Develop in Swift

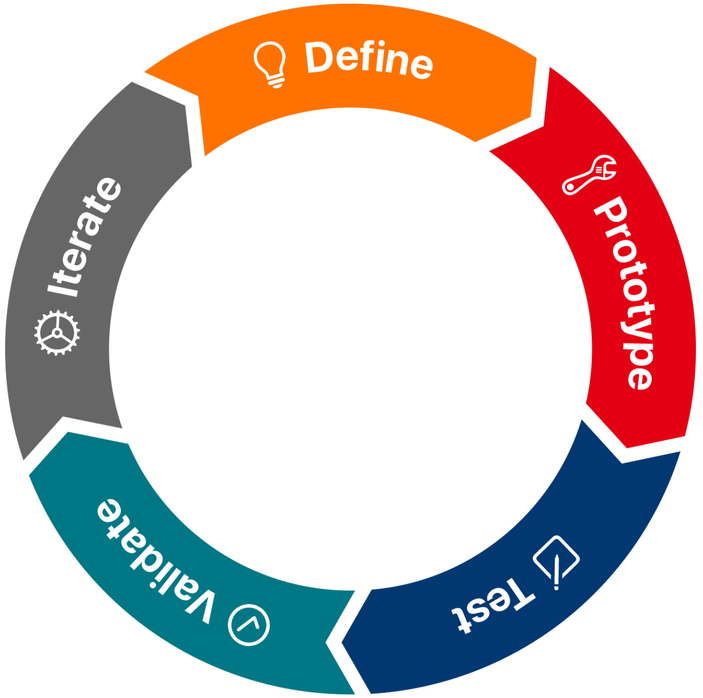
App Design Workbook

Welcome

The App Design Workbook guides you through the app design cycle to help you bring your iOS app ideas to life. You'll define, prototype, test, validate and iterate on your design as you relate your design concept to the Swift code that powers iOS apps.

App designers spend a lot of time getting the look and feel of their app just right. But that's just one part of a much longer process. And design isn't linear; the best designs are refined and improved over time. Good app design begins with understanding the user, and extends to every decision you make, both big and small.

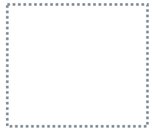
Behind every great app is an individual or team that started with an idea and a commitment to improve and refine it, step by step. Get ready — you're about to take the first step in a rewarding journey.



App Design Cycle

How to Use This Workbook

This workbook is designed so that you can customise it as you go. The result will be unique to you and your app, and it will reflect all the decisions you make along the way. You can add notes and highlighting to exercises and bring in your own images and other resources.



Templates

Some slides include templates that you'll fill out. Make as many copies as you need.

Prototype exercises

Slides marked with the app prototype icon indicate that you'll work in a separate Keynote document to build your prototype.

Code explorations

Slides marked with the Swift Playgrounds app icon indicate that you'll dive into code in Swift Playgrounds.

What You'll Need

Swift Playgrounds

Swift Playgrounds is a revolutionary app for iPad and Mac that helps you learn and explore coding in Swift, the same powerful language used to create world-class apps for the App Store. You'll use Swift Playgrounds in optional coding explorations in this workbook, diving into code to learn concepts that relate directly to your app. Download Swift Playgrounds for Mac >

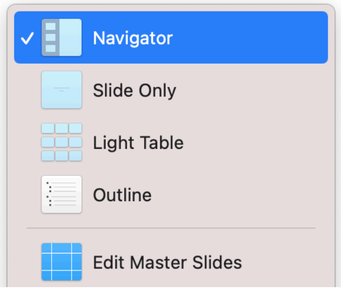
Go Green prototype

The examples in this workbook are based on Go Green — a demonstration app prototype in a Keynote file. To simulate the app, play the slideshow and click to advance through screens. Download the Go Green app prototype >

iOS Keynote Kit

You'll use a library of iOS interface elements to build your Keynote prototype. Download the iOS Keynote Kit >

Keynote Basics



Before you start, make sure you're familiar with the basics of navigating Keynote. To see the navigator, click the icon in the toolbar or the View menu, and choose Navigator. Groups of slides have a disclosure indicator for hidden (›) or visible (⌄) content. To show or hide slides in a group, click the disclosure indicator.

To move a slide, click and drag it in the navigator. To duplicate a slide, select it in the navigator, then choose Edit > Duplicate Selection or press Command (⌘)-D.

Fill out templates by editing text in boxes. To edit the text in a box, double-click it.

Add images to placeholders by dragging and dropping.

Remember: You can always undo mistakes by choosing Edit > Undo or by pressing Command (⌘)-Z.

App Design Cycle

Define

Discover
- Observe
- Explore Your Users
- Consider Diversity
- Summarise Your Audience

Analyse
- Analyse Causes
- Research Competitors
- Leverage Capabilities

Plan
- Find Differentiators
- Define Features
- Prioritise Features
- Describe Key Functions
- Define an MVP

Prototype

Map
- Outline Screens
- Group Screens
- Link Screens

Wireframe
- Create Tabs
- Add Navigation
- Create Modals
- Add Interface Elements

Refine
- Tap Targets
- Content Insets
- Weight and Balance
- Alignment

Style
- Personality
- Icon

Test

Architect
- Define Tests
- Create User Journeys
- Define a Process
- Plan an Introduction

Script
- Outline Scripts
- Write Scripts
- Anticipate Errors

Prepare
- Gather Users
- Last Check

Validate

Validate
- Gather Notes
- Form Key Insights
- Draw Conclusions

Iterate

Iterate
- Organise Your Conclusions
- Go Back: Define
- Go Back: Prototype
- Go Back: Test

Define

Your journey starts by defining your app. An initial discovery process will help you identify the challenge you want to solve and understand your audience. Then you'll analyse how an app can tackle the challenge before building towards a list of goals, features and key functions.

Observe
Explore Your Users
Consider Diversity
Summarise Your Audience

Discover

You'll begin by identifying a challenge and the people it affects. By the end of this stage, you'll have a thorough understanding of a challenge, and an insight into the people who would benefit from a solution.

Be observant and keep an open mind. The questions you ask — and the scenarios and points of view you imagine — will determine the direction of your app and its ultimate success.

Even if you already have an idea for your app, the following exercises can help you validate your current thinking.

Observe

Spark your imagination. People who create great apps are often motivated by addressing a challenge that they or their community experience.

Create as many copies of this slide as necessary to capture your thoughts. Don't try to filter them too much! You never know which one will lead you to a great app idea.

Are there questions you or others think about often?

What challenges do you or others face in your daily lives?

Have you or others used workarounds in existing apps?

Are there apps that partially address a challenge, but that require you to use them in unintended ways or augment their capabilities using other apps or activities?

Explore Code

In this exercise, you'll:

- Write your first line of code.
- Print a message to the console.

Hello, World!

You're using the App Design Workbook because you have an idea that you want to turn into an app. But design isn't the whole story — every app is built with code. The design and code of an app are related. Maybe more closely than you might imagine.

If you're new to coding, it might seem mysterious and complex. While it takes some time to learn the skills necessary to build an app from top to bottom, the basic concepts and practices are easy to understand. The perspective you get from understanding even a little of the code behind an app will give you an advantage in the design process.

In this and future exercises, you'll discover how Swift code — the same language used by professional app developers around the world — powers the features of an app.

One last thing: don't worry about mistakes. All coders get stuck — from the newest beginner to the most seasoned expert. You won't need any of the code from these exercises, so use them to play, explore and get curious.

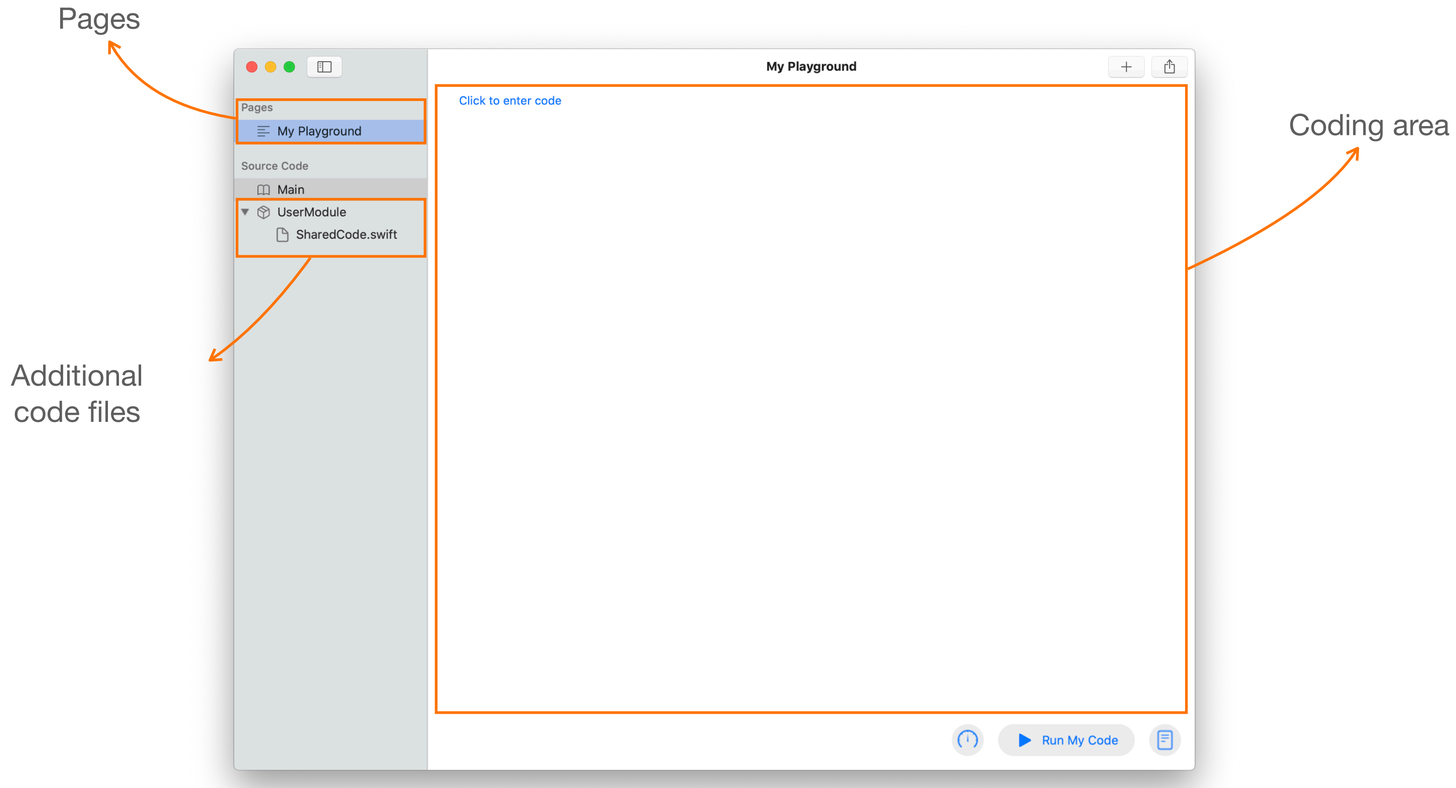
Get ready to write your first lines of Swift!

Hello, World!

Download the Swift Playgrounds app and create a new playground.

To get started, download the Swift Playgrounds app from the App Store. It's available for both Mac and iPad. Instructions in all Explore Code exercises are for Mac.

Open the app. To make a new playground, find the More Playgrounds section at the bottom of the screen and choose Blank, or choose File > New Blank Playground. A playground starts with one page named "My Playground". You'll add more pages in coming code explorations. Double-click your new My Playground to open it. You'll see the window below:



Hello, World!

Add a line of code that will display text in the console.

Look for the blue text that says “Click to enter code” in the coding area, and enter the following Swift code:

```
print("Explore Code!")
```

Run your code by clicking the Run My Code button in the lower right.

This code produces a message in the console. Notice that a red badge has appeared on the button in the lower right.

That’s the console button. Click it to display the output of your program to the right of your code.

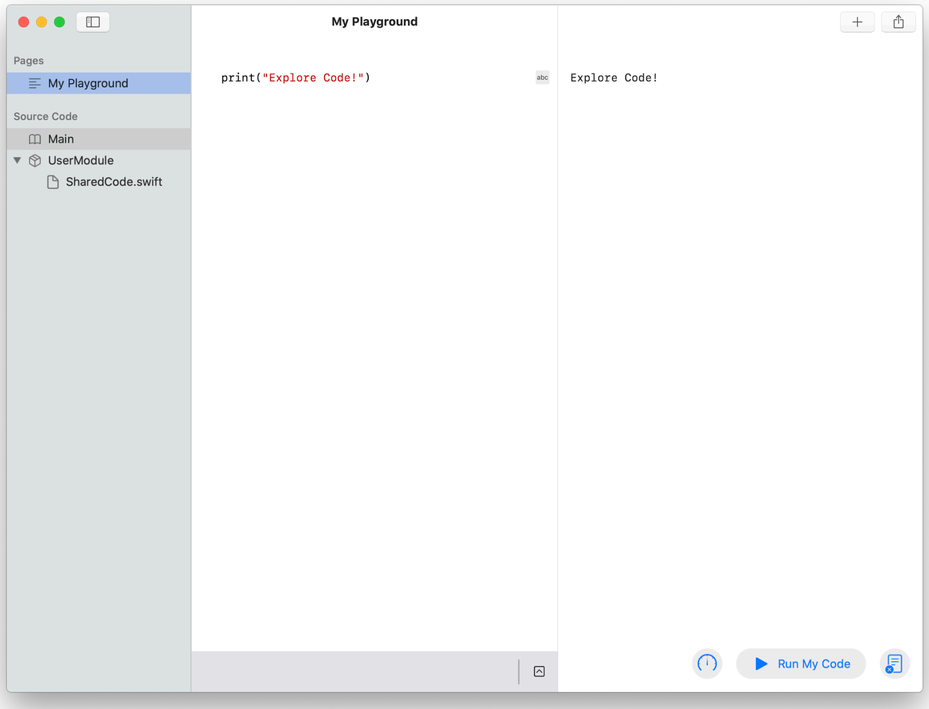
```
print("Explore Code!")
```

```
Explore Code!
```

Hello, World!

Review your work and try a challenge.

Your playground page should now look like this:



Congratulations! You've just completed your first code exploration.

Challenge: Programmers who are first learning a new language usually write a simple program that produces just the message "Hello, World!" You're now ready to do this in Swift. Give it a try and join the ranks.

Hints: Replace the characters in between the two quotation marks. Leaving matching quotes at the beginning and the end is very important. Did you remember to click Run My Code again?

Explore Your Users

Pick one of the challenges and gather information about individuals who experience it. Each person is different. It's important to think broadly to capture as much diversity as you can.

Good design is user-centred. You've had a good start by thinking about the challenges that you and others face. Keep it going! By narrowing down from the general to the specific, you'll place individual people at the core of your process.

Personal stories from real people can give you perspective you might not otherwise have. Consider interviewing people from your community to create authentic profiles.

Example

Who is this person? How do they describe themselves?

A father of two, kindergarten teacher, taking online classes in photography.

How old are they?

31

What are important aspects of their environment?

Live in an apartment on the third floor. Not enough space for large bins in the house, so they only have a small recycling container.

How do they describe the challenge they face?

I don't really understand what's recyclable and what belongs in the rubbish. The labels are hard to find, and I don't really how to distinguish between things like different kinds of paper.

What do they most want in a solution? How would it make their lives easier?

I need help quickly identifying what's recyclable. If I could sort through items quickly every evening, I'd be more likely to spend mental energy on it, since my kids deserve as much energy as I can give them.

In which specific circumstances might they use an app that addresses their challenge?

I could spend a little time every evening with my kids sorting through our daily waste.

Explore Your Users

Pick one of the challenges and gather information about individuals who experience it. Each person is different. It's important to think broadly to capture as much diversity as you can.

Good design is user-centred. You've had a good start by thinking about the challenges that you and others face. Keep it going! By narrowing down from the general to the specific, you'll place individual people at the core of your process.

Personal stories from real people can give you perspective you might not otherwise have. Consider interviewing people from your community to create authentic profiles.

Who is this person? How do they describe themselves?

How old are they?

What are important aspects of their environment?

How do they describe the challenge they face?

What do they most want in a solution? How would it make their lives easier?

In which specific circumstances might they use an app that addresses their challenge?

Consider Diversity

Identify things about your users you may have overlooked.

A user's identity and circumstances will have a huge impact on how they'll experience and use an app. Summarise all your users with these different aspects in mind.

Everyone has biases that affect the way they perceive the world. Compensate for your biases so that they don't creep into your app's design.

Did you identify something that you didn't consider when imagining your audience? For example, were all your users a similar age? Consider going back to the earlier exercises with your new insights in mind.

Ages

Genders

Languages

Disabilities

Cultures

Economic circumstances

Living situations

Summarise Your Audience

Summarise your findings about individual users. Refer to your earlier research and use it to draw some conclusions.

Example

What's the most important concern in a solution?

Understanding the percentage of rubbish vs recycling.

The age range of the users is:

15 to 30

Our app will be opened when …

Throwing things in the rubbish or recycling.

Our app will be used in this environment:

Inside, with a connection to Wi-Fi or mobile reception.

Our environment will have these limitations:

Users may have their hands full.

When designing our app, we need to consider:

Users might not know what qualifies as a recyclable.

Summarise Your Audience

Summarise your findings about individual users. Refer to your earlier research and use it to draw some conclusions.

What's the most important concern in a solution?

The age range of the users is:

Our app will be opened when …

Our app will be used in this environment:

Our environment will have these limitations:

When designing our app, we need to consider:

Analyse Causes
Research Competitors
Leverage Capabilities

Analyse

Now that you've identified who your app serves and the challenges they face, it's time to get specific. By the end of this stage, you'll have a clearer picture of the form your app might take.

You'll look at the root causes of your users' challenges. And you'll use them to drive feature ideas that take advantage of key iOS capabilities, while contrasting your ideas with existing apps.

Analyse Causes

Dig deeper into the issues you've observed and find the core problem. Then consider how your app could solve it.

Asking why something happens will help you discover hidden causes behind what you observe directly. The deeper you dig, the closer you'll get to the core motivating need for your solution.

Create as many copies of the following template as you need to describe the problems you've identified in your research.

Example

Users are having this problem:

1. They'd like to begin recycling but struggle to hold themselves accountable.

This happens because:

2. It's easier to throw everything into one bin.

This is because:

3. They struggle to differentiate between rubbish and recycling.

Which is because:

4. Recycling seems complicated and hard to remember.

And the result is this core problem:

5. People are rarely taught how to recycle.

We can solve this issue in our app by:

Educating people on what qualifies as recycling, and gamifying the experience so they can hold themselves accountable with their peers.

Analyse Causes

Dig deeper into the issues you've observed and find the core problem. Then consider how your app could solve it.

Asking why something happens will help you discover hidden causes behind what you observe directly. The deeper you dig, the closer you'll get to the core motivating need for your solution.

Create as many copies of the following template as you need to describe the problems you've identified in your research.

1. Users are having this problem:

↓

2. This happens because:

↓

3. This is because:

↓

4. Which is because:

↓

5. And the result is this core problem:

We can solve this issue in our app by:

Research Competitors

Find and describe apps that relate to the problem you've identified.

Discover what people are currently using to solve the problem. Search the App Store for similar apps to find out what users enjoy or dislike about their solution. This will give you insight into what your app will be competing with.

	This app is interesting because:	I like/dislike this app because:

Leverage Capabilities

Note iOS capabilities that you might use in your app.

iOS comes with an array of great technologies for addressing how users want to interact with an app. You'll see many listed here, but keep in mind that there are many more.

What features do your competitors have in common that you might need to use in your app? Which features might be game changers?

Visit the **iOS Human Interface Guidelines site** and look at User Interaction and System Capabilities to learn more.

Map

Display interactive maps that locate the user, provide directions, indicate points of interest, display satellite images and more.

Near field communication

Detect when your device is near a sensor to interact with payment systems and more.

Augmented reality

Place virtual objects in the world that users can see and interact with on screen.

Image processing

Use sophisticated algorithms to adjust images and apply filters.

Speech recognition

Convert spoken audio into text.

Haptics

Provide feedback through touch by vibrating the device.

Machine learning

Use sophisticated algorithms to analyse and categorise visual, auditory, textual and other forms of information.

Camera

Use the built-in front and back cameras and their powerful processing capabilities.

Bluetooth®

Communicate wirelessly with nearby devices using a standard, secure, low-power interface.

GPS

Locate the device anywhere in the world and look up corresponding information, such as country and city.

Context menus

Provide quick access to actions for an onscreen object.

Microphone and speakers

Capture and play back high-fidelity stereo audio.

Drag and drop

Move items by pressing and dragging.

Accelerometer and gyroscope

Track the device's orientation and movement.

Widgets

Display information related to your app on the Home Screen, in a variety of sizes and styles.

Notifications

Provide updates to the user on the Lock Screen when they're not using your app.

Explore Code

In this exercise, you'll:

- Create a new page.
- Import a framework.
- Learn about the live view.
- Display an interactive map.

Map

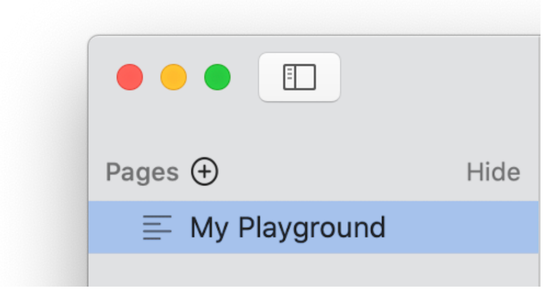
A major component of coding is recognising what work has already been done for you and figuring out how to use it. The many advanced built-in capabilities of iOS are organised in frameworks.

In this exercise, you'll display an interactive map using an iOS framework. To start, be sure "My Playground" is open in the Swift Playgrounds app.

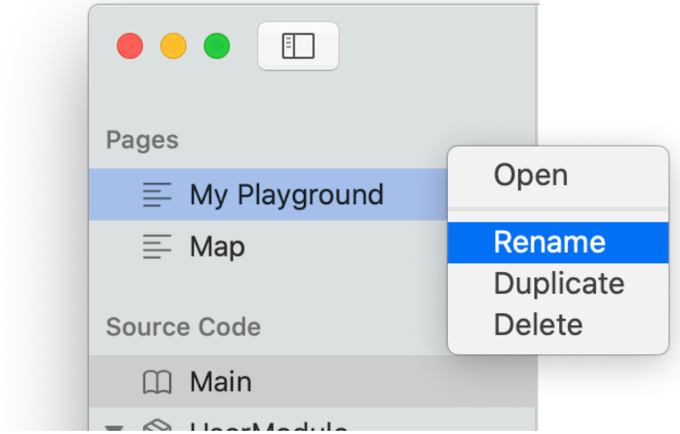
Map

Create a new page and add code to create a map.

If it’s not showing, open the sidebar by clicking . To create a new page, hover your mouse next to “Pages” and click the ⊕ button, or choose **File > New Page.**



Change the page title to “Map”. While you’re editing page names, right-click the “My Playground” page, click Rename, then type “Hello World” to give it a more descriptive name.



In the editor, type the following Swift code:

```
MKMapView()
```

You’ll get a red dot, indicating an error:

- `MKMapView()`

Map

Import the MapKit framework.

This is the right code to create a map, but `MKMapView` is in a specialised framework. You'll need to import that framework to use it in your code.

Above your first line of code, enter:

```
import MapKit
```

Your code should now look like this:

```
import MapKit
MKMapView()
```

Swift uses different font styles and colours (syntax highlighting) to help you read your code.

`import` is a Swift keyword. Keywords have special meaning in Swift that sets them apart from the rest of your code.

Now that you've imported the `MapKit` framework, the error will go away. Click Run My Code.

The code runs, but you won't see a map. To view it, you'll need to do one more thing.

Map

Import the PlaygroundSupport framework and display the map.

You've created a map, but displaying it requires another framework. The `PlaygroundSupport` framework lets you control aspects of the playground itself. Enter one more line of code above the first line:

```
import PlaygroundSupport
```

Then add the highlighted code to the beginning of the final line:

```
PlaygroundPage.current.liveView = MKMapView()
```

Be sure to include a space both before and after the equal sign. Your completed playground should now read:

```
import PlaygroundSupport
import MapKit
PlaygroundPage.current.liveView = MKMapView()
```

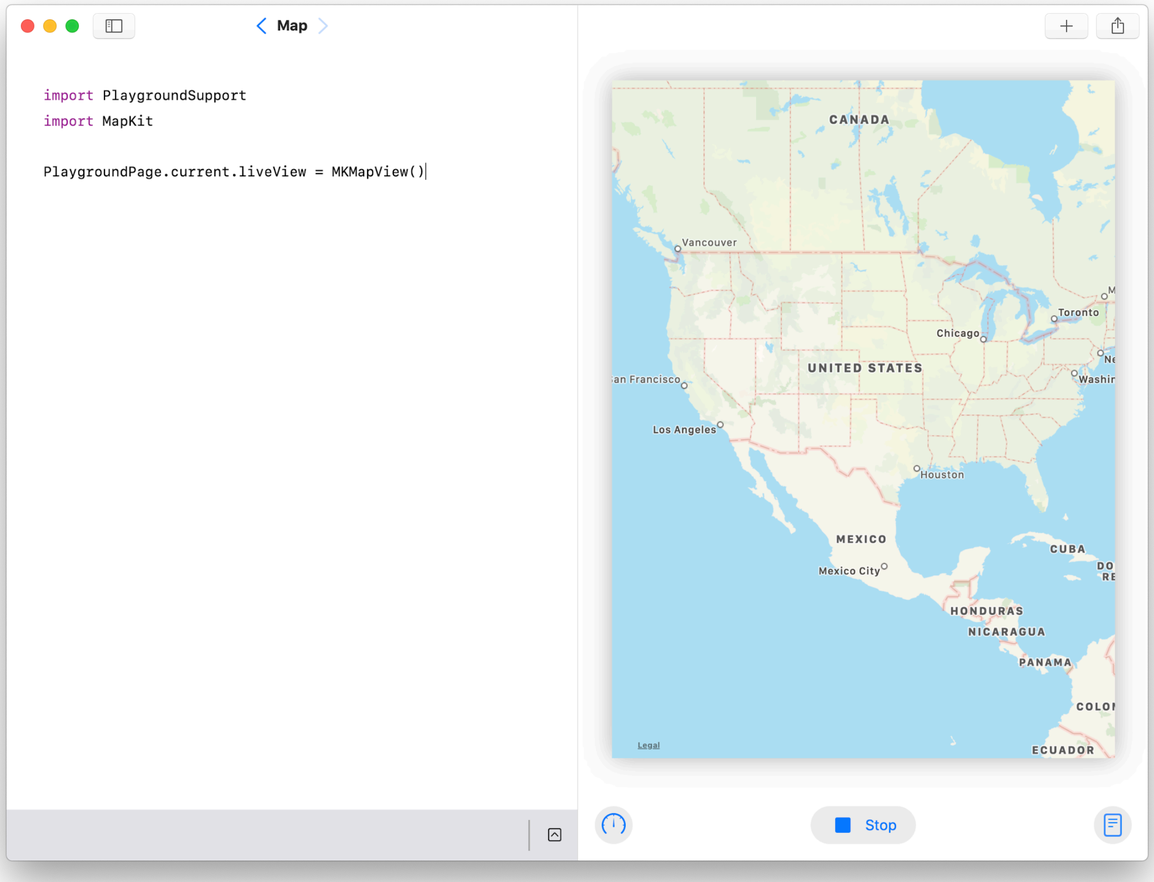
Now run your code.

The live view in the playground opens to display the map you created. You can interact with it as you would any app that embeds a map. Try clicking and dragging to display different regions of the map. If you're using a trackpad, use two-finger gestures to zoom.

Map

Review your work.

Your completed exercise should now look like this:



This is a simple example of the power of Swift frameworks. You get a large body of familiar functionality just by importing the ones you want. In fact, other than the import statements themselves, it took just one line of code to create the map and show it in the live view!

Find Differentiators
Define Features
Prioritise Features
Describe Key Functions
Define an MVP

Plan

You’re ready to put together a concrete plan for your app. By the end of this stage, you’ll have a concise, well-defined plan that you can begin building into a prototype.

You’ll build your plan by identifying key differentiators, setting goals, and then narrowing down your feature set to exactly those you’ll need to test whether your app will impact real users.

Find Differentiators

Create a list of ways your app will be different from existing apps.

Your app's differentiators set it apart from others in the market.

A differentiator is a core feature of your app. The following things are not differentiators:

- Style, such as colour, fonts, icons and images.
- Arrangement of onscreen items.
- How screens are organised.

Our app will be different from these apps by:

Explore Code

In this exercise, you'll:

- Learn how to separate models and views.
- Create variables.
- Discover another way to display text.

Model and View

As a designer, your focus is largely on the visible interface and usability of your app. Developers use the term 'view' to describe the parts of an app a user sees and interacts with. The model of an app defines its data. It's a companion to the app's views, and equally important.

Developers separate views from models because they're independent. A view might display different model data at different times, and the same model data might appear in multiple views.

In this exercise, you'll create model data and display it in two different views.

To start, be sure "My Playground" is open in the Swift Playgrounds app. Then create a new page and name it "Model and View".

Model and View

Display text data in the console.

Enter the following line of Swift code:

```
"Explore Code!"
```

This text value is a model. If you run your code now, the text will be created but you won't see anything. But you've already displayed data using `print()`. The console is a kind of view, and the `print()` command adds a line of text to it.

Add the following line of code, being sure to keep the first line; you'll come back to it.

```
print("Explore Code!")
```

Run your code, and open the console to see the message.

Model and View

Display text data in the live view.

The console is useful for developers to examine data in their app, but it's not visible to the user. To display text on a user's device, you'll need another kind of view. Insert the following code at the beginning of your playground:

```
import SwiftUI
```

Now add the following line at the end of your playground to create a view to display the string:

```
var textView = Text("Explore Code!")
```

`Text` is a kind of view that can display text data, which you provide in parentheses.

`textView` is a variable declared using the keyword `var`. You use variables in Swift to refer to things by name.

The equal sign associates the value on the right with the variable on the left. This is called assignment to a variable.

You've created a new view for your text. Now it's time to display it using the live view. Add two more lines of code as highlighted below:

```
import PlaygroundSupport
import SwiftUI

"Explore Code!"
print("Explore Code!")
var textView = Text("Explore Code!")
PlaygroundPage.current.setLiveView(textView)
```

Adds playground support.

Sets the `Text` view as the current page's live view.

Run My Code

Now run your code to see your text in the live view.

Model and View

Create a model using a variable.

Notice that you've entered "Explore Code!" as a text value three times. It's not just repetitive — a typo in one value would break your intention to make all three represent the same model data.

You've just learned to create a variable to refer to something by name. You can use the same technique here with your text data. Update the line where you first created the value `"Explore Code!"` by assigning it to a new variable.

```
var message = "Explore Code!"
```

Now you can use the `message` variable anywhere you want to refer to this text. Replace the remaining instances of `"Explore Code!"` with `message`. Your completed code should now look like this:

```
import PlaygroundSupport
import SwiftUI

var message = "Explore Code!"
print(message)
var textView = Text(message)
PlaygroundPage.current.setLiveView(textView)
```

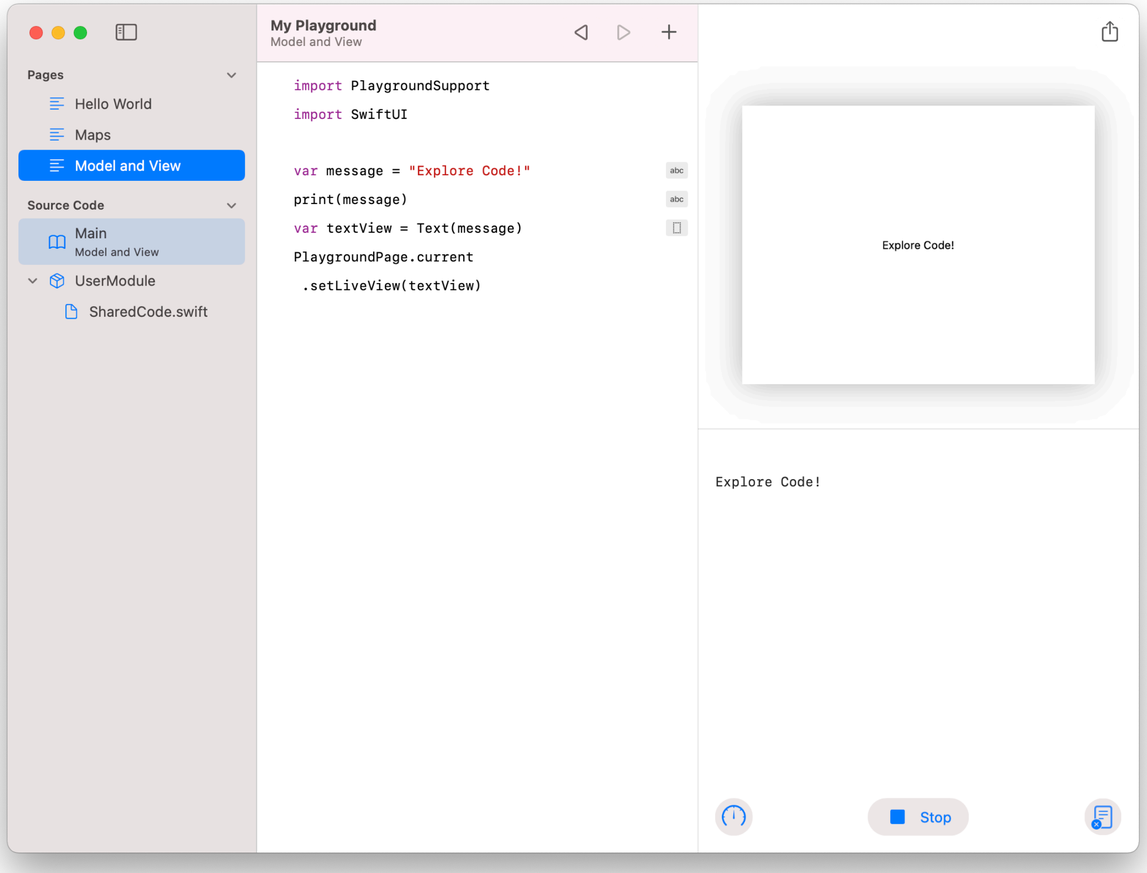
Run your code. As before, you should see this text value displayed in both the live view and the console.

You've taken the final step to separate your model from its views. The variable `message` stores the model text `"Explore Code!"`, and you've displayed it in two different views.

Model and View

Review your work and try a challenge.

Your completed exercise should now look like this:



A major part of developing an app is defining and creating the models and views that power it. By separating the two concepts, you can focus on them separately. Consider the kinds of models and views your app might have as you continue designing.

Challenge: Experiment with changing the text you assigned to `message` and observe the result.

Explore Code

Define Features

Summarise the baseline and differentiating features of your app.

Establishing clear, achievable goals helps you focus on the problems you're trying to solve, and the way you're hoping to solve them. All of your previous research will help you define the basic features and differentiators of your app.

Example

Current

recycling apps

have focused primarily on

what is considered recycling.

As a baseline, our app will need to:

Log rubbish and recycling.

Scan product barcodes to determine recyclability.

Educate about what is recyclable.

Provide quick links to environmental causes.

Shop for environmentally friendly products that reduce rubbish.

Breakdown of rubbish to recycling.

Our app will be different by:

Encouraging recycling through gamification.

Communicating with smart bins to calculate weight automatically.

Creating challenges to spark ideas.

Letting users shop for recycling supplies like bins, bags and magnets.

Define Features

Summarise the baseline and differentiating features of your app.

Establishing clear, achievable goals helps you focus on the problems you're trying to solve, and the way you're hoping to solve them. All of your previous research will help you define the basic features and differentiators of your app.

Current

have focused primarily on

As a baseline, our app will need to:

Our app will be different by:

Explore Code

In this exercise, you'll:

- Learn about strings.
- Create a string with concatenation.
- Create a string with interpolation.

Strings

Just about every app displays text, and many allow the user to enter it using the keyboard. In Swift, the concept of text — and all that you can do with it — goes by the name 'string'. You've already used strings to create values such as "Explore Code!" In this exercise, you'll explore more of what you can do with strings.

To start, be sure "My Playground" is open in the Swift Playgrounds app. Then create a new page and name it "Strings".

Strings

Create a string with concatenation.

The simplest way to work with string values is to simply enter text between matching double quotes. Create two new strings by entering the following code:

```
"No Palm Oil Challenge"
"Pick Up Trash Challenge"
```

Note the common text, “Challenge”, in each string. Now imagine that your app might want to create many such strings on the fly. Swift allows you to compose strings from multiple parts. Start by defining a new variable below to store the unchanging text:

```
var challengeString = "Challenge"
```

Use another variable for the changing beginning of the string by entering the following code:

```
var name = "No Palm Oil"
```

You can combine these two strings using the “+” operator. This is called string concatenation. Enter the following two lines to create a new string and print it.

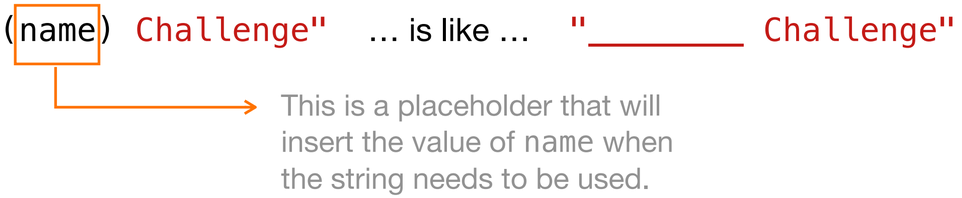
```
var noPalmOilChallenge = name + challengeString
print(noPalmOilChallenge)
```

Run your code, and open the console to see the message.

Strings

Create a string with interpolation.

Another way to compose strings is with string interpolation, which is similar to filling in the blanks in a sentence. Think of it as a placeholder that will insert the value of name when the string needs to be used.



Use string interpolation to produce the same “No Palm Oil Challenge” string by entering the following code:

```
var anotherNoPalmOilChallenge = "\(name) Challenge"
print(anotherNoPalmOilChallenge)
```

Notice that the constant part of the string is coloured red, while the interpolated part is black. This is another great example of syntax highlighting.

String interpolation is powerful. For example, it works with numbers. Enter the following code:

```
var pointsForCompletion = 8
print("You could score \(pointsForCompletion) points by completing it.")
```

A string can have any number of interpolated values. Enter the following code to substitute both the name of the challenge and the number of points in one longer string:

```
print("Complete the \(noPalmOilChallenge) to score \(pointsForCompletion)
points!")
```

Run My Code

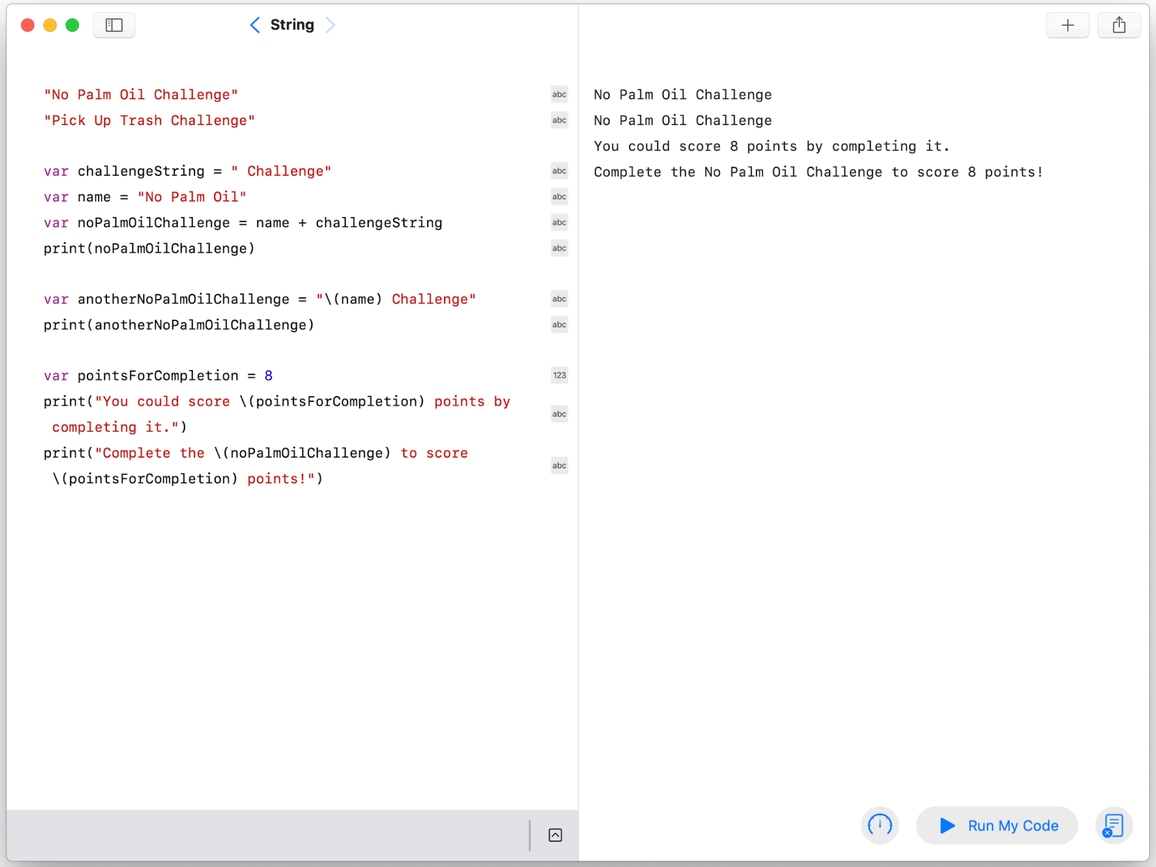
Run your code and review the Console output.

Explore Code

Strings

Review your work and try a challenge.

Your completed exercise should now look like this:



Challenge: Experiment with what you've learned about constructing strings. Try assigning your given name to `givenName` and your family name to `familyName`, then construct `fullName` using concatenation (+). Use interpolation to embed them in a string like so: "My full name is _____ _____." Can you add your age? Review the earlier explorations and try to display this using a text view.

Prioritise Features

Make a copy of the previous exercise. Annotate the most important features and order them by priority. Ask yourself which features are absolutely necessary. Can you pare your list down to three or four key features? Do you need to reorder your priorities?

Identifying the most important features of an app helps you work towards your minimum viable product (MVP). Your first iteration of the app should include just the features necessary to validate your idea.

To avoid ‘feature creep’, it’s important to distinguish between must-have features and those that would be great enhancements. Clear, minimal focus is essential to the success of the design process.

Example

Current

recycling apps

have focused primarily on

what is considered recycling.

As a baseline, our app will need to:

- Log rubbish and recycling. **1**
- Educate about what is recyclable. **2**
- Let users shop for environmentally friendly products that reduce rubbish. **7**
- Scan product barcodes to determine recyclability. **8**
- Provide quick links to environmental causes. **5**
- Provide a breakdown of rubbish to recycling. **6**

Our app will be different by:

- Encouraging recycling through gamification. **3**
- Creating challenges to spark ideas. **4**
- Communicating with smart bins to calculate weight automatically. **9**
- Letting users shop for recycling supplies like bins, bags and magnets. **10**

Describe Key Functions

For each priority feature, describe its key functions. Copy and paste this slide as many times as you need.

