SwiftUI Essentials ~ Landmarks iOS App[1]

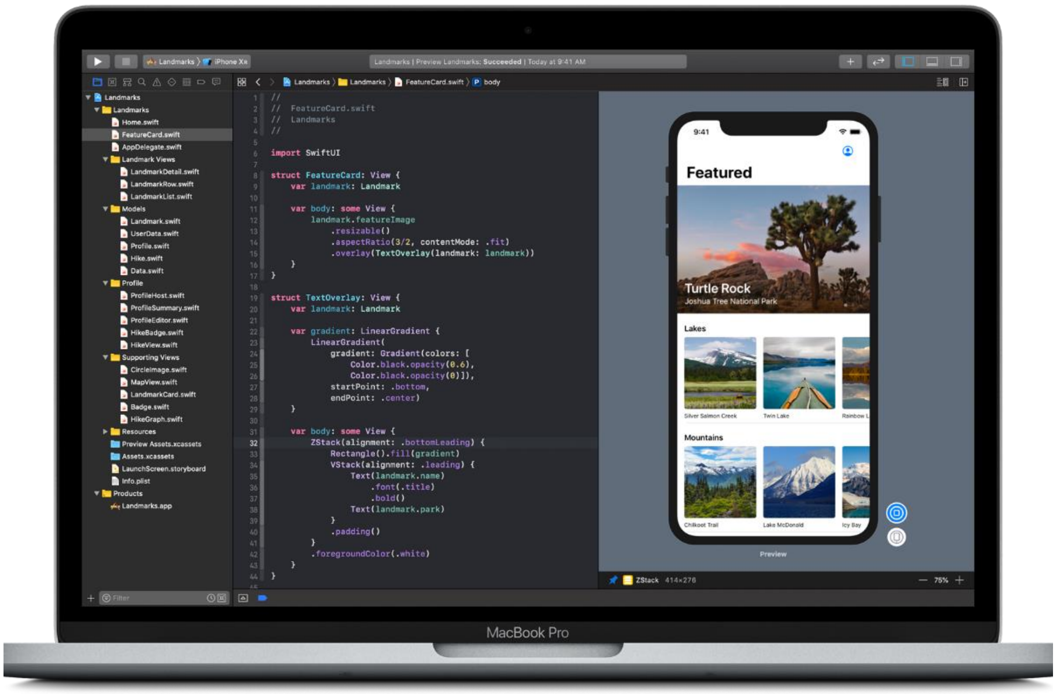
Creating and Combining Views

This lab will guide you through building *Landmarks* - an iOS app for discovering and sharing the places you love. You will start by building the view that shows a landmark's details.

To lay out the views, Landmarks uses *stacks* to combine and layer the image and text view components. To add a map to the view, you will include a standard MapKit component. As you refine the view's design, Xcode provides real-time feedback so you can see how those changes translate into code.

NOTE: At the end of the lab instructions there will be some questions which need to be answered and included in the final zip file you will upload onto NextCloud ~ you can enclose the answer on the same pdf or creating a new text (textEdit, Word or others) document with the answers.

Estimated completion time ~ 40min



```
//
//  FeatureCard.swift
//  Landmarks
//

import SwiftUI

struct FeatureCard: View {
    var landmark: Landmark

    var body: some View {
        landmark.featureImage
            .resizable()
            .aspectRatio(3/2, contentMode: .fit)
            .overlay(TextOverlay(landmark: landmark))
    }
}

struct TextOverlay: View {
    var landmark: Landmark

    var gradient: LinearGradient {
        LinearGradient(
            gradient: Gradient(colors: [
                Color.black.opacity(0.6),
                Color.black.opacity(0)]),
            startPoint: .bottom,
            endPoint: .center)
    }

    var body: some View {
        ZStack(alignment: .bottomLeading) {
            Rectangle().fill(gradient)
            VStack(alignment: .leading) {
                Text(landmark.name)
                    .font(.title)
                    .bold()
                Text(landmark.park)
            }
            .padding()
        }
        .foregroundColor(.white)
    }
}
```

Landmarks | Preview Landmarks: Succeeded | Today at 9:41 AM
Featured
Turtle Rock
Joshua Tree National Park
Lakes
Mountains
Preview
MacBook Pro

Landmarks - iOS App built with SwiftUI

[1] https://developer.apple.com/tutorials/swiftui/creating-and-combining-views

Section 1

Create a New Project and Explore the Canvas

Create a new Xcode project that uses SwiftUI. Explore the canvas, previews and the SwiftUI template code.

To preview and interact with views from the canvas in Xcode, ensure your Mac is running macOS Catalina 10.15.

Step 1

Open Xcode and either click **Create a new Xcode project** in Xcode's startup window, or choose