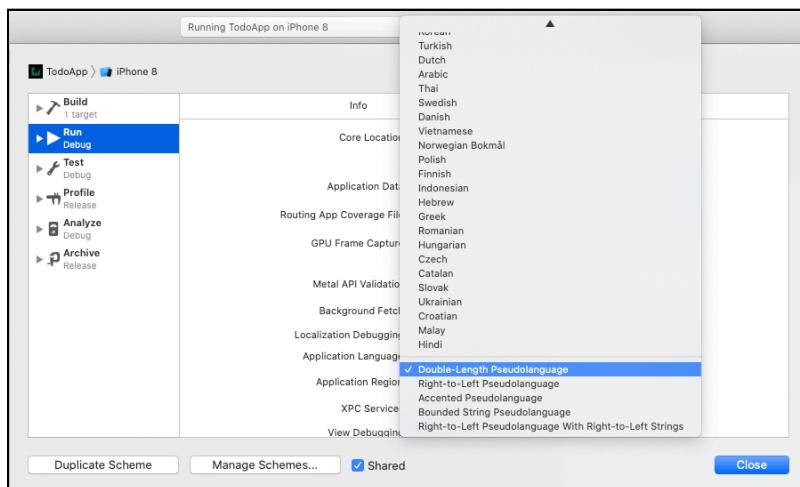
Here's the preview for Double-Length Pseudolanguage:



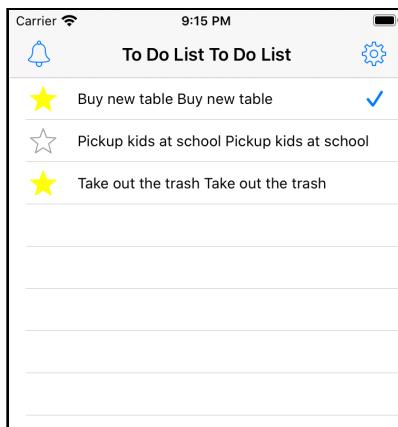
You can also test apps without using the preview or changing the language directly on the simulator. Click the active scheme in the toolbar, and choose **Edit Scheme....**



A modal window appears. Click the drop-down for **Application Language**, and scroll to the bottom. Select **Double-Length Pseudolanguage**:



Build and run.



All of the text is duplicated, and notice the app handles it well. Switching the app's language via the scheme editor is another tool you can use to test how your app handles content changes.

## Key points

- Unless strictly necessary, don't use `rightAnchor` or `leftAnchor` to create constraints; instead, use `leadingAnchor` and `trailingAnchor`.
- Use the preview on Interface Builder to test your apps for internationalization.
- Don't use fixed constraints on elements that can contain text unless it's absolutely necessary.

# Chapter 13: Common Auto Layout Issues

Libranner Santos

Auto Layout is great, but it's not a magic formula for creating bug-free interfaces. With Auto Layout, you'll occasionally run into problems. For instance, your layout may have too few constraints, too many constraints or conflicting constraints. Although the Auto Layout Engine will try to solve most of these problems for you, it's crucial to understand how to handle them.

## Understanding and solving Auto Layout issues

Under the hood, the Auto Layout Engine reads each of your constraints as a formula for calculating the position and size of your views. When the Auto Layout Engine is unable to satisfy all of the constraints, you'll receive an error.

There are three types of errors you'll encounter while working with Auto Layout:

- **Logical Errors:** These types of errors usually occur when you inadvertently set up incorrect constraints. Maybe you entered incorrect values for the constants, choose the wrong relationship between the elements or didn't take into account the different orientations and screen sizes.
- **Unsatisfiable Constraints:** You'll get these errors when it's impossible for the Auto Layout Engine to calculate a solution. A simple example is when you set one view to have a width  $< 20$  **and** a width = 100. Since both of those constraints cannot be true at the same time, you'll get an error.

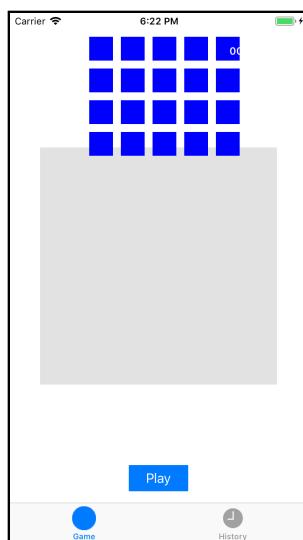


- **Ambiguous Constraints:** When the Auto Layout Engine is trying to satisfy multiple constraints, and there's more than one possible solution, you'll get this type of error. When this happens, the engine will randomly select a solution, which may produce unexpected results with your UI.

It's time to see how these issues look in a real project, starting with Unsatisfiable Constraints.

## Unsatisfiable Constraints

Open the starter project for this chapter and build and run.



**Note:** Your view may look different because of the random selections of the Auto Layout Engine.

Throughout this chapter, you'll work with this project. It's a simple memory game where the user has to repeat the sequence by tapping the corresponding box.

With the project still running, switch to Xcode and open **Main.storyboard**. This is the main screen for the memory game. You should see a header at the top, but there are some conflicts you need to resolve. This is where the console logs can help!

## How to read the logs

Look at the console, and you'll see some useful information. What you're seeing is a combination of informative text, constraints in visual format language and even some helpful suggestions:

```
2019-05-05 04:19:42.439737+0200 DebuggingAutoLayout[4310:346296]
[LayoutConstraints] Unable to simultaneously satisfy
constraints.
    Probably at least one of the constraints in the following list
    is one you don't want.
        Try this:
            (1) look at each constraint and try to figure out which you
                don't expect;
            (2) find the code that added the unwanted constraint or
                constraints and fix it.
(
    "<NSLayoutConstraint:0x600002069270
    UIView:0x7fad7dc0f2f0.height == 100    (active)>",
    "<NSLayoutConstraint:0x600002069900
    UILayoutGuide:0x600003a355e0'UIViewSafeAreaLayoutGuide'.bottom
    == UIView:0x7fad7dc0f2f0.bottom + 800    (active)>",
    "<NSLayoutConstraint:0x600002069950 V:|-0-
    [UIView:0x7fad7dc0f2f0]    (active, names:
    '|':UIView:0x7fad7dc0d510 )>",
    "<NSLayoutConstraint:0x600002074910 'UIView-Encapsulated-
    Layout-Height' UIView:0x7fad7dc0d510.height == 667    (active)>",
    "<NSLayoutConstraint:0x600002069860
    'UIViewSafeAreaLayoutGuide-bottom' V:
    [UILayoutGuide:0x600003a355e0'UIViewSafeAreaLayoutGuide']-(0)-|
    (active, names: '|':UIView:0x7fad7dc0d510 )>"
)

Will attempt to recover by breaking constraint
<NSLayoutConstraint:0x600002069270 UIView:0x7fad7dc0f2f0.height
== 100    (active)>

Make a symbolic breakpoint at
UIViewAlertForUnsatisfiableConstraints to catch this in the
debugger.
The methods in the UIConstraintBasedLayoutDebugging category on
UIView listed in <UIKitCore/UIView.h> may also be helpful.
```

Although your log may look slightly different, the general information is the same:

1. First, there's a suggestion that you might have too many constraints. If you see one that you didn't expect, you might need to remove it.
2. Next, you'll see all of the constraints related to the affected view using the Visual Format Language. If you need a refresher on this format, refer to Chapter 4, “Construct Auto Layout with Code.”

3. "<NSLayoutConstraint:0x600002069270 UIView:0x7fad7dc0f2f0.height == 100 (active)>". The first piece, NSLayoutConstraint:0x600002069270, gives the memory address of the NSLayoutConstraint object for each line. The next part, UIView:0x7fad7dc0f2f0, gives the memory address of the view this constraint is attached to. This memory address may not be meaningful to you, but it can help you know when two constraints refer to the same view if they have the same memory address. The last part, .height == 100, tells you what the constraint is. This one simply sets the height to be equal to 100.
4. "<NSLayoutConstraint:0x600002069900 UILayoutGuide:0x600003a355e0'UIViewSafeAreaLayoutGuide'.bottom == UIView:0x7fad7dc0f2f0.bottom + 800 (active)>". This constraint sets the bottom of a view — you can see it's the same one as the view in the first line by the memory address — to 800 points above the bottom of the safe area.
5. "<NSLayoutConstraint:0x600002069950 V:|-0)-[UIView:0x7fad7dc0f2f0] (active, names: '|':UIView:0x7fad7dc0d510 )>". In this line, after the memory address of the constraint, you see V:. From the Visual Format Language, you know that this means a vertical constraint. This is followed by |-0)-[UIView:0x7fad7dc0f2f0]. You know that | refers to a view's superview, -(0)- means there is 0 vertical distance between the superview and the view, and [UIView:0x7fad7dc0f2f0] tells you which view, again, the same one from the first two lines. So, a space of 0 between the top of a view and its superview means the top of the view is aligned with its superview. The last part of the line, names: '|':UIView:0x7fad7dc0d510, gives you the memory address of the superview.
6. "<NSLayoutConstraint:0x600002074910 'UIView-Encapsulated-Layout-Height' UIView:0x7fad7dc0d510.height == 667 (active)>". This one refers to the superview height constraint. This is not a constraint that you added; it was added by the system to set the height of the view controller's root view. You can see from the memory address that it's the same superview referenced in the line above.
7. "<NSLayoutConstraint:0x600002069860 'UIViewSafeAreaLayoutGuide-bottom' V:[UILayoutGuide:0x600003a355e0'UIViewSafeAreaLayoutGuide']-(0)-|(active, names: '|':UIView:0x7fad7dc0d510 )>". This one sets the bottom of the safe area layout guide to the bottom of the superview. Just like the previous one, this one is created by the system.
8. Next, it's telling you that it will try to fix the problem by removing one constraint for you. In this case: <NSLayoutConstraint:0x600002069270 UIView:0x7fad7dc0f2f0.height == 100 (active)>. However, because it picks a random constraint to remove, you should never rely on this.

- Finally, it suggests that you create a symbolic breakpoint, which you'll do in the next section.

When you put all of this together, you'll see that the view is set to be:

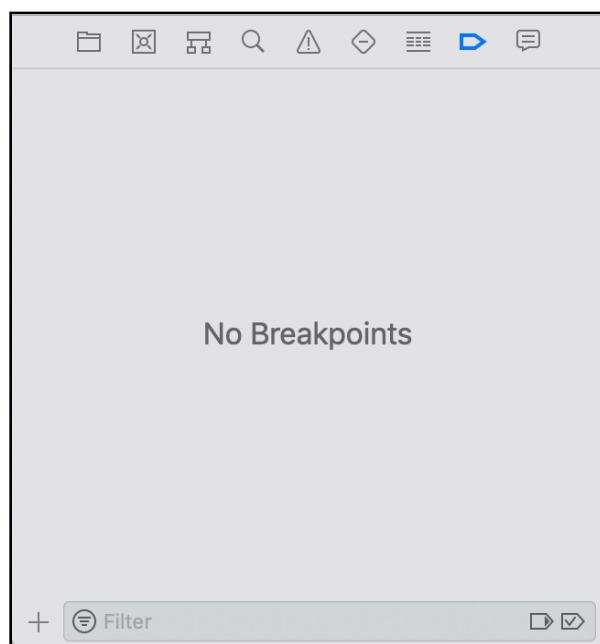
- 100 points high
- at the top of the superview
- 800 points from the bottom of the superview

This requires the superview to be 900 points high. However, the superview itself is only 667 points high. There's no way for Auto Layout to satisfy all of the constraints!

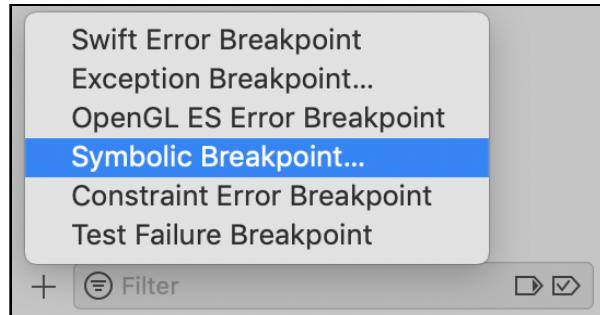
## Symbolic Breakpoints

When you create a Symbolic Breakpoint at `UIViewAlertForUnsatisfiableConstraints`, the app will stop running if it finds an unsatisfiable constraint. This is useful because sometimes the app may appear OK, when in fact, it has some issues you need to address.

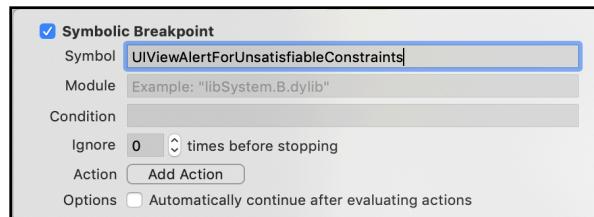
Press **Command-8** to go to the **Breakpoint navigator**.



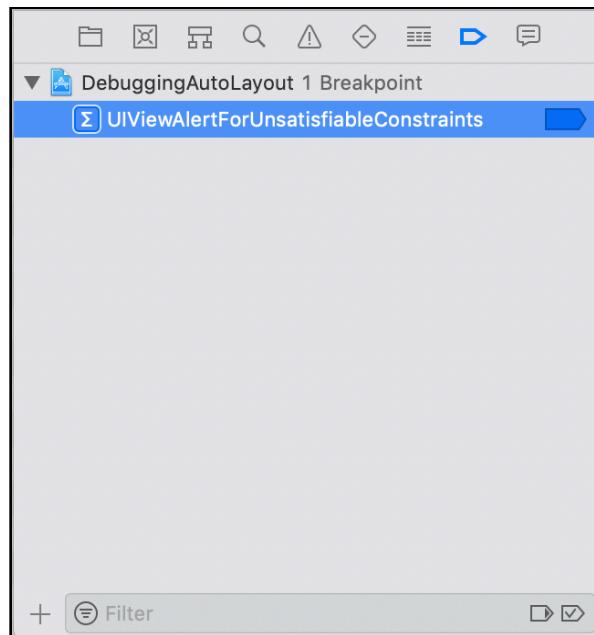
On the bottom left, click +, and then select **Symbolic Breakpoint...**



When the pop-up appears, enter **UIViewAlertForUnsatisfiableConstraints** in the Symbol field and press **Enter**.



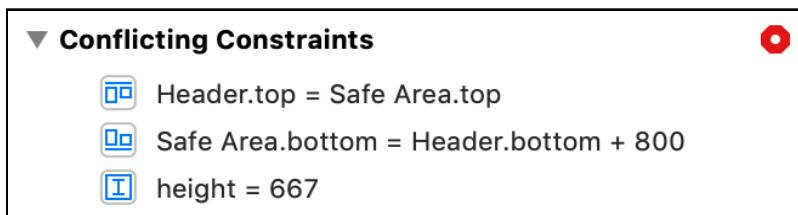
You'll now see the new breakpoint in the Breakpoint navigator on the left.



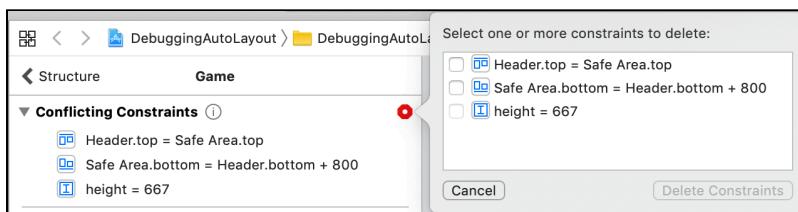
Build and run. Notice the app pauses when it encounters this issue.

## Using Interface Builder to solve conflicts

Open **Main.storyboard**. In the document outline, click the **red circle with an arrow** at the right of the Game Scene, and you'll see a list of the conflicting constraints.



Now, click on the **red circle** at the right of the conflicting constraints header to get more information.



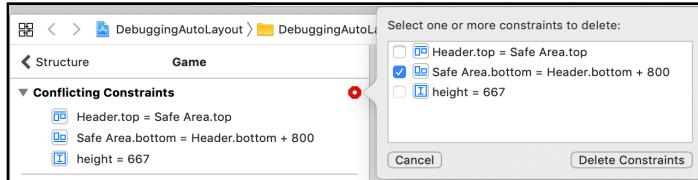
Since Auto Layout can not satisfy all of these constraints, you'll need to delete one. But how do you know which one to remove?

Well, that's something that comes from a combination of experience and knowing how the interface needs to look. One way to get a better look at these issues is by selecting the element in the document outline.

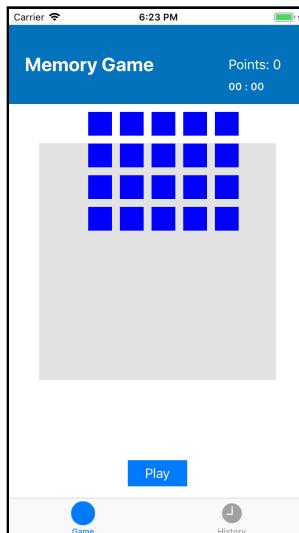
Click the **< Structure** button to go back to the document outline. Then, select the **Header** view, you'll see some red lines, which indicates that there are conflicts related to that element.

You can also go to the Size inspector and look at the constraints for that element. For the header, you have height, trailing, leading, bottom and top constraints. If you think about it, it doesn't make much sense to have height, top and bottom constraints since you only need two of those for the engine to be able to infer the other. If you specify top and height, the engine can calculate the bottom; or if you specify top and bottom, the engine can calculate the height; but when you specify top, bottom and height, you have unsatisfiable constraints.

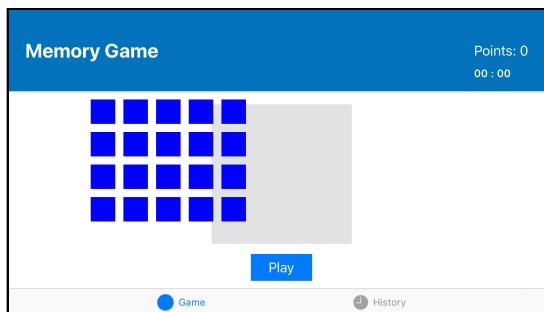
Remove **Safe Area.bottom = Header.bottom + 800**. To do this, click the **red circle** again, and this time select the constraint to be removed and click **Delete Constraints**.



The screen no longer shows red lines. Fantastic, you just got rid of the conflicting constraint! Now, build and run.



Well, things are looking better, but some views are still not quite right. Also, if you rotate the simulator (**Command-left arrow**), you'll see the app does not manage the rotation well, and a lot of things are out of place.



First, the board should stay in the center regardless of the orientation.

Open **BoardViewController.swift** and go to `createBoard()`, which is where the logic is that creates the board.

There is one fundamental problem with the way the board is created, all of the boxes are created independently, and there isn't a container that you can center. Also, it will be nice to use stack views to group the boxes in a better way, as you saw in Chapter 3, "StackViews," many benefits come with the use of them.

Replace `createBoard()` with this new one:

```
private func createBoard() {
    var tagFirstColumn = 0
    let numberOfRows = 4
    let numberOfColumns = 4

    //1
    let boardStackView = UIStackView()
    boardStackView.axis = .vertical
    boardStackView.distribution = .fillEqually
    boardStackView.spacing = 10
    boardStackView.translatesAutoresizingMaskIntoConstraints =
        false

    //2
    containerView.addSubview(boardStackView)
    view.addSubview(containerView)

    //3
    NSLayoutConstraint.activate([
        containerView.topAnchor.constraint(equalTo:
            boardStackView.topAnchor),
        containerView.leadingAnchor.constraint(equalTo:
            boardStackView.leadingAnchor),
        containerView.trailingAnchor.constraint(equalTo:
            boardStackView.trailingAnchor),
        containerView.bottomAnchor.constraint(equalTo:
            boardStackView.bottomAnchor)
    ])

    for _ in 0..<numberOfRows {
        //4
        let boardRowStackView = UIStackView()
        boardRowStackView.axis = .horizontal
        boardRowStackView.distribution = .equalSpacing
        boardRowStackView.spacing = 10

        //5
        for otherIndex in 0..<numberOfColumns {
            let button =
```

```
        createButtonWithTag(otherIndex + tagFirstColumn)
    buttons.append(button)
    boardRowStackView.addArrangedSubview(button)
}

//6
boardStackView.addArrangedSubview(boardRowStackView)

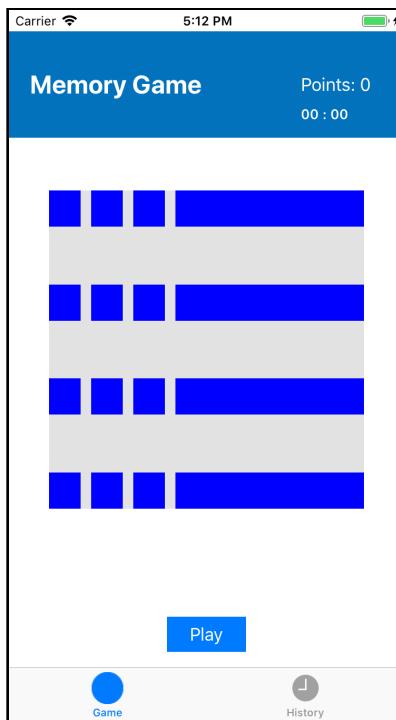
//7
tagFirstColumn += numberOfRows
}

//8
blockBoard()
}
```

With this new function, you:

1. Create a `boardStackView` `UIStackView` and set its axis, distribution and spacing. You also set its `translatesAutoresizingMaskIntoConstraints` property to `false`. This stack view will contain all of the rows, which is why its orientation is vertical. The distribution is set to `fillEqually` and the spacing to `10`. This set up guarantees the rows will have an equal separation of `10` between each other.
2. Add `boardStackView` to `containerView`. This view is centered, so it makes sense to put the board inside of it.
3. Set top, leading, trailing and bottom constraints to be equal between `boardStackView` and `containerView`. Now, the container can grow depending on the board size.
4. Create a `boardRowStackView`. These are the rows that form the board. You set the axis to be horizontal and the distribution to be `equalSpacing` with a spacing of `10` to get some padding between the boxes.
5. Create a loop to add the boxes to the row.
6. Add `boardRowStackView` to `boardStackView`.
7. Increment `tagFirstColumn` so that you start with the right value for the next row.
8. Call `blockBoard()` so that users cannot play until they tap the play button.

Build and run.



**Spoiler alert:** This layout is still ambiguous, so you may see something slightly different here. You'll address that in the next section.

There's still some work to do, but the board is now properly centered.

Before moving on to the next section, here are some tips to prevent Unsatisfiable Constraints:

- Remember to set `translatesAutoresizingMaskIntoConstraints` property to `false` when creating layouts by code.
- Try to use priorities wisely, i.e., set priority to be 999, when possible, so the Engine knows which constraint can be broken, if necessary. Not all constraints should be required.
- Avoid giving views with an intrinsic content size a required content hugging or compression resistance. Unless it's a special case, let the Layout handle it.

## Ambiguous layouts

When the Auto Layout Engine arrives at multiple solutions for a system of constraints, you'll have what's known as an ambiguous layout.

Although, Interface Builder can offer you solutions with problematic views during design, what happens when your Auto Layout issues happen during runtime?

Fortunately, there are view debug methods available that can help.

### UIView debug methods

UIView comes with four useful methods that you can use for debugging your Auto Layout issues:

- **hasAmbiguousLayout**: Returns `true` if the view frame is ambiguous.
- **exerciseAmbiguityInLayout**: Randomly selects a solution to illustrate the ambiguity.
- **constraintsAffectingLayout**: Returns an array of the constraints affecting the view on the specified axis.
- **\_autolayoutTrace**: Returns a string with the information regarding the view hierarchy that contains the view.

### Dealing with ambiguous layouts

Go back to the starter project and build and run.

Once the app starts, check the logs. There's nothing there, but the layout doesn't look quite right. Press **Control-Command-Y** to pause the program execution.

Now, type the following in the console:

```
po [[UIWindow keyWindow] _autolayoutTrace]
```

Scroll to the top, and you'll see something like this:

```
(lldb) po [[UIWindow keyWindow] _autolayoutTrace]

UIWindow:0x7f9ff8c245b0
|   UILayoutContainerView:0x7f9ff8b01060
|   |   UITransitionView:0x7f9ff8b08bc0
|   |   |   UIViewControllerWrapperView:0x7f9ff8b0eab0
|   |   |   |   *UIView:0x7f9ff8b0bff0
|   |   |   |   *UILayoutGuide: 0x600002340620 -
|   |   |   |   "UIViewSafeAreaLayoutGuide", layoutFrame = {{0, 20}, {375, 598}},
|   |   |   |   owningView = <UIView: 0x7f9ff8b0bff0; frame = (0 0; 375 667);
|   |   |   |   autoresize = W+H; layer = <CALayer: 0x600001a6f1e0>>>
|   |   |   |   *UIView:0x7f9ff8b0ace0
|   |   |   |   |   *UILabel:0x7f9ff8b0aec0'Memory Game'
|   |   |   |   |   |   *UILabel:0x7f9ff8b0bd00'Points: 0'
|   |   |   |   |   *UILabel:0x7f9ff8b0c1d0
|   |   |   |   |   *UIButton:0x7f9ff8b09fb0'Play'
|   |   |   |   |   |   *UIButtonLabel:0x7f9ff740cd60'Play'
|   |   |   |   *ContainerView:0x7f9ff8b0a9f0
|   |   |   |   |   *UIStackView:0x7f9ff8b09cd0
|   |   |   |   |   |   *UIStackView:0x7f9ff8b0f2f0
|   |   |   |   |   |   |   *_UIOLAGapGuide: 0x600002654900 -
|   |   |   |   |   |   |   "UISV-distributing", layoutFrame = {{30, 0}, {59.5, 0}}, owningView
|   |   |   |   |   |   = <UIStackView: 0x7f9ff8b0f2f0; frame = (0 0; 299 67.5); layer =
|   |   |   |   |   |   <CATransformLayer: 0x600001a6fc40>>>- AMBIGUOUS LAYOUT for
|   |   |   |   |   |   |_UIOLAGapGuide:0x600002654900'UISV-distributing'.minY{id: 441},
|   |   |   |   |   |   |_UIOLAGapGuide:0x600002654900'UISV-distributing'.Height{id: 442}
|   |   |   |   |   |   |   *_UIOLAGapGuide: 0x600002654200 -
|   |   |   |   |   |   |   "UISV-distributing", layoutFrame = {{119.5, 0}, {60, 0}},
|   |   |   |   |   |   |   owningView = <UIStackView: 0x7f9ff8b0f2f0; frame = (0 0; 299 67.5);
|   |   |   |   |   |   |   layer = <CATransformLayer: 0x600001a6fc40>>>- AMBIGUOUS LAYOUT for
|   |   |   |   |   |   |   |_UIOLAGapGuide:0x600002654200'UISV-distributing'.minY{id: 443},
|   |   |   |   |   |   |   |_UIOLAGapGuide:0x600002654200'UISV-distributing'.Height{id: 444}
|   |   |   |   |   |   |   |   *_UIOLAGapGuide: 0x600002654400 -
|   |   |   |   |   |   |   |   "UISV-distributing", layoutFrame = {{209.5, 0}, {59.5, 0}},
|   |   |   |   |   |   |   |   owningView = <UIStackView: 0x7f9ff8b0f2f0; frame = (0 0; 299 67.5);
|   |   |   |   |   |   |   |   layer = <CATransformLayer: 0x600001a6fc40>>>- AMBIGUOUS LAYOUT for
|   |   |   |   |   |   |   |   |_UIOLAGapGuide:0x600002654400'UISV-distributing'.minY{id: 445},
|   |   |   |   |   |   |   |   |_UIOLAGapGuide:0x600002654400'UISV-distributing'.Height{id: 446}
|   |   |   |   |   |   |   |   |   *_UIOLAGapGuide: 0x600002654000 -
|   |   |   |   |   |   |   |   |   "UISV-distributing", layoutFrame = {{30, 0}, {59.5, 0}}, owningView
|   |   |   |   |   |   = <UIStackView: 0x7f9ff8b10ab0; frame = (0 77.5; 299 67); layer =
```

**Note:** The command you entered uses Objective-C syntax. When you stop the debugger like this, it stops in framework code, which is written in Objective-C. If you get to this point by setting a breakpoint inside your Swift code, you can use expression `-l objc -O -- [[UIWindow keyWindow] _autolayoutTrace]`. The first part of the command expression `-l objc -O --` tells LLDB that you want to run Objective-C code. After that, you type the command you want to run.

What you see is the view hierarchy of the app. Notice there are several guides in a stack view with the notation “AMBIGUOUS LAYOUT.” By looking through the hierarchy, you can see that this stack view contains several others, so you can identify it as the boardStackView.

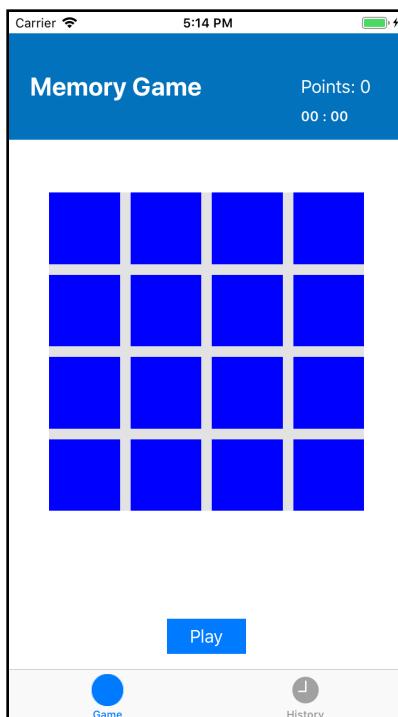
```
*UIStackView:0x7f8be3405bf0
| | | | *-<_UIOLAGapGuide: 0x6000018f9e00 - "UISV-
distributing", layoutFrame = {{0, 50}, {0, 10}}, owningView =
<UIStackView: 0x7f8be3405bf0; frame = (0 0; 230 230); layer =
<CATransformLayer: 0x600002497940>>>- AMBIGUOUS LAYOUT for
_UIOLAGapGuide:0x6000018f9e00'UISV-distributing'.minX{id: 595},
_UIOLAGapGuide:0x6000018f9e00'UISV-distributing'.Width{id: 596}
```

Notice there are some ambiguities between the horizontal position and the width of the UIOLAGapGuide elements; these are responsible for the space between the views inside the UIStackView.

You want the subviews of all of the stack views to fill the space, so you’ll need to change the distribution for the stack view.

In `createBoard()`, change `boardRowStackView.distribution` to `.fillEqually`.

Now, build and run.



Once again, pause the program execution (**Control-Command-Y**), and type this on the console:

```
po [[UIWindow keyWindow] _autolayoutTrace]
```

The view tree is already looking better, but there's another issue with a label at the bottom.

```
| | | | *DebuggingAutoLayout.TimerView:0x7fc5d609630
| | | | <UILayoutGuide: 0x6000021081c0 - "UIViewsSafeAreaLayoutGuide",
| | | | owningView = <DebuggingAutoLayout.TimerView:
| | | | 0x7fc5d609630; frame = (277 90.5; 67 15); layer = <CALayer: 0x600001801600>>
| | | | | *UILabel:0x7fc5d609c60'00 : 00'- AMBIGUOUS LAYOUT for
| | | | UILabel:0x7fc5d609c60'00 : 00'.minX{id: 346}
| | | | UITabBar:0x7fc5ec09790
| | | | | _UIBarBackground:0x7fc5d5d40b770
| | | | | UIImageView:0x7fc5d5d40bc60
| | | | | UIImageView:0x7fc5d5d40be30
| | | | | _UIVisualEffectView:0x7fc5d5d40be30
| | | | | _UIVisualEffectBackdropView:0x7fc5d5d502f50
| | | | | _UIVisualEffectSubview:0x7fc5d5d500040
| | | | UITabBarButton:0x7fc5ec08a00
| | | | | UITabBarSwappableImageview:0x7fc5ec0c7c0
| | | | | UITabBarButtonLabel:0x7fc5ec0a050'Game'
| | | | UITabBarButton:0x7fc5d609830
| | | | | UITabBarSwappableImageview:0x7fc5d5d40b520
| | | | | UITabBarButtonLabel:0x7fc5d4ac90'History'

Legend:
* - is laid out with auto layout
+ - is laid out manually, but is represented in the layout engine because
  translatesAutoresizingMaskIntoConstraints = YES
• - layout engine host
```

Copy the memory address for the label that has the issue; in this case, it's **0x7fc5d609c60**; however, that address will be different for you. Once you have it copied, hang on to it because you'll need it soon.

## Visual debugging

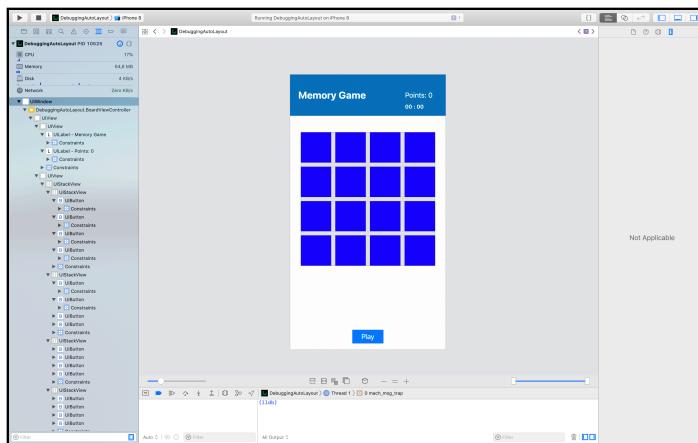
To solve the remaining issue, you'll use the **Debug View Hierarchy**, which gives a visual way to look at how the layout is structured.

Click **Debug View Hierarchy** in the debug bar. It doesn't matter if the app is running or paused; it'll work in either case.



Xcode now displays a new window with the view of the app in the middle. Also, there's a new toolbar you haven't seen before. This new interactive 3D model gives you a lot of useful information.

You can use the controls in the toolbar at the bottom to navigate the elements that make up the view.



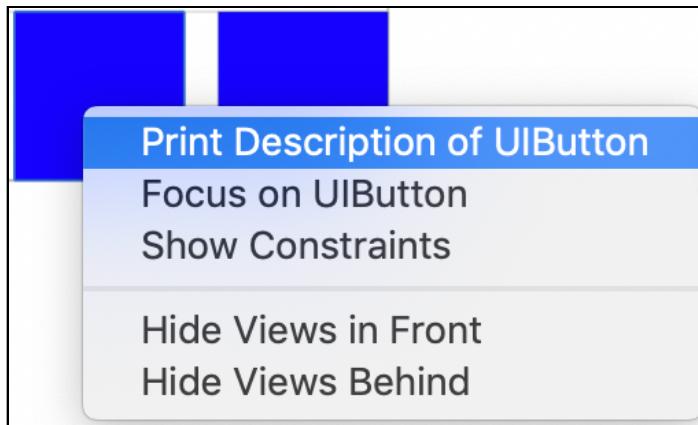
## View hierarchy toolbar

So what does this new toolbar do? From left to right:



- **First slider:** This sets the amount of spacing between layers of views. If you're in 2D mode, it switches to the 3D mode when you change this.
- **Clipping button:** Shows/hides clipped content. This view doesn't have any clipped content, so it doesn't do anything obvious.
- **Show constraints button:** With this mode on, when you select an object, it shows its constraints in a similar way to how they are displayed in Interface Builder.
- **Adjust view mode:** With this option, you can choose between three display modes: content, wireframes or both. It hides one or another depending on your selection.
- **Change canvas background color:** Changes the canvas to light or dark, which lets you see the views for each mode.
- **Orient to 3D/2D:** Switches between 3D or 2D mode for visualization.
- **Zoom controls:** Zooms in or out.
- **Range slider:** Filters the range of views shown by depth. This allows you to focus only on the layers of views you're working with.

**Right-click** any of the blue boxes, and you'll see a menu.

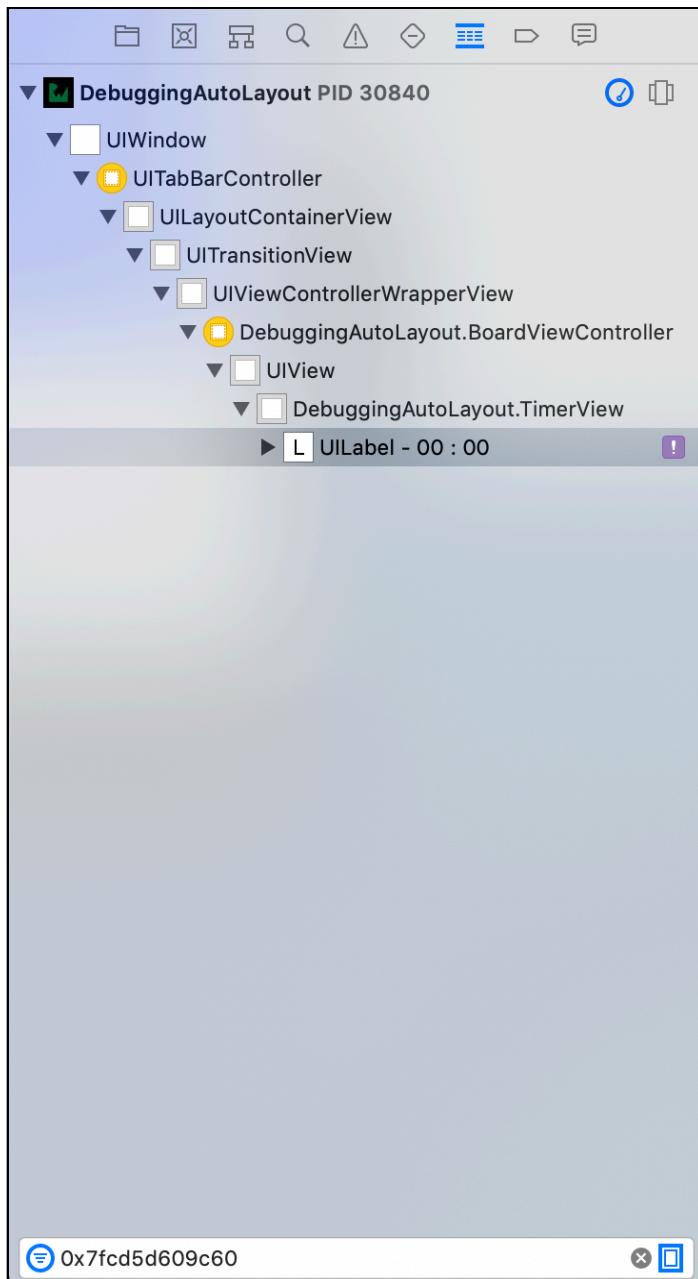


Here's what you can do, from top to bottom:

- **Print description:** Performs a `po` command on the selected object and prints the information in the console.
- **Focus on -name of the object-:** Filters the display to only show that view's subviews.
- **Show constraints:** Shows constraints just like using the button in the toolbar.
- **Hide views in front/behind:** Adjusts the range sliders to hide views above or below this view.

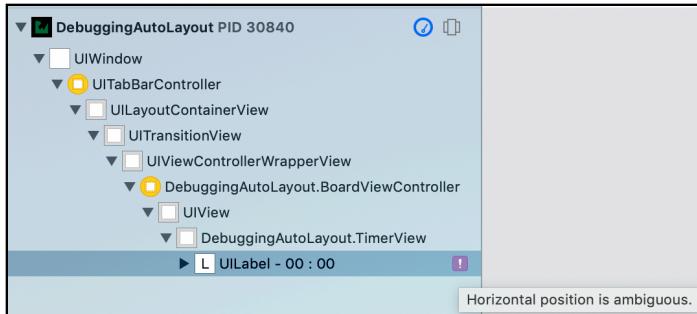
## Handling runtime issues

Using the memory address copied from the layout trace, paste it on the filter located at the bottom-left side of the screen.

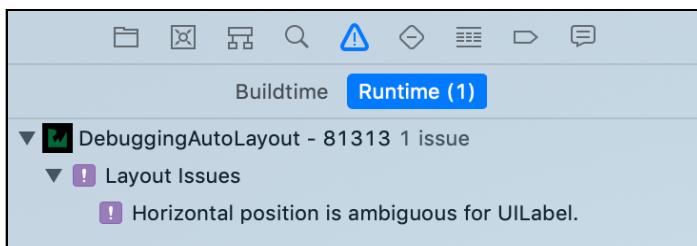


You'll notice that the view hierarchy changed, and now it's showing just the parts of the tree related to the view you specified, pretty neat. This comes in handy when you want to see the view causing the trouble, and you only have the memory address.

There's a purple icon with an exclamation sign. This tells you there's something wrong with that view. If you put your mouse over this icon, you'll see a tooltip telling you the reason behind the issue.



There's still one more way to get a list of issues: Press **Command-5** to show the Issue navigator. Now, click the **Runtime** tab.



Here, you can see issues that are affecting your layout at runtime. Normally, you'd use this to see build time issues, but for this chapter, you'll focus only on the runtime issues.

Notice there's an issue indicating that **Horizontal position is ambiguous for UILabel**. Click on the message in the Issue navigator, and the affected view, in this case, the label displaying "00:00", is highlighted. It looks like there's a missing constraint for the label that shows the timer.

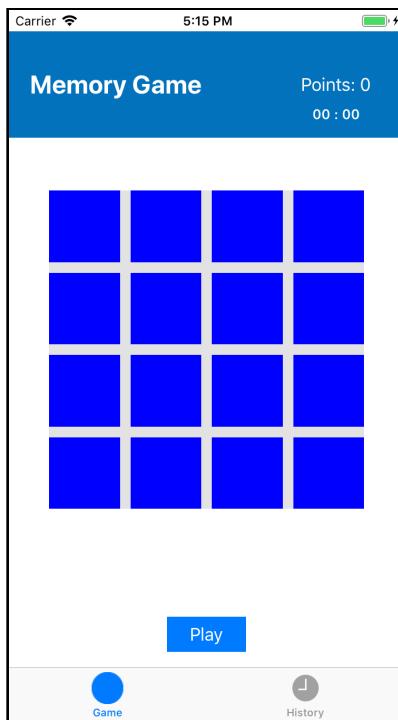
Stop the project. Go to **TimerView.swift** and locate `updateConstraints()`. There's only one constraint for `timerLabel`: its vertical position. To center it horizontally, you need to add one more constraint.

Add this new constraint immediately before the existing one:

```
labelConstraints.append(  
    timerLabel.centerXAnchor.constraint(  
        equalTo: safeAreaLayoutGuide.centerXAnchor))
```

Build and run.

Perfect, the label is now centered, and there are no runtime issues.



## Common performance issues

The Auto Layout Engine does a lot of things for you. It acts as a dependency cache manager, making things faster, and Apple makes performance-related improvements nearly every year. Nonetheless, you have to be aware of how it works to avoid the unnecessary use of resources.

In the next chapter, you'll learn about the Render Loop and the Auto Layout Engine, which is key to understanding how to avoid performance issues with Auto Layout.

## Churning

Churning is a common issue and one that can easily fly above any developer's head. It usually happens when you create multiple constraints — many times without them being necessary.

Go to **TimerView.swift** and look at `updateConstraints()`:

```
override func updateConstraints() {
    NSLayoutConstraint.deactivate(labelConstraints)
    labelConstraints.removeAll()

    labelConstraints.append(
        timerLabel.centerXAnchor.constraint(
            equalTo: safeAreaLayoutGuide.centerXAnchor))
    labelConstraints.append(
        timerLabel.centerYAnchor.constraint(
            equalTo: safeAreaLayoutGuide.centerYAnchor))

    NSLayoutConstraint.activate(labelConstraints)
    super.updateConstraints()
}
```

This code works fine; however, the Render Loop runs 120 times every second, and in this scenario, that means you're deactivating and activating this set of constraints every time.

Since this is a simple UI, you won't notice performance issues, but that won't be the case when you create collection views or any complicated layout.

To fix this problem, change the code to this:

```
override func updateConstraints() {
    if labelConstraints.isEmpty {
        labelConstraints.append(
            timerLabel.centerXAnchor.constraint(
                equalTo: safeAreaLayoutGuide.centerXAnchor))
        labelConstraints.append(
            timerLabel.centerYAnchor.constraint(
                equalTo: safeAreaLayoutGuide.centerYAnchor))

        NSLayoutConstraint.activate(labelConstraints)
    }
    super.updateConstraints()
}
```

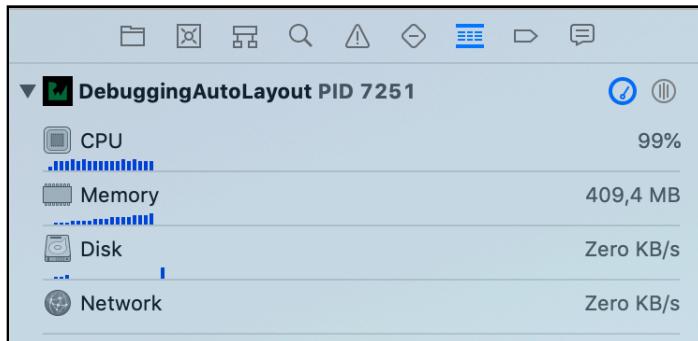
Here, you check if `labelConstraints` contains any data. If data exists, that means the constraints were already added, so you don't need to add them again. You also removed some unnecessary code since you don't have to deactivate the constraints anymore. With these few changes, your code is much more efficient.

## Layout feedback loop

Until now, you haven't seen the second tab of the app: History.

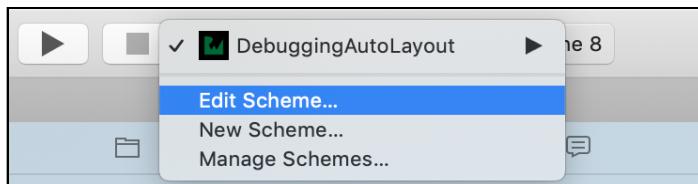
Build and run, and tap the History tab. Oh no, the app is unresponsive! Don't worry; you're about to fix that.

Go to Xcode and press **Command-7** to show the Debug navigator.



Pay close attention to the CPU and Memory indicators. See that? There's definitely something wrong with the app.

Open the Scheme editor by clicking on top of the app name on the top left side, like this:

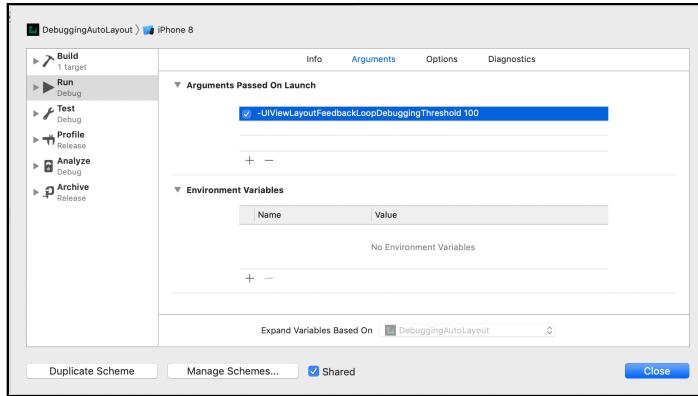


Click on **Edit Scheme**.

Add a new argument on the **Arguments Passed on Launch** section, and enter:

```
-UIViewLayoutFeedbackLoopDebuggingThreshold 100
```

Once you're done, click **Close**.



Build and run. Tap the History tab, and the app still freezes. But this time, when you go to Xcode, you'll see a log. Now, the app is properly crashing.

```
2019-05-08 20:49:48.193149+0200 DebuggingAutoLayout[10441:75492] [LayoutLoop] Layout
feedback loop debugging enabled via -UIViewLayoutFeedbackLoopDebuggingThreshold launch
argument with a threshold of 100
2019-05-08 20:49:51.071877+0200 DebuggingAutoLayout[10441:75492] [LayoutLoop] About to
send -setNeedsLayout to layer for <UITableView>: 0x7fd8ec82ae00; f={0, 0}, {375, 667}
(
    0 UIKitCore          0x0000000114bc4f12 -
    [_UIVViewLayoutFeedbackLoopDebugger _recordSetNeedsLayoutToLayerOfView:] + 760
    1 UIKitCore          0x0000000114b7a9b7-[UIView(Hierarchy)
    setNeedsLayout] + 164
    2 UIKitCore          0x0000000114b27317 -
    [UIScrollView(UIScrollViewInternal) setNeedsLayout] + 76
    3 DebuggingAutoLayout
    $S19DebuggingAutoLayout25ScoresTableViewControllerC07viewDidC8SubviewsyF + 225
    4 DebuggingAutoLayout
    $S19DebuggingAutoLayout25ScoresTableViewControllerC07viewDidC8SubviewsyFTo + 36
    5 UIKitCore          0x0000000114b90914-[UIView(CALayerDelegate)
    layoutSublayersOfLayer:] + 1824
    6 QuartzCore         0x0000000116118b19-[CALayer layoutSublayers]
    + 175
    7 QuartzCore         0x000000011611d9d3
    _ZN2CA5Layer16layout_if_neededEPNS_11TransactionE + 395
    8 QuartzCore         0x000000011609967ca
    _ZN2CA7Context18commit_transactionEPNS_11TransactionE + 342
    9 QuartzCore         0x00000001160cd97e
    _ZN2CA11Transaction6commitEv + 576
    10 QuartzCore         0x00000001160ce6fa
    _ZN2CA11Transaction17observer_callbackEP19__CFRunLoopObservermPv + 76
    11 CoreFoundation
    __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__ + 23
    12 CoreFoundation
    0x0000000110cb98be __CFRunLoopDoObservers +
    438
    13 CoreFoundation
    0x0000000110cb9751 __CFRunLoopRun + 1537
    14 CoreFoundation
    0x0000000110cb8e11 CFRunLoopRunSpecific + 625
    15 GraphicsServices
    0x0000000119e371dd GSEventRunModal + 62
    16 UIKitCore
    0x00000001146a681d UIApplicationMain + 140
    17 DebuggingAutoLayout
    0x0000000110fb98a27 main + 71
    18 libdyld.dylib
    0x00000001131b3575 start + 1
)
2019-05-08 20:49:51.071877+0200 DebuggingAutoLayout[10441:75492] [LayoutLoop] About to
send -layoutSubviews to <UITableView>: 0x7fd8ec82ae00; f={0, 0}, {375, 667} >.
(
    0 UIKitCore          0x0000000114bc4311 -
    [_UIVViewLayoutFeedbackLoopDebugger willSendLayoutSubviewsToView:] + 1042
    1 UIKitCore          0x0000000114b9077d-[UIView(CALayerDelegate)
    layoutSublayersOfLayer:] + 1417
    2 QuartzCore         0x0000000116118b19-[CALayer layoutSublayers]
    + 175
    3 QuartzCore         0x000000011611d9d3
    _ZN2CA5Layer16layout_if_neededEPNS_11TransactionE + 395
    4 QuartzCore         0x000000011609967ca
    _ZN2CA7Context18commit_transactionEPNS_11TransactionE + 342
    5 QuartzCore         0x00000001160cd97e
    _ZN2CA11Transaction6commitEv + 576
    6 QuartzCore         0x00000001160ce6fa
    _ZN2CA11Transaction17observer_callbackEP19__CFRunLoopObservermPv + 76
    7 CoreFoundation
    0x0000000110cbec27
    __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__ + 23
    8 CoreFoundation
    0x0000000110cb98be __CFRunLoopDoObservers +
    438
    9 CoreFoundation
    0x0000000110cb9751 __CFRunLoopRun + 1537
```

This is a rather large log. When dealing with the feedback loop you'll get this kind of log, and it's never too easy to find the issue. Reading the logs, you'll see some mentions to **setNeedsLayout** getting called and also there's this message:

```
DebuggingAutoLayout[10441:75492] [LayoutLoop] >>>UPSTREAM LAYOUT  
DIRTYING<<< About to send -setNeedsLayout to layer for  
<UITableView: 0x7fd8ec82ae00; f={{0, 0}, {375, 667}} > under  
-viewDidLayoutSubviews for <UITableView: 0x7fd8ec82ae00; f={{0,  
0}, {375, 667}}
```

It looks like **setNeedsLayout** might be causing this issue.

Go to **ScoresTableViewController.swift** and look for any calls to that function. You'll see it here:

```
override func viewDidLayoutSubviews() {  
    view.setNeedsLayout()  
}
```

By calling **setNeedsLayout()** in **viewDidLayoutSubviews()**, the app is creating an infinite loop because it's telling the Render Loop to schedule another update pass just after the first one is finished. Since that's not of any use, remove the function completely, then build and run.

You can now see the list of scores. If the list is empty, play until you lose and come back again.

Fantastic, you got rid of the bugs, the app is in great shape, and you now know how to deal with some of the common issues that affect Auto Layout.

## Key points

- There are three main types of Auto Layout issues: ambiguous layouts, unsatisfiable constraints and logic errors.
- You can solve Auto Layout issues faster using **UIView** methods, **Debug View Hierarchy** and **Interface Builder**.
- To avoid performance issues, you need to understand how the **Render Loop** and **Auto Layout Engine** works.

# Section III: Advanced Auto Layout

Deepen your knowledge of Auto Layout by exploring these more advanced topics. Specifically, you'll cover:

- **Chapter 14: Under the Hood of Auto Layout:** Pull back the curtain and see the inner workings of Auto Layout. Explore the math behind the magic and discover the “why” behind the behavior of the Auto Layout engine.
- **Chapter 15: Optimizing Auto Layout Performance:** Learn to fine-tune your app’s Auto Layout performance. Learn about best practices and about common mistakes that cost you performance.
- **Chapter 16: Layout Prototyping with Playgrounds:** Learn to use Xcode playgrounds to prototype your user interfaces. See how this technique can streamline your development process and make you more efficient.
- **Chapter 17: Auto Layout for External Displays:** Learn how to support external displays in your app. Learn to build a layout for external display, how to handle display connect and disconnect events, and how to accommodate different external display resolutions.
- **Chapter 18: Designing Custom Controls:** Learn techniques for creating custom user interface controls that work well with Auto Layout. Integrate many of the things you’ve learned in this book to create control animations, adjust to size classes and more.



# Chapter 14: Under the Hood of Auto Layout

By Jayven Nhan

By now, you've learned the different ways to add Auto Layout constraints. However, you may be wondering how the Auto Layout engine works its magic. Spoiler alert: It's not magic!

In this chapter, you'll dive into the inner workings of the Auto Layout engine and discover the math that makes Auto Layout so powerful. This chapter is less hands-on than the others, but if you're curious enough about the math behind the magic, you're in for a real treat.

This chapter will answer the following questions:

- What does Auto Layout use to create constraints?
- Where does Auto Layout fit in the grand scheme of processes in the app?
- What's the purpose of solving linear programming problems?
- How can I solve linear programming problems?
- What's the math behind solving equality and inequality constraints?
- How does the Cassowary algorithm work?

Rest assured, Auto Layout isn't a framework made by wizards from Hogwarts. At the low-level, Auto Layout computes arithmetic algorithms. But before looking at the Auto Layout algorithm, you're going to look at the differences between alignment and frame rectangles.

## Alignment rectangles versus frame rectangles

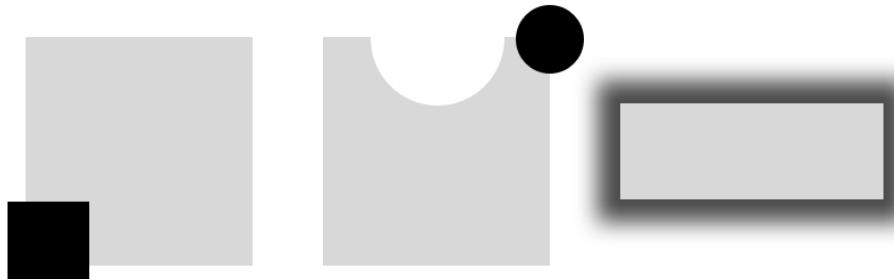
As you already know, Auto Layout helps you layout many views using relativity. Unfortunately, there's a common misconception on the "relativity" part. This section answers the question: What part of a view does the Auto Layout engine use to create a relativity relationship with another view? In other words, what does Auto Layout use to create constraints?

## Theory

It's a common misconception that Auto Layout uses **view frames** to create constraints. It doesn't; it uses **alignment rectangles** to create them.

A view's superview uses the view's frame rectangle values to position and size the view within its superview. Although a view can have identical alignment and frame rectangle values, knowing what Auto Layout uses to create constraints can help you avoid layout misalignments.

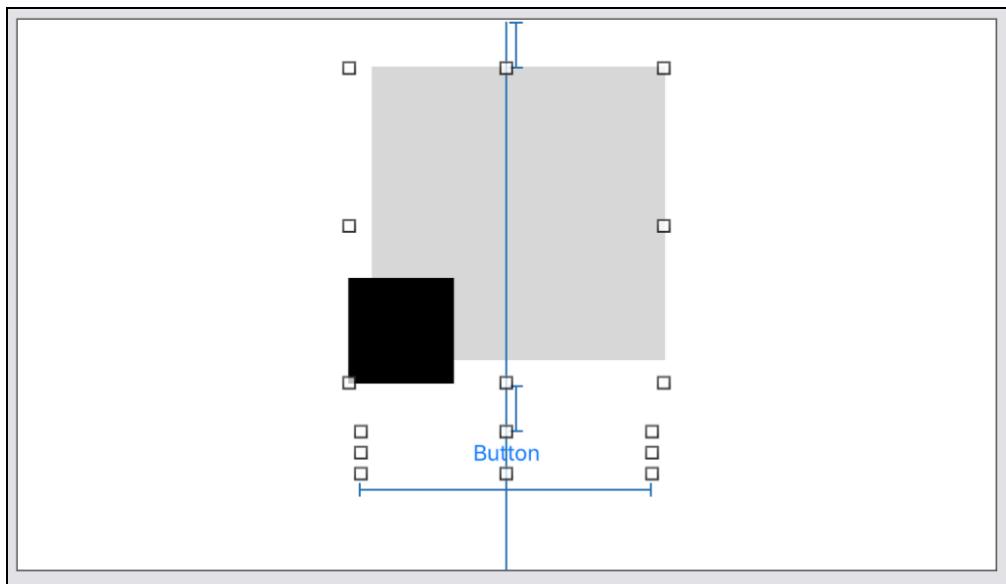
Review the following images:



If you make the wrong assumption about Auto Layout and view frames, any of the image assets in the previous figure could be problematic for your layout, so you need to be clear on the alignment edges of each view. If you're not, a view's alignment may appear different from what you expect. The incorrectly aligned view can also offset the alignments of other views, which can turn a single layout misalignment into a collection of additional misalignments.

## Working with alignment rectangles

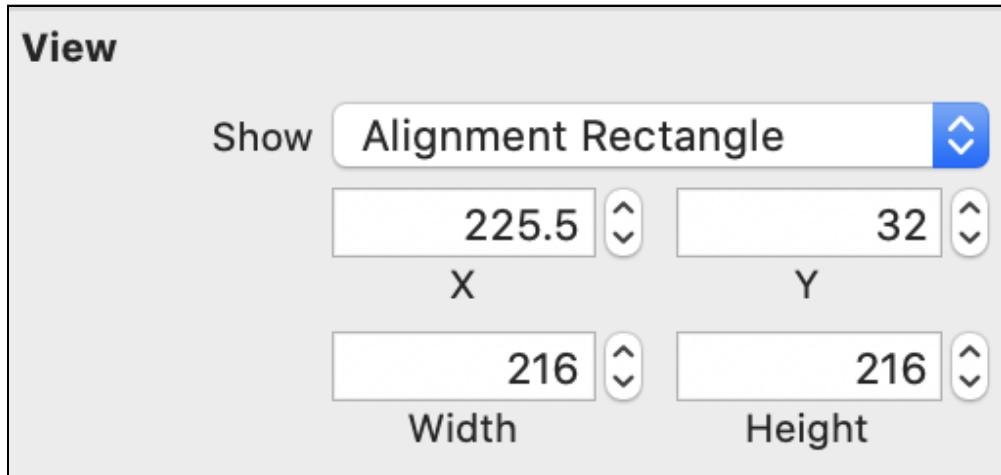
Look at the following image:



Here, you can see the image view's alignment rectangle extends to the black square's left, black square's bottom, grey square's top and grey square's right edges. The image view is horizontally centered based on the black and grey squares' combined width. The button's top edge aligns below the black square's bottom edge.

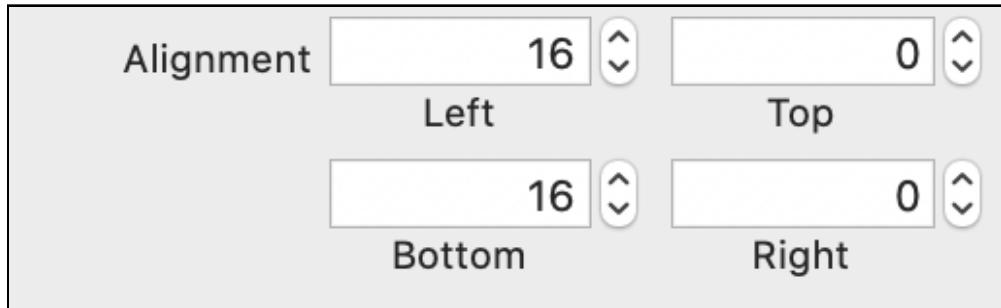
However, this may not be the layout you want. If you want the grey box edges as the alignment rectangle, you can update the view's alignment rectangle for the layout engine to correctly align the view.

Before making any adjustments, look at the image view's current alignment rectangle values:

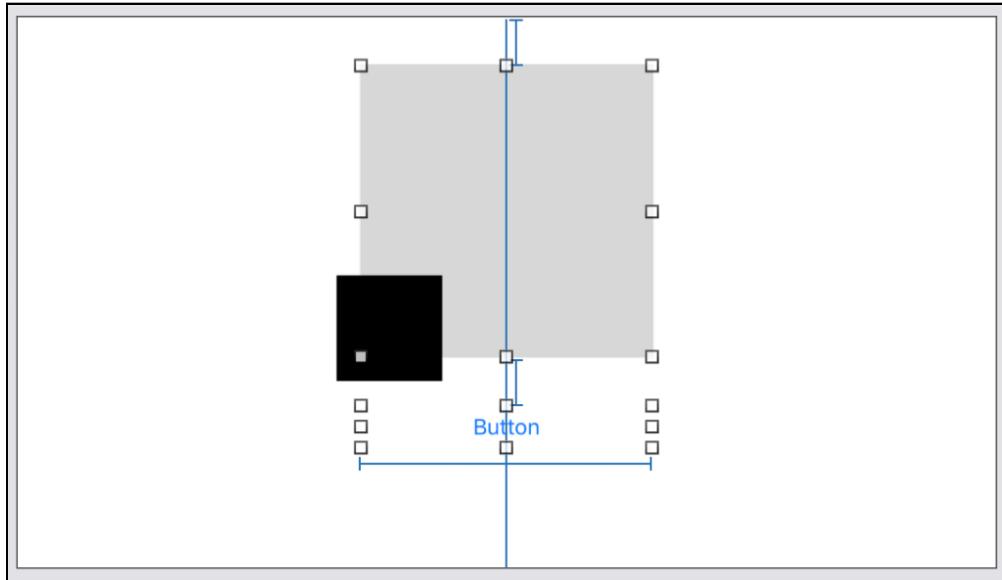


You can adjust a view's alignment rectangle with the assistance of the asset catalog. To make the alignment rectangle adjustments in your project, here's how you'd do it:

1. Open the **asset catalog**.
2. Click the **image asset**.
3. Click the **desired image(s)**. It's possible to have different image versions for different scales.
4. Click the **Attributes inspector**.
5. In the **Alignment** section, enter the inset values for the selected image. The alignment section looks something like this:



After the alignment adjustment, you'll see the changes in the storyboard. Here's the updated image view:



The image view's alignment rectangle extends to the grey square edges; the image view is horizontally centered within the container using the grey square; and the button's top edge aligns below the grey square's bottom edge.

To achieve the same solution using code, you can create a custom image view:

```
import UIKit

final class CustomImageView: UIImageView {
    override init(image: UIImage?) {
        let edgeInsets = UIEdgeInsets(
            top: 0,
            left: 16,
            bottom: 16,
            right: 0)
        let image = image?.withAlignmentRectInsets(edgeInsets)
        super.init(image: image)
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }
}
```

You can think of a view's alignment rectangle as containing core visual elements, while a view's frame contains the core visual elements plus ornaments.

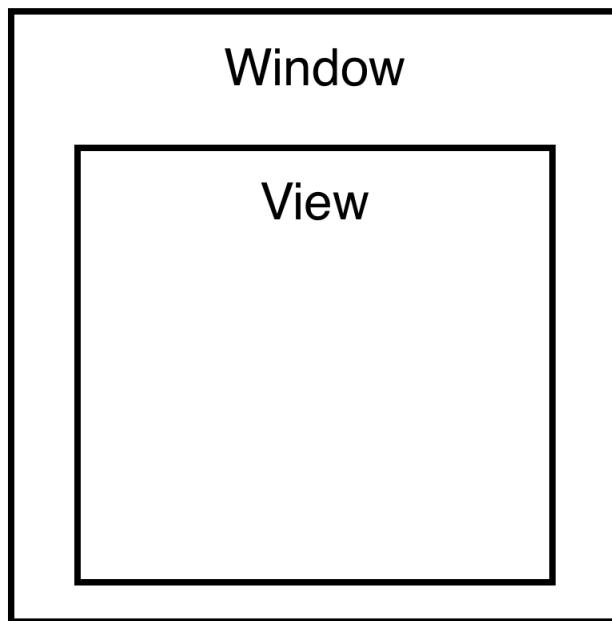
Now that you know Auto Layout uses alignment rectangles to create a constraint relationship between views, another question you may have is: How does Auto Layout fit in the big picture of the different parts of the app?

## Fitting Auto Layout in the big picture

In the process of view layouts, Auto Layout seems to “just” compute view layouts. Then, your user interface looks a certain way. This section covers the in-between view processes.

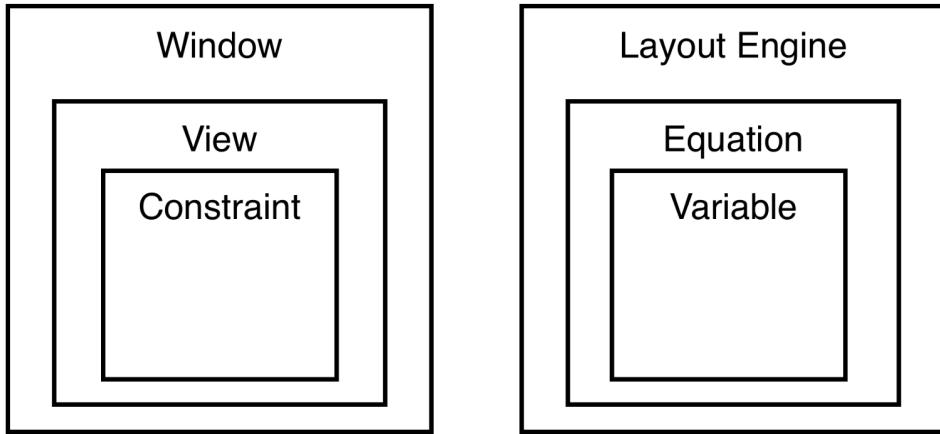
So, where is the layout engine in the grand scheme of processes in an app? And, how does it interact with other user interface components to create your view layouts?

Look at the diagram below:



Your app contains a window, which contains a view. This view is your blank canvas.

Next, you add additional user interface components. You add constraints onto the initial view, and more components come into play:



The initial view contains the constraints, and constraints convert into linear equations. The linear equations then pass into the Auto Layout engine for the layout calculations.

The layout engine calculates for layout variables. After solving the view's layout variables (x, y, width and height), the layout engine manages the caching of view layout variables and the tracking of view dependencies.

Saving the view's information is vital for Auto Layout's performance. It allows Auto Layout to mitigate extraneous or repetitive layout calculations. For example, a view's constraint constant can increment by a single point. Because of the cache, the layout engine doesn't need to re-calculate the layout variables on the view. It also doesn't need to re-calculate other views, which depend on the view's alignment rectangle.

Repeatedly doing this type of layout re-calculation can take a toll on your app's performance. Instead, the layout engine efficiently calculates the required parameters for the change in layout. After the layout engine calculates and saves the view's layout information, it passes the view's frame (size and position) back to the view. The view now has its essential layout information.

That's how Auto Layout works in the grand scheme of processes in an app. Now, on a high-level, you understand the in-between processes from adding constraints to the views appearing on an app. Next, you'll look at the low-level workings on how Auto Layout solves constraints.

## Solving constraints

Auto Layout uses Cassowary, a constraint solving algorithm, to solve constraints. After placing constraints on a view, the view still needs to determine its layout variables. For this reason, you need a constraints solver. Cassowary helps translate a view's constraints to layout variables.

With Auto Layout, you can create equality and inequality constraints, which represent either a linear equation or inequality, and add them to a view. The layout engine solves constraints for the layout variables.

Here's an example linear equation:

- Item1.Attribute = Multiplier x Item2.Attribute + Constant

And, here's a linear inequality:

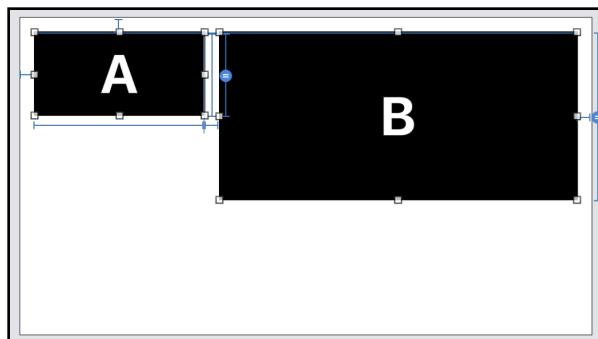
- Item1.Attribute >= Multiplier x Item2.Attribute + Constant

## Solving equality constraints

This section focuses on equality constraints and linear equations.

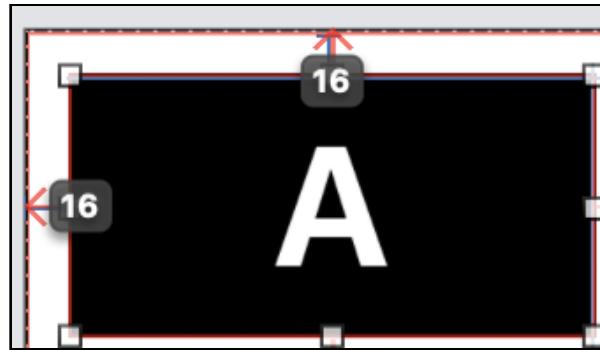
There are scenarios where you might have a constraint that positions a view a fixed number of points away from another view. Or maybe you have a constraint that indicates a view's width is a multiplier of another view's width. These are both examples of equality constraints. You'll need to solve equality constraints to translate them into layout variables.

Look at these two labels with equality constraints:



Label A has the following constraints:

- 200 points width.
- 100 points height.
- Top edge is 16 points from the superview's top edge.
- Leading edge is 16 points from the superview's leading edge.



The linear equations associated with the constraints are:

- $\text{LabelA.width} = 1.0 * \text{NotAnAttribute} + 200.0$
- $\text{LabelA.height} = 1.0 * \text{NotAnAttribute} + 100.0$
- $\text{LabelA.top} = 1.0 * \text{Superview.top} + 16.0$
- $\text{LabelA.leading} = 1.0 * \text{Superview.leading} + 16.0$

Note that `NotAnAttribute` equals 0.

From solving the constraints added, the layout frame properties come out as:

```
labelA.frame.minX = 16
labelA.frame.minY = 16
labelA.frame.width = 200
labelA.frame.height = 100
```

The above pseudocode assumes Label A's position is within its superview's coordinate system, and the superview's top and leading edges are zero in its own coordinate space.

Label A's layout properties calculations are straight forward here. The view positions 16 points down and right from the superview's top-left corner. Also, Label A has a constant width and height.

Now, the constraints added to Label B are more interesting:

- Top edge aligns with Label A's top edge.
- Leading edge is 16 points from Label A's trailing edge.
- Trailing edge is 16 points from the superview's trailing edge.
- Height equals two times Label A's height.

The linear equations are:

- $\text{LabelB.top} = 1.0 * \text{LabelA.top} + 0.0$
- $\text{LabelB.leading} = 1.0 * \text{LabelA.trailing} + 16$
- $\text{LabelB.trailing} = 1.0 * \text{Superview.trailing} - 16.0$
- $\text{LabelB.height} = 2.0 * \text{LabelA.height} + 0.0$

From solving the constraints added, the layout properties come out as follows:

```
labelB.frame.minY = labelA.frame.minY  
                  = 16  
  
labelB.frame minX = labelA.frame.minX + labelA.frame.width + 16  
                  = 16 + 200 + 16  
                  = 232  
  
labelB.frame.width = superview.frame.width - labelB.frame.minX  
                     - 16  
                     = 667 - 232 - 16  
                     = 419  
  
labelB.frame.height = labelA.frame.height * 2  
                     = 100 * 2  
                     = 200
```

The layout engine solves layout variables by making value substitutions. Did you know that you can solve constraints on paper? It's true! Contrary to popular belief, the layout engine does not cast *Harry Potter* spells. Instead, it solves linear equations for equality constraints using math.

When it comes to solving inequality constraints, the constraint solving methods get a little more complex, and linear programming comes into play.

## Solving linear programming problems

Linear programming problems ask questions relating to how to minimize or maximize a variable given constraints. Linear programming is a mathematical technique used in solving linear equations. Whether you’re looking to minimize cost or maximize output, linear programming can come in handy.

For instance, linear programming can assist a hardware company in maximizing product outputs or minimizing production costs given a set of constraints. The constraints, for example, can relate to a company’s budget, the number of available employees or certain hardware output per hour.

**Note:** This section walks through the process of solving a linear programming problem (LPP). You’re not expected to understand all of the intricacies involved with LPP, but you’ll have enough information here to gain a better understanding of Cassowary. The point is to get a feel for the high-level patterns of solving LPP — you’ll take a closer and more specific look at LPP when dealing directly with Cassowary.

Imagine yourself running a software company, and you’re trying to maximize the profit your company makes over the next month.

Here are your constraints:

- You have 100 software engineers.
- SWE work either at night or in the morning.
- The office’s capacity is 80 at night and 60 in the morning.
- SWE contributes \$600 profit/hour at night and \$400/hour in the morning.
- Each employee can contribute up to 100 hours/month.
- Employees can work up to 5 hours at night or 8 hours in the morning.

How many employees should work at night or in the morning to maximize profits for the next month?

Solving LPP on paper looks something like this:

```
//// Profit equation
P = 600x + 400y

//// Variables
x = # of SWE working at night.
y = # of SWE working in the morning.

//// Inequalities
// Employees can not work negative hours.
x >= 0, y >= 0
// The number of possible employees.
x + y <= 100
// The number of possible employees hours.
5x + 8y <= 10000
```

Here's how the math works:

1. You have your profit equation. P represents the profit. x and y represent the software engineers working at night and in the morning, respectively. The coefficients represent profits for respective work environments (night/morning).
2. You have your set of constraints represented in inequalities. Employees can not work negative hours. The maximum number of employees given is 100. Total possible employee hours is 10,000 deriving from 100 maximum available employees and their available work hours per month.

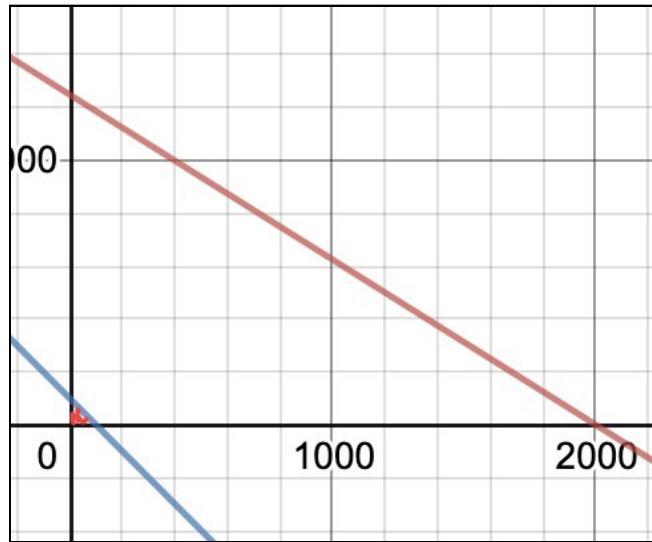
In linear programming, optimal values are found at one of the vertices of the **feasible region**. The feasible region is the area that satisfies all the constraints.

To find the feasible region, you can graph the inequalities. Simply find the x and y-intercept of the inequalities by substituting zero for the other variable:

```
//// Substitutions
x + y <= 100
x = 100
y = 100

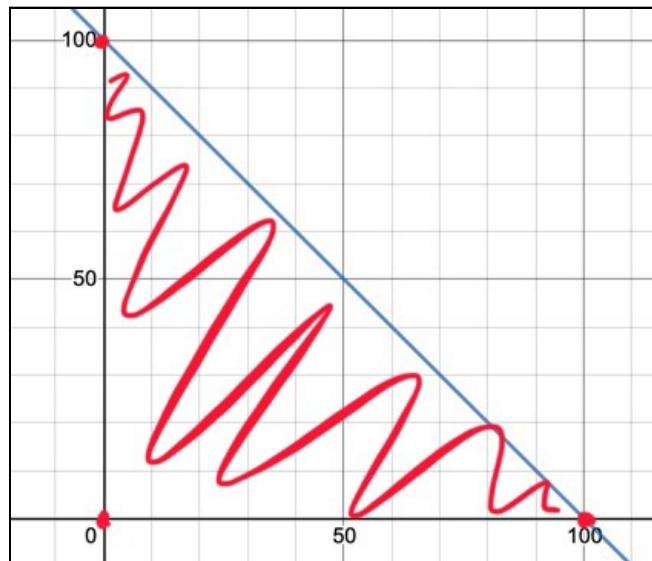
//// More substitutions
5x + 8y <= 10000
x = 2000
y = 1250
```

You now have your x and y-intercepts. When you graph the inequalities on paper, it looks something like this:



Below the red and blue lines, you'll see a red shaded area. The red shaded area is the feasible region — an area that satisfies the four inequalities.

When you zoom into the feasible region, you'll see a clearer image of the feasible region vertices:



The three red points are the vertices of the feasible region and the corner points. The optimal point sits in one of the following coordinates:

- (0, 0)
- (0, 100)
- (100, 0)

You can now test each of these points to see the coordinate pair that gives you the highest profit given the inequality constraints. You can pull up the profit equation and test corner point coordinates:

```
P = 600x + 400y  
// (0,0)  
P = 600(0) + 400(0)  
P = 0  
  
// (0,100)  
P = 600(0) + 400(100)  
P = 40,000  
  
// (100,0)  
P = 600(100) + 400(0)  
P = 60,000
```

From testing the corner point coordinates, it turns out that allocating 100 software engineers into working at night and 0 software engineers into working in the morning produce the highest profits for the next month.

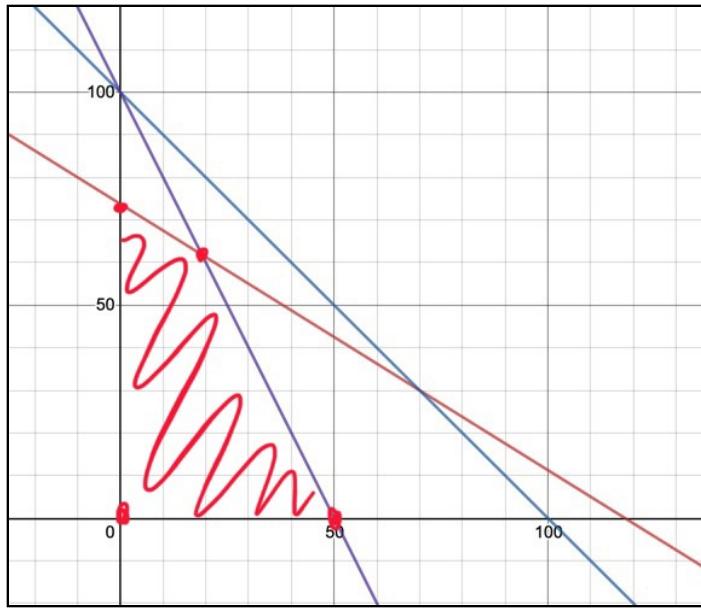
OK, it's time to twist the problem by replacing previous inequalities with the following inequalities:

```
//// Inequalities  
x >= 0, y >= 0  
x + y <= 100  
5x + 8y <= 590  
2x + y <= 100
```

What do you think, can you solve the given LPP?



With the inequalities above, the graph looks something like this with the feasible region:



On the graph, you can see four vertices. The feasible region vertices at the x/y-intercepts are:

- (0, 0)
- (0, 73.75)
- (50, 0)

An extreme vertex in a feasible region like one of the above is known as a basic feasible solution. You can calculate the last vertex that makes up the feasible region. But first, you need to turn the inequalities into equations:

- Red line:  $5x + 8y \leq 590 \rightarrow 5x + 8y = 590$
- Purple line:  $2x + y \leq 100 \rightarrow 2x + y = 100$

After that, solve for the x and y coordinates using elimination and substitution:

```
//// Elimination
// 1
5x + 8y = 590
2x + y = 100 (multiply every value by -2.5 to cancel x)
```

```

// 2
5x + 8y = 590
-5x - 2.5y = -250 (add values together)

// 3
5.5y = 340 (divide both side by 5.5)

// 4
y = 61.81 (rounded to the nearest hundredth)

//// Substitution
// 5
5x + 8y = 590
5x + 8(61.82) = 590
x = (590 - 494.56) / 5
x = 19.09 (rounded to the nearest hundredth)

```

Here's how the math works out:

1. Multiply  $2x + y = 100$  by  $-2.5$ .
2. Add  $-5x - 2.5y$  and  $-250$  to  $5x + 8y = 590$ 's left and right hand side respectively.
3.  $x$  is eliminated from  $5x + 8y = 590$ . You're left with  $5.5y = 340$ . To solve for  $y$ , you divide both sides of the equation by  $5.5$ .
4. You get  $61.81$  when rounded to the nearest hundredth for  $y$ .
5. Now, enter  $y$  into either equation to solve for  $x$ . Entering  $y$  into  $5x + 8y = 590$  gives you  $x$  as  $19.09$  when rounded to the nearest hundredth.

You now have all of the corner points of the feasible region. To test for the optimal point, pass the coordinates back into  $P = 600x + 400y$ :

- $(0, 0) - 0$
- $(0, 73.75) - 29,500$
- $(50, 0) - 30,000$
- $(19.09, 61.81) - 36,178$

For this particular problem, having a fraction of a person doesn't make sense. But don't worry; you can round down the numbers.

It turns out that 19 people working at night and 61 people working the morning is optimal given the constraints. The resulting profit for this allocation procedure is \$35,800 / month.



Just like you did with that example, the layout engine is always trying to satisfy all of the constraints to the best of its ability — except the layout engine answers questions like “what’s the widest Label A can be while simultaneously satisfying all of Label A’s other constraints and Label B’s constraints?”

Now that you’re getting the hang of linear programming, it’s time to transition to Cassowary.

## Cassowary

Cassowary optimizes layout computations for interactive user interface components. On June 29, 1998, Greg J. Badros and Alan Borning published an academic paper titled *The Cassowary Linear Arithmetic Constraint Solving Algorithm: Interface and Implementation*. According to the authors, constraint solvers at the time couldn’t efficiently handle simultaneous linear equations and inequalities for graphical applications. Hence, the birth of Cassowary.

Cassowary is an incremental algorithm and is based on the dual simplex method. In this section, you’ll solve inequalities using Cassowary. The subsections involved in solving inequalities are:

1. Augmented simplex form.
2. Basic feasible solve form.
3. Basic feasible solution.
4. Simplex optimization.
5. Error minimization.

## Sample inequality constraint problem

First, you’ll see a sample inequality constraint problem you need to solve. To prime an inequality constraint problem in augmented simplex form, you’ll represent the problem’s constraints in equations and inequalities. You’ll also learn what an objective function is to better understand its role in solving a linear programming problem.



## Constraints

For time efficiency, you'll solve inequalities in the horizontal axis. However, you can apply the same logic in the vertical axis context as well.

Below are three points ( $x_l$ ,  $x_m$ ,  $x_r$ ), which are all on the horizontal axis:

- - -  $x_l$  - -  $x_m$  - -  $x_r$  - - -

With the following constraints:

- $x_m$  is the midpoint of  $x_l$  and  $x_r$ .
- $x_l$  is at least 10 points to the left of  $x_r$ .
- The points are within the 0 to 100 range.

Given the constraints, the solution is:

$x_l < x_m < x_r$

You can simplify and remove redundant bound constraints, and represent the constraints shown above in equation and inequalities as follows:

- $2x_m = x_l + x_r$
- $x_l + 10 \leq x_r$
- $x_r \leq 100$
- $0 \leq x_l$

## Problem/objective function

In linear programming, an **objective function** is a function that minimizes or maximizes for a specific goal. The objective function is the problem you want to solve in linear programming.

Your objective function is to minimize the distance between  $x_m$  and  $x_l$  given the constraints you saw earlier. You'll begin to solve the objective function by converting the constraints into an augmented simplex form.

## Augmented simplex form

The Cassowary algorithm builds on top of the simplex algorithm. The simplex algorithm is a method of solving optimization problems in linear programming and it doesn't take negative values (un-restricted values).



All of the variables are  $\geq 0$  in the algorithm's equations.

Augmented simplex form, on the other hand, allows and handles unrestricted variables. It can handle both positive and negative variables by using two tableaux — the unrestricted simplex tableau for unrestricted values and the simplex tableau for non-negative values. A tableau is a table of values representing a linear program in standard form. Tableau and standard form are interchangeable ways to represent a linear program.

The purpose of the augmented simplex form is to incrementally add and delete required constraints with restricted and unrestricted variables from the constraints system.

## Converting inequalities to equations

The first step to working with augmented simplex form is to convert inequalities to equations.

Inequalities in the form of a linear real expression being less than or equal to a number, like this:

- $e \leq n$

Can be converted to this:

- $e + s = n$ , where  $s \geq 0$

The  $s$  is a non-negative number known as a slack variable.

Go ahead and convert inequalities in the form of a linear real expression to equations:

- $x_l + 10 \leq x_r \rightarrow x_l + 10 + s_1 = x_r$
- $x_r \leq 100 \rightarrow x_r + s_2 = 100$

You now have the following equalities and inequalities:

- $2x_m = x_l + x_r$
- $x_l + 10 + s_1 = x_r$
- $x_r + s_2 = 100$
- $0 \leq x_l, s_1, s_2$

## Separating equations into tableaus

It's time to separate the equalities into unrestricted and simplex tableaus. Initially, all of the equations are placed into the simplex tableau.

To separate the equalities into the unrestricted tableau, you'll use Gauss-Jordan elimination. This includes the following steps:

- Find an equation with an unrestricted variable in the simplex tableau.
- Remove the equation from the simplex tableau.
- Solve for the unrestricted variable.

At this time, the unrestricted variable equals an expression. Using the expression you find, substitute the expression into both tableau's equations.

Afterward, repeat removing unrestricted variables from the simplex tableau until there are no more unrestricted variables in the simplex tableau.

## Application

Looking back at the equations above, you know that  $0 \leq x_l, s_1, s_2$ . These variables have values that are confined to any number greater than or equal to 0. They're **restricted** variables.

$x_r + s_2 = 100$  has an unrestricted variable,  $x_r$ . Remove  $x_r$  from the simplex tableau. Solve for the unrestricted variable:

- $x_r + s_2 = 100 \rightarrow x_r = 100 - s_2$

Substitute the expression into occurrences of  $x_r$  in the simplex tableau:

- $2x_m = x_l + x_r \rightarrow 2x_m = x_l + \mathbf{100 - s_2}$
- $x_l + 10 + s_1 = \mathbf{x_r} \rightarrow x_l + 10 + s_1 = \mathbf{100 - s_2}$
- $0 \leq x_l, s_1, s_2$

Look back into the simple tableau;  $2x_m = x_l + 100 - s_2$  has an unrestricted variable,  $x_m$ . After the substitution, you get:

- $2x_m = x_l + 100 - s_2 \rightarrow x_m = (1/2)(x_l) + 50 - (1/2)(s_2)$

Substitute  $x_m$  into the minimization problem; here's how:

1. minimize  $x_m - x_l$

2. minimize  $(1/2)(xl) + 50 - (1/2)(s2) - xl$
3. minimize  $50 - (1/2)(xl) - (1/2)(s2)$

The simplex tableau is left with the following equations:

- $xl + 10 + s1 = 100 - s2$

At this point, there are no more unrestricted variables in the simplex tableau.

Next, you'll learn about basic feasible solve form to continue solving your objective function.

## Basic feasible solve form

You started with inequalities and converted the inequalities to an augmented simplex form optimization problem. It's now time to solve the problem.

You solve an augmented simplex form optimization problem by having the equations in basic feasible solved form, looking like this:

- $x_0 = c + a_1x_1 + \dots + a_nx_n$

With the following conditions:

- $x_0$  is basic. It isn't found in the objective function or any other equation.
- Other variables are parameters.

In a basic feasible solved form, you have a basic variable and parametric variables.

Your objective function is this:

- minimize  $50 - (1/2)(xl) - (1/2)(s2)$

Convert the equations into basic feasible solved form, and they'll look like this:

- $x_m = (1/2)(xl) + 50 - (1/2)(s2)$
- $x_r = 100 - s2$
- $s1 = 90 - xl - s2$

Notice that  $x_m$ ,  $x_r$  and  $s1$  do not occur in the objective function or any other equation. Those are your basic variables. The other variables are the parametric variables.

With the equations set up in basic feasible solved form, you can solve for a basic feasible solution.

## Basic feasible solution

As introduced earlier, a basic feasible solution is an extreme vertex of a feasible solution region. You look for these vertexes because you care about maximizing or minimizing outputs when solving linear programming problems.

To solve for a basic feasible solution, you:

1. Set the parametric variables to 0.
2. Have the basic variables equal to a constant.

Like this:

- $x_m = (1/2)(0) + 50 - (1/2)(0) \rightarrow x_m = 50$
- $x_r = 100 - (0) \rightarrow x_r = 100$
- $s_1 = 90 - (0) - (0) \rightarrow s_1 = 90$

Afterward, you have  $(x_m, x_r, s_1, x_l, s_2) = (50, 100, 90, 0, 0)$ .

Substitute the variable values into the objective function:

- $50 - (1/2)(0) - (1/2)(0) = 50$

The value for the objective function using the basic feasible solution is 50. 50 is a viable solution, but it isn't optimized — well, not yet!

## Simplex optimization

The simplex algorithm finds the optimum solution by continually searching for an adjacent basic feasible solved form. The adjacent basic feasible solved form consists of a basic feasible solution. The basic feasible solution decreases the value in an objective function. When there's no more adjacent basic feasible solved form left, the optimum solution is found.

Simplex optimization does the following:

1. It takes a basic feasible solved form.
2. It applies matrix operations known as pivoting.

3. If further optimization is applicable, it creates a new basic feasible solved form and returns to Step 1.

When decreasing a variable value in the objective function, you can start with zero and go up from there. Ensure that as the variable increases, it doesn't cause non-negative numbers to become negative.

Pivoting is an operation which exchanges a basic and a parameter using matrix operations. You pivot to search for an adjacent basic feasible solved form. You can reach an adjacent feasible solved form by performing a single pivot.

## Application

First, look at how you can decrease the value in the objective function:

- $50 - (1/2)(x_l) - (1/2)(s_2)$

Subject to the following non-negative confinements:

- $0 \leq x_l, s_1, s_2$

Look at the objective function. As you increase  $x_l$  starting at zero, the objective function value continues to decrease. You'll want to increase this value until you can no longer do so.

Look at the simplex and unrestricted tableau.

From the simplex tableau, you have the following equation:

- $s_1 = 90 - x_l - s_2$

Pivot by swapping out  $x_l$  and  $s_1$ :

- $x_l = 90 - s_1 - s_2$

You've turned  $s_1$  into a parametric variable.  $x_l$  is now basic. Now, substitute  $x_l$  into the other equations and objective function.

The unrestricted tableau:

- $x_m = (1/2)(90 - s_1 - s_2) + 50 - (1/2)(s_2) \rightarrow x_m = 95 - (1/2)(s_1) - s_2$
- $x_r = 100 - s_2$

And, the objective function:

- $50 - (1/2)(90 - s_1 - s_2) - (1/2)(s_2) \rightarrow 5 + (1/2)(s_1)$

At this point, you can decrease  $s_1$  to decrease the objective function value. However,  $s_1$  subjects to be greater than or equal to zero.  $s_1$  has already taken the value of zero, which is the minimum it can be.

There's no further adjacent basic feasible solved form that can further decrease the objective function at this point. Hence, you've arrived at the optimal solution.

To get the objective function value, enter zero for the parametric variables in the objective function:

- $5 + (1/2)(0)$

The objective function value is 5. Finally, you get the optimized solution.

That's how Cassowary solves constraints involving inequalities. Cassowary also helps deal with non-required constraints.

## Error minimization

For constraints with a non-required priority or priority  $< 1000$ , error minimization comes into play. With non-required constraints, the layout engine goes through additional optimizations before providing the final variables.

If you'd like to read more on the math behind handling constraint priorities, review Cassowary's quasi-linear and quadratic optimization algorithms.

## Key points

- Auto Layout uses alignment rectangles to create a relationship between two view items.
- Linear programming problems are minimization/maximization problems.
- Cassowary is a linear arithmetic constraint solving algorithm.
- The Layout engine uses Cassowary to solve equality and inequality constraints.
- Solving inequality constraints using Cassowary consists of working with augmented simplex form, basic feasible solve form, basic feasible solution and simplex optimization.

# Chapter 15: Optimizing Auto Layout Performance

By Jayven Nhan

App performance is critical for delivering a great end-user experience. A sluggish and unresponsive app tends to frustrate its users to the point where they delete the app. On the other hand, a fast and responsive app helps its users achieve their core tasks.

With Auto Layout, you have some handy tools available at your disposal to help fine-tune your app's performance. In this chapter, you'll learn about the following Auto Layout performance optimization topics:

- Betting safe on performance with Interface Builder.
- Factoring in the render loop.
- Understanding constraints churning.
- Utilizing static and dynamic constraints.
- Batching and updating constraint changes.
- Understanding the cost of using Auto Layout.
- Making performance gains using best practices.



## Betting safe on performance with Interface Builder

One of the benefits of using Interface Builder is that it reduces the room for errors when handling Auto Layout constraints. You show Interface Builder how you want your layout to look using constraints, and you put the rest into the hands of Apple.

In other words, Interface Builder handles the behind-the-scenes code.

However, many of today's production apps implement Auto Layout partially or entirely using code. When you implement Auto Layout programmatically, you allow for greater customization — but you do so at the cost of widening the error margin for performance degradation. In short: The more you're able to do, the more things can go wrong.

When using code for Auto Layout, it's not only important to know what to do. It's also important to know what not to do. For example, overriding and adding suboptimal code in `updateConstraints()` can cause disastrous performance or even a crash.

So, what's better? It all depends. If you want Apple's "it just works" experience with regard to optimizing Auto Layout performance, then use Interface Builder. If, however, you want finer control over how your Auto Layout is implemented, then the programmatic approach has more merit.

## Factoring in the render loop

The render loop is not often talked about, yet it's the backbone for laying out your user interface at every frame. Understanding how the render loop works will help you see the larger picture of how Auto Layout works with other system components. At the core, the render loop's purpose is to ensure all of your views are presented as intended for every frame.

The render loop can be utilized to mitigate extraneous layout work. However, if it's used incorrectly, the render loop can cause your app to take a performance hit. After all, the render loop has the potential to run 60 or even 120 times per second, depending on the device's refresh rate (e.g., iPhone 11 Pro and iPad Pro 12.9" respectively).