List information the user will see and actions that the user will take. Add notes to clarify your thoughts or ask questions that you'll want to come back to as you work on your design.

Don't think in terms of user interface (UI) yet. Elements such as buttons, tabs and icons are supported and defined by the data and actions in your app. The better you understand them, the better your design will fit your MVP features.

Example

For this feature:

Log rubbish and recycling

The user will need to be able to do and see these things:	Notes:
Add log entry	Select either weight or item
Describe items	How much they recycled on what day
Record-keeping of past items	Day, week, month
Choose a day that recycling was done	
Default to today's date	Might need a quick jump to 'today'
Browse past dates	
Share log entry to social media	

Describe Key Functions

For each priority feature, describe its key functions. Copy and paste this slide as many times as you need.

List information the user will see and actions that the user will take. Add notes to clarify your thoughts or ask questions that you'll want to come back to as you work on your design.

Don't think in terms of UI yet. Elements such as buttons, tabs and icons are supported and defined by the data and actions in your app. The better you understand them, the better your design will fit your MVP features.

For this feature:

The user will need to be able to do and see these things:	Notes:

Define an MVP

Go back to your Describe Key Functions slides, or duplicate them, and prioritise based on what is most important and crucial to implementing the feature.

You'll base your MVP on the features you identify here.

Remember that you're not your user. If you get stuck, think about what would most help the user achieve the goals you identified.

Example

For this feature:

Log rubbish and recycling

The user will need to be able to do and see these things:		Notes:
Add log entry	1	Select either weight or item
Describe items	2	How much they recycled on what day
Record-keeping of past items	3	Day, week, month
Choose a day that recycling was done	5	
Default to today's date	4	Might need a quick jump to 'today'
Browse past dates	6	
Share log entry to social media	7	

Prototype

It’s time to build a working Keynote prototype of your app. You’ll start by mapping screens to form an app architecture, then apply basic UI elements to create a wireframe. Then you’ll refine your prototype using common design guidelines to ensure that it meets an iOS user’s expectations. And finally, you’ll define the personality of your app with colour, icons and more.

Outline Screens
Group Screens
Link Screens

Map

An app map is a set of outlines that describe the information and functions on each screen — and how they relate to each other. By the end of this stage, you'll have a set of screen outlines with well-defined groupings and relationships.

By mapping your app, you define its architecture. iOS users have a set of expectations — a mental model — for the way an app should behave. They expect related information to be grouped together, and for activities to be easy to navigate. You'll derive your app's architecture from the key functions of the MVP, making decisions based on how you expect users to work with the app.

Outline Screens

Review the key functions of your MVP and organise them into screens, describing the content and actions of each.

The key functions you listed have natural relationships that suggest which ones belong together. Feel free to break up a function into more than one item in a screen outline, or combine several key functions into one item. Your notes will help you make these decisions.

Try to keep each screen focused on a single concept or activity, summarised in its title, without worrying about how many there are. Next you'll organise them in a way that users understand.

You'll work with these outlines in the map stage. During the wireframe stage, you'll translate your screen outlines into the text, images, controls, icons and other UI elements in your app.

Example

Day Summary

- Summary of today's recycling
- Itemised list of today's recycling
- View details for a logged item
- Add new items to rubbish or recycling

Item Detail

- Type of item
- Weight
- Date

Calendar View

- Calendar with current date
- Summary of the month's items
- View a day's recycling log

Add Item

- Select rubbish or recycling
- Select by weight or by item
- Input details: date, type, weight
- Recycling information
- Submit items
- Cancel entry

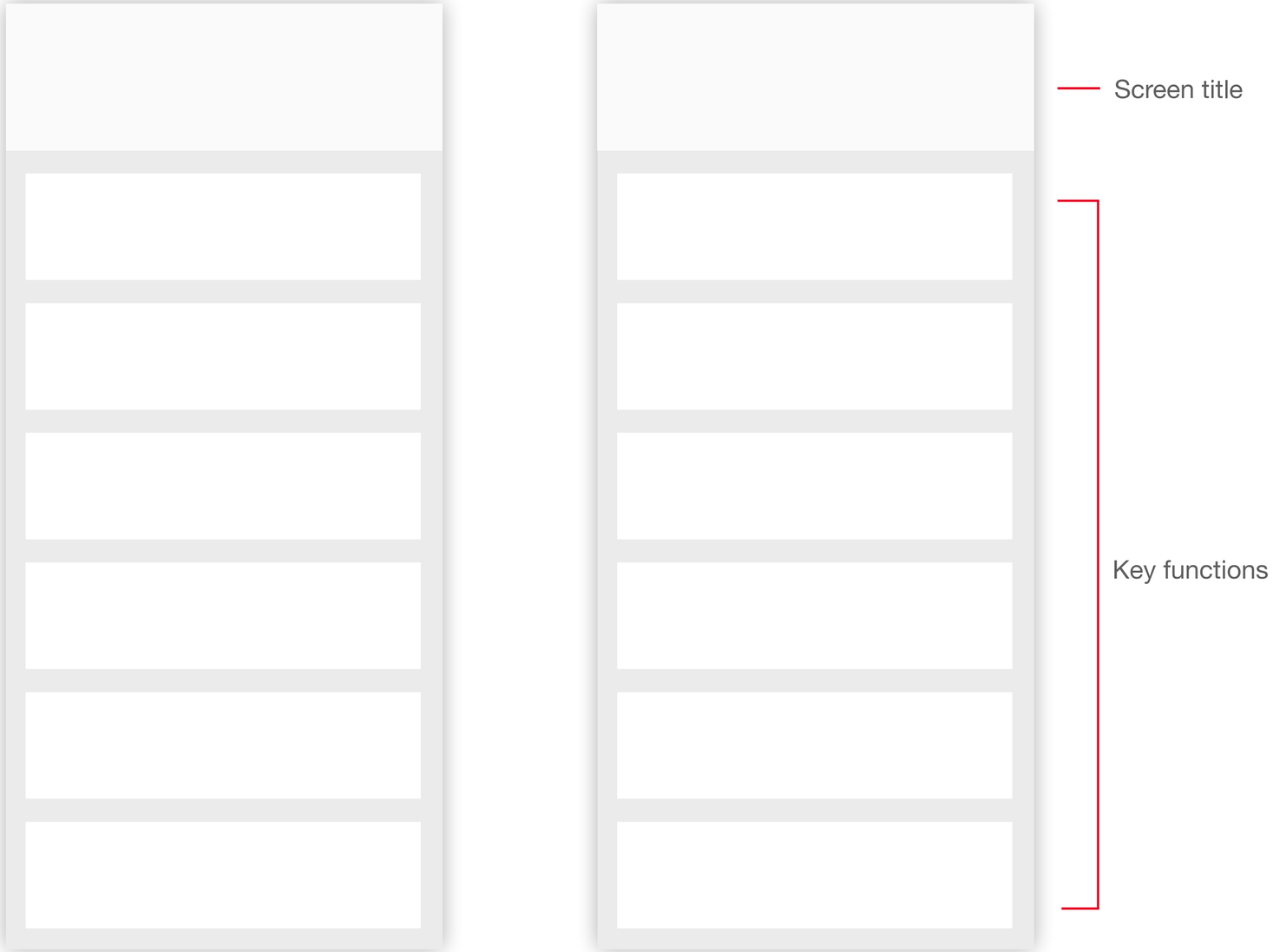
Outline Screens

Review the key functions of your MVP and organise them into screens, describing the content and actions of each.

The key functions you listed have natural relationships that suggest which ones belong together. Feel free to break up a function into more than one item in a screen outline, or combine several key functions into one item. Your notes will help you make these decisions.

Try to keep each screen focused on a single concept or activity, summarised in its title, without worrying about how many there are. Next you'll organise them in a way that users understand.

You'll work with these outlines in the map stage. During the wireframe stage, you'll translate your screen outlines into the text, images, controls, icons and other UI elements in your app.

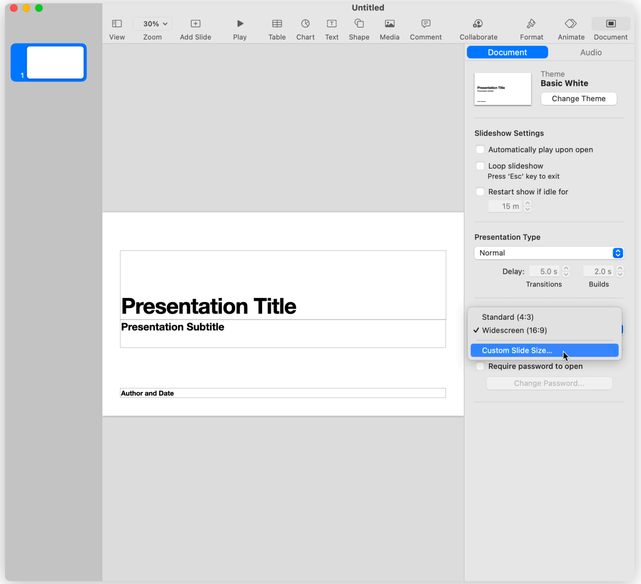


Prototype

Create a new Keynote file for your prototype and add a slide for each screen outline.

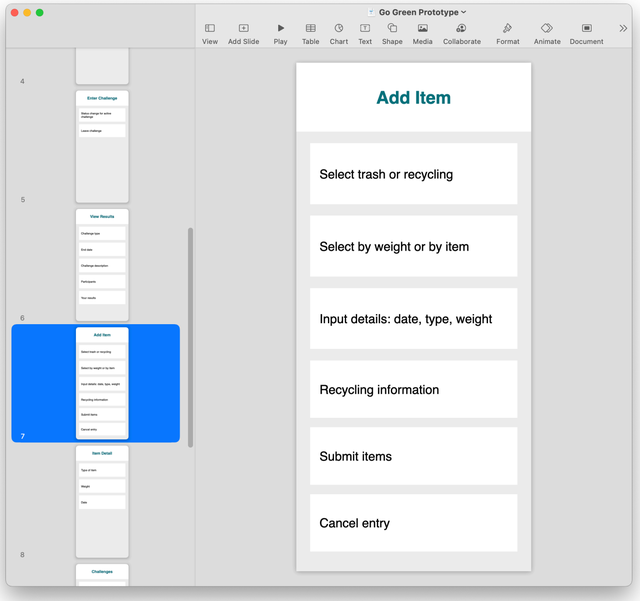
It’s time to begin creating your app prototype in a new Keynote document. Keynote is a great way to make a prototype that you can view on the device you’re testing on.

1. Create a new Keynote file.
2. Open the Document inspector or choose View > Inspector > Document.
3. Under Slide Size, choose Custom Slide Size.
4. Enter 375 for width and 812 for height. Click OK.
5. Open the Format inspector or choose View > Inspector > Format.
6. Under Background, select Colour Fill and choose light grey (or another background to contrast with white).
7. At the top of the inspector, use the Change Master button to select the Blank master slide.



Now create one slide per screen outline and copy your outlines to the slides.

To copy a screen outline, click and drag to select all of its boxes. Copy, then paste them to the blank slide in your prototype, and drag them into position.



In this exercise, you'll:

- Learn about structures.
- Learn about naming.
- Use code comments.

Data and Naming

Text isn't the only kind of data in an app. Swift can work with many other common kinds of values, such as numbers and dates. Different kinds of data are called types. All data in Swift belongs to a type — including ones you create to represent the information your app works with. In this exercise, you'll create your own customised type.

To start, be sure "My Playground" is open in the Swift Playgrounds app. Then create a new page and name it "Data and Naming".

Data and Naming

Define and name a structure.

To create your own data type in Swift, you'll use a structure. Each customised structure in your code should have a recognisable and understandable name — for example, `Challenge`, `RecyclingItem` and `Achievement`. By convention, the name of a type always begins with a capital letter. Spaces aren't allowed in type names; if you need more than one word to describe a type, use camel case to capitalise the first letter of each word.

Enter the following code to create a structure (denoted by the keyword `struct`). Be sure to include the curly braces, which you'll find near the P on your keyboard.

```
struct Challenge {
}
```

You can create a new challenge value (an instance) by typing its name followed by parentheses. Add the following code to create two instances of `Challenge` and assign each to an identifier:

```
var noStraws = Challenge()
var pickUpTrash = Challenge()
```

A structure instance is basically a value like any other. Try printing both to the console:

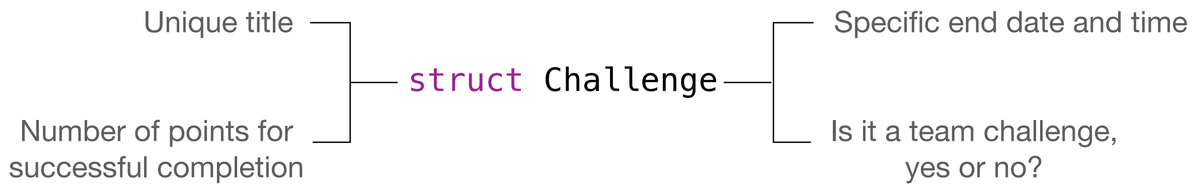
```
print(noStraws)
print(pickUpTrash)
```

You can see that while they're two different instances assigned to two different variables, there's nothing to distinguish one challenge from another.

Data and Naming

List the properties of a structure.

Obviously no two challenges are alike. Consider what distinguishes one challenge from another. Which properties will the `Challenge` type need to support the app's functionality? Making a list or diagram can help:



The next exploration will show how to add properties inside the definition of a type. But for now, you can use comments in your code to plan. Swift will ignore anything you type from after two forward slashes (`//`) all the way to the end of the line. Programmers use comments to take notes and describe their code.

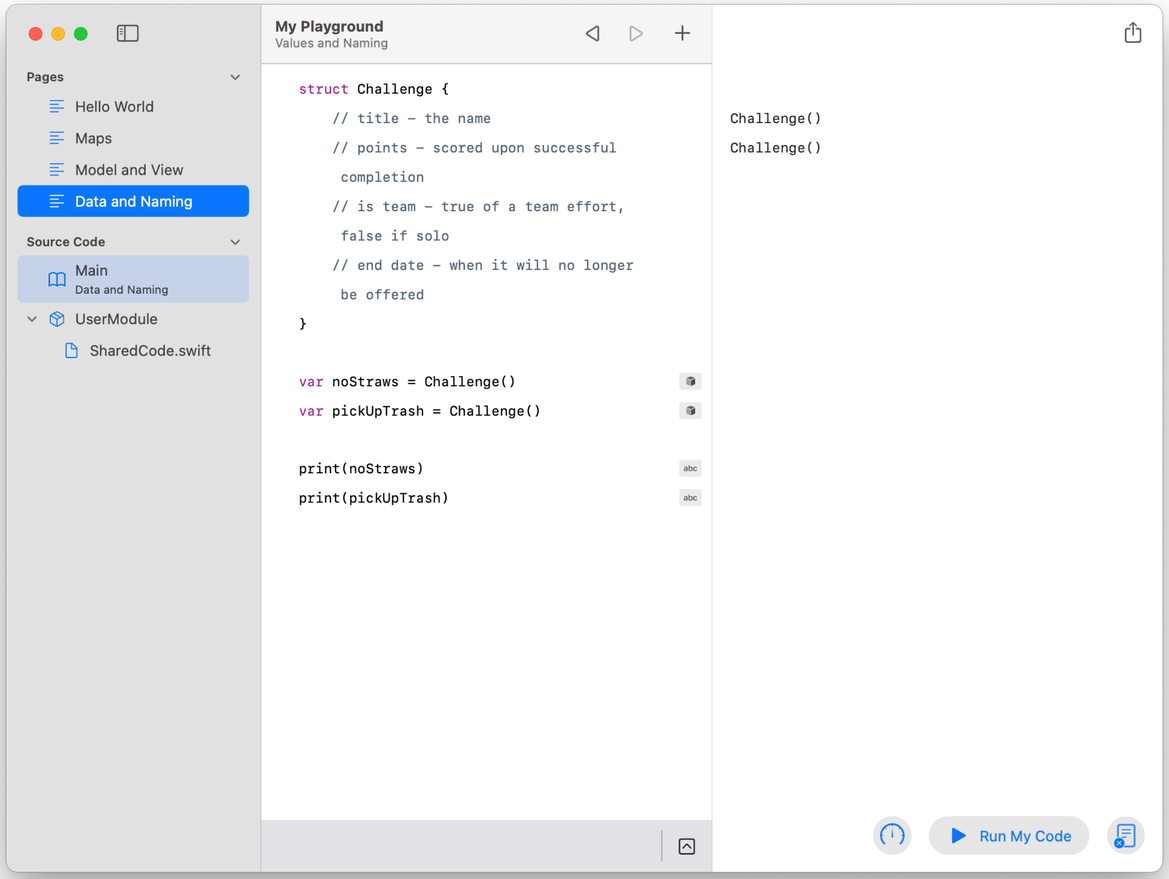
Enter some comments as placeholders for the `Challenge` structure's properties between curly braces. You can replace them with code later.

```
struct Challenge {
    // title - the name
    // points - scored upon successful completion
    // is team - true of a team effort, false if solo
    // end date - when it will no longer be offered
}
```

Data and Naming

Review your work.

Your completed exercise should now look like this:



Challenge: Think about the kinds of data you'll need in your own app. Create a new structure for each type, and use comments to describe their properties.

Group Screens

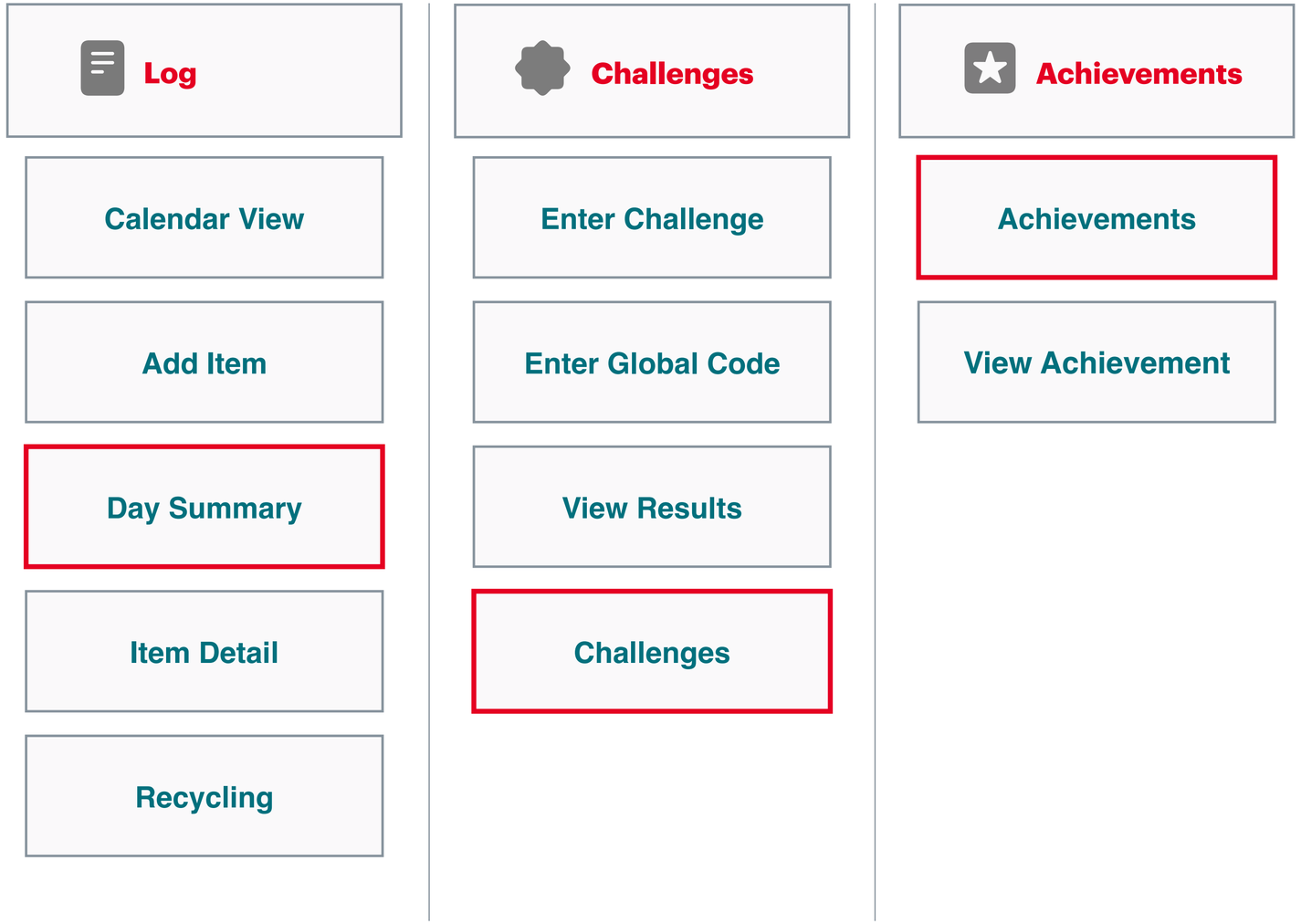
Group your screen titles into categories and name them. Use the SF Symbols app to choose an icon that best describes each category. Then highlight the default screen for each category.

The architecture of an app often breaks down into several global categories of activities that the user can switch between fluidly. These screen groups will translate into UI elements in the next stage of prototyping.

Don’t worry if your screens don’t fall naturally into multiple categories. Some apps focus on just one activity.

Example

A one- or two-word summary of what these categories are about:



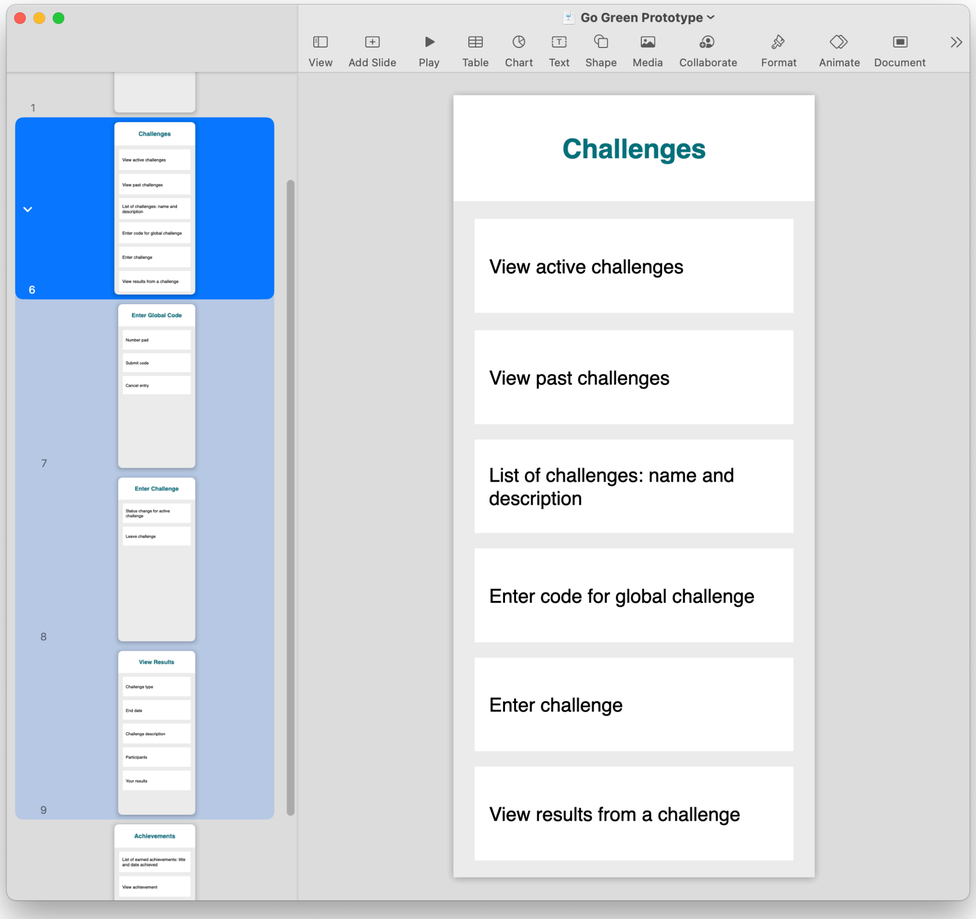
Group Screens

1. Copy the title box for each screen outline here.
2. Group the screen titles into categories.
3. Duplicate the Category box as needed to provide titles and icons for each group of screens.
4. Choose the title of the main screen for each category and highlight that box.

Prototype

Use the slide navigator to group screen outlines by category.

1. Order the screen outlines by group.
2. For each group, use the main screen as the parent and drag the others below it as children.



Explore Code

In this exercise, you'll:

- Define properties for a structure.
- Use different data types.
- Supply values for an instance of a structure.

Types and Properties

The structures you create are determined by the data in your app. You organise each thing your app represents by grouping related data together and giving it a name. In Swift, you define a structure and declare its properties. In this exercise you'll learn how to create properties for a structure, to further your understanding of Swift types.

To start, be sure "My Playground" is open in the Swift Playgrounds app. Then open the page named "Data and Naming".

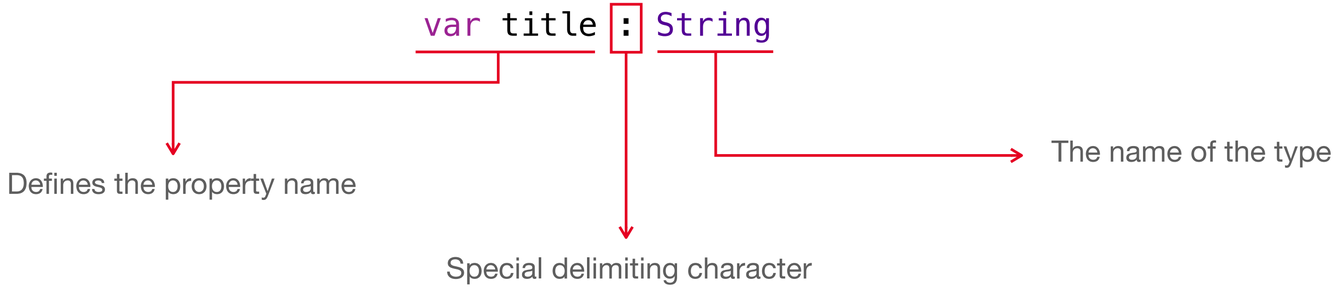
Types and Properties

Create structure properties.

Start by deleting everything but the first few lines where you created the `Challenge` structure. Update the first two lines inside the structure by creating two variables:

```
struct Challenge {
   var title: String // the name
   var points: Int // scored upon successful completion
   // is team - true of a team effort, false if solo
   // end date - when it will no longer be offered
}
```

A variable inside a structure has special meaning and is called a property of the structure. You created two properties, `title` and `points`. Each one has a type. Here's how that works:



Unlike the variables you've made previously, you didn't assign values to `title` or `points`. You'll assign the values to all of a structure's properties each time you create one.

Note the names of the two types you used. You've already learned about strings; the official Swift type goes by the same name, and uses the capital letter naming convention. `Int` (short for integer) is a type that represents whole numbers, such as 0, 42 and –8.

Types and Properties

Make an instance of a struct by supplying values for its properties.

Enter the following to start creating an instance of your challenge struct, stopping with the open parenthesis (:

```
var noStrawChallenge = Challenge(
```

Look at the bottom of the playground. This time, there's a code completion highlighted in blue that you can use as a shortcut.

```
(title: String, points: Int)
```

Press Return to insert it into your code:

```
Challenge(title: String , points: Int )
```

Now that your structure has properties, you'll need to provide values for each instance. Each placeholder indicates the type of the property, which you replace with a specific value. Replace the first placeholder with `"No Straw Challenge"` and the second placeholder with `5` to make a specific unique `Challenge` instance. Then print it to the console:

```
var noStrawChallenge = Challenge(title: "No Straw Challenge",
points: 5)
print(noStrawsChallenge)
```

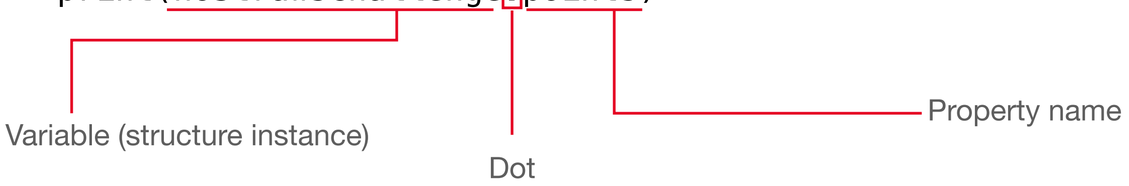
You can also review the values directly without printing, by clicking the result icons (◆) and (abc) in the right margin of the code editor.

Types and Properties

Access individual properties of a structure instance.

You can access the values of these properties individually using dot notation. Try printing the values of each property using the following code:

```
print(noStrawsChallenge.title)
print(noStrawsChallenge.points)
```



Run My Code

Run your code and review the console output.

You can also use dot notation to modify an existing structure instance. Use the equal sign to assign a new value of the correct type to the desired property. For example, add the following line to the end of your page:

```
noStrawChallenge.points = 8
```

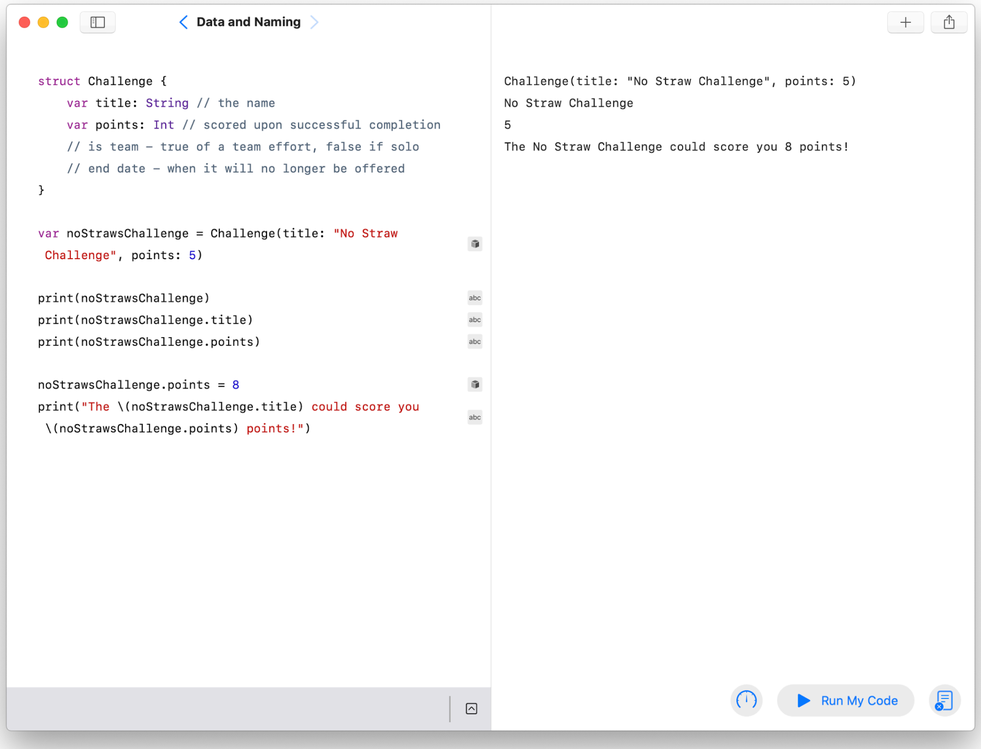
Add another print line at the end to show the new modified state of the challenge using string interpolation:

```
print("The \(noStrawChallenge.title) could score you
 \(noStrawChallenge.points) points!”)
```

Types and Properties

Review your work.

Your completed exercise should now look like this:



Challenge: In addition to `String` and `Int`, you can also use `Bool` to represent values that can either be true or false, and `Double` to represent decimal numbers, such as 3.14. If you've created structures for your app's data types, practise replacing your comments with real properties and creating instances of them.

Is there a property that doesn't match one of those four basic types? Try to create a new structure to represent it, then use it as the type of the property. When you create an instance, you'll have to create an instance of the property's structure as well. If you're feeling adventurous, you can take this exercise as far as you want, nesting structures inside others to create complex data types.

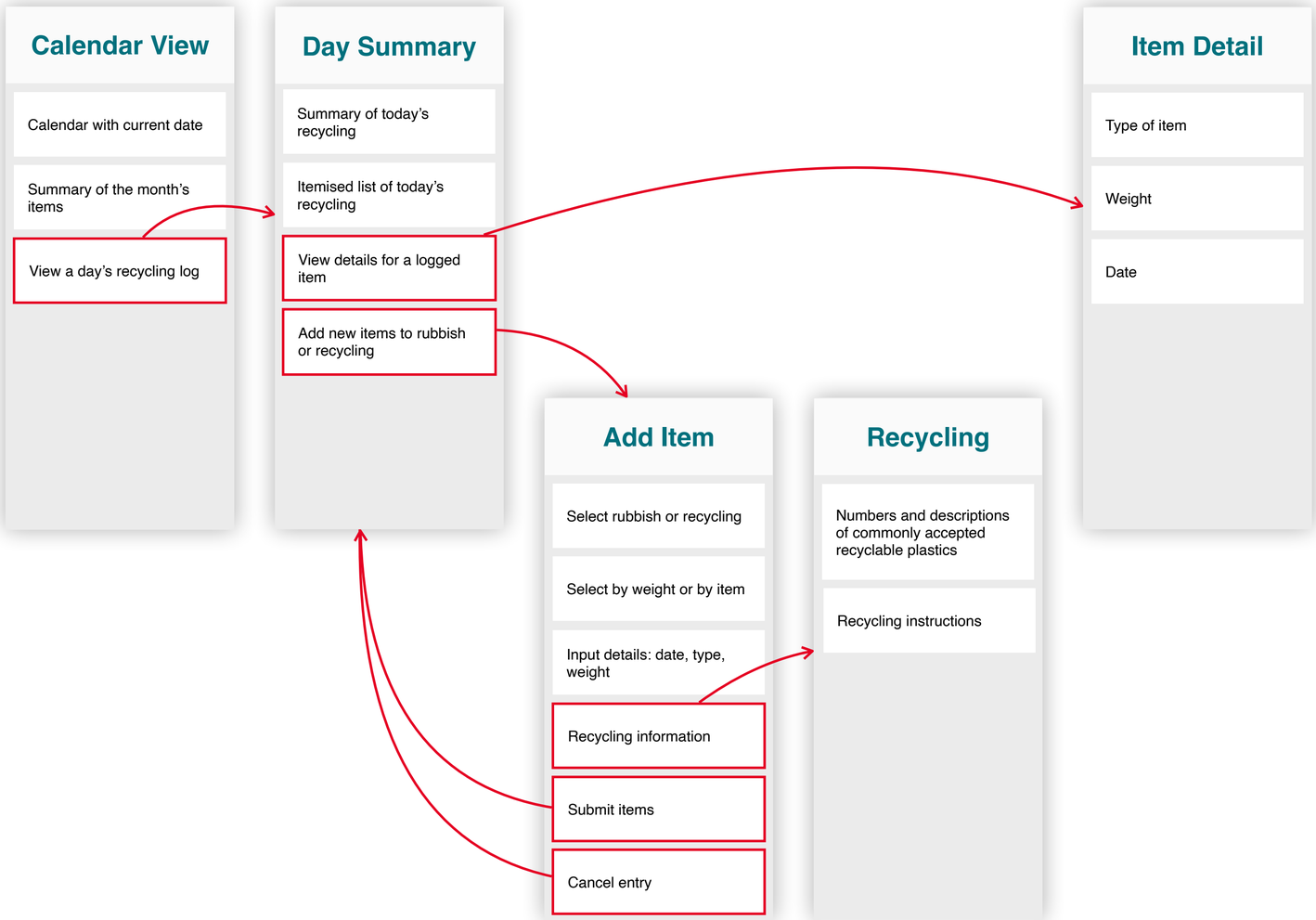
Explore Code

Link Screens

Draw lines that connect content to the screens you outlined and grouped.

Screens within each global category are often organised in a sequential flow from one to the next.

You might notice that some screens have many outgoing lines while others have none. Don't worry — you'll organise them in the next exercise.



Link Screens

Copy your existing screen outlines here and link them together.

You'll probably need one slide per group. After you paste your outlines, resize them so they all fit on the screen. Arrange the screens so that related ones are close to each other.

Ungroup your slides before you add links. Select all the screens and choose Arrange > Ungroup or press Option-Command-G.

Change the colour of boxes that trigger the presentation of a different screen. Use a connection line to link from a coloured box to the screen it leads to.

Prototype

Add links between related screens.

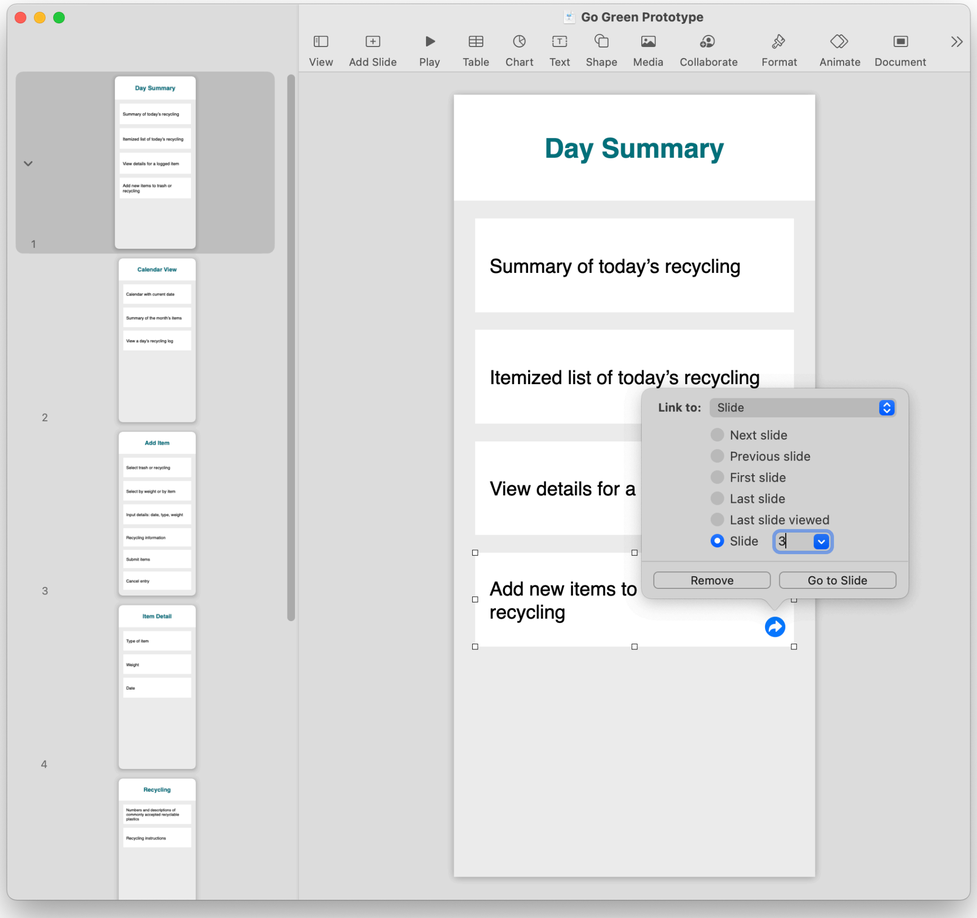
Linking slides is the process of creating tap targets on a Keynote slide that jump to a corresponding slide.

You may want to add highlights to the same items you just highlighted in your app map to make this exercise easier.

After you've finished, try playing your Keynote prototype and click the links. (You can play your presentation on an iPhone to get a feel for how your app will look and feel on a device.)

For each item that presents another screen, add a link from that item to the screen it presents.

To add a link to another slide, right-click an object and choose Add Link > Slide or press Command-K. Choose Slide, then select Slide (the last item in the list), and enter the slide number.



Create Tabs
Add Navigation
Create Modals
Add Interface Elements

Wireframe

A wireframe is a minimal working prototype. By the end of this stage, you'll have a functioning Keynote prototype that simulates the behaviour of your app.

You'll build a wireframe from your app's architecture map by converting screen outlines into a sketch of the interface. This is an organised process, starting with the top-level navigation elements and progressively drilling down to the elements on each screen.

Create Tabs

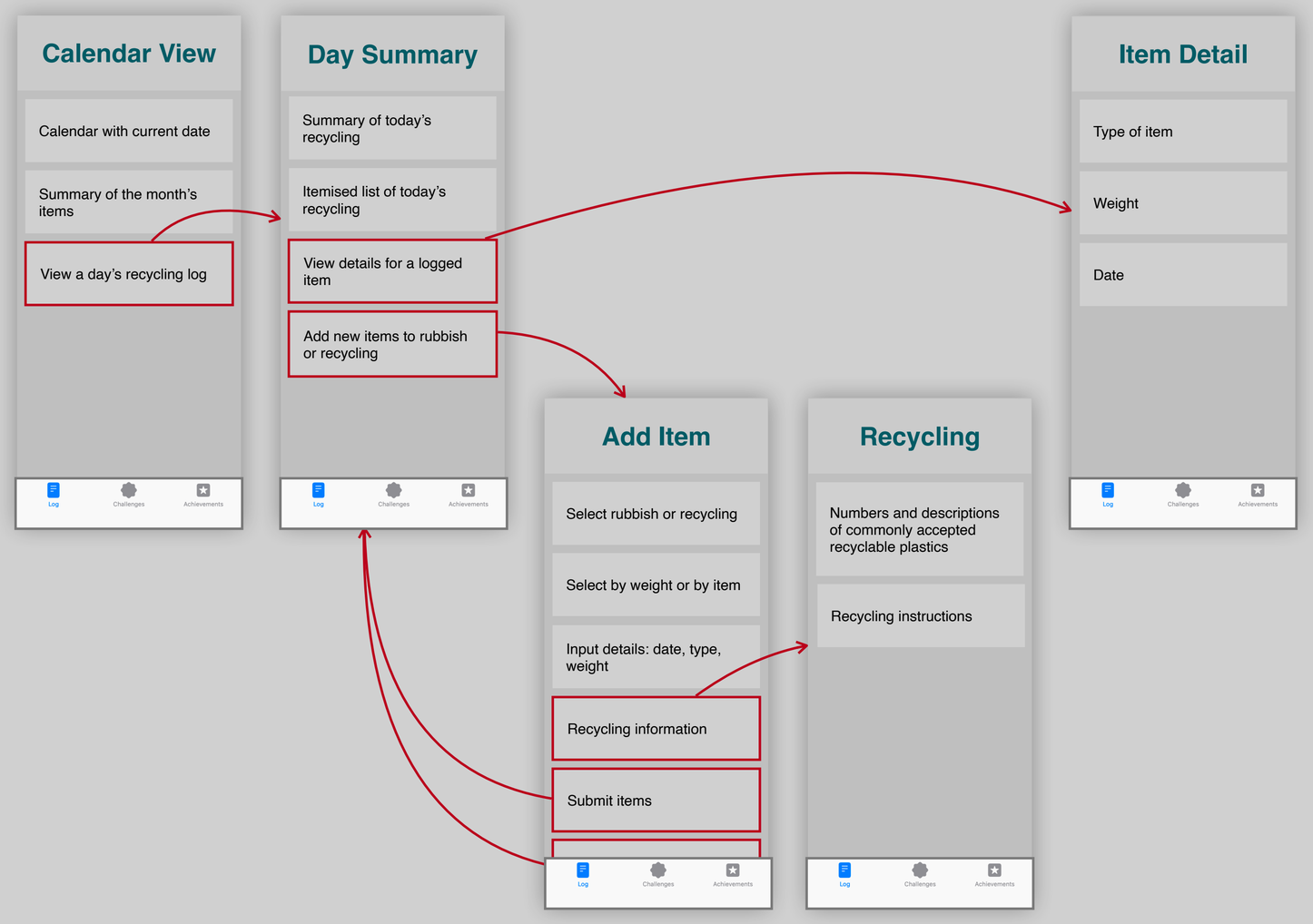
A tab bar is the most common form of global navigation in apps. It lets people quickly switch among different sections of an app. Because it's always at the bottom of the screen and doesn't change, users can rely on the tab bar no matter where they are in the app.