Apple exposes the render loop API methods for developers to optimize layout performance. The goal is to let the system do the minimum amount of work to produce the desired user interface for every frame. The render loop's methods can be segregated into three phases: **update constraints**, **layout** and **display**.

In order, each phase does the followings:

1. **Update constraints:** Layout calculation.
2. **Layout:** Input layout values from the previous step into the view.
3. **Display:** Render the user interface of the view within the layout bounds.

Each render loop phase has its own set of methods:

1. **Update constraints:** `updateConstraints()`, `setNeedsUpdateConstraints()`, `updateConstraintsIfNeeded()`
2. **Layout:** `layoutSubviews()`, `setNeedsLayout()`, `layoutIfNeeded()`
3. **Display:** `draw(_:)`, `setNeedsDisplay()`

In the update constraints phase, its operations start from the leaf views and travel up the view hierarchy. You can use this to think about the order in which constraints are calculated. For the layout and display phases, the operations start at the root of the view hierarchy and travel down to the child views. The latter phases consist of transferring layout information from the layout engine and visualizing a view from the given rectangle.

You may have noticed that in each render loop phase, there are two or more operations. For the most part, you don't need to and shouldn't touch them. However, you may be able to find room to reduce repetitive layout work and fine-tune performance to give your layout speed an edge.

More important than improving the layout speed, however, is to avoid the pitfalls, which will do you a lot more harm than good. By understanding the components of the render loop, you'll be less susceptible to fall into the possible pitfalls, such as layout churning. Layout churning is when the layout engine repeatedly and unnecessarily computes layout variables. Here's a breakdown of the functions within the render loop:

- **updateConstraints():** Make constraints changes.
- **setNeedsUpdateConstraints():** Invoke constraint changes in the next render loop.

- **updateConstraintsIfNeeded()**: Ensures the latest constraint changes are applied to the view and its subviews.
- **layoutSubviews()**: Set the size and position of the view using constraints information.
- **setNeedsLayout()**: Invoke layout changes in the next render loop.
- **layoutIfNeeded()**: Ensures the latest layout updates are applied to the view and its subviews.
- **draw(\_:)**: Draw view's content within the view's bounds.
- **setNeedsDisplay**: Invoke view redrawing in the next render loop.

Although these methods are accessible to developers, take great care when using them because they are sensitive code. You should minimize your interaction with them, and only touch them when you're able to reduce the overall layout work.

For example, say you have a `UITextView`: The intrinsic size of the text view depends upon the font size, font style, text, padding and so on. One thing you can do is to recompute the text view's size each time there's an update to one of the properties. This is inefficient since these updates operate consecutively. In other words, everything that's computed before the final update is extraneous work. Therefore, instead of calling a method like `updateConstraints()`, you can call `setNeedsUpdateConstraints()` after updating a property. This ensures that you only call `updateConstraints()` at the end of the render loop before the frame gets sent onto the screen.

## Why update constraints

As introduced earlier, making use of `updateConstraints()` can be a recipe for disaster. So, why use it? The title of this chapter may give it away: It's for performance. There are two ways to make constraint changes when using code: in place or in batches by using `updateConstraints()`.

In place constraint changes are found in places such as `viewDidLoad()`, view initializers or user interaction. These constraint changes have the layout engine activate and deactivate constraints individually.

On the other hand, `updateConstraints()` gets the layout engine to batch constraint changes. Imagine activating and deactivating all of your constraints at once. To schedule a change in constraints, call `setNeedsUpdateConstraints()`.

You use `updateConstraints()` when making constraint changes in place is slow, or you can mitigate redundant work for the layout engine. However, because this is sensitive code, it has the possibility of making your app susceptible to constraints churn.

## Constraints churn

Constraints churn happens when constraints are repeatedly added and removed. Although this may not happen frequently, it's essential to know what it looks like to avoid any misstep.

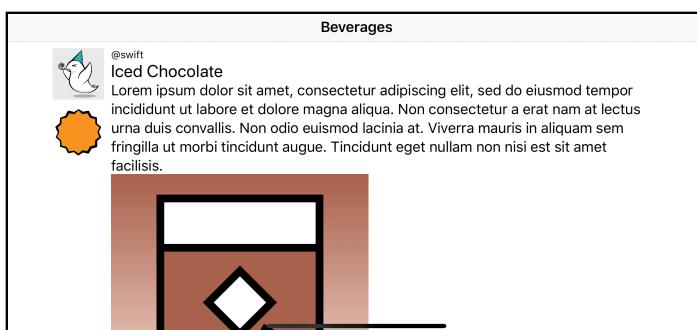
To avoid constraints churn, ensure the following:

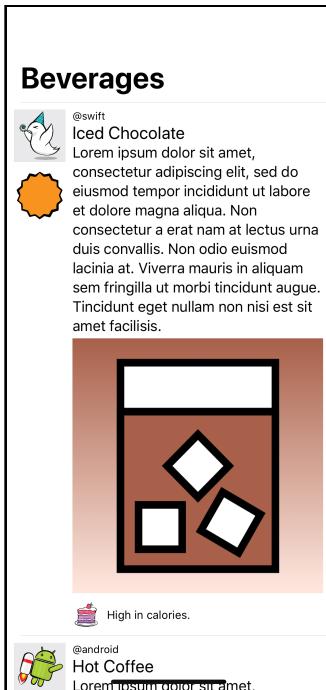
- Static constraints are set once.
- Dynamic constraints only activate and deactivate when necessary.

Static constraints are constraints that never change throughout the lifetime of a view. Dynamic constraints are constraints that can change. They may be used in one scenario and discarded in another. You'll see examples of this as you work on the sample project.

Before running the sample project, it's important to note that the simulator uses your Mac's hardware, which can be magnitudes faster than your iOS device. By using an iOS device, you'll be better able to see performance dents if there are any. You might even consider using an older iOS device to benchmark performance.

Open `AutoLayoutPerformance.xcodeproj` in the **starter** folder. Build and run.





Go ahead and scroll up and down on your device. You'll see the console log prints `Update Constraints` every time a cell is dequeued from the table view. You may notice that the scroll frame rate is a bit choppy; yet, there are only eight views per cell. Just imagine the performance dent for views with more subviews. It's time to look inside to see what's wrong and what you can do to smooth out the frames of your app.

Open **TableViewCell.swift** from the **Views** group. Look inside `updateConstraints()`, and you'll see the following:

```
// 1
contentView.subviews.forEach { $0.removeFromSuperview() }
// 2
contentView.addSubview(usernameLabel)
contentView.addSubview(titleLabel)
contentView.addSubview(profileImageView)
contentView.addSubview(badgeImageView)
contentView.addSubview(descriptionLabel)
contentView.addSubview(postImageView)
contentView.addSubview(cakeImageView)
contentView.addSubview(isHighCalorieLabel)
// 3
NSLayoutConstraint.deactivate(layoutConstraints)
// 4
```

```
layoutConstraints =
    usernameLabelConstraints
    + titleLabelConstraints
    + profileConstraints
    + descriptionLabelConstraints
    + postImageViewConstraints
// 5
if beverage?.isHighCalorie ?? false {
    layoutConstraints += highCalorieConstraints
} else {
    layoutConstraints += lowCalorieConstraints
}
// 6
if beverage?.isFrequentUser ?? false {
    layoutConstraints += badgeImageViewConstraints
}
// 7
NSLayoutConstraint.activate(layoutConstraints)
```

Here's the code breakdown:

1. Content view will remove its subviews by calling `removeFromSuperview()` on each subview.
2. Content view will add the re-add the subviews.
3. The layout engine will deactivate all constraints inside of `layoutConstraints`.
4. `layoutConstraints` is set with title label, profile, description label and page image view constraints.
5. Then, depending on if the beverage is high-calorie, add a constraint to `layoutConstraints`.
6. Then, depending on if the beverage belongs to a frequent user, add a constraint to `layoutConstraints`.
7. Activate the constraints.

At first, the code may not seem problematic — but it is. Although you may not be able to call this method 60 times per second due to the size of the table view cell and how fast you can scroll, the layout engine does extraneous work that can be mitigated, which you'll fix next.

## Adding and removing subviews

Notably, there's no need to add and remove subviews constantly. Instead, you only need to do this once when the view initializes.

First, remove the following code from `updateConstraints()`:

```
contentView.subviews.forEach { $0.removeFromSuperview() }
```

Second, move the following code from `updateConstraints()` to `commonInit()`:

```
contentView.addSubview(usernameLabel)
contentView.addSubview(titleLabel)
contentView.addSubview(profileImageView)
contentView.addSubview(badgeImageView)
contentView.addSubview(descriptionLabel)
contentView.addSubview(postImageView)
contentView.addSubview(cakeImageView)
contentView.addSubview(isHighCalorieLabel)
```

You've removed the extraneous workload for adding and removing subviews inside `updateConstraints()`. Any time `TableViewCell` calls `updateConstraints()`, it no longer unnecessarily rips off all its subviews and adds back the same subviews.

## Activating and deactivating static constraints

Most of the constraints are static constraints, which don't change throughout the lifetime of the view. They are there for good as long as the view doesn't get deallocated. For static constraints, you need to ensure that they're only going to activate once.

First, add the following property to `TableViewCell`:

```
private var staticConstraints: [NSLayoutConstraint] = []
```

To mitigate the extraneous workload for the layout engine, you'll use this property to reference and keep track of activated static constraints.

Second, replace the following code in `updateConstraints()`:

```
NSLayoutConstraint.deactivate(layoutConstraints)
layoutConstraints =
    usernameLabelConstraints
    + titleLabelConstraints
    + profileConstraints
    + descriptionLabelConstraints
    + postImageViewConstraints
```

With the following:

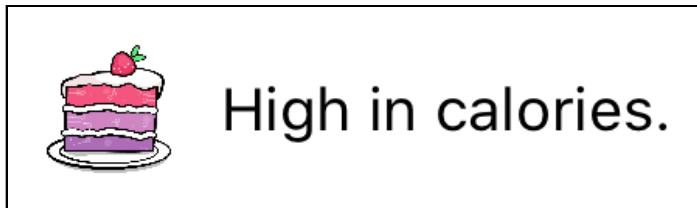
```
if staticConstraints.isEmpty {  
    staticConstraints =  
        usernameLabelConstraints  
        + titleLabelConstraints  
        + profileConstraints  
        + descriptionLabelConstraints  
        + postImageViewConstraints  
    NSLayoutConstraint.activate(staticConstraints)  
}
```

TableViewCell no longer deactivates and activates the same constraints repeatedly. The app checks if there are static constraints. If there are none, the app sets the static constraints and activates them. For subsequent render loops, static constraint changes are out of the equation.

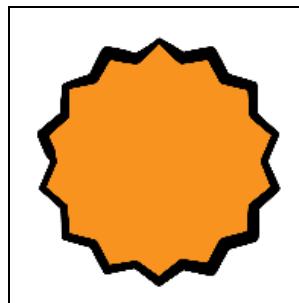
## Activating and deactivating dynamic constraints

Dynamic constraints are constraints that may change throughout the lifetime of a view. Part of the table view cell's layout is dependent on the beverage properties.

A high-calorie beverage lays out an additional image and label:



A beverage post from a frequent user lays out an orange badge:



These constraints are dynamically activated and deactivated based on the beverage properties. Once again, you should aim to mitigate any unnecessary work for the layout engine.

First, add the following property to `TableViewCell`:

```
private var dynamicConstraints: [NSLayoutConstraint] = []
```

Here, similar to `staticConstraints`, you create an empty array of `NSLayoutConstraint`. With this collection, you'll use it to reference activated dynamic constraints.

Then, add the following helper method to `TableViewCell`:

```
private func updateDynamicConstraints(isHighCalorie: Bool) {
    NSLayoutConstraint.deactivate(dynamicConstraints)
    dynamicConstraints = isHighCalorie
        ? highCalorieConstraints : lowCalorieConstraints
    NSLayoutConstraint.activate(dynamicConstraints)
}
```

You'll use this method to deactivate old dynamic constraints and set the new dynamic constraints to either the high-calorie or low-calorie constraints. Afterward, you'll activate the new dynamic constraints.

Finally, replace the following code in `TableViewCell`:

```
if beverage?.isHighCalorie ?? false {
    layoutConstraints += highCalorieConstraints
} else {
    layoutConstraints += lowCalorieConstraints
}
```

With the following:

```
// 1
if beverage?.isHighCalorie ?? false
    && dynamicConstraints != highCalorieConstraints {
    updateDynamicConstraints(isHighCalorie: true)
// 2
} else if dynamicConstraints != lowCalorieConstraints {
    updateDynamicConstraints(isHighCalorie: false)
}
```

Here's how it works:

1. The decision to compute the high-calorie constraints is predicated on two conditions: First, the beverage is high-calorie. Second, the current dynamic constraints referenced within `TableViewCell` differ from the high-calorie constraints. Only when these two conditions are true do you pull the Auto Layout engine out of bed.
2. Similarly, the decision to update dynamic constraints is predicated on the beverage's high-calorie status and if the previous dynamic constraints differ from the low-calorie constraints. Otherwise, you let the Auto Layout engine rest.

## Avoiding unnecessary constraints activation and deactivation

Similar to the beverage's high-calorie status, whether the beverage post belongs to a frequent user also has a role in `TableViewCell` user interface. You can activate and deactivate the badge image view's constraints depending on `isFrequentUser`. However, this is extraneous work on the layout engine. Whenever possible, choose to activate the constraints once, and you can use `isHidden` to show or hide a view, which is less taxing on the system.

Remove the following code from `updateConstraints()`:

```
if beverage?.isFrequentUser ?? false {
    layoutConstraints += badgeImageViewConstraints
}
```

Replace `profileConstraints` in `TableViewCell` with:

```
private var profileConstraints: [NSLayoutConstraint] {
    profileImageViewConstraints + badgeImageViewConstraints
}
```

Now, `TableViewCell` no longer unnecessarily activates and deactivates `badgeImageViewConstraints`. Instead, the operation only runs inside of `updateConstraints()` when `staticConstraints` is empty inside a `TableViewCell`.

Lastly, remove the following code from `updateConstraints()`:

```
NSLayoutConstraint.activate(layoutConstraints)
```

Also, remove `layoutConstraints` from `TableViewCell`. The code above and the associated property are no longer needed anywhere in the view.

Build and run. You'll now have a much smoother scrolling experience. Cheers!

Each constraint computation isn't computationally significant in any way. However, when these computations repeatedly compound beside other operations within your app, the overall computation is a significant drain on your device — possibly translating to a rather poor user experience.

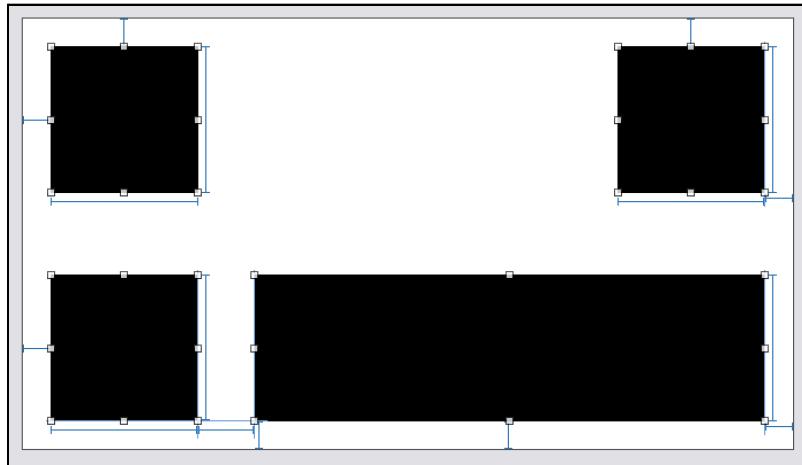
## Unsatisfiable constraints

It's good to know that unsatisfiable constraints can cause performance issues. Not only that, other problems may stem from unsatisfiable constraints. When there are unsatisfiable constraints, the layout engine will need to go through the process of figuring out which constraints to break in the hopes of giving you the desired layout.

This is unnecessary work for the layout engine, and you risk getting a layout that isn't what you want. As a result, other parts of your app that interconnect with the constraints can have problems now too, which can be difficult to debug since unsatisfiable constraints mask the root of the problem. As a rule of thumb, ensure that there are no unsatisfiable constraints for optimal app performance with Auto Layout.

## Constraints dependencies

Now, you'll look at how layout dependencies affect Auto Layout's performance. Look at the following diagram:



The top two views are independent of each other. They are constrained to the root view. As a result, you'd have linear time complexity for both layout operations.

The bottom two views have a dependency. Both views are constrained to be a specific distance apart. When there's a dependency, the layout engine will do additional substitution work.

Perhaps, the first thing that comes to a lot of people's minds is to make the views independent of each other as much as possible. However, the layout engine caches layout and tracks dependencies. As long as the dependencies are correlated to the layout you want, it's highly unlikely that avoiding constraints and using other methods such as manual calculation is worthwhile. The key takeaway here is not to be afraid to add constraints and avoid doing a bunch of manual calculations for Auto Layout.

However, there's a fine line between having the constraints to display a particular layout versus having too many constraints or constraints settings to accommodate for additional layouts. In the latter case, there will be false dependencies where constraints don't need to be dependent on each other. In addition to higher computation power requirements, debugging becomes difficult. The solution is to put two different layouts in two different views and don't use constraints to make it work in a single view for performance and debugging sanity.

## Advanced Auto Layout features and cost

You may wonder how expensive it is to set layout constraint inequalities, constants and priorities. When given an inequality relation, the layout engine sees it as a single additional variable to solve. Inequalities are lightweight for the layout engine.

Setting a layout constraint constant makes use of the layout engine's dependency tracker. Because the layout engine tracks dependencies, even having a constraint react directly to the user's swipe gesture is performant. Apple uses the dependency tracker to optimize Auto Layout's set of constraint operation performance.

Unlike constraint inequalities and constants, constraint priorities take additional work to run the simplex algorithm for error minimization. This is something to be mindful of and not be afraid of using it. Instead, you should use it only when you need to.

In a view that tries to accommodate for layouts that should be separated into multiple views, you may find many constraints and constraint priorities settings all over the place. When this is the case, it's time to separate the layouts into their compartmental views.

You may have seen this in some legacy apps. When this nightmare shows itself, it's an excellent idea to use performance and debugging clarity as talking points for a view refactor.

## Apple optimizing UIKit

Apple is continuously optimizing UIKit's performance. Every year at WWDC, Apple introduces a more performant way to integrate your data, behind the scene optimizations, new frameworks to make building dynamic and responsive layouts easy and so much more.

To take advantage of the latest Apple technologies, upgrading to the project's latest deployment target can sometimes do the trick. Other times, you'd need to do some more manual work. For example, in WWDC18, Apple introduced UIKit layout improvements for iOS 12. The improvements shown are no laughable matter. They included more performant ways to help developers achieve the layout they want, which includes improvements to the OS system, core of UIKit, client code and more. Plus, for devices running on iOS 12 and above, these improvements are free.

Although you can do more with Auto Layout in code, it's easier to get Auto Layout right with Interface Builder. You can optimize Auto Layout by helping the layout engine do the least amount of work for your desired layout. As long as that is your north star, you have the correct mental model and are on the right path.

## Key points

- Auto Layout done in the Interface Builder gives you automatic layout performance optimization. It's a great place to start and stick to it for performance if you can.
- When creating Auto Layout constraints in code, you can do it in place or in batches.
- When you find a need to optimize Auto Layout performance, update the constraints inside of `updateConstraints()`.

- Beware when working with sensitive code like `updateConstraints()`. You use it to make layout changes and defer extraneous work.
- Get rid of any unsatisfiable constraints to mitigate the risks of performance dent and problems masked from it.
- Additional view dependency means an additional substitution for the layout engine when solving constraints.
- Typically, letting the layout engine compute view frames is faster than the computation you can manually do in the client code for your view(s).
- Inequalities aren't expensive. They're an additional variable for the layout engine to solve.
- Constraint priorities utilizes the simplex algorithm for error minimization. This takes more work than inequalities. It's good to know that it does take more work, but you should use it when you need to.
- When Apple optimizes UIKit, this can give your app a performance boost. Devices running on the latest iOS version should deliver the fastest Auto Layout performances.

# Chapter 16: Layout Prototyping with Playgrounds

By Jayven Nhan

With a Swift playground, developers can quickly prototype their ideas and layouts using a minimal amount of code, allowing for instant feedback. In this chapter, you'll look at some of the benefits of using playgrounds over a full Xcode project.

There are three main issues when using full Xcode projects for layout prototyping:

- **Boilerplate code and extra files:** When you create a new single view project, Xcode adds boilerplate code along with additional files that you often don't need.
- **Lack of Speed:** Iteration is a vital component while building new apps, especially for layout prototyping. With full Xcode projects, you need to repeatedly build and run your project to see how everything works. With playgrounds, you can immediately see the results.
- **Dependency:** When you build UIs on top of existing projects, you're dealing with existing code, which means there's a good chance your new code will influence the existing code and vice versa.

When you prototype your layout, you iterate. Because your time is valuable, playgrounds are useful for prototyping; they also give you some additional benefits, such as:

- **Documentation:** Playground pages provide clear, convenient step-by-step documentation.
- **Xcode integration:** You can integrate a playground with an Xcode project. This makes the project's documentation convenient and accessible. This is something you can find in many of Apple's latest sample projects.



- **Mobility:** With the newer iPads, developers can create and run code right on the iPads using Swift Playgrounds. In doing so, they'll have direct access to features not available on a simulator such as a camera.

These are just some of the benefits of using playgrounds. By the end of this chapter, you'll have learned how to prototype layouts and provide markup documentation using playgrounds.

## Getting started with playgrounds

Open the **starter project** and select the **Working with Live View** page.

At the top of the page, add the following code:

```
import PlaygroundSupport
```

This code imports the `PlaygroundSupport` framework, giving you access to the live view.

A live view allows Xcode to interact with a playground to display the executed code. A live view is the view you'll be able to see and interact with. You can put almost any UI component into a live view.

## Setting up the live view

Working with a live view requires minimal setup — all you need to do is give the live view a view to display. You can have the live view show `UIView`, `UIButton`, `UIViewController` and more.

At the bottom of the playground page, add the following code:

```
// 1
let size = CGSize(width: 400, height: 400)
let frame = CGRect(origin: .zero, size: size)
let view = UIView(frame: frame)

// 2
PlaygroundColor.current.liveView = view
```

With this code, you:

1. Initialize a 400×400 `UIView`.
2. Set the playground page's live view to the initialized `UIView`.



Now that you've attached a view to the live view, you'll learn how to execute code within a playground.

## Executing playground code

To execute your code, click **Execute Playground**, which you'll find below the standard editor.



To execute your code up to a certain line, hover over the line number and click or press **Shift-Return**.



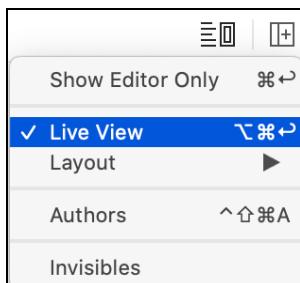
With the line execution feature, you can run the code line-by-line and gain the following benefits:

- **Understanding:** You can break down mysterious presentation or business logic behind lengthy code.
- **Save time:** You can quickly execute either the next line, the next few lines or the remaining lines of code; you don't have to run everything.

Now that you know how to execute code within a playground, you'll learn how to display the live view.

## Displaying the live view

When you execute your playground code, Xcode shows the live view on the right by default. If Xcode disabled the live view, click the **Adjust Editor Options** menu and select **Live View**:



With the live view in the Assistant editor, **execute** the playground. You'll see a view with a black background.

At this point, you're ready to use markup formatting to document your playground.

## Markup formatting

Markup is a computer language made for document annotations. Swift playgrounds support markup, so you can use it to document your playground, which helps organize and clarify your code.

To tell Xcode you're writing a line of markup, start your code comment followed by a colon, `//:`.

Make sure Xcode is set to show you the raw markup code. Choose **Editor** ▶ **Show Raw Markup** from the menu. Then, add the following markup text to the top of the page:

```
//: # Sampling Pads
//: ## Featuring rock, jazz and pop samples.
//: ### By: Your Name
```

Here, you declare three single-line markup comments. The first has the largest heading font size. Each line that follows has a smaller font size due to the number of consecutive `#`.

To see the markup annotation, you need the playground to render the markup text. If you open the **Editor** menu, you'll see the **Show Raw Markup** option has changed to **Show Rendered Markup**.

Afterward, your page will render something like this:

**Sampling Pads**

Featuring rock, jazz and pop samples.

By: Jayven Nhan

That's a great start to your playground documentation, but you've got more work to do.

Replace the following markup text:

```
//: # Sampling Pads
//: ## Featuring rock, jazz and pop samples.
//: ### By: Your Name
```



With this:

```
/*:  
 # Working with [live view](https://developer.apple.com/  
 documentation/playgroundsupport/playgroundpage/1964506-liveview)  
 ## Featuring rock, jazz and pop samples.  
 ### By: Your Name  
 */
```

By starting with `/*:` and ending with `*/`, you've declared multiline markup. You also added a link reference to the live view documentation.

After `import PlaygroundSupport`, add the following markup text:

```
//: `view` is used for experimental purposes on this page.
```

This text describes the role of `view`. When rendered, you see this:

view is used for experimental purposes on this page.

To declare inline code, you can put the code between two backticks.

Next, add the following to the top of your playground page:

```
//: [Previous Page](@previous)
```

When you're done with that, at the bottom of the playground page, add this:

```
//: [Next Page](@next)
```

These two markup formatting syntaxes are rather straightforward. The first link navigates to the previous playground page, and the second link navigates to the next playground page.

Open the **Table of Contents** page and add the following markup text:

```
/*:  
 # Table of Contents  
  
 1. [Working with Live View](Working%20with%20Live%20View)  
 2. [Music Button](Music%20Button)  
 3. [Sampling Pad](Sampling%20Pad)  
  
 */
```

This block of code creates links to the specific playground pages. When adding links, always encode the playground page name according to the URI rules in RFC 3986. Essentially, replace any whitespace with %20.

Documentation is an essential part of code maintenance and sharing. Now that you know how to document your playground, it's time to fire up your creativity and start prototyping.

## Experimenting with layout

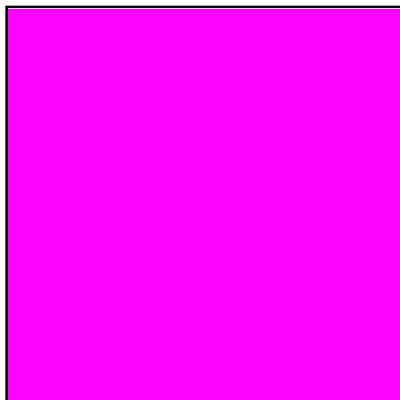
In this section, you'll go through a layout experimentation workflow in a playground.

Open the **Working with Live View** page, and below `PlaygroundPage.current.liveView = view`, add the following code:

```
view.backgroundColor = .lightGray  
view.backgroundColor = .blue  
view.backgroundColor = .red  
view.backgroundColor = .magenta
```

With the code above in the playground, you can see incremental changes. In this experimentation stage, suppose you want to see how `view` looks with different background colors. To do this, you can run the code above line-by-line, which helps you iterate through the different background colors and immediately see the results.

**Execute** the code you added, and your live view will look like this:

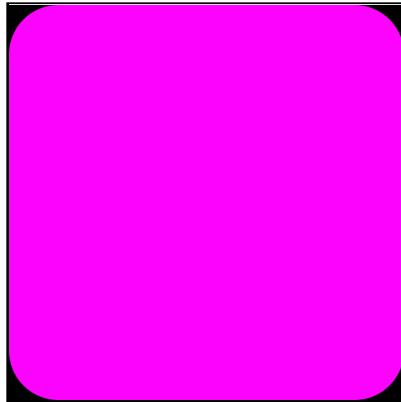


Below the code you already added, add this:

```
view.layer.cornerRadius = 50  
view.layer.masksToBounds = true  
view.layoutIfNeeded()
```

Now, let's say you want to alter the view shape. As you execute your code line-by-line, you need to call `layoutIfNeeded()`. In doing so, the view knows to take the latest layout changes into effect.

**Execute** the code you added, and your live view now looks like this:



Now, you're going to add a label to customize your view further. Add the following code:

```
// 1
let label = UILabel()
label.backgroundColor = .white
view.addSubview(label)
// 2
label.translatesAutoresizingMaskIntoConstraints = false
let labelLeadingAnchorConstraint =
    label.leadingAnchor.constraint(
        equalTo: view.leadingAnchor,
        constant: 8)
let labelTrailingAnchorConstraint =
    label.trailingAnchor.constraint(
        equalTo: view.trailingAnchor,
        constant: -8)
let labelTopAnchorConstraint =
    label.topAnchor.constraint(
        equalTo: view.topAnchor,
        constant: 8)
labelLeadingAnchorConstraint.isActive = true
labelTrailingAnchorConstraint.isActive = true
labelTopAnchorConstraint.isActive = true
// 3
label.text = "Hello, wonderful people!"
view.layoutIfNeeded()
```

With this code, you:

1. Add a label with a white background to `view`.
2. Create and activate constraints programmatically, keeping references to the constraints.
3. Set the label's text and update any `view`'s pending layout changes.

**Execute** the code, and your live view now looks like this:



Notice the label is clipped. You can update the label's position by updating the constraints. In addition to that change, you can also make a few more UI adjustments to give the label a facelift.

Next, you're going to set up the layout and user interface for the label. Below the code you just added, add the following:

```
// 1
label.font = UIFont.systemFont(ofSize: 64, weight: .bold)
label.adjustsFontSizeToFitWidth = true
// 2
labelLeadingAnchorConstraint.constant = 24
labelTrailingAnchorConstraint.constant = -24
labelTopAnchorConstraint.constant = 24
// 3
view.layoutIfNeeded()
// 4
label.textAlignment = .center
label.backgroundColor = .clear
label.textColor = .white
label.text = "WONDERFUL PEOPLE!"
```

With this code, you:

1. Set the label's font and tell the label to adjust its font to fit the available width.
2. Update the leading, trailing and top anchor constraint constants.
3. Update any view's pending layout changes.
4. Update the label's text alignment, background color, text and text color.

**Execute** the code, and your live view now looks like this:



Hmm, not bad. But suppose you want to center the label's y position in the container and animate the vertical axis constraint. To do that, add the following code below what you've just added:

```
// 1
label.removeConstraint(labelTopAnchorConstraint)
// 2
let labelCenterYAnchorConstraint =
    label.centerYAnchor.constraint(
        equalTo: view.centerYAnchor,
        constant: -32)
labelCenterYAnchorConstraint.isActive = true
// 3
view.layoutIfNeeded()
// 4
UIView.animate(
    withDuration: 3,
    delay: 1,
    usingSpringWithDamping: 0.1,
    initialSpringVelocity: 0.1,
```

```
options: [.curveEaseInOut, .autoreverse, .repeat],  
animations: {  
    labelCenterYAnchorConstraint.constant -= 32  
    view.layoutIfNeeded()  
},  
completion: nil)
```

With this code, you:

1. Remove `labelTopAnchorConstraint` from `label`.
2. Add and activate `labelCenterYAnchorConstraint`, which centers `label` in its container's on the vertical axis.
3. Update any `view`'s pending layout changes.
4. Animate `label`'s vertical center constraint.

**Execute** the code you added, and your live view now looks like this:



Great! It looks like you've got the hang of experimentation using a playground. Next, you'll create a custom button.

## Creating a custom button

With structured and focused playground pages, developers visiting your playground can understand your code with less cognitive load. In this section, you'll up your game and create a custom button: `MusicButton`.

Open the **Music Button** page. Inside this page, you'll see `MusicButton`, a subclass of `UIButton`. `MusicButton` contains `MusicGenre`, which dictates `MusicButton`'s appearance and its associated audio track.

At the end of the page, add the following code:

```
let size = CGSize(width: 200, height: 300)
let frame = CGRect(origin: .zero, size: size)
let musicButton = MusicButton(frame: frame)
PlaygroundColor.current.liveView = musicButton
//: Different music genre gives `MusicButton` a different look.
musicButton.musicGenre = .rock
```

First, you initialize `musicButton` with a frame. You then set `musicButton` to the live view. After that, you create some documentation explaining the code. Finally, you set the music genre of the music button to rock.

**Execute** the playground, and you'll see this:



When developers visit your playground, they may want to understand the effect a music genre has on `MusicButton`. At the same time, you may want to let developers know that the button's music genre is what makes it special.

Below the code you added, add the following to see the jazz effect:

```
musicButton.musicGenre = .jazz
```

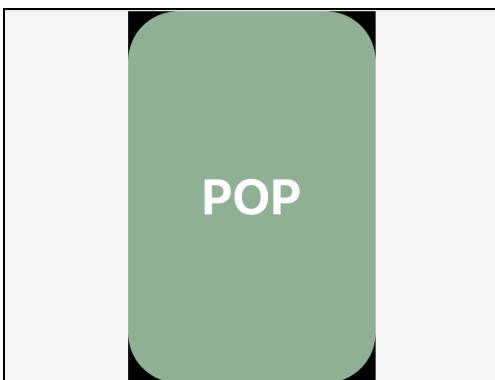
**Execute** the next line of code, and you'll see this:



Now, add the following code to see the pop effect:

```
musicButton.musicGenre = .pop
```

**Execute** the next line of code, and you'll see this:



Next, add the following code to document the audio playing feature of `musicButton`:

```
/*
  Each music genre is associated with an audio track from the
  **Resources** folder.

  You can prepare the audio player by calling
  `makeAudioPlayer()`.

  Afterward, you can call `play()` on the audio player to play
  the associated audio track`.

*/
```

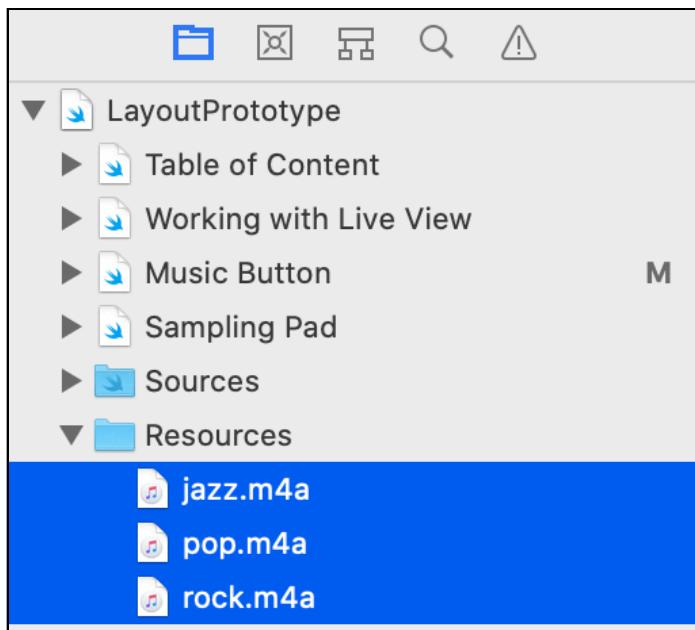
```
let audioPlayer = musicButton.makeAudioPlayer()  
audioPlayer?.play()
```

Here, you add some documentation informing developers about audio tracks, creating an audio player, and getting the audio player to play.

**Execute** the playground to the final line of code.

You'll hear a song, specifically an audio track that's associated with the button's music genre, so make sure you turn up your sound.

To integrate assets like image, video and audio, you can drag them into the playground's **Resources** folder; you'll find that folder in the Project navigator.



To decrease and increase the audio player's volume in real-time, add the following code:

```
audioPlayer?.volume -= 0.5  
audioPlayer?.volume += 0.5
```

**Execute** the code above line-by-line. Notice how the audio player first decreases and then increases the volume.

Now that you have a grasp on how to prototype in a playground, you'll put what you've learned into creating a more fun and complex view. How about a sampling pad?

## Moving code to the Sources folder

Playgrounds allow you to share code between playground pages. In this section, you're going to make `MusicButton` accessible to all playground pages.

Open the **Music Button** page. Comment out `MusicButton`'s class declaration.

In the playground's main **Sources** folder, open **MusicButton.swift**, and then uncomment all of the code inside that file.

The `MusicButton` code you've uncommented is a refactored version of the previous `MusicButton` with two main differences:

1. It uses `public` access modifiers to make the code accessible to all playground pages.
2. There's a new delegate property, and when `touchesEnded(_:with:)` triggers, it notifies that delegate.

Now that you've made your code reusable across all playground pages, you'll create a sampling pad.

## Working with more complex custom views

When views get complex, it's a good idea to include documentation on how a complex view works. With clear documentation, other developers can understand your views. In addition to documentation, the live view can make any custom view interactive for developers. They can test, experiment and observe empirical evidence.

Open the **Sampling Pad** page. Below the frameworks import, add the following code to begin creating the sampling pad:

```
final class SamplingPad: UIView {
    private var audioPlayer: AVAudioPlayer?

    init() {
        let size = CGSize(width: 600, height: 400)
        let frame = CGRect(origin: .zero, size: size)
```

```
    super.init(frame: frame)
}

required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
}

func set(_ audioPlayer: AVAudioPlayer) {
    audioPlayer.enableRate = true
    self.audioPlayer = audioPlayer
}

func playAudioPlayer() {
    audioPlayer?.play()
}

func update(_ volume: Float) {
    audioPlayer?.volume = volume
}
```

The code declares `SamplingPad`, a custom `UIView`. `SamplingPad` has an audio player and methods that play sound and adjust the audio player's volume.

Immediately below the code you just added, add the following to customize the sampling pad with music buttons:

```
//: ### Embed sampling pad and music buttons in a horizontal
stack view.
let samplingPad = SamplingPad()
PlaygroundPage.current.liveView = samplingPad

let rockMusicButton = MusicButton(type: .system)
rockMusicButton.musicGenre = .rock

let jazzMusicButton = MusicButton(type: .system)
jazzMusicButton.musicGenre = .jazz

let popMusicButton = MusicButton(type: .system)
popMusicButton.musicGenre = .pop

let horizontalStackView = HorizontalStackView(arrangedSubviews:
    [rockMusicButton,
     jazzMusicButton,
     popMusicButton])
horizontalStackView.distribution = .fillEqually
```

With this code, you set the live view to SamplingPad. SamplingPad initializes with the default 600×400 frame size. You then create the rock, jazz and pop music buttons. Afterward, you embed the buttons into a horizontal stack view with its distribution set to fill equally.

Next, add the following code:

```
//: ### Embed the horizontal stack view into a vertical stack view.  
let verticalStackView = VerticalStackView(  
    arrangedSubviews: [horizontalStackView])  
verticalStackView.translatesAutoresizingMaskIntoConstraints =  
    false  
verticalStackView.axis = .vertical  
samplingPad.addSubview(verticalStackView)
```

Here, you embed the horizontal stack view into a vertical stack view. You also prepare verticalStackView for Auto Layout, set its axis to vertical, and then add it to samplingPad.

Below that code, add the following:

```
//: ### Set up vertical stack view layout.  
let verticalSpacing: CGFloat = 16  
let horizontalSpacing: CGFloat = 16  
NSLayoutConstraint.activate(  
    [verticalStackView.leadingAnchor.constraint(  
        equalTo: samplingPad.leadingAnchor,  
        constant: horizontalSpacing),  
     verticalStackView.topAnchor.constraint(  
        equalTo: samplingPad.topAnchor,  
        constant: verticalSpacing),  
     verticalStackView.trailingAnchor.constraint(  
        equalTo: samplingPad.trailingAnchor,  
        constant: -horizontalSpacing),  
     verticalStackView.bottomAnchor.constraint(  
        equalTo: samplingPad.bottomAnchor,  
        constant: -verticalSpacing)])  
samplingPad.layoutIfNeeded()
```

This code adds and activates constraints for verticalStackView. It also gets samplingPad to update any pending layout changes.

Execute the playground, and you'll see this:



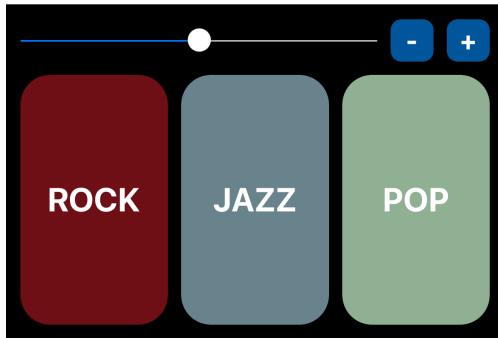
Immediately below the code you already added, add the following to begin integrating VolumeButtons into the sampling pad:

```
//: ### Create and set up layouts for volume controls.  
// 1  
let decreaseVolumeButton = VolumeButton(type: .system)  
decreaseVolumeButton.volumeButtonType = .decrease  
let increaseVolumeButton = VolumeButton(type: .system)  
increaseVolumeButton.volumeButtonType = .increase  
let volumeSlider = VolumeSlider()  
// 2  
let volumeButtonsStackView =  
    HorizontalStackView(  
        arrangedSubviews: [  
            decreaseVolumeButton,  
            increaseVolumeButton])  
volumeButtonsStackView.distribution = .fillEqually  
let volumeControlsStackView =  
    HorizontalStackView(  
        arrangedSubviews: [  
            volumeSlider,  
            volumeButtonsStackView])  
verticalStackView.insertArrangedSubview(  
    volumeControlsStackView,  
    at: 0)  
// 3  
volumeButtonsStackView.widthAnchor.constraint(  
    equalToConstant: 120).isActive = true  
samplingPad.layoutIfNeeded()
```

With this code, you:

1. Create an increase VolumeButton, decrease VolumeButton and VolumeSlider.
2. Arrange the controls inside of the stack views, and insert the final stack view into verticalStackView.
3. Give the volume buttons stack view a width. You also update any pending layout changes in samplingPad.

**Execute** to the last of line code, and you'll see this:

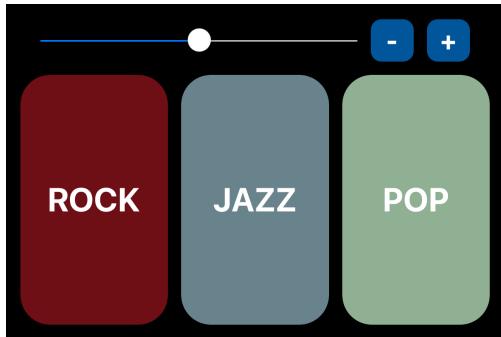


Now, add the following code to customize the volume control's stack view layout:

```
//: ### Add spacer view to `volumeControlsStackView`.  
let leftSpacerView = UIView()  
let rightSpacerView = UIView()  
volumeControlsStackView.insertArrangedSubview(  
    leftSpacerView,  
    at: 0)  
volumeControlsStackView.addArrangedSubview(rightSpacerView)  
  
//: ### Add width constraints to spacer views.  
NSLayoutConstraint.activate(  
    [leftSpacerView.widthAnchor.constraint(equalToConstant: 8),  
     rightSpacerView.widthAnchor.constraint(  
         equalTo: leftSpacerView.widthAnchor)])  
samplingPad.layoutIfNeeded()
```

With this code, you push the custom volume controls inward using spacer views, 8 points from the left and the right.

**Execute** to the last line of code, and you'll see this:



To notify the sampling pad of a `MusicButton` touch action, add the following code to the end of the page:

```
//: ### Set up `samplingPad` with `MusicButtonDelegate`.
extension SamplingPad: MusicButtonDelegate {
    func touchesEnded(_ sender: MusicButton) {
        guard let audioPlayer = sender.makeAudioPlayer()
            else { return }
        set(audioPlayer)
        playAudioPlayer()
        volumeSlider.setValue(1, animated: true)
    }
}
```

Here, you extend `SamplingPad` to handle setting up the audio player for the music buttons. You also adjust the audio player's volume back to 1 anytime a music button triggers `didTouchUpInside(_)`.

Add the following code to the end of the page:

```
//: ### Set up `SamplingPad` with `MusicButtonDelegate`
adoption.
[rockMusicButton, jazzMusicButton, popMusicButton]
    .forEach { $0.delegate = samplingPad }
```

This code sets the delegate of all rock, jazz and pop music buttons to `samplingPad`.

**Execute** to the last line of code. **Click** a music button in the live view, and you'll hear some music playing.

You're getting close, but you're not done yet, so add the following code to the end of the page:

```
// ### Update volume slider to associate with volume button
action.
extension SamplingPad {
    @objc func volumeSliderValueDidChange(
        _ sender: VolumeSlider) {
        audioPlayer?.volume = sender.value
    }
}
volumeSlider.addTarget(
    samplingPad,
    action: #selector(
        SamplingPad.volumeSliderValueDidChange),
    for: .valueChanged)
```

Now, when you slide the volume knob across the slider, the sample pad's audio player will adjust its volume accordingly.

Next, add the following code to the end of the page:

```
// ### Set up `SamplingPad` with volume buttons.
extension SamplingPad {
    @objc func volumeButtonDidTouchUpInside(
        _ sender: VolumeButton) {
        let change: Float =
            sender.volumeButtonType == .increase ? 0.2 : -0.2
        volumeSlider.value += change
        audioPlayer?.volume = volumeSlider.value
    }
}
```

Here, you extend `SamplingPad` to make `VolumeButton` increase or decrease the `volumeSlider`'s value and `audioPlayer`'s volume. The change in volume and slider is either an increase or decrease of `0.2` depending on the volume button's type.

Below the code you just added, add the following:

```
[increaseVolumeButton, decreaseVolumeButton].forEach {
    $0.addTarget(
        samplingPad,
        action: #selector(
            SamplingPad.volumeButtonDidTouchUpInside(_:)),
        for: .touchUpInside) }
```

This code sets the volume button's delegate to `samplingPad`, which enables the volume increase/decrease features using the plus or minus volume buttons in `samplingPad`.

Finally, **execute** to the last line of code — and there you have it, your own sampling pad. Go crazy, mix-up some sick tracks and blow some minds!

You've gone a long way in this chapter. You started by prototyping a simple magenta view. You then took what you learned early on and developed a sampling pad — and you've done it all using playgrounds.

Playgrounds help you get your ideas quickly off the ground. They also let you run code line-by-line, get responsive and immediate feedback and help you to create wonderful documentation for your code, which is perhaps one of the most overlooked gems in a development process. Documentation is vital for clear communication, productivity, code sharing.

## Key points

- Playgrounds are arguably the quickest way to go from idea to prototype.
- In comparison to an Xcode project, playgrounds reduce the amount of boilerplate code developers must see. Developers also won't need to build and run in the prototype phase continuously and will no longer need to risk having existing code unintentionally influencing new code and vice versa.
- Use markup formatting to structure and format your playground content. Aim for clarity, focus, and ease of experimentation.
- Playgrounds have a live view to display interactive content.
- Playgrounds allow developers to run code up to a particular line; then, write and execute new code without having to stop and execute the playground entirely again.
- Place playground assets in the Resources folder.
- Place code you'd like to make accessible to all playground pages in the Sources folder.

# Chapter 17: Auto Layout for External Displays

By Jayven Nhan



Nowadays, using an external display goes well beyond merely connecting a monitor to a stationary computer. In recent years, more and more people are looking to connect their mobile devices to some type of external display — and as an app developer, it's your responsibility to know how to handle this demand.

For iOS, there are currently three popular external display solutions:

- AirPlay
- Physical connections (cables)
- Chromecast



Generally speaking, an iPhone user will often look for this type of functionality for:

- **Video playback:** The iOS device becomes the video playback controller, and the external display becomes the core content provider.
- **Powerpoint presentations:** The iOS device shows the slide notes to the presenter while the external display shows the slide content to the audience.
- **Gaming:** The iOS device becomes the game controller, and the external display becomes the core content provider.

While this type of symbiotic relationship is common — where the device is the controller and the display shows the content — it's not the only reason users rely on external displays. Some like to use external displays simply because they offer more screen real estate than their mobile counterparts. Whatever the reason may be, by adding support for external displays, you can provide a better overall experience for your users.

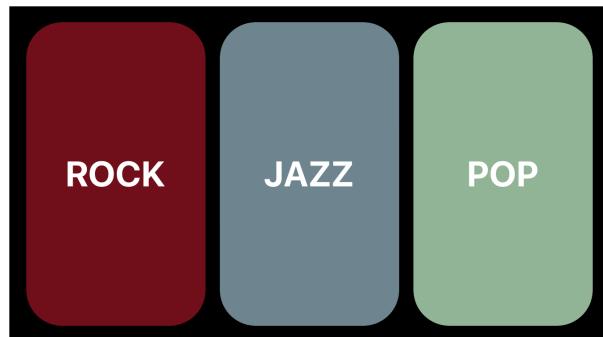
In this chapter, you'll learn how to:

- Build a layout for an external display.
- Configure an external display window.
- Handle new external display connections.
- Handle existing external display connections.
- Handle external display disconnections.
- Accommodate different external display resolutions.

You'll accomplish all of this by building a music playback app that supports external displays. By the end of this chapter, you'll know how to support external displays in your iOS project, and in the process, you'll gain greater clarity of using windows within your apps.

## Getting started

Open **ExternalDisplay.xcodeproj** in the **starter** folder. Build and run the app on the simulator.

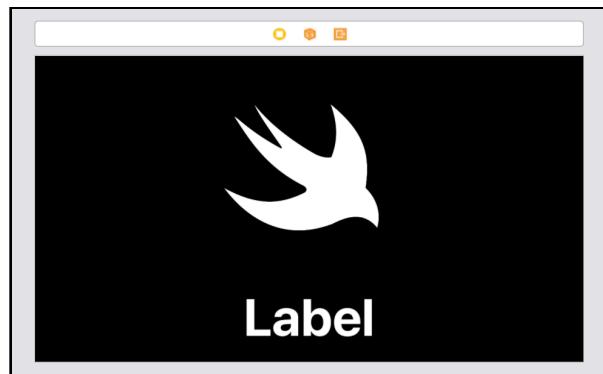


From within the **Simulator** app, select **Hardware** ▶ **External Displays** ▶ **1920×1080 (1080p)**. For now, you won't see anything but a blank screen. Close the simulator, and return to Xcode.

The good news is that you can use the Auto Layout fundamentals you already know to build layouts for external displays — provided you understand the key differences.

## Building a layout for an external display

Open **Main.storyboard** and look at **MusicPlayerViewController**. This will be the view controller for the external display.



Notice the existing Auto Layout constraints; these will save you the trouble of having to implement them yourself. For **Stack View**, you have the following constraints:

- Center aligned to the superview's horizontal axis.
- Center aligned to the superview's vertical axis.
- Leading edge greater than or equal to 16 of the safe area leading edge.
- Trailing edge greater than or equal to 16 of the safe area trailing edge.
- Top edge greater than or equal to 16 of the safe area top edge.
- Bottom edge greater than or equal to 16 of the safe area bottom edge.

With these constraints, you center the **Stack View** within the container and set limitations on how much the stack view can grow, so that it avoids having its content spill beyond the visible screen area.

For **Artwork Image View** and **Music Genre Label**, you have a greater than or equal to width relation constraint between the two. This gives the label an equal width to the image view while also giving it the flexibility to outgrow the image view's width.

If the label's intrinsic content width fits within the image view's width, it looks like this:



If the label's intrinsic content width is greater than the image view's width, it instead looks like this:



These constraints are applied to `MusicPlayerViewController`. Your next task is to configure an external display window.

## Configuring an external display window

For every display that you plan to show to your users, you contain it inside of a view. A window is the parent of all views for a display, whether it's an iPhone, iPad or external display. For supporting an additional external display, it's paramount that you correctly create and manage the external display's window to show separate content and present a smooth user experience to your users.

In your project, you have the main window that displays your app's main content. Similarly, to display content on an external display, you'll need to create an additional window.

Open `MusicSamplingController.swift` and add the following method:

```
private func makeWindow(from screen: UIScreen) -> UIWindow {  
    // 1  
    let bounds = screen.bounds  
    let window = UIWindow(frame: bounds)  
    // 2  
    window.screen = screen  
    // 3  
    window.rootViewController = musicPlayerViewController  
    // 4  
    window.isHidden = false  
    return window  
}
```

A few things are occurring here:

1. Given a `UIScreen`, take its bounds as the size of the `UIWindow`.
2. Set the window's screen to the one in the argument. You do this to let the window know on which screen to show up.
3. Set the window's root view controller to `musicPlayerViewController`.
4. Unhide the window before returning it.

Apple recommends as a best practice to provide the window with a screen before showing it because changing the screen of a visible window is an expensive operation.

The structure of the code you've added may remind you of programmatically initializing a view controller in earlier chapters. And that's right! It's very similar.

Here's a sample chunk of code that programmatically initializes a view controller:

```
let storyboard = UIStoryboard(name: "Name", bundle: nil)
let viewController =
    storyboard.instantiateInitialViewController()
window = UIWindow(frame: UIScreen.main.bounds)
window?.rootViewController = viewController
window?.makeKeyAndVisible()
```

The code above creates a window to display your app content on an iOS device. It creates a window using the size of the device. On the other hand, `makeWindow(from:)` creates a window using the external screen size.

Next, you'll handle an existing external display connection and place your layout onto the screen.

# Handling an existing external display connection

Your app can launch with or without an external display connection established. In this section, you'll learn how to handle cases in which an external display is already connected.

Open **MusicSamplingController.swift**. Add the following code to `configureExistingScreen()`:

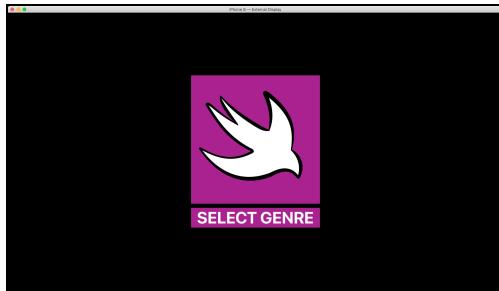
```
// 1
let screens = UIScreen.screens
// 2
guard
    screens.count > 1,
    let screen = screens.last
    else { return }
// 3
let window = makeWindow(from: screen)
externalWindows.append(window)
```

Here are the configuration amendments:

1. Reference all of the screens into a constant.
2. Using the guard statement, ensure the screen count is greater than 1. If this condition passes, then safely unwrap the last screen in the array of available screens.
3. Using the safely unwrapped screen, create a window. Then, append the window into the external windows array for future references.

With the simulated external display open and ready in the Simulator app, build and run.

You'll see the following on the external display:



Close the external display window. Build and run the app again. After the app launches, open the simulated external display, and this time, you'll see a black screen.

Currently, the app doesn't know how to handle a new external display connection. It's time to fix that!

## Connecting a new external display

Although most people find at-most two external displays on their desk, Apple's documentation doesn't state the upper bound of simultaneously connected external displays. Whether you set up your app to support one or ten external displays, you need to implement business logic to deal with the connectivity of the displays. One way to do that is to observe external display event notifications.

To receive external display event notifications, you use `NotificationCenter`. This API broadcasts specific information to specific registered observers. The information sent to these observers depends on the notification name registered under the object. You can utilize a custom or system notification name to register an object to receive notifications. Since external display events are system events, you'll use system notification names for convenience and standardization.

Add the following code to `observeConnectedScreen()` in `MusicSamplingController.swift`:

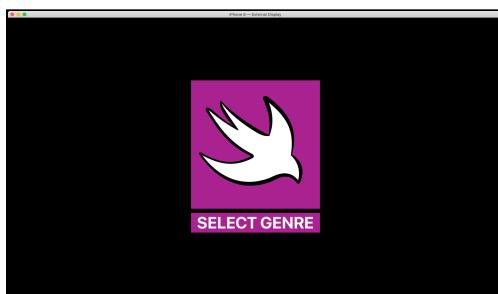
```
// 1
NotificationCenter.addObserver(
    forName: UIScreen.didConnectNotification,
    object: nil,
    queue: nil) { [weak self] notification in
    // 2
    guard
```

```
let self = self,
let screen = notification.object as? UIScreen
else { return }
// 3
let window = self.makeWindow(from: screen)
self.externalWindows.append(window)
}
```

Here's what you did:

1. Observe the screen connection notification.
2. Safely unwrap the notification object as UIScreen.
3. Create the external display window and append the window into `externalWindows`.

Close the external display window. Build and run. After the app launches, open an external display. This time, you'll see `MusicPlayerViewController` in the external display.



Great! You now have the new external display connection workflow established. Next up, you'll handle external display disconnection events.

## Disconnecting an external display

Similar to the previous implementation, you'll use `NotificationCenter` to handle disconnecting an external display connection.

Add the following code to `observeDisconnectedScreen()` in `MusicSamplingController.swift`:

```
// 1
notificationCenter.addObserver(
    forName: UIScreen.didDisconnectNotification,
```

```
object: nil,
queue: nil) { [weak self] notification in
    // 2
    guard
        let self = self,
        let screen = notification.object as? UIScreen
    else { return }

    // 3
    for (index, window) in self.externalWindows.enumerated() {
        guard window.screen == screen else { continue }
        self.externalWindows.remove(at: index)
    }
}
```

With this code, you:

1. Observe for a screen disconnection notification.
2. Safely unwrap the notification object as a UIScreen.
3. Enumerate the externalWindows. In the loop where the disconnected screen equals the current iteration screen, remove the window object at the loop index.

This completes the external display disconnection logic flow. Next, you'll learn to accommodate for different external display resolutions.

## Accommodating external display resolutions

Depending on the external display size, you may want to display the UI differently. In this section, you'll set the music genre label's visibility based on the display resolution.

Add the following method to `MusicSamplingController` in `MusicSamplingController.swift`:

```
// 1
private func setMusicGenreLabelVisibility(screen: UIScreen) {
    // 2
    guard let currentMode = screen.currentMode else { return }
    // 3
    screen.availableModes.forEach {
        print("Available mode:", $0)
    }
    // 4
```

```
let lowerBoundSize = CGSize(width: 1024, height: 768)
// 5
self.musicPlayerViewController.musicGenreLabel.isHidden =
    currentMode.size.width <= lowerBoundSize.width
}
```

With this code, you:

1. Create a method that takes a UIScreen parameter.
2. Safely unwrap the screen's current mode for the screen's resolution.
3. Print the available modes of the screen to the console, which lets you know about the available resolutions.
4. Create a lower bound size object to compare width with the current screen mode size. The comparison will help decide which user interface to show/hide.
5. The music genre label is hidden when the screen width is less than or equal to the lower bound size width.

Add the following line of code to observeConnectedScreen() as the last line inside the notification closure:

```
self.setMusicGenreLabelVisibility(screen: screen)
```

This ensures the external display connection event invokes the music genre label's visibility presentation logic.

Finally, add the following code to observeScreenResolutionChanges():

```
// 1
notificationCenter.addObserver(
    forName: UIScreen.modeDidChangeNotification,
    object: nil,
    queue: nil) { [weak self] notification in
    // 2
    guard
        let self = self,
        let screen = notification.object as? UIScreen
    else { return }

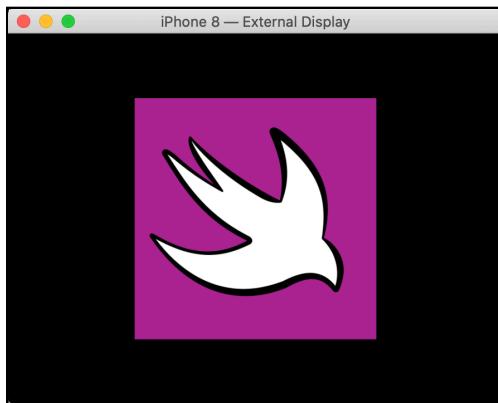
    // 3
    self.setMusicGenreLabelVisibility(screen: screen)
}
```

Here's what you did:

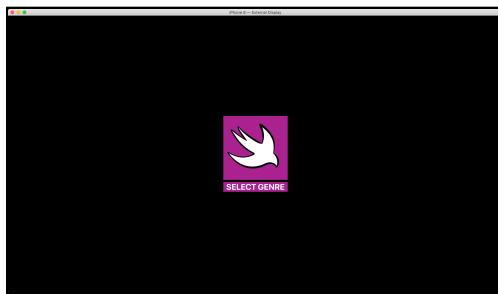
1. Observe for the screen resolution change notification.
2. Safely unwrap the notification object as UIScreen.
3. Call `setMusicGenreLabelVisibility(screen:)` and pass in the safely unwrapped screen object.

Build and run.

An app running on a 1024×768 resolution display will look like this:



And an app running on a 3840×2160 (4K) resolution display will look like this:



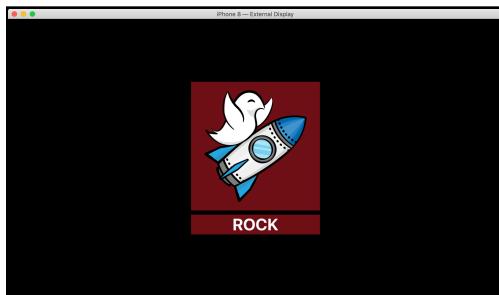
**Note:** If you're testing on the simulator, launching an app with a connected external display defaults to the screen's first screen mode. For example, a connected 1280×720 simulator external display two screen modes, 720×480 and 1280×720. The app defaults to the former screen mode.

There you go! You've learned how to extract a screen's resolution, check the available resolutions and use screen resolution information to adjust your layout.

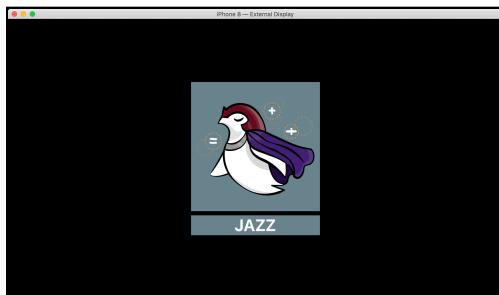
Now, ensure the rest of the app looks great.

Build and run. Tap the rock, jazz and pop buttons. You will see the following on the external display:

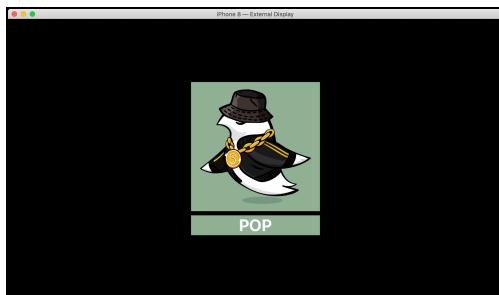
Rock:



Jazz:



Pop:



Excellent, the UI looks great on the external display. You've learned the intricacies of supporting external displays. By supporting an extensive range of iOS features, your users can expect seamless plug-and-play experiences from your apps.

Even when the display size is a 24" monitor, Auto Layout adapts accordingly. Your users can expect beautifully laid out user interfaces from your app no matter how or on which device they use it on.

## Key points

- The two native external display solutions in iOS are AirPlay and a physical cable.
- Use `NotificationCenter` to observe existing, new and disconnected external display connections.
- Extract and adapt the app's UI to the screen resolution using the external display screen mode attribute.

# 18

# Chapter 18: Designing Custom Controls

By Jayven Nhan

Standard UIKit controls are native, intuitive, and they support out-the-box features for iOS users. However, as you develop your app, you might discover that the standard controls limit you to a specific set of features that don't meet your app's requirements. Whether you want to implement a different user interface layout or some non-standard user interaction, understanding how to create custom controls will help you achieve the desired results.

Depending on the level of customizations you want to make, creating custom controls can require a substantial amount of work. The more customizations or features you want to add, the more work you'll need to do. But that's not all you need to think about. Your approach to making a custom control also dictates the amount of work you have ahead of you.

To make a custom UIKit control, you have two options: You can subclass a standard control like a `UIButton`. Or, you can subclass a more generic class like `UIView` or `UIControl`. Each approach has its pros and cons, which you'll discover as you build your first custom control.



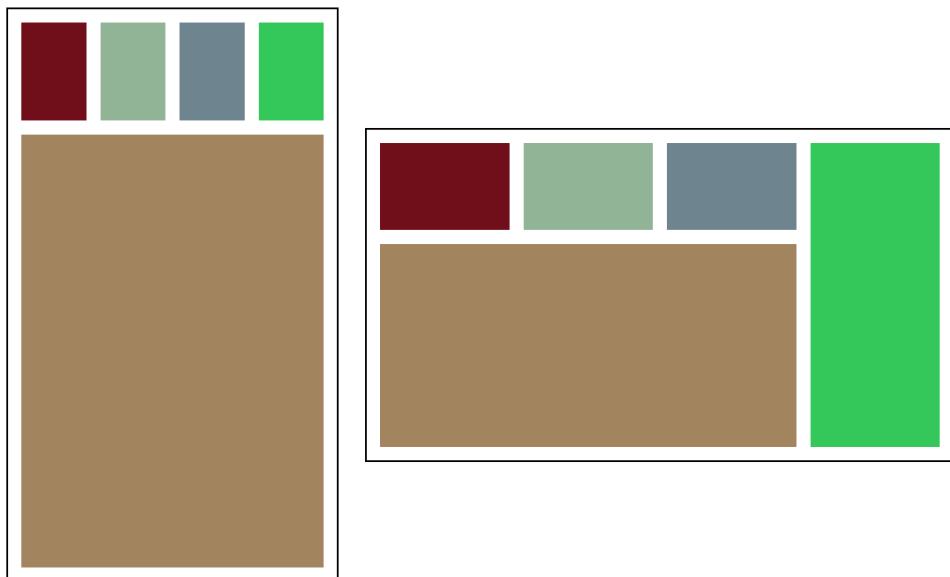
In this chapter, you'll create a custom DJ deck user interface. In doing so, you'll learn the following topics:

- Subclassing and making a custom `UIView`.
- Visualizing XIB/NIB files in a storyboard.
- Customizing user interfaces according to size classes.
- Subclassing and making a custom `UIButton`.
- Subclassing and making a custom `UIControl`.
- Adopting accessibility features.

## Getting started

You'll start by subclassing a `UIView` to create a custom view. Open the **starter project**. To keep this chapter focused, you'll use an existing view and NIB file, located inside the project.

For the custom DJ deck, the layout blueprint for landscape and portrait orientations is as follows:



For this control, you'll implement the following custom user interfaces:

- Backlit Button.
- Disc Spinner View.
- Pitch Control.

These three custom user interfaces make up the custom DJ deck interface.

## Applying adaptive layout to a custom view

Open **DJControllerView.xib** in the **Xibs** group.

You have some standard UIKit views that are laid out without using any adaptive layout. It's your job to make the view's layout adaptive for the different screen sizes and size classes based on the layout blueprint.

You already learned how to work with size classes and adaptive layouts in Chapter 10, "Adaptive Layout". Before moving on to the implementation guide — and to practice and help solidifying your understanding — you'll first implement the adaptive layout on the custom view on your own.

The next section walks you through the custom view's adaptive layout implementations. It assumes you remember how to perform the individual tasks. Refer back to the earlier chapters if you need a refresher.

## Laying out the stack view

First, you'll handle the layout of the backlit buttons (three top-left buttons).

Implement the following layout changes:

1. Select the **three backlit buttons** from the top-left corner of the container view. Embed the views into a **stack view**.
2. Set the stack view's alignment equal to **Fill**. Set its distribution to **Fill Equally** and the spacing to **16**.
3. Set the stack view's **leading edge and top edge 16 points from the superview**.
4. Set a **width-to-height** aspect ratio constraint on the red backlit button with a multiplier of **1:1.5**.

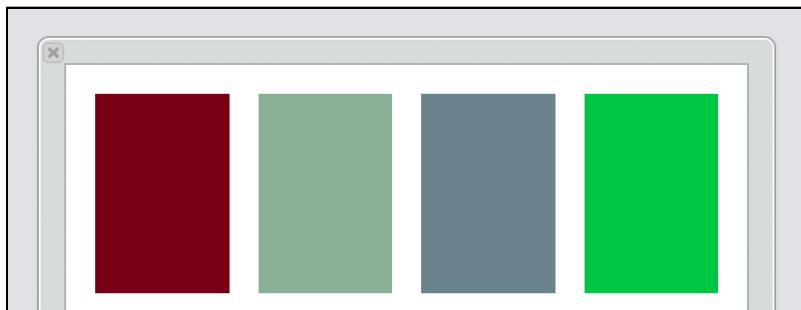


Here, you're embedding the backlit buttons into a stack view. You then configure the stack view's subviews to fill the container space with equal size and spacing distributions. Finally, you apply Auto Layout constraints onto the stack view and the red backlit buttons.

## Laying out the pitch control

Next, you'll work on the pitch control (top-right view). Implement the following layout changes to the **pitch control**:

1. Leading edge equals **16** points from the stack view's trailing edge.
2. Top edge aligns to the stack view's top edge.
3. Trailing edge equals **16** points from the superview's trailing edge.
4. Height equals to the red backlit button's height with a **1:1** ratio.
5. Width equals to the red backlit button's width with a **1:1** ratio.



With these changes, you apply equality constraints to the pitch control's leading, top and trailing edges. You also apply width and height aspect ratio constraints between the pitch control and the red backlit button.

## Laying out the disc spinner view

Finally, you'll set up the disc spinner view (bottom view). Implement the following layout changes to the **disc spinner view**:

1. Leading, bottom and trailing edges equal to **16** spaces to its superview's respective edges.
2. Top edge equals **16** spaces from the stack view's bottom edge. Make sure the relationship is with the stack view and not the pitch control.

With these changes, you apply equality constraints onto the edges of the disc spinner view.

The DJ controller view's layout should now match its design blueprint in portrait orientation; however, the custom view currently looks like this in landscape:



You have some more adaptive layout work to do. Now might be a great time to call on size classes for assistance.

## Making user interface variations

To begin implementing user interface variations, you'll need to first set your Interface Builder's preview orientation to **landscape** so you'll be able to see how things look in landscape.

Now, apply the following changes to the **red backlit button**:

1. Select **width to height aspect ratio** constraint. Add **any width compact height** variation. Uncheck the added variation's **Installed** checkbox.
2. Create another **width to height aspect ratio** constraint. Set the multiplier to **1.5:1**. Uncheck the **Installed** checkbox. Add an **any width compact height** variation. This time, check the **Installed** checkbox.

The first step ensures Auto Layout knows to uninstall the first aspect ratio constraint when the height is compact. The second step ensures Auto Layout knows to install a different aspect ratio constraint when the height is compact.

You'll see the following layout on an iPhone 8 in landscape orientation:



For devices that fall into the regular height category, the aspect ratio constraint that Auto Layout installs is the constraint with the 1:1.5 width-to-height aspect ratio.

**Tip:** Using Interface Builder's device preview, you can conveniently verify your size class layout assumptions on larger and smaller devices.

Next, select **disc spinner view** and apply the following layout changes:

1. Select **trailing alignment** constraint. Add an **any width compact height** variation. Uncheck the added variation's **Installed** checkbox.
2. Align the **trailing edge** to the stack view's trailing edge. Uncheck the **Installed** checkbox. Add an **any width compact height** variation. Check the added variation's **Installed** checkbox.

Here, you variate the disc spinner's trailing constraints based on the available vertical space on the device.

You'll see the following layout on an iPhone 8 in landscape orientation:



Finally, to complete the user interface variations, you'll need to configure the pitch control.

Select **pitch control**, and apply the following layout changes:

1. Select the **equal heights** constraint. Add an **any width compact height** variation. Uncheck the added variation's **Installed** checkbox.
2. Align the bottom edge to the disc spinner's **bottom edge**. In the newly added constraint, uncheck the default **Installed** checkbox. Add an **any width compact height** variation. Ensure the added variation's **Installed** checkbox is checked.

Your view will now look amazing and as intended by design.



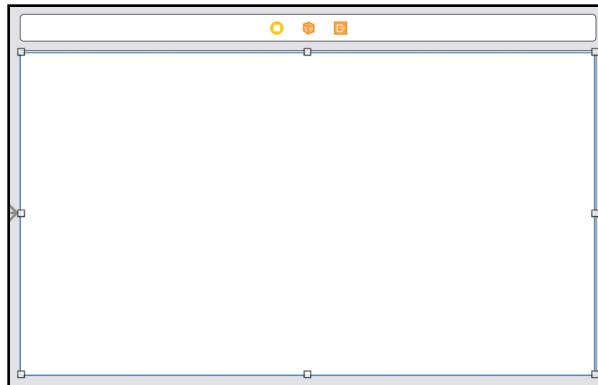
Next, you'll learn how to integrate your custom view designed in NIBs to storyboards.