In the following Keynote exercise, you'll add a tab bar to your prototype.

If your app doesn't have multiple top-level navigation categories, you might find a tool bar useful. Use a tool bar at the bottom of a screen to provide important actions for that screen.

Never use a tab bar and a tool bar on the same screen.

Example



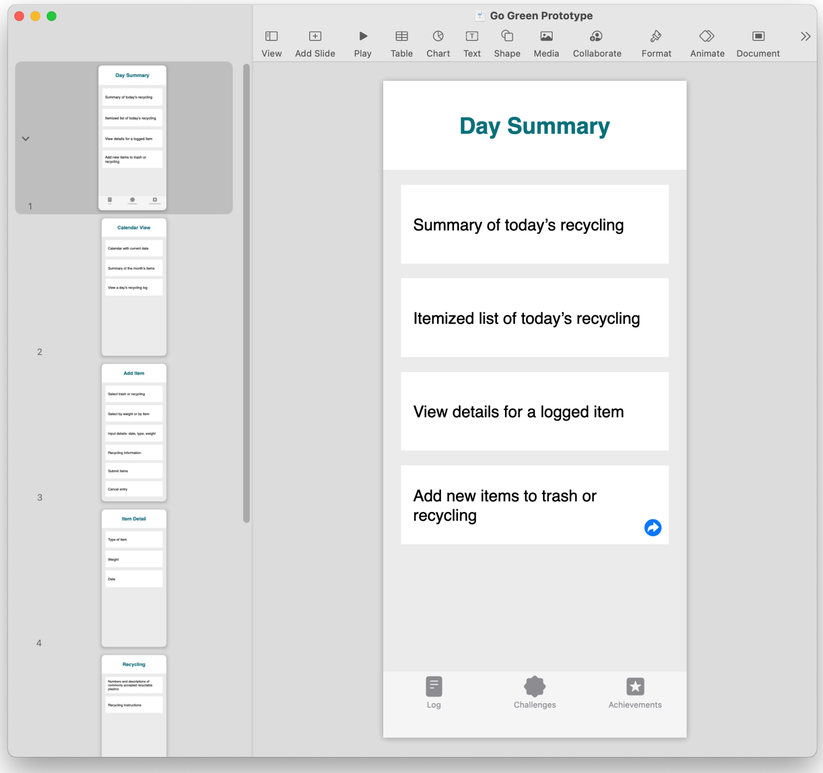
Prototype

Add a tab bar to your prototype screens and link each tab to its main screen.

After you've finished, try playing your Keynote prototype and click the tabs to navigate between screens. Congratulations! You're one step closer to a prototype that looks and feels like a native iOS app.

Set up a global tab bar.

1. Copy and paste a tab bar from the iOS Templates+UI-Elements Keynote presentation into a slide of your prototype.
2. Ungroup it by right-clicking and choosing Ungroup, or by pressing Option-Shift-Command-G.
3. Set the names and icons of the tab items to match your navigation categories.
4. For each tab item, add a link from the tab item to the main page for that navigation category.
5. Group the tab bar again. Select all its items, then either right-click and choose Group or press Option-Command-G.



Prototype

Add a tab bar to your prototype screens and link each tab to its main screen.

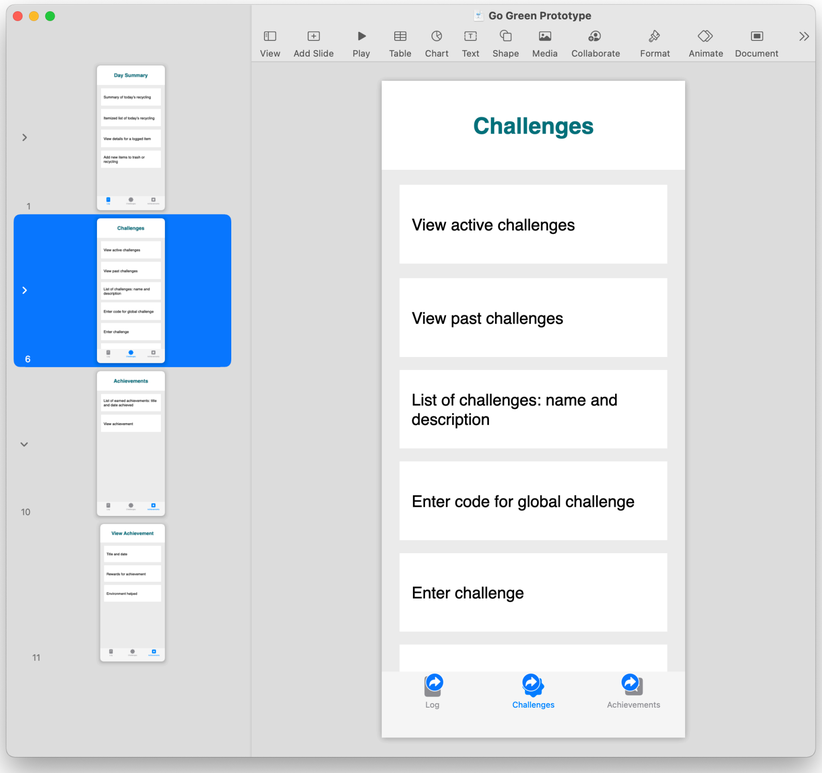
After you've finished, try playing your Keynote prototype and click the tabs to navigate between screens. Congratulations! You're one step closer to a prototype that looks and feels like a native iOS app.

Create a tab bar for each main screen.

1. Copy and paste the updated tab bar into each main screen outline in your prototype.
2. For each main screen, use the standard iOS blue to highlight the icon and title of its tab; make the other screens grey.

Create tab bars for all child screens.

1. Copy and paste the tab bars from each main screen to its children.



Add Navigation

Find linear paths between the screens in your app map.

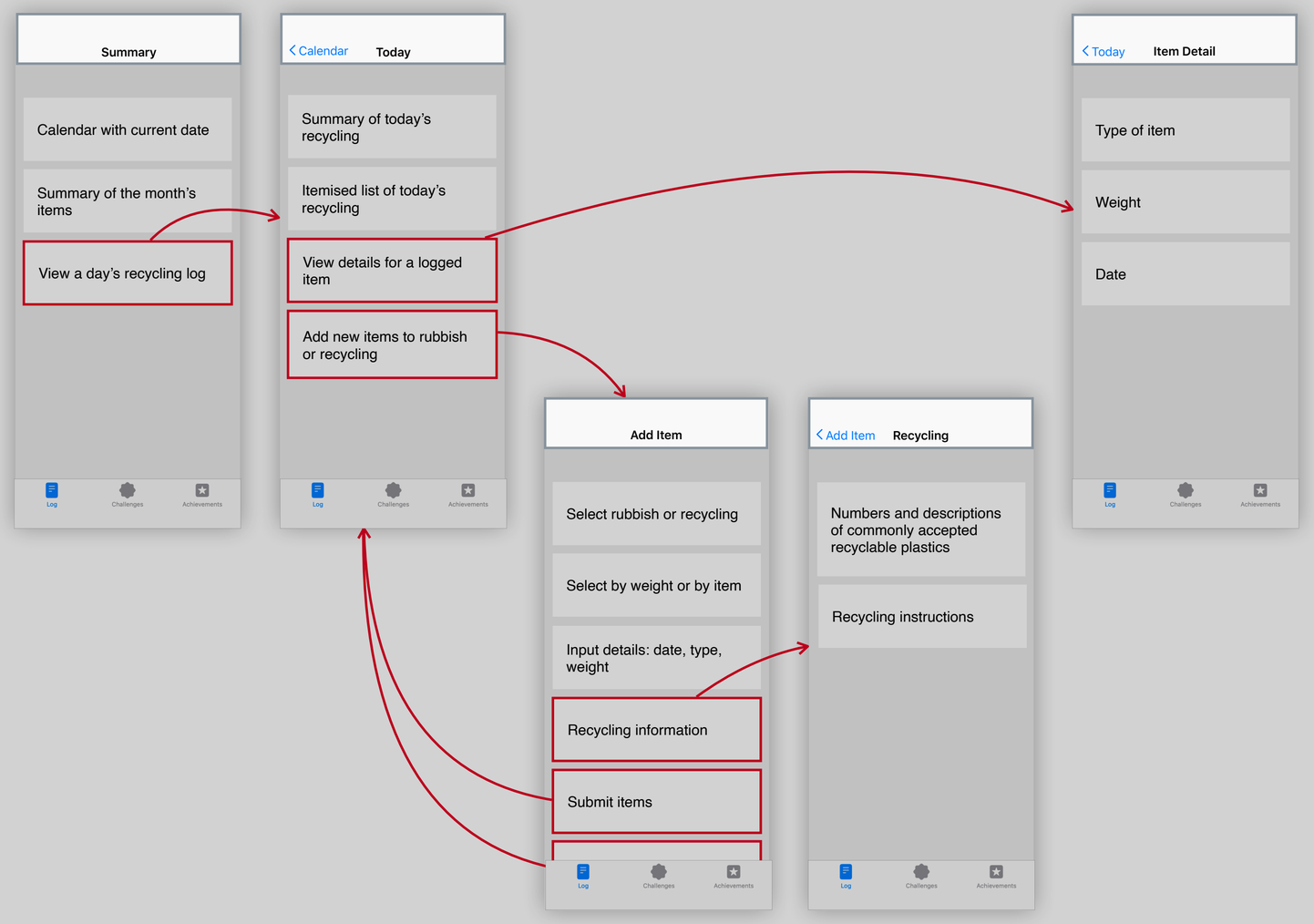
Linear paths between screens are usually managed through hierarchical navigation.

The navigation bar manages a sequence of hierarchical screens. Choosing an onscreen item pushes the next one in from the right, and tapping the Back button allows the user to go to the previous screen.

The title of the current screen appears in the centre of the navigation bar. The Back button appears on the left, and often takes the title of the previous screen. The right side of a navigation bar can contain actions such as Add and Search.

In the following Keynote exercise, you'll add navigation bars to your prototype.

Example

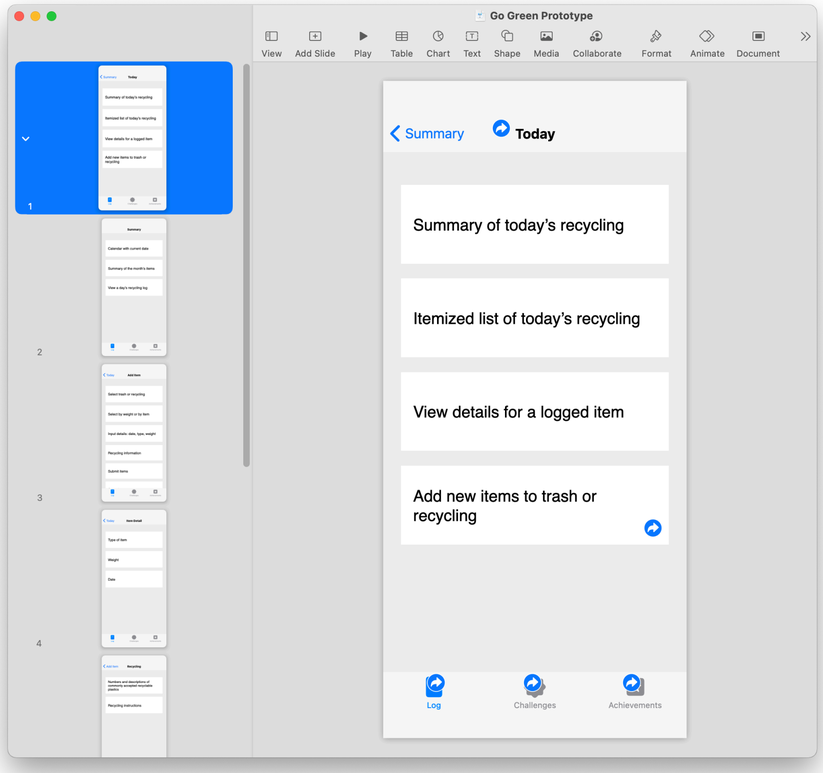


Prototype

Add a navigation bar to the top of each screen outline and link their Back buttons.

For each screen in your prototype:

1. Delete its name.
2. Copy and paste a navigation bar from the iOS Templates+UI-Elements Keynote presentation. (Choose the one most appropriate to this screen.)
3. Set the title of the navigation bar. (If the screen's name is long, you might choose a different title for the navigation bar.)
4. Delete extraneous items in the navigation bar.
5. Link the Back button to the previous screen if applicable.



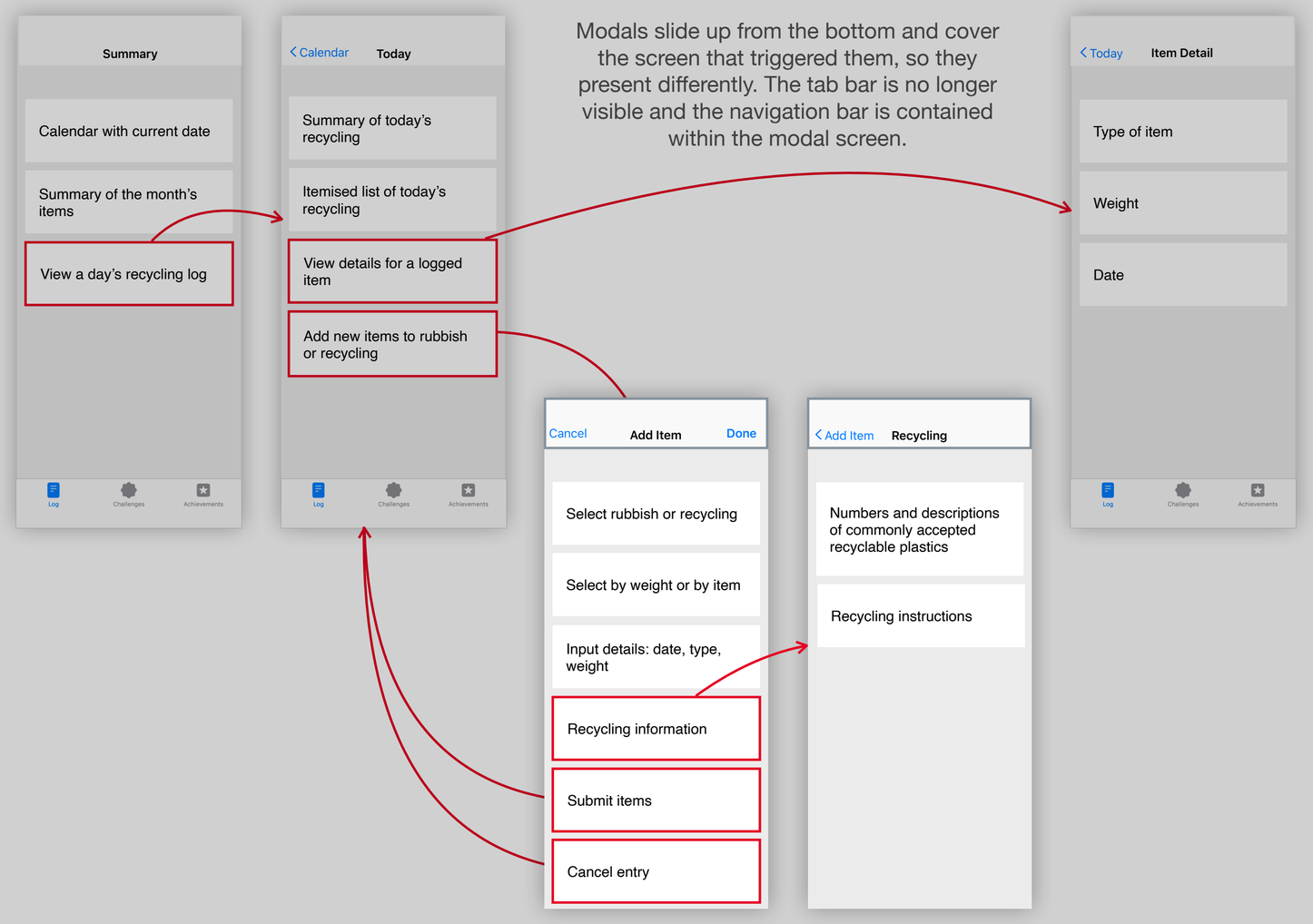
Create Modals

Identify focused activities and create modal screens.

Modality is a design technique that helps people focus on a self-contained task or set of closely related options. A modal presents content in a temporary mode that's separate from the user's previous context and requires an explicit action to exit.

Always include a button that dismisses the modal view — in the main screen, the navigation bar or both. For example, you might use a Done or Cancel button. Including a button ensures that the modal view is accessible to assistive technologies and provides an alternative to dismissal gestures.

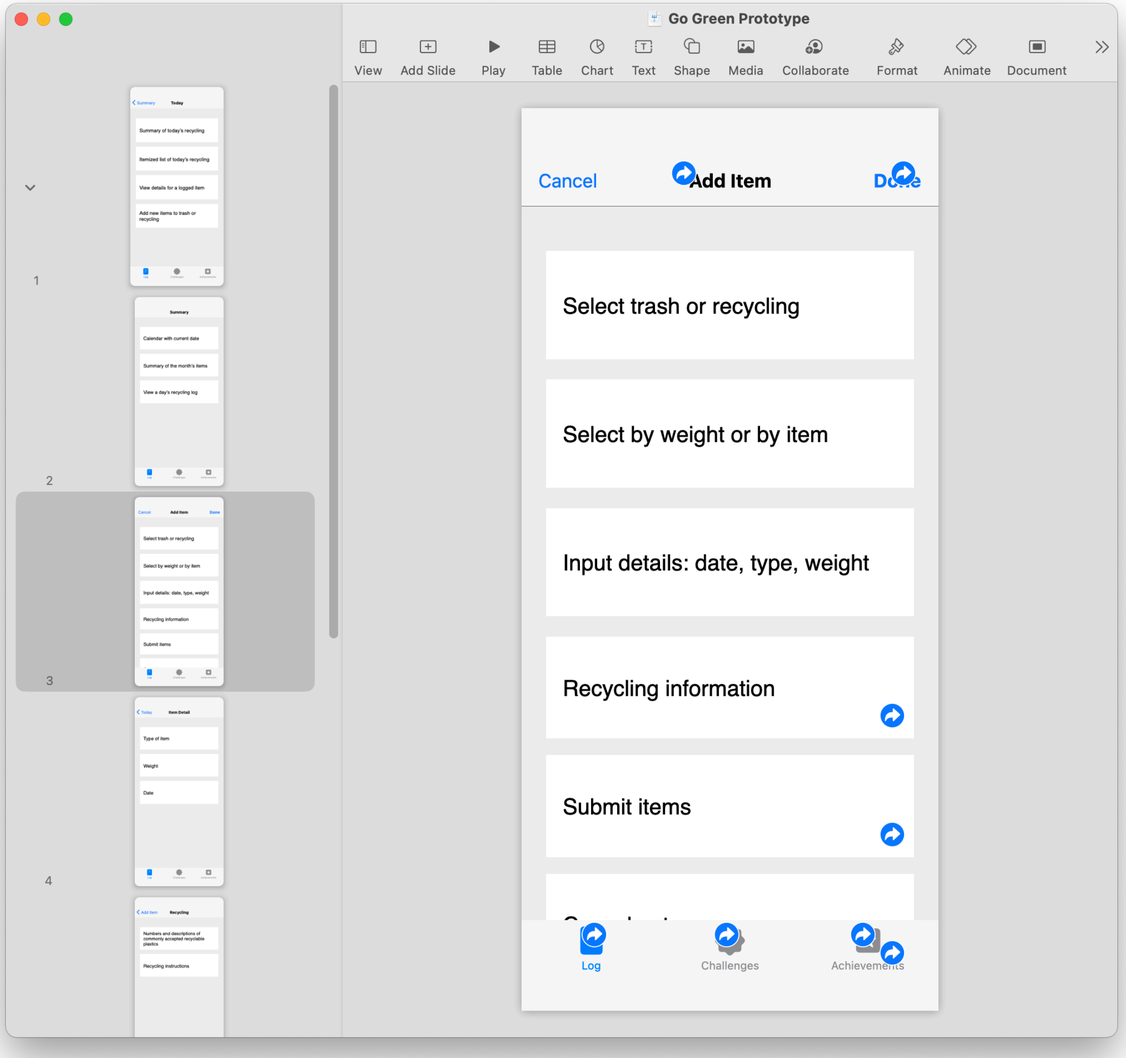
Example



The first screen in a modal does not have a Back button. If your modal moves to a secondary screen, then a Back button is used.

Prototype

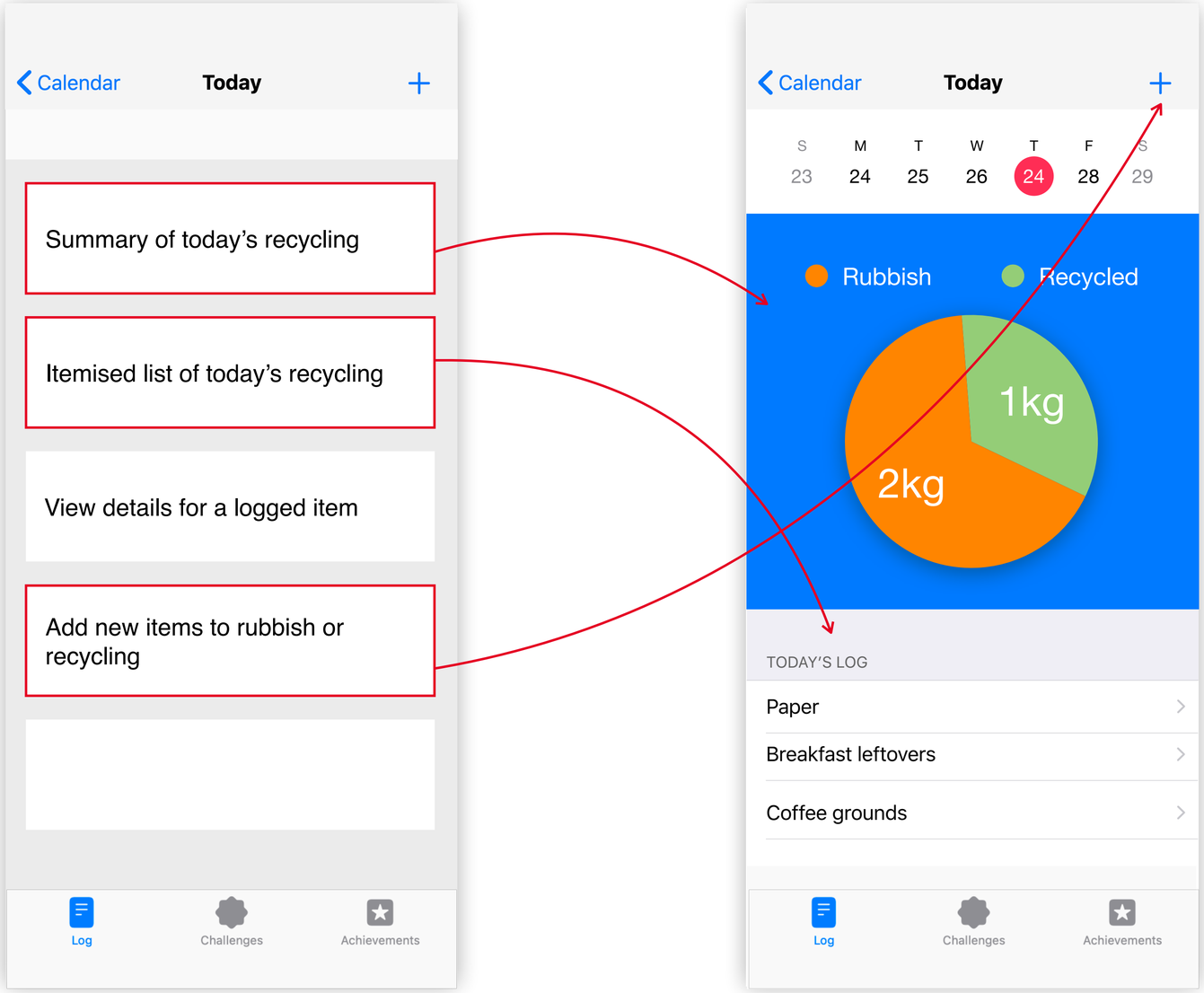
Find screens that present a focused activity to the user and convert them to modals by removing the tab bar and updating the navigation bar.



Add Interface Elements

Use standard iOS elements to convert your screen outlines into wireframes.

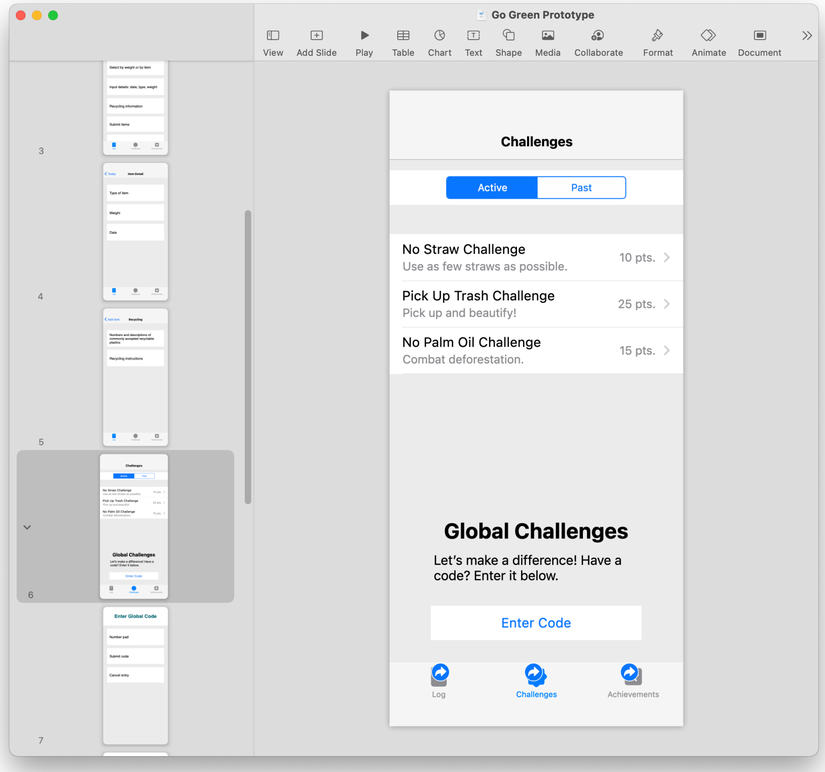
iOS users expect standard interface elements in apps when presented with information, controls or navigational elements. And iOS developers consistently adopt a set of common practices, which are documented in the Human Interface Guidelines.



Prototype

Convert the remainder of your screen outlines into UI elements.

1. Review the UI elements in the Human Interface Guidelines.
2. Decide which elements will be helpful in your app.
3. Paste the elements from the iOS Templates+UI-Elements Keynote presentation.
4. Create any customised elements you need that aren't in the template.
5. Change your screen outline boxes from text to UI elements.



Explore Code

In this exercise, you'll:

- Create a button.
- Update the console when the button is clicked.
- Update the value of a variable.

Event-Based Programming

In an iOS app, the user is in control. Their interactions are represented to your app as a series of events. Your app responds to each event by interpreting the user's action and acting on their command. You write event handlers — blocks of code that perform the work — and attach them to interactive elements such as buttons, text fields, sliders and switches.

Consider a button in your app. How do you know when the user has tapped the button? The SwiftUI framework provides a `Button` type that detects user taps and lets you attach the specific action you want when the button is tapped. In this exercise, you'll create a button that performs an action.

To start, be sure "My Playground" is open in the Swift Playgrounds app. Then create a new page and name it "Event-Based Programming".

Event-Based Programming

Create a button that prints to the console.

Enter the following two lines to import the PlaygroundSupport and SwiftUI frameworks:

```
import PlaygroundSupport
import SwiftUI
```

Now create a button with the following three lines of Swift, being careful that the curly braces { }, parentheses ( ) and double quotes " " all match:

```
var challengeButton = Button("Complete Challenge") {
   print("Completed a challenge.")
}
```

From start to finish, here's what this code does:

```
var challengeButton
```

Creates a variable for the button.

```
Button("Complete Challenge")
```

Creates an instance of `Button` — a type of SwiftUI view — and sets its title with a `String` value.

```
{
  print("Completed a challenge.")
}
```

Defines the button's action, which will execute each time the button is clicked.

Now add one more line to add the button to the live view:

```
PlaygroundPage.current.setLiveView(challengeButton)
```

Event-Based Programming

Create a variable that the button will update.

Run your code. You'll see the live view with your button. Click the console button; you'll see an empty console. Now click the Complete Challenge button. You should see the result of the action appear in the console.

As the user interacts with the views in an app, the app often updates its model data in response. For example, you might want to keep track of the number of challenges the user has completed.

Add a variable to keep track of the total number of challenges completed and include it in your printed string:

```
var numberCompleted = 0
var challengeButton = Button ("Complete Challenge") {
   print("Challenge Completed! Total number of challenges completed: \
   (numberCompleted).")
}
```

Run your code and click the button a few times.

You should see the following in the console view:

```
Challenge Completed! Total number of challenges completed: 0.
Challenge Completed! Total number of challenges completed: 0.
Challenge Completed! Total number of challenges completed: 0.
Challenge Completed! Total number of challenges completed: 0.
```

Event-Based Programming

Update the variable when the button is clicked.

Of course, you want the number of challenges to increase each time the button is clicked.

To do this, you'll update the value of the variable in the button's action. You've changed the property of a structure instance by assigning it a new value. But this time, you have to base the new value on the existing value of `numberCompleted`. You might think to assign 1 to the variable as shown below:

```
var challengeButton = Button ("Complete Challenge") {
   numberCompleted = 1
   print("Challenge Completed! Total number of challenges completed:
   \(numberCompleted).")
}
```

However, this won't work because the variable will change from 0 to 1 the first time the button is clicked, but won't change afterwards.

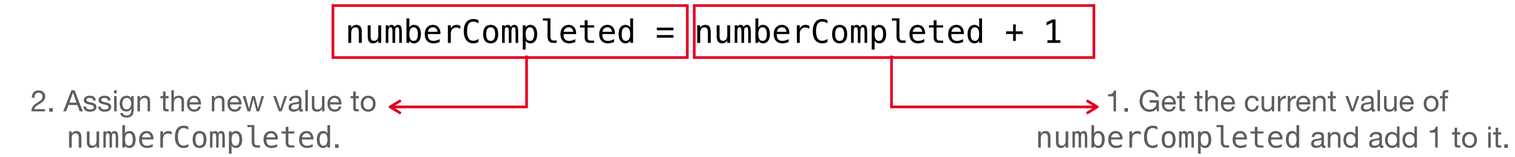
Event-Based Programming

Update the variable when the button is clicked.

Instead, you'll refer to the value of the variable itself when updating it. Modify your code so it looks like this:

```
var numberCompleted = 0
var challengeButton = Button ("Complete Challenge") {
   numberCompleted = numberCompleted + 1
   print("Challenge Completed! Total number of challenges completed:
   \(numberCompleted).")
}
```

To understand what's happening, read the line in order from right to left:



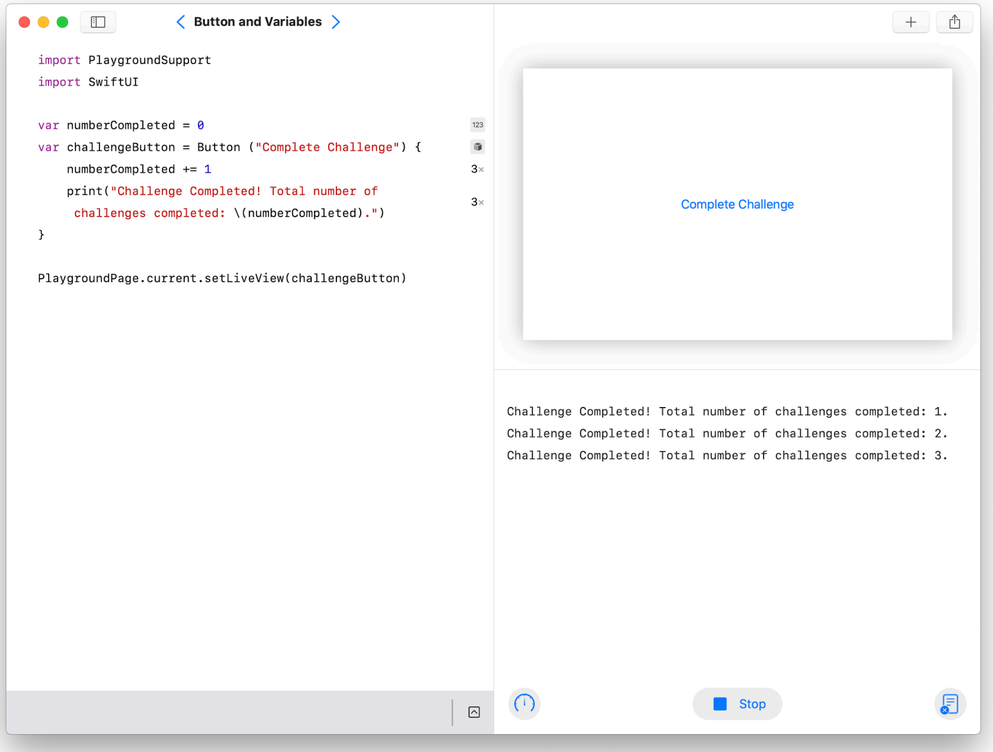
Run My Code

Run your code again. Click the Challenge Completed button a few times. Each time you complete a new challenge, the total number of challenges completed should change in the console.

Event-Based Programming

Review your work and try a challenge.

Your completed exercise should now look like this:



Challenge: Change the title of the button, for example, “Another Challenge Down!”

Change the code in the action to construct and print a message of your choosing.

Make a button that counts down from a starting value.

Make a button that updates the value of a String variable using the same technique you used with `numberCompleted`.

Tap Targets
Content Insets
Weight and Balance
Alignment

Refine

Now that you have a functioning prototype, it's time to apply important interface design guidelines. By the end of this stage, your prototype will feel at home on iOS and in the hands of your users.

A solid interface design is critical to a good iOS experience. You'll learn about the most important properties of a good interface and apply those lessons to make your prototype a pleasure to use.

Tap Targets

Users should be able to tap the icons or buttons in your app. If the tap target is too small, users will have trouble triggering it. If it's too big, it can interfere with another button that's close to it.

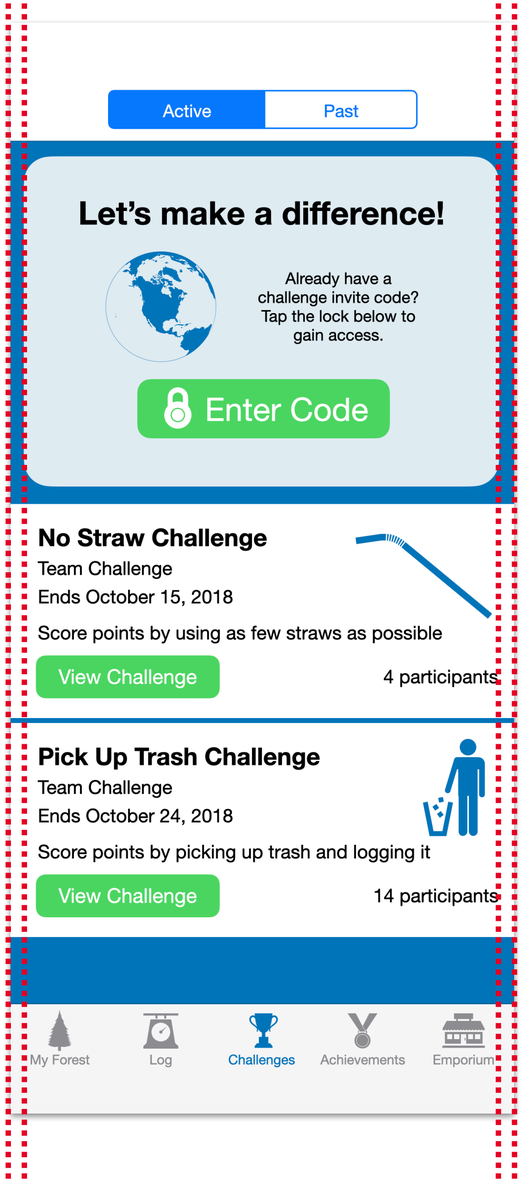
Try to maintain a minimum tappable area of 44pt by 44pt for all controls.

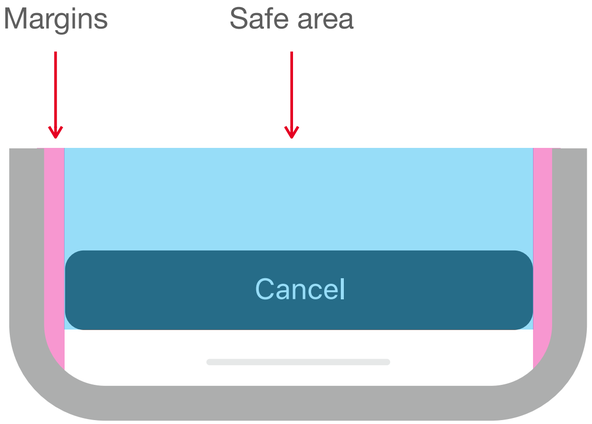
Content Insets

Many actions in iOS require the user to use a swipe gesture to trigger an action.

People use swipe gestures at the bottom edge of the display to access features like the Home Screen and App Switcher. These gestures could cancel customised gestures you implement in this area. The far corners of the screen can be difficult areas for people to reach comfortably.

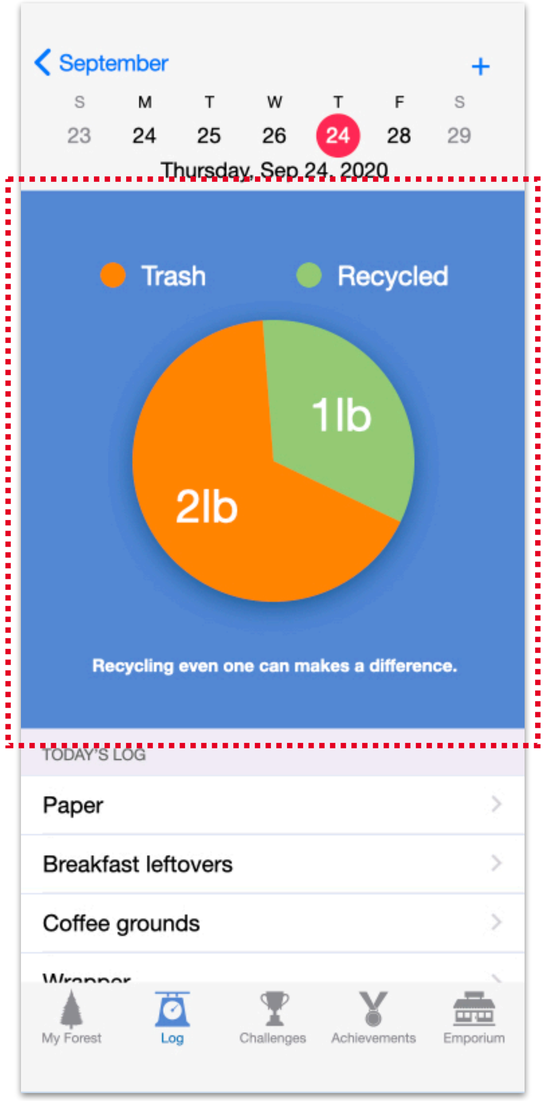
In general, content should be centred and symmetrically inset so it looks great in any orientation. You should also make sure the content isn't clipped by rounded corners, hidden by a sensor housing or obscured by the indicator for accessing the Home Screen.





Weight and Balance

Large items catch the eye and appear more important than smaller ones. They're also easier to tap, which is especially important when an app is used in distracting surroundings, such as in the kitchen or a gym. In general, place principal items in the upper half and near the left side of the screen in a left-to-right reading context.



Explore Code

In this exercise, you'll:

- Learn how to display colours.
- Arrange views in horizontal and vertical stacks.
- Nest stacks.
- Add other kinds of views to stacks.

Composing Views

All apps have a view hierarchy. Larger views in your app (such as a screen) contain smaller ones (such as lists), which contain even smaller ones (a list's individual rows). How much further could you break down the hierarchy?

Views are powerful because you can compose them together in myriad ways to create intricate and beautiful interfaces. In this exercise, you'll get a taste of the power of view composition.

This is the only code exploration that won't explain every line of code you write. SwiftUI is a complex and powerful framework. To see it at work, you'll have to forgo understanding exactly how your code works.

To start, be sure "My Playground" is open in the Swift Playgrounds app. Then create a new page and name it "Composing Views".

Composing Views

Display a colour view.

Start by importing your two favourite frameworks:

```
import PlaygroundSupport
import SwiftUI
```

Now enter the following code. You won't understand all of it, though you should notice that you're creating a new structure named `ContentView` and a new kind of property named `body`. Be sure you're nesting the curly braces correctly, and that you have one closing brace for each opening brace.

```
struct ContentView: View {
    var body: some View {
        Colour.red
    }
}

PlaygroundPage.current.setLiveView(ContentView())
```

The live view should open to show a red view that fills all the available space, as shown above. As you may have guessed, `Colour` is a type of SwiftUI view that simply displays a colour.

For the remainder of this exercise, focus only on the innermost code as indicated below.

```
struct ContentView: View {
    var body: some View {
        Colour.red
    }
}
```

This code defines the views inside ContentView.

Composing Views

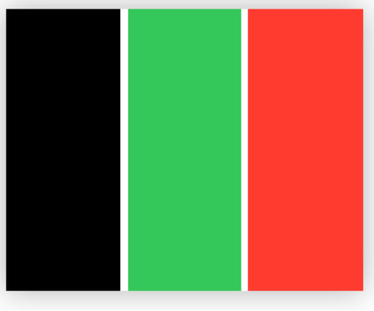
Arrange views in stacks.