## Integrating a custom view from NIB to storyboard

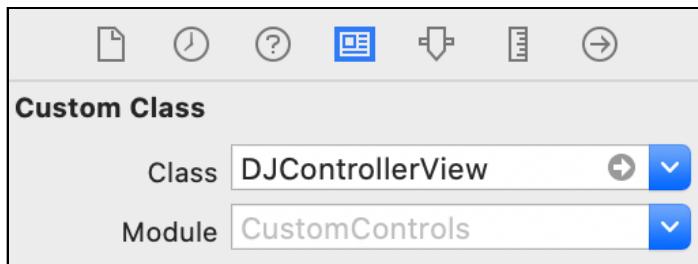
Now that you've configured your custom view in the NIB, you'll integrate it into the main storyboard.

Open **Main.storyboard**. You'll see a blank canvas view controller.

Drag a **UIView** from the Object Library onto the view controller. Then, pin the view to the **edges of its container**.



Select the **view**. In the Inspectors panel, click **Show the Identity inspector**. Then, set the custom class to **DJControllerView**.



At the moment, there are no visible changes in the storyboard. As you can imagine, this is a sub-optimal solution for when you want to see how a custom view looks in the Interface Builder. After all, one of Interface Builder's greatest strengths is giving you the ability to see the user interface without needing to always build and run.

But don't worry. You can get this superpower back — even when you create a custom view using a NIB.

Next, you'll learn to prepare your custom view for Interface Builder visualization.

## Visualizing XIB/NIB files in storyboards

Build and run, and you'll see an empty-looking view controller. The reason for this is because you need to initialize the NIB file inside of your custom control.

Open **UIView+UINib.swift** located inside the **Extensions** group. Add the following extension code:

```
extension UIView {
    func instantiateNib<T: UIView>(view: T) -> UIView {
        // 1
        let type = T.self
        // 2
        let nibName = String(describing: type)
        // 3
        let bundle = Bundle(for: type)
        // 4
        let nib = UINib(nibName: nibName, bundle: bundle)
        // 5
        guard let view = nib.instantiate(
            withOwner: self,
            options: nil).first as? UIView
            else { fatalError("Failed to instantiate: \(nibName)") }
        // 6
    }
}
```

```
        return view  
    }  
}
```

With the extension method, you:

1. Create a simple and reusable method to instantiate views from NIBs.
2. Take the view's type and convert it into a string.
3. Initialize a bundle object using the view's type.
4. Construct a NIB object by passing in the NIB name and bundle into the respective parameters.
5. Instantiate the NIB object and set the caller as the NIB object's owner. You also safely extract the first object from the instantiated NIB file as a `UIView`.
6. Return the instantiated view.

In **DJControllerView.swift**, located inside the **Views** group, add the following property:

```
private lazy var view = instantiateNib(view: self)
```

To initialize the view from the NIB, you declare a lazy variable. This ensures the property initializer runs after the `self` initialization.

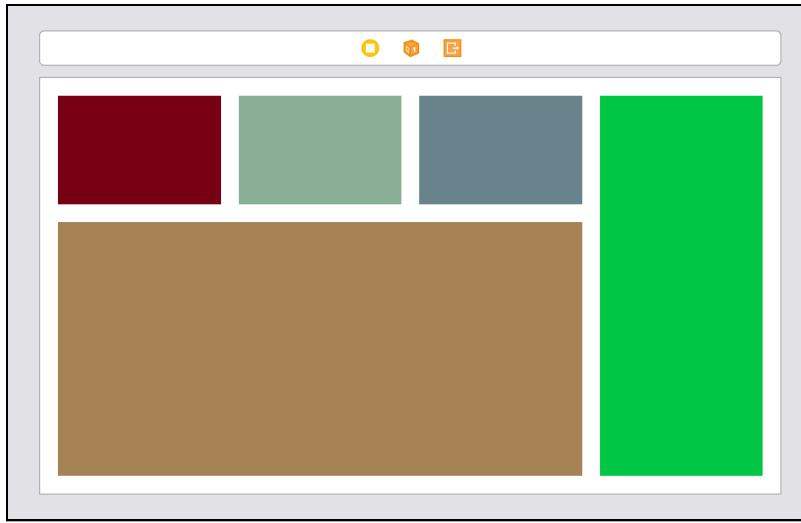
Now, add following code to `commonInit()`:

```
addSubview(view)  
view.fillSuperview(self)
```

Here, you add the view initiated from the NIB file. You then use an Auto Layout extension method to make `view` fill the edges of the custom view.



Open **Main.storyboard**. You'll see the following on an iPhone 8:



This is cool. But, there's more customization you can do with Interface Builder and custom views.

## Preparing custom views for Interface Builder

You can segregate the user interface in Interface Builder from runtime. In other words, you can customize your custom view to different user interfaces in Interface Builder and at runtime. You can use this feature, for example, to help other developers better understand your custom view.

Next, you'll customize your custom view to have a label that's indicative of the custom view's class name.

Open **UIView+InterfaceBuilder.swift**. Add the following extension code:

```
extension UIView {
    func addLabelDescribing<T: UIView>(
        view: T,
        insideSuperview superview: UIView
    ) {
        // 1
        let viewDescriptionLabel = ViewDescriptionLabel()
        // 2
    }
}
```

```
viewDescriptionLabel.text = String(describing: T.self)
// 3
superview.addSubview(viewDescriptionLabel)
// 4
viewDescriptionLabel.center(superview)
}
}
```

With the extension method, you:

1. Initialize a custom label with default properties.
2. Set the label's text to the class name.
3. Add the label onto the superview.
4. Center the label in the superview using a helper method.

In `DJControllerView`, add the following method override:

```
override func prepareForInterfaceBuilder() {
    super.prepareForInterfaceBuilder()
    addLabelDescribing(view: self, insideSuperview: view)
}
```

After the view loads in Interface Builder, you inject the code to add a label describing the custom view in the Interface Builder.

Open `Main.storyboard`, and you'll see the following:



That's pretty cool, right? Next, you'll learn to make a custom control from subclassing a standard control.

## Making a custom UIButton

In this section, you'll create another custom control by subclassing a standard UIKit control. In particular, you'll work with a `UIButton`.

A standard `UIButton` can sometimes feel boring. You'll implement some custom animation based on the button's user interaction event to spice things up!

Open `BacklitButton.swift`. Notice that the custom class already has the standard `UIButton` as a subclass.

Add the following animation methods:

```
// 1
private func shrinkAnimation() {
    let scale: CGFloat = 0.9
    UIView.animate(withDuration: 0.2) { [weak self] in
        self?.transform = CGAffineTransform(
            scaleX: scale,
            y: scale)
    }
}
// 2
private func resetAnimation() {
    UIView.animate(withDuration: 0.2) { [weak self] in
        self?.transform = .identity
    }
}
```

The first animation method scales the view to 90 percent of its original size. The second animation method resets to the view's original size.

Next, you need to implement the animation inside the custom button. Add the method overrides:

```
// 1
override func touchesBegan(
    _ touches: Set<UITouch>,
    with event: UIEvent?
) {
    super.touchesBegan(touches, with: event)
    shrinkAnimation()
}
```

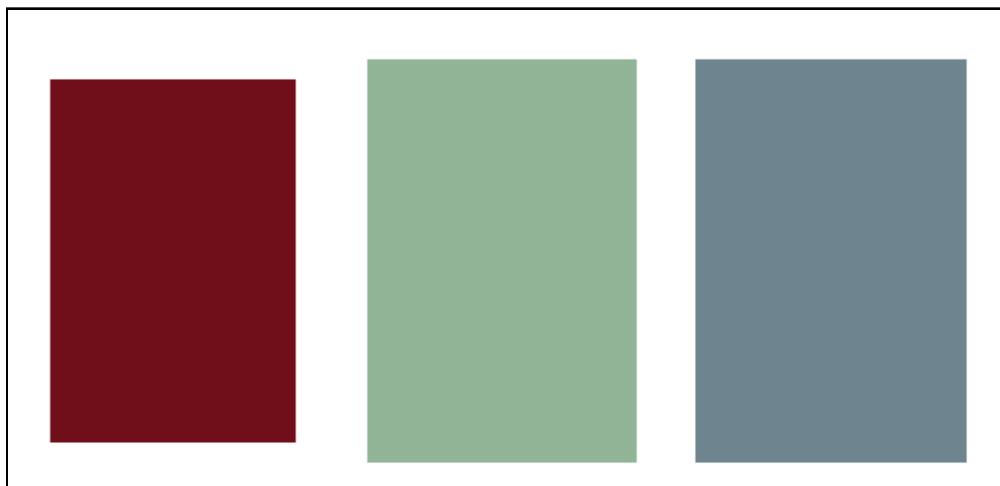
```
// 2
override func touchesEnded(
    _ touches: Set<UITouch>,
    with event: UIEvent?
) {
    super.touchesEnded(touches, with: event)
    resetAnimation()
}

override func touchesCancelled(
    _ touches: Set<UITouch>,
    with event: UIEvent?
) {
    super.touchesCancelled(touches, with: event)
    resetAnimation()
}
```

UIResponder is the superclass of UIView. For creating a custom view, UIResponder's primary purpose is to handle control events. Here's how the method overrides work:

1. Upon detecting a touches began event, you trigger the shrink animation.
2. Upon detecting a touches cancelled or ended event, you trigger the reset animation.

Build and run. When you tap a backlit button, you'll see the shrink animation in effect:



When you release the button, you'll see the button animate smoothly to its original form.

Next, you'll build out the pitch control.

## Making a custom UIControl

Standard UIKit controls inherit from `UIControl`. Standard controls include `UIButton`, `UISegmentedControl`, `UITextField`, `UISlider`, `UISwitch`, `UIPageControl` and more.

`UIControl` subclasses from `UIView`. In addition to the built-in `UIView` functionalities, `UIControl` also provides touch tracking, control state, touch events handling functionalities as application programming interfaces for developers. In this section, you'll learn to use `UIControl`'s functionalities to create your custom controls.

Particularly, you'll create a knob control and a slider control, and you'll place them inside `DJControllerView`. Depending on the size class, one of the two custom controls will show. The knob control will show for devices with compact heights, and a vertical slider will show on any larger devices.

## Customizing size class variations

The knob will consist of only a static image with no user interactivity. Your job is to have it show up and hidden in the corresponding size classes.

Open `DiscSpinnerView.swift`. Uncomment `view` and the code inside of `commonInit()`.

With this change, you add the NIB initiated view as a subview, and you pin the added view's edges to the container view.

Open **Main.storyboard**, and you'll see the following:



It's time to make this control show or hide depending on the screen size.

Open **PitchControl.swift**. Replace `view` with the following:

```
private lazy var view = instantiateNib(view: self)
```

With this code, you replace a standard `UIView` with a custom view initiated from NIB.

Open **PitchControl.xib**, and do the following:

1. From the document outline, select **Pitch Knob Control**.
2. Open the Attributes inspector, and click the + button next to the **Hidden** checkbox.
3. Add and enable an **any width and compact height** variation.
4. Add and enable a **regular width and regular height** variation.

The knob control shows or hides depending on the screen size. For devices such as the iPhone 11, the app will hide the pitch control only in landscape mode. For iPad devices, the app will show the pitch control in landscape orientation with exception of compact width split views.

Also, to avoid repeating a similar task, user interface variations for Slider Background Image View and Thumb Image View are implemented for you. Basically, it shows vertical slider user interfaces when the pitch control is hidden and vice versa.

Next, you'll start to implement control features for the vertical slider.

## Implementing control features for vertical slider

The vertical slider will have an interactive slidable thumb that enables users to adjust the control's value. In addition, the vertical slider will also implement Accessibility features.

Open **PitchControl.xib**, if it's not already open.

You have the slider background image view and the thumb image view. The Auto Layout constraint to focus on here is the top alignment equality constraint between the thumb image view's top edge and the slider background image view's top ledge. Later, you'll use this constraint to position the thumb image view as the user slides the thumb up or down.

It's time to set up the control's data properties.

## Setting up data properties

Your control will store various data properties to implement different business and presentation logic. First, add the following properties to **PitchControl**:

```
// 1
private let minValue: CGFloat = 0
private let maxValue: CGFloat = 10
// 2
private var value: CGFloat = 1 {
    didSet {
        print("Value:", value)
    }
}
// 3
private var previousTouchLocation = CGPoint()
```

With this code, you:

1. Define the slider's lower and upper bounds.
2. Set the slider's initial value to 1. Set a property observer to see the most up to date control's value in the console log.
3. Initialize a touch location variable to keep track and compare the user's touch input later.



Next, add the following computed properties:

```
// 1
private var valueRange: CGFloat {
    return maxValue - minValue + 1
}
// 2
private var halfThumbImageViewHeight: CGFloat {
    return thumbImageView.bounds.height / 2
}
// 3
private var distancePerUnit: CGFloat {
    return (sliderBackgroundImageView.bounds.height / valueRange)
        - (halfThumbImageViewHeight / 2)
}
```

With these computations, you:

1. Derive the control's value range from the lower and upper bounds defined earlier.
2. Halve the thumb image view's bounds height. This is a convenient way to access a value that you'll use more than once.
3. Calculate the distance travel between value increment and decrement. This is used when you implement adjustable accessibility values.

Your properties are settled for tracking user touch inputs.

Next, you'll make use of touch tracking handlers to implement movements on the thumb image view.

## Implementing touch tracking handlers

The touch tracking handler methods derived from `UIControl`. You'll now override the touch tracking handler methods to integrate custom logic into your project.

To begin, add the following method override:

```
override func beginTracking(
    _ touch: UITouch,
    with event: UIEvent?
) -> Bool {
    super.beginTracking(touch, with: event)
    // 1
    previousTouchLocation = touch.location(in: self)
    // 2
    let isTouchingThumbImageView = thumbImageView.frame
        .contains(previousTouchLocation)
    // 3
```

```
thumbImageView.isHighlighted = isTouchingThumbImageView  
// 4  
return isTouchingThumbImageView  
}
```

Here's the code breakdown:

1. Cache the touch location in `previousTouchLocation`. You'll need this value when comparing touch locations later in `continueTracking(_:with:)`.
2. Check if the user's touch input is on the thumb image view.
3. Set the thumb image view's highlight state to whether the user is touching the thumb image view.
4. The decision to begin tracking user input is defined by whether the user is touching the thumb image view.

The image view's highlight state provides visual aids to indicate that a control is currently being interacted with. By default, the control draws the highlight image when the property is `true`.

When the user is touching the thumb image view, the touch handling continues. Otherwise, the touch handling logic stops right there.

To continue handling user's touch input logic, add the following method override:

```
override func continueTracking(  
    _ touch: UITouch,  
    with event: UIEvent?  
) -> Bool {  
    super.continueTracking(touch, with: event)  
    // 1  
    let touchLocation = touch.location(in: self)  
    let deltaLocation = touchLocation.y  
        - previousTouchLocation.y  
    // 2  
    let deltaValue = (maxValue - minValue)  
        * deltaLocation / bounds.height  
    // 3  
    previousTouchLocation = touchLocation  
}
```

Here's the code breakdown:

1. Compute the change in y position between the current and previous touch locations.

2. Calculate the value change using the defined value bounds, position difference from touch locations and the control's height. You'll use this information to amend the control's value.
3. Cache the latest touch location.

Now, add the following code at the end of `continueTracking(_:with:)`:

```
// 1
value = boundValue(
    value + deltaValue,
    toLowerValue: minValue,
    andUpperValue: maxValue)
// 2
let isTouchingBackgroundImage =
    sliderBackgroundImageView.frame
        .contains(previousTouchLocation)
if isTouchingBackgroundImage {
    thumbImageViewTopConstraint.constant =
        touchLocation.y - self.halfThumbImageViewHeight
}
// 3
return true
```

Here's the code breakdown:

1. Use a helper method to ensure the control's value is within the upper and lower value bounds. Then, set the value accordingly.
2. If the user's touch input is within the frame of the slider's background image view, then update the thumb image view's top constraint constant to the current touch location minus half of the thumb image view's height. You minus half of the thumb image view's height to align the image view's center with the touch location.
3. Continue tracking unless your users release their fingers from the screen.

You use `boundValue(_:toLowerValue:andUpperValue:)` for edge cases such as when users slide their finger way above or below the control frame. In other words, you bound the value according to the value's constraints. You then update the thumb image view's top constraint constant. You're actually almost done with the touch tracking implementation.

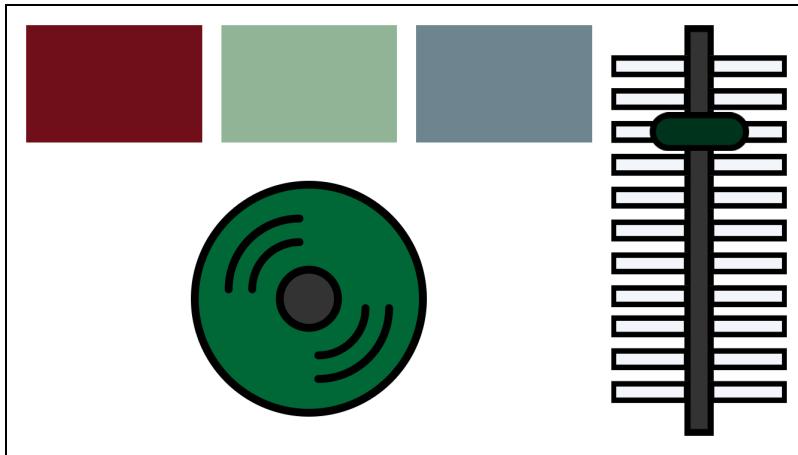
To handle the touch tracking's end state, add the following method override:

```
override func endTracking(
    _ touch: UITouch?,
    with event: UIEvent?
```

```
) {  
    super.endTracking(touch, with: event)  
    thumbImageView.isHighlighted = false  
}
```

Here, you reset the thumb image view's highlight state to `false`. Also, the image on the thumb image view will adjust to the unhighlighted version.

Build and run.



Now, you'll be able to:

- Touch and move the thumb image view.
- See the thumb image view's highlight state changes as you touch, continue to touch and release your finger from the screen.
- The thumb image view's y position and the control's value are constraints to define the upper and lower bounds.
- See the control's value change in the console log as you slide the thumb.

In the next section, you'll implement an exciting and critical custom control feature: Accessibility.

## Implementing accessibility features.

Especially on Apple's platform, iOS users are accustomed to and expect Accessibility support, so your custom control should support Accessibility features. By implementing Accessibility features, you make your app usable by a larger audience.

This section focuses on the implementation of Accessibility features and assumes that you have familiarity with using VoiceOver.

**Note:** If you're unfamiliar with using VoiceOver, check out <https://www.raywenderlich.com/6827616-ios-accessibility-getting-started> to get you on the right track.

## Setting up the basics

At the moment, you won't even be able to select the custom control when using the app in assistive mode. Making the control selectable is one thing, but you'll also need to make the control's purpose clear. The assistive users shouldn't need to spend time figuring out what to do or how to use your custom control.

First, add the following code to `PitchControl`:

```
private func setupAccessibilityElements() {  
    // 1  
    isAccessibilityElement = true  
    // 2  
    accessibilityLabel = "Pitch"  
    // 3  
    accessibilityTraits = [.adjustable]  
    // 4  
    accessibilityHint = "Adjust pitch"  
}
```

With this code, you:

1. Make the custom control visible to assistive apps.
2. Return a text that succinctly describes the control.
3. Set the characteristic of the control to `adjustable`. As the property name suggests, this indicates that the slider control can be adjusted. To fully support an adjustable control, you'll need to implement `accessibilityIncrement()` and `accessibilityDecrement()`. You'll do this later.
4. Indicate the result of performing an action on the control.

Now, add a call to this method at the end of `commonInit()`:

```
setupAccessibilityElements()
```

Build and run. Turn on VoiceOver. At this point, you'll be able to select the custom control, and you'll hear the announcer announce the control's accessibility label, characteristic and hint. Great, you've made the custom control available and clear to assistive users. However, your users have no way of controlling the value of the control. That's a problem you need to fix!

## Implementing value adjustability

Implementing value adjustability isn't an easy task. When a VoiceOver user wants to change the value of your control, you'll need to think about a value increment/decrement that fits your user's criteria.

You first need to consider how the value increment/decrement works. It shouldn't be too large where users can't choose a specific value. At the same time, it can't be too small where it'll take too much of your user's time to get the right value. It's a give-and-take scenario, and the more of these you make, the better you'll get at finding the right balance.

Add the following property to PitchControl:

```
private let valueIncrement: CGFloat = 1
```

You've defined the increment/decrement value for the pitch control. When a user adjusts the pitch control's value, it'll either increase/decrease value by one. One is a sweet spot as it lets the user change pitch quickly to reach the approximate desired value. At the same time, it allows for a broad enough selection of significant pitch values.

OK, time to get going with making your control adjustable. When an assistive user changes the value of your control, you'll need the announcer to announce the control's value accordingly. Add the following code:

```
override var accessibilityValue: String? {
    get {
        return "\\(Int(value))"
    }
    set {
        super.accessibilityValue = newValue
    }
}
```

This code overrides the accessibility value property and sets what the assistive technology announcer will announce each time the control's value changes. In this case, you're using a value that's closest to a whole integer number. Why? Because it can be difficult for your user when the announcer announces a number such as 7.1284769201. Also, the user likely doesn't need or care to know about the decimal places.

Now, add the following value type to the bottom of the file:

```
fileprivate enum Direction {
    case up
    case down
}
```

You'll use the value type added to communicate the user's swipe gesture direction from the accessibility's increment/decrement action.

Next, add the following method to PitchControl:

```
private func slideThumbInDirection(_ direction: Direction) {
    // 1
    let valueChange: CGFloat
    switch direction {
        case .up:
            valueChange = valueIncrement
        case .down:
            valueChange = valueIncrement * -1
    }
    // 2
    let newValue = value + valueChange
    if newValue < minValue {
        value = minValue
    } else if newValue > maxValue {
        value = maxValue
    } else {
        value = newValue
    }
    // 3
    thumbImageViewTopConstraint.constant =
        value * distancePerUnit
}
```

Here's how the code works:

1. Define the change in value in the positive or negative direction, depending on the swipe direction.
2. With the new value deriving from current value and the change in value, bound the new value to the control's value bounds.

3. Move the thumb view vertically using the latest control's value and the calculated distance for each control's value unit.

Finally, to put the accessibility adjustable and value control logic in place, add the following methods:

```
override func accessibilityIncrement() {
    super.accessibilityIncrement()
    slideThumbInDirection(.down)
}

override func accessibilityDecrement() {
    super.accessibilityDecrement()
    slideThumbInDirection(.up)
}
```

With the accessibility's increment/decrement methods, you pass in the swipe direction into `slideThumbInDirection(_:)`. The logic you've previously implemented will take care of the intended features here.

Build and run on a physical device. With VoiceOver switched on and your pitch control selected, you can adjust the custom control's value as you swipe up or down. The thumb view will move according to the value change from the swipes. Also, the control's value is limited within the bounds you've set, and the user interface reflects it as well.

## Key points

- Custom controls allow you to create interactive user interfaces to your app's specifications.
- Creating custom controls is fun; however, standard controls are intuitive to iOS users and support out-the-box application features, so use standard controls over custom controls whenever possible.
- Custom controls aren't complete without adopting accessibility features.



# Conclusion

Throughout this book, you've learned the techniques you need to build awesome, adaptable, accessible and flexible user interfaces using Auto Layout. We encourage you to use these techniques in your own projects and hope this book provides a useful reference to Auto Layout as you do so.

If you have any questions or comments as you work through this book, please stop by our forums at <http://forums.raywenderlich.com> and look for the particular forum category for this book.

Thank you again for purchasing this book. Your continued support is what makes the tutorials, books, videos, conferences and other things we do at raywenderlich.com possible, and we truly appreciate it!

— Jayven, Libranner, Jerry, Tammy and Richard

The *Auto Layout by Tutorials* team