SwiftUI provides several types of views that allow you to ‘stack’ other views inside of them. The `VStack` view can arrange many views vertically. Try it by replacing `Colour.red` in your code with a `VStack` that arranges three differently coloured views.

```
import PlaygroundSupport
import SwiftUI
struct ContentView: View {
    var body: some View {
        VStack {
          Colour.black
          Colour.green
          Colour.red
         }
    }
}
PlaygroundPage.current.setLiveView(ContentView())
```

The HStack view arranges your views horizontally. Replace `VStack` with `HStack` and click Run My Code again.

```
        HStack {
          Colour.black
          Colour.green
          Colour.red
        }
```



Explore Code

Composing Views

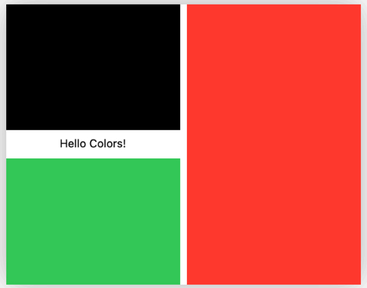
Nest stacks.

Vertical and horizontal stacks can be placed inside each other. Replace the contents of the `HStack` with the code below and click Run My Code.

```
HStack {
   VStack {
      Colour.black
      Colour.green
   }
   Colour.red
}
```

You already know two other kinds of SwiftUI views. Try adding a `Text` view:

```
HStack {
   VStack {
      Colour.black
      Text(“Hello, Colours!")
      Colour.green
   }
   Colour.red
}
```

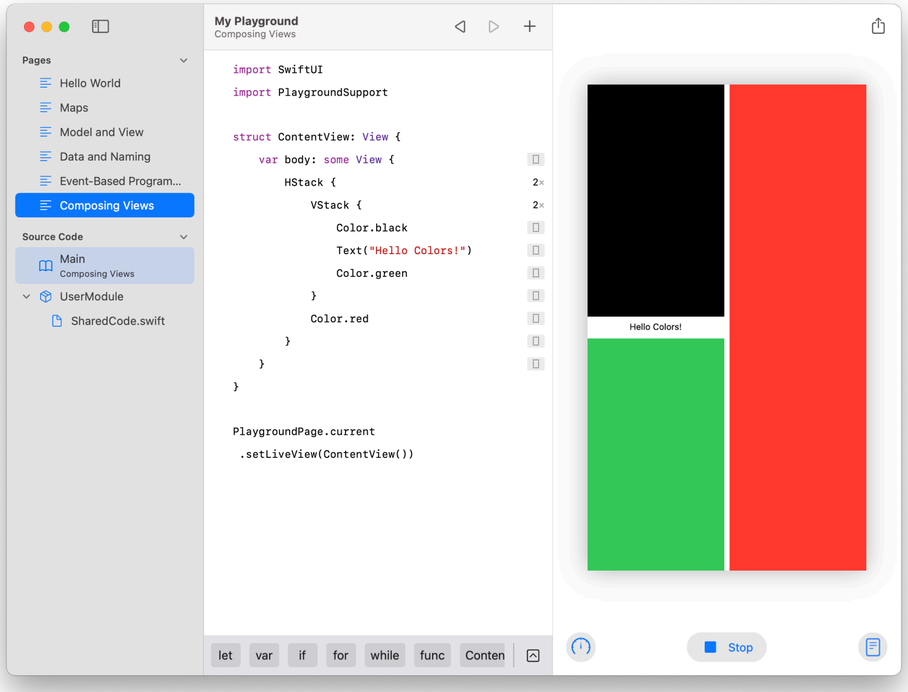


Explore Code

Composing Views

Review your work and try a challenge.

Your completed exercise should now look like this:



You've just scratched the surface of view composability in SwiftUI. Views can be nested to an arbitrary level and combined to create the most complex of interfaces.

Challenge: There are many ways to specify `Colour` instances, but for now try displaying some other common ones using dot notation, for example, `.yellow`, `.purple` or `.blue`.

How many flags can you replicate with this technique?

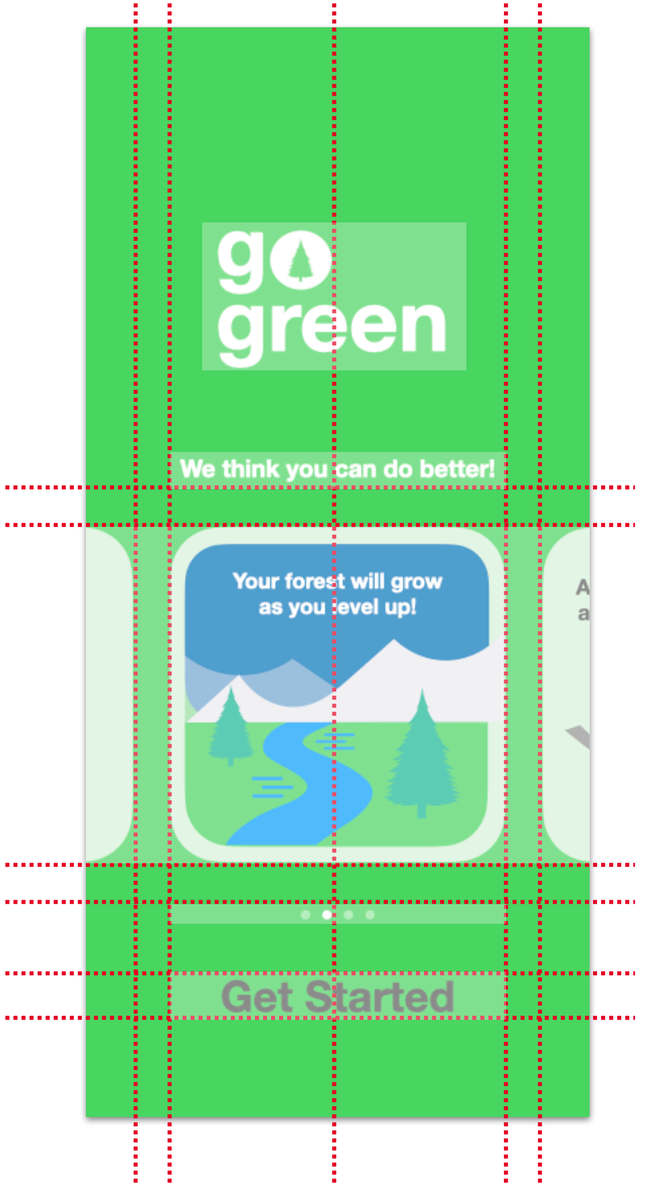
Try adding a button (or several!) to your view hierarchy.

What kinds of interfaces can you simulate using just these five SwiftUI views?

Explore Code

Alignment

Alignment makes an app look neat and organised, helps people focus while scrolling and makes it easier to find information. Indentation and alignment can also indicate how groups of content are related.



Personality
Icon

Style

The last stage of prototyping is defining the personality of your app to set it apart from its peers. By the end of this stage, you'll have a prototype that's as close as it can come to a real app — one that you'll be proud to share with your testers.

Style encompasses a range of elements from colour and font to icons. Now you can use your imagination to create a cohesive identity for your app.

Personality

Complete the style guide template to apply to your app.

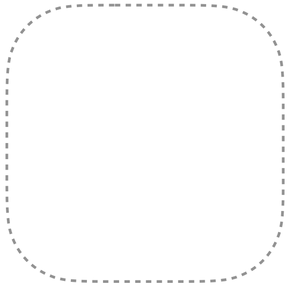
Picking out colours, typography, images and icons — in other words, branding your app — can be fun. But it's important to keep accessibility in mind when choosing these UI elements.

San Francisco Typeface
Designed to be consistent with the simple and clean iOS aesthetic, system fonts are legible and neutral.

SF Symbols
Apple created an icon set that supports Dynamic Type and the Bold Text accessibility feature.

Colour
iOS offers a range of system colours that automatically adapt to vibrancy and changes in accessibility settings.

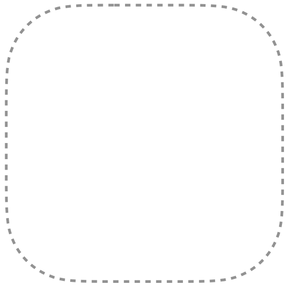
Primary colour

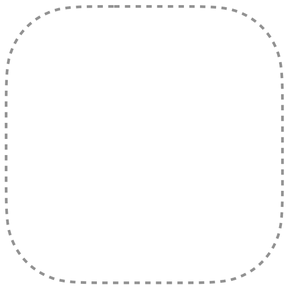


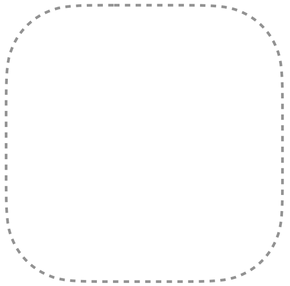
Secondary colour

Icon set

Imagery examples







Typography

The quick brown fox jumps over the lazy dog.

Prototype

Incorporating style into the provided resource.

1. Update important elements in your app (for example, highlighted tabs and buttons) to match the primary colour in your style guide.
2. Use the secondary colour sparingly, to call attention to important details.
3. Update icons to match your style guide.
4. Update fonts to match your style guide.
5. Standardise imagery.

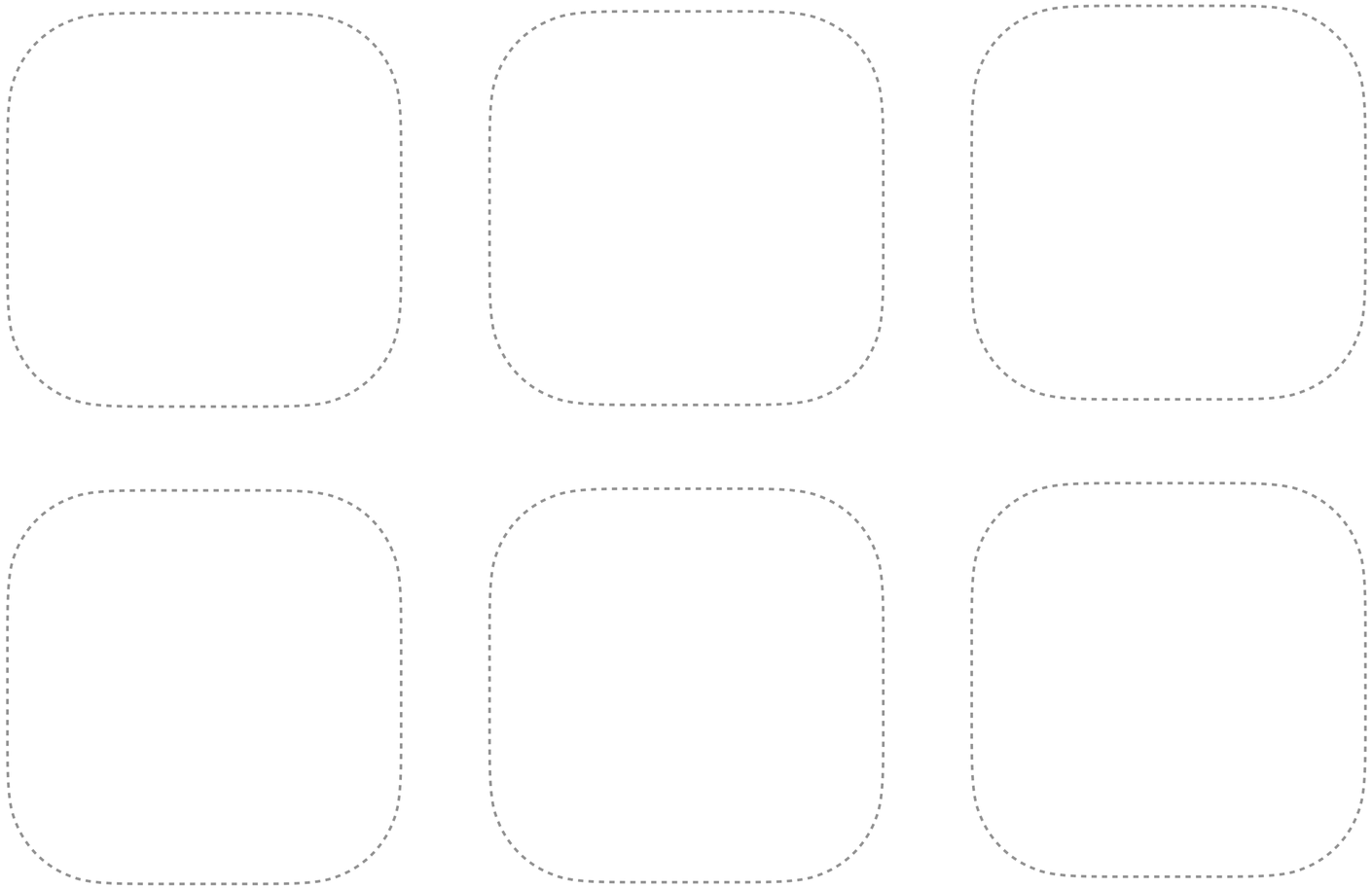
Icon

Use the templates to try out a few icon designs. Make more copies of this slide if you need to.

Your app's icon will distinguish it on a user's screen from all the other apps they use on their phone.

Make it simple
Find a single element that captures the essence of your app and express that element in a simple, unique shape. Add details cautiously. An icon should have a single, centred focus point that immediately captures attention. If an icon's content or shape is too complex, its essence may not be discernible, especially at smaller sizes.

Make it recognisable
You don't want your users to have to examine the icon to figure out what it represents and what your app does; they should get the gist immediately. Using transparency or a busy background can impede recognition. Test your icon against varying backgrounds — dark and light, simple colours, patterns and photos — so you can be sure that it stands out in all contexts.

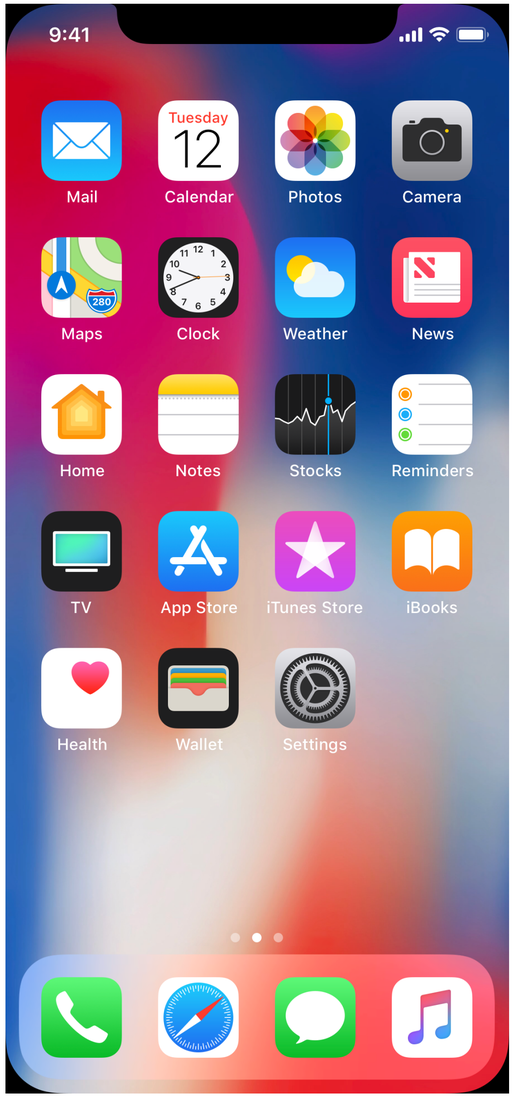


Prototype

Create a Home Screen in your prototype so that users can tap your app icon.

Test your icon on different backgrounds.

1. Create a tap target link of your icon.
2. Use the Magic Move transition from opening your app to a launch screen.
3. Set a timer to move to the next screen that the user will land on.



Test

Testing your prototype will help you understand whether your ideas and assumptions are correct. In the test phase, you'll architect your tests, create a plan to execute them, and prepare by gathering users and creating a checklist.

Define Tests
Create User Journeys
Define a Process
Plan an Introduction

Architect

The first part of testing your app is understanding what and how to test. By the end of this stage, you'll have a plan that you can use to write your test scripts.

You've defined your app's goals; how will you determine whether you've achieved them? You've implemented a prototype; how do you expect it to be used? You'll define tests that will answer those questions, and you'll also take a step back to think about setting expectations — yours, and those of your users.

Define Tests

For each goal that users want to accomplish with your app, define the steps they'll take to accomplish it and describe any existing flow the user may be in.

Before you design your tests, you need to decide what's important to test. Your tests will teach you what users find useful, as well as how well you've designed your app. You'll also learn about the assumptions you made along the way.

If you select the right set of tests with your broad goals in mind, the results will help you draw clear conclusions about where you're on track and where you need to correct course.

Example

Users want to do this with our app:

Add rubbish or recycling weight to a day

The user will have needed to complete these steps first:

Have either rubbish or recycling to throw out

Users need to complete these critical tasks:

Classify whether the item is rubbish or recycling	
Estimate the weight of the item	
Submit the entry	

Define Tests

For each goal that users want to accomplish with your app, define the steps they'll take to accomplish it and describe any existing flow the user may be in.

Before you design your tests, you need to decide what's important to test. Your tests will teach you what users find useful, as well as how well you've designed your app. You'll also learn about the assumptions you made along the way.

If you select the right set of tests with your broad goals in mind, the results will help you draw clear conclusions about where you're on track and where you need to correct course.

Users want to do this with our app:

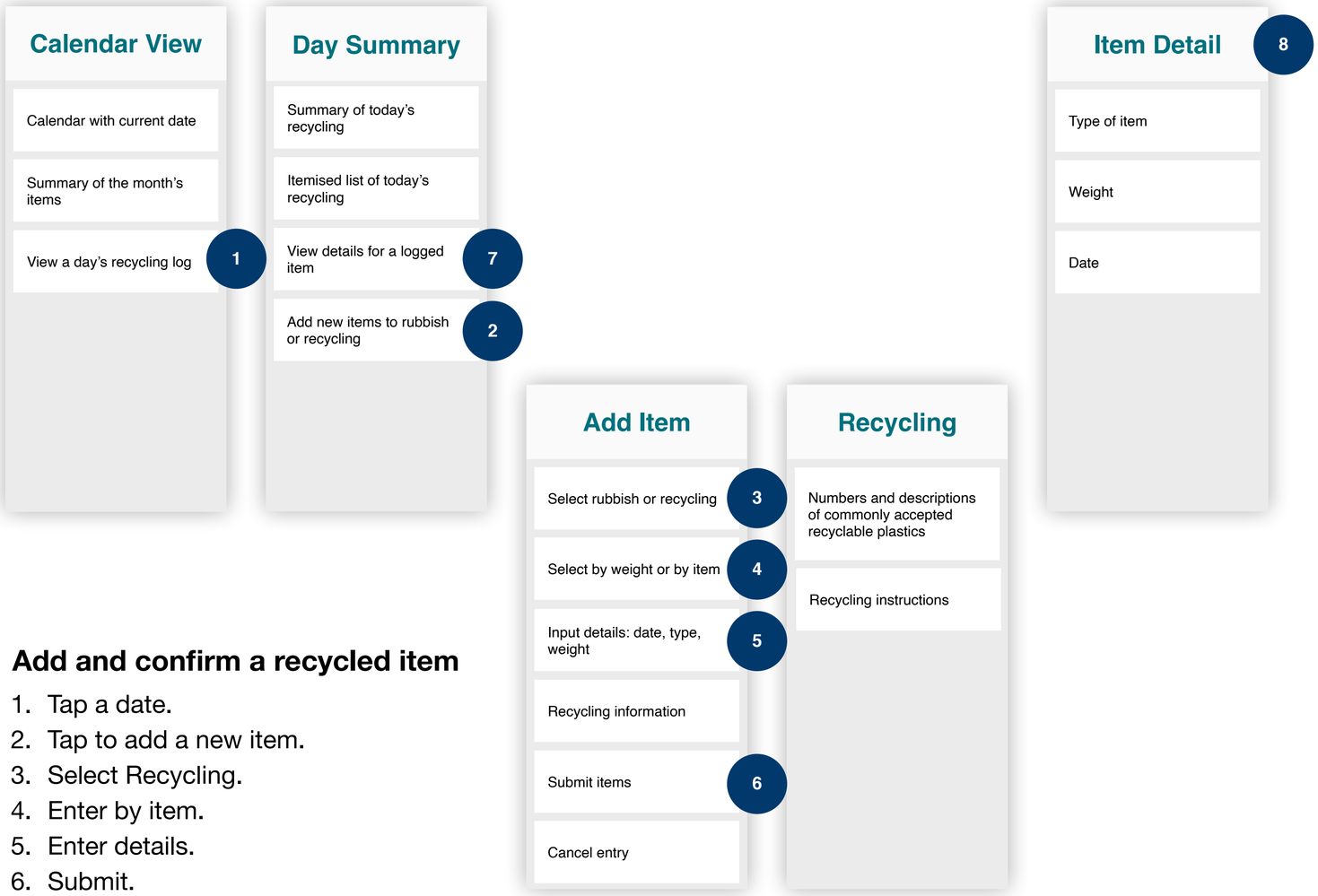
The user will have needed to complete these steps first:

Users need to complete these critical tasks:

Create User Journeys

For each feature you want to test, use a copy of your screen outline map to create a user journey map. Number and label each step from your previous exercises, attaching each to the screen that the user will be on.

A journey map visually represents what a user will need to do to complete a task. These might be interactions like taps or swipes, but they may also include taking a picture, speaking into the microphone, or bringing their device close to an object in the world (for example, an NFC reader).



Add and confirm a recycled item

1. Tap a date.
2. Tap to add a new item.
3. Select Recycling.
4. Enter by item.
5. Enter details.
6. Submit.
7. Choose the new item from the list.
8. View the details.

Define a Process

Summarise the requirements and parameters of your test.

Before you build the tests themselves, you should create a standardised process for how you'll test individual users. Having consistent data is critical to a successful analysis of the test results.

Example

We plan on testing these user journeys:

A list of the user journeys

We will record our findings using:

Paper, video, audio only, screen recording

We will show our prototype by:

Screen share, in-person

We will conduct the test at:

Time, Date, Location

We will need the following equipment:

MacBook, iPhone, chairs, desk, camera, tripod

This person will take notes/record:

John Appleseed

This person will conduct the user test:

Jane Macintosh

Define a Process

Summarise the requirements and parameters of your test.

Before you build the tests themselves, you should create a standardised process for how you'll test individual users. Having consistent data is critical to a successful analysis of the test results.

We plan on testing these user journeys:

We will record our findings using:

We will show our prototype by:

We will conduct the test at:

We will need the following equipment:

This person will take notes/record:

This person will conduct the user test:

Plan an Introduction

Create an introduction that you'll use with each participant.

You'll want to set the tone prior to each test so that all your participants have consistent expectations. By anticipating issues that might arise, both you and your test subjects will feel at ease, and you'll have the best chance to gather useful information.

Write a short introduction about yourself and your project, and explain the goals of your test.

Participants can be intimidated by the word 'test'. Make sure you let your user know that there are no wrong answers and any feedback is useful feedback. What else might make your participant feel uncomfortable? How might you mitigate those feelings?

If you're recording a participant, be sure to get their consent for audio and/or video. Let the participant know why recording is beneficial to your test. Consider what you'll do if they decline to consent.

Participants usually want to know how much time they need to commit to the testing session. How long do you think it will take for users to complete tasks and answer your follow-up questions?

Outline Scripts
Write Scripts
Anticipate Errors

Script

Now that you've planned your testing, it's time to focus on the details. By the end of this stage, you'll have a complete set of test scripts.

You'll define the flow of your tests to keep the user engaged and oriented, dig into the kinds of questions each test can answer and prepare for the unexpected.

Outline Scripts

Describe each test and the order in which they'll be conducted. Be sure to include any contextual information the user will need to complete the task successfully.

Your testing script should tell a story that the user can relate to. The order of tasks should create a natural flow that puts the user into the right frame of mind and keeps them engaged throughout the process. Where possible, put your tests in the order the user would encounter them in their everyday lives.

Remember that some tasks are more critical to test than others. What are the most important features to test? You might want to test those first in case you run out of time.

For tests that don't flow naturally from one to the next, it's especially important to provide context so you don't interrupt the flow of the script. Make sure you identify those situations and think carefully about how you'll keep the user focused and oriented.

Example

This is the sequence of tasks to test:	Context we should provide prior to the test:
Add recycling or rubbish to daily summary.	Give the participant a group of objects that are rubbish and recycling.
Discover information about recycling numbers.	
Check their achievements.	User should reach the achievement when they successfully add recycling.
Understand challenges to being more environmentally conscious.	

Outline Scripts

Describe each test and the order in which they'll be conducted. Be sure to include any contextual information the user will need to complete the task successfully.

Your testing script should tell a story that the user can relate to. The order of tasks should create a natural flow that puts the user into the right frame of mind and keeps them engaged throughout the process. Where possible, put your tests in the order the user would encounter them in their everyday lives.

Remember that some tasks are more critical to test than others. What are the most important features to test? You might want to test those first in case you run out of time.

For tests that don't flow naturally from one to the next, it's especially important to provide context so you don't interrupt the flow of the script. Make sure you identify those situations and think carefully about how you'll keep the user focused and oriented.

This is the sequence of tasks to test:

Context we should provide prior to the test:

Write Scripts

For each test, determine which questions and observations you'll use.

You can choose from different types of questions when building a testing script. You might use one or more of them for a given test, and you might ask them (more than once!) during a particular user journey.

Refer back to your user journeys; they'll help you decide when and how to gather your data.

Example

We want to understand more about:

> Will users be able to successfully sort and enter rubbish and recycling into the app?

We could answer this by having users complete a rank order of the screen.

> Looking at this screen, rank the items you see from most to least important for discarding rubbish or recycling.

We could gain insights for this by having users complete a task observation.

Your participant should be starting from a screen that they're most likely to be on when discarding rubbish and recycling. Observe and note where they're successful or unsuccessful completing parts of the task. To gather more information, ask what they think about completing the task.

> If you were to discard these items, how might you record that in the app?

We can dig deeper about why a user makes decisions by asking about expectations.

Your participant should look at the screen and talk through what they are seeing and what they expect each element will do. For example, if a participant mentions the 'plus icon', you can use it as an opportunity to ask more questions on how it might work.

> When looking at this (Today) screen, what do you expect you'll be able to complete on this screen?

Write Scripts

For each test, determine which questions and observations you'll use.

You can choose from different types of questions when building a testing script. You might use one or more of them for a given test, and you might ask them (more than once!) during a particular user journey.

Refer back to your user journeys; they'll help you decide when and how to gather your data.

We want to understand more about:

We could answer this by having users complete a rank order of the screen.

We could gain insights for this by having users complete a task observation.

Your participant should be starting from a screen that they're most likely to be on when discarding rubbish and recycling. Observe and note where they're successful or unsuccessful completing parts of the task. To gather more information, ask what they think about completing the task.

We can dig deeper about why a user makes decisions by asking about expectations.

Your participant should look at the screen and talk through what they are seeing and what they expect each element will do. For example, if a participant mentions the 'plus icon', you can use it as an opportunity to ask more questions on how it might work.

Anticipate Errors

Make a plan for what you'll do when the user gets stuck or asks you a question.

It's especially important not to lead the user through a test. You'll be most tempted to step in when something goes wrong. Your interactions can bias the user and rob you of important insights, so be sure you've planned for how to ask and answer questions to minimise their influence on the test.

Participants who get stuck completing a task will often ask you for help. It's important to dig deeper into why they're stuck. Ask things such as, "What do you expect it to do?" How might you get your user back on track without leading them? What questions will help you better understand why they're stuck in a task?

When participants are completing a task, there might be smaller elements that you can test. Things like icon recognition, text clarity and colour contrast can impact how the user completes the task. Consider what small tests can happen while a participant completes tasks.

When participants get quiet, they're usually figuring things out. You want them to talk through what they're experiencing. How might you gently remind them to talk through their thought process? Where might they need a moment to think during the test?

Prepare

You’re almost there! By the end of this stage, you’ll be ready to test your prototype.

The quality of your data depends on the users you test with, so it’s important to select them carefully. And you’ll want to make sure that you’re ready at the start to provide each participant with an enjoyable experience.

Gather Users

Make a list of the users you'll enrol in your testing, and plan a date and location for each one.

The information you gathered in the Discover phase will be beneficial during testing. Select users who are directly affected by the challenges you identified, and who would be most likely to use your app.

It takes at least three people to begin to see patterns in user tests, so be sure to enrol enough participants so that you can accommodate a cancellation or two.

Participant's name	Date and time	Location

Last Check

Use the checklist to double-check that you're ready to start testing.

Be sure that the testing script will run smoothly when talking to users. Use this checklist to complete a dry run of your testing script.

- [ ] Are all your questions nonleading?
- [ ] Can you run a question in your sketch or prototype smoothly?
- [ ] Do you have a plan for how you'll restart the task process if the user gives up?
- [ ] Does your testing script cover the features that are most important to the goal?
- [ ] Do you have a plan for where you'll conduct the testing? Will it be remote or in person?

Validate

You'll have a lot of information to digest after testing your prototype. It's important to summarise and draw the correct conclusions from your testing data so that you know how to improve your app. You'll start by formatting your data to make it digestible. Then you'll summarise your observations by discovering relationships between them. Then you'll zoom out to root causes and identify core issues.

Gather Notes

Create succinct notes from your observations for each participant.

After user testing, you'll have a lot of raw data. Before you draw any conclusions, it's important to convert it to a consistent format. Don't worry about how to organise or categorise it.

The more you can narrow your observations down to single, focused data points, the easier it will be to organise them.

This activity could involve transferring and splitting up written notes, summarising survey answers, or analysing video or audio.

You might need many notes for each participant. If so, consider organising the notes by task to be completed per participant.

Example

Participant:	Christina Ahmed			
Wanted to use the scale because she has one at home.	Able to enter rubbish and recycling.	Wished she could enter rubbish and recycling on the same screen.	Unclear on what classified something as recycling.	Didn't find it necessary to change the date. She wouldn't have entered something after the fact.
Would like to have seen possible achievements rather than just ones she's earned.	Unclear on what Rewards and Waterway mean.	Unsure where she would get an invite code from.	Would like to see which of her friends are participating rather than just the number of friends.	Likes that she can see both active and past challenges.

Gather Notes

Create succinct notes from your observations for each participant.

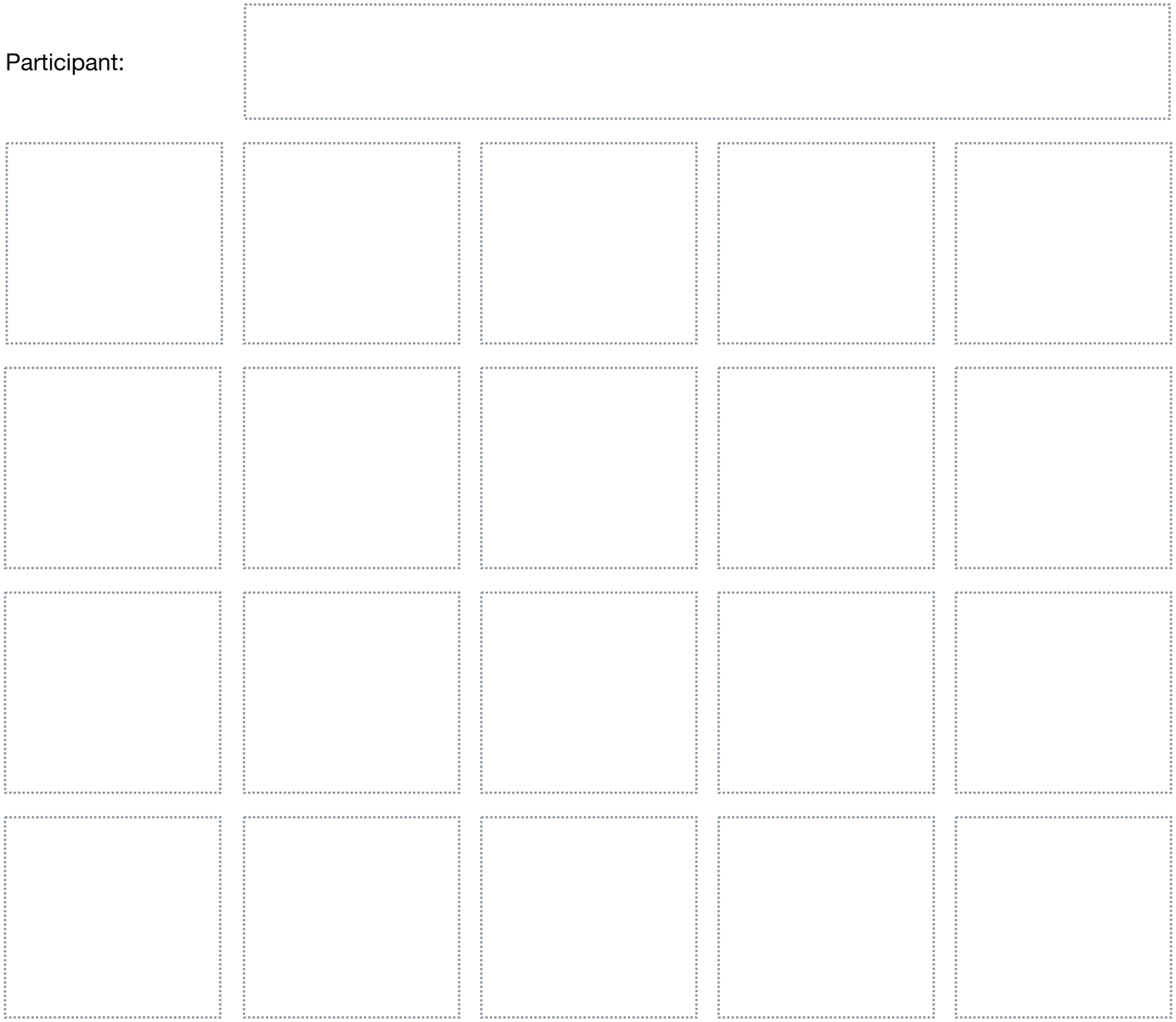
After user testing, you'll have a lot of raw data. Before you draw any conclusions, it's important to convert it to a consistent format. Don't worry about how to organise or categorise it.

The more you can narrow your observations down to single, focused data points, the easier it will be to organise them.

This activity could involve transferring and splitting up written notes, summarising survey answers, or analysing video or audio.

You might need many notes for each participant. If so, consider organising the notes by task to be completed per participant.

Participant:



Form Key Insights

Look through your notes and group similar ones. Summarise those groups as key insights into your users' behaviours.

An affinity diagram helps you visualise similarities across participants and identify patterns.

Once you start to see a pattern in your groupings, you can summarise those groups as key insights into user behaviour.

Don't analyse the reasons for their behaviour yet; just focus on finding themes.

Example

Jamil Green
Unclear on what classified something as recycling.

Karla Gonzalez
"I wonder what this ⓘ is for next to recycling?"

Karla Gonzalez
"Oh I wish that information was somewhere more visible."

Jody Akana
"I don't know if this is recycling or not … what does the 5 mean?"

Jody Akana
Threw out recycling. Unable to find information about recycling numbers.

This is our key insight:

Users are confused about whether an item is recyclable.

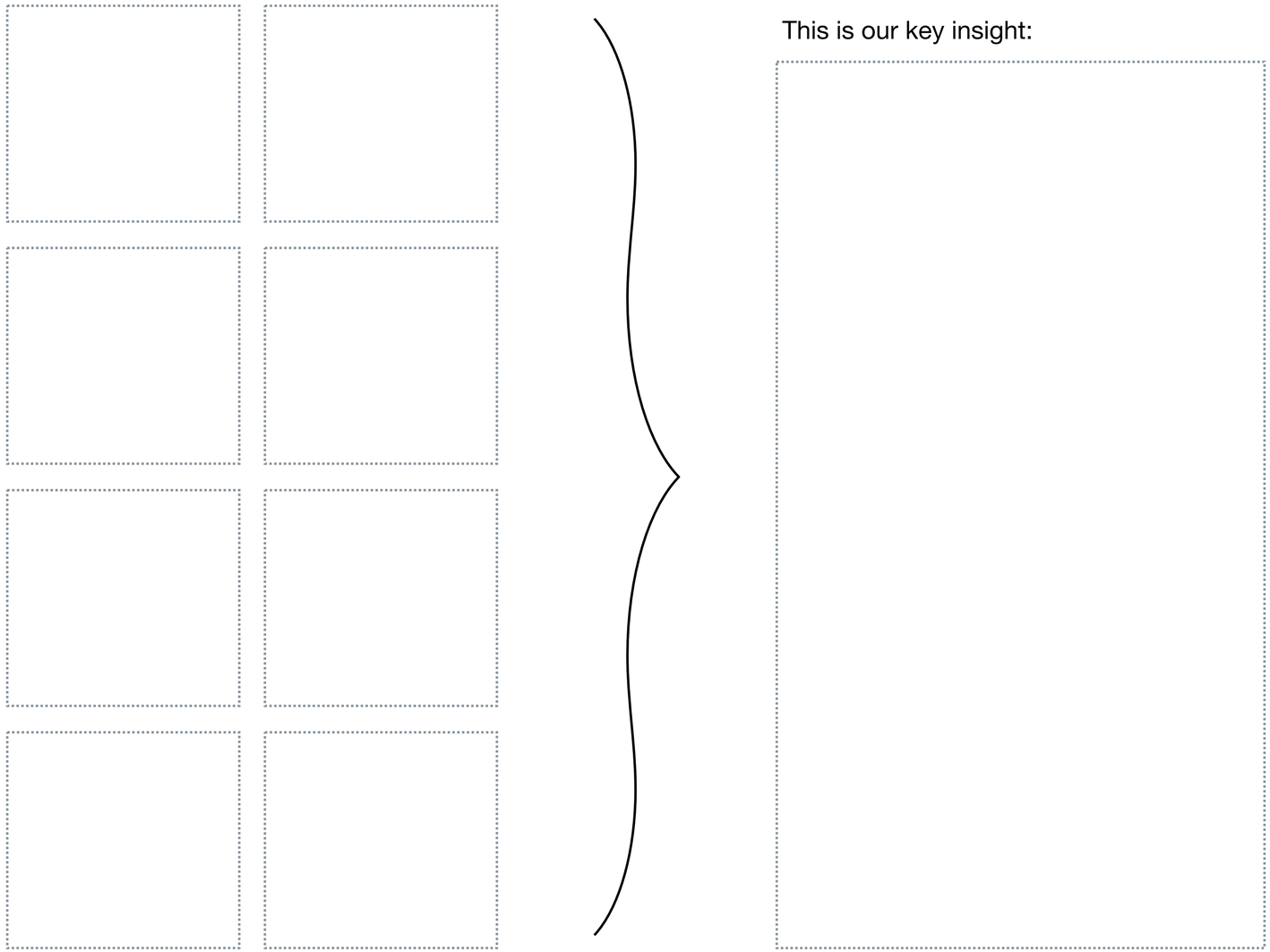
Form Key Insights

Look through your notes and group similar ones. Summarise those groups as key insights into your users' behaviours.

An affinity diagram helps you visualise similarities across participants and identify patterns.

Once you start to see a pattern in your groupings, you can summarise those groups as key insights into user behaviour.

Don't analyse the reasons for their behaviour yet; just focus on finding themes.



Draw Conclusions

For each task, summarise your key insights and review them in light of your assumptions to draw overall conclusions.

The final phase of analysing your results is to look for big patterns. Identifying these patterns will guide you towards decisions about iterating on your design.

When drawing conclusions from key insights, keep things general. Focus on root causes rather than particular issues.

Example

We tested this task:

Do users discover the information about what's considered recycling?

We expected users would complete the task by:

Tapping the ⓘ icon to reveal more information.

We observed this instead:

Users are confused about whether an item is recyclable.

We've made these conclusions:

The ⓘ icon isn't well understood.

Users want a more prominent reminder of what the recycling numbers mean.

Users have difficulty remembering recycling numbers.

It's important to also tell users where to find recycling numbers on products.

Draw Conclusions

For each task, summarise your key insights and review them in light of your assumptions to draw overall conclusions.

The final phase of analysing your results is to look for big patterns. Identifying these patterns will guide you towards decisions about iterating on your design.

When drawing conclusions from key insights, keep things general. Focus on root causes rather than particular issues.

We tested this task:

We expected users would complete the task by:

We observed this instead:

We've made these conclusions:

Iterate

Look closely at your first prototype and you'll see a world-changing app beginning to take form. Now comes the critical phase of any design — working towards that vision by applying all you've learned during your design process. You'll use the conclusions from your analysis as a guide to re-evaluate choices you made throughout your app design journey. Then you'll revisit different elements of your design, looking for opportunities to make improvements large and small.

Organise Your Conclusions

Before you use your conclusions to revisit steps in your design process, group similar conclusions. Seeing important themes will help you decide where to focus your efforts.

Your conclusions may range from surfacing important content to making it easier to navigate the app and reducing visual clutter.

You might also discover that you've misunderstood your users, or you've omitted a critical element of your test plan.

Example

Interface elements

The ⓘ icon isn't well understood.

It's too hard to find the button to add a new item.

Graphs and charts don't have enough contrast.

Navigation

The flow for adding a new recycling item requires too many actions.

Adding multiple items requires the user to repeat the same flow for each one.

Wording

Many users don't understand the phrase 'Recycling Number'.

Go Back: Define

If your groupings include the following topics, consider revisiting these elements of Define.

If you heard comments like these:	Revisit these sections in Define
Your competitor did a feature better	Competitor Analysis, Differentiator
Needing to use two hands	Diversity
Issue not being solved	Observe, Analyse Causes, Prioritise Features
Wished for a different feature	Observe, Cause and Effect, Goal Statement

Go Back: Prototype

If your groupings include the following topics, consider revisiting these parts of Prototype.

If you heard comments like these:	Revisit these sections in Prototype
Unable to understand a task	Content, Grouping Outlines, Linking Outlines
Unable to read content	Style, Hierarchy
Unable to complete a task	Global Navigation, Navigation Bar, Modals, Elements
A feature not being recognised	Feature Creep, Content
Disconnect of what the app icon means	App Icon
Didn’t understand categories	Content, Grouping Outline, Linking Outlines

Go Back: Test

If your groupings include the following topics, consider revisiting these parts of Test.

If you heard comments like these:	Revisit these sections in Test
Confused as to what you want them to do	Planning, Questions
I know that because you called it this	Nonleading Questions, Questions
Script didn’t match thought process	User Journey, What to Test

Learn to Code with Apple.

You don’t need prior experience to dive straight into creating apps for Apple platforms. Apple’s app development curriculum makes it easy for anyone to code in Swift just like the pros — whether it’s for a term in school, for professional certification or to advance your skills. Learn more at developer.apple.com/learn/curriculum.

App Showcase Guide

Demonstrate your ingenuity by sharing your achievements with community events, such as project demonstration events or app showcases. The App Showcase Guide provides practical support to help you host an in-person or virtual app showcase event. Download: apple.com/au/education/docs/app-showcase-guide-AU.pdf

Swift Coding Club

Swift Coding Clubs are a fun way to design apps. Activities are built on learning Swift programming concepts in Xcode playgrounds on Mac. Collaborate with peers to prototype apps and think about how code can make a difference in the world around you. Download: apple.com/au/education/docs/swift-club-xcode-AU.pdf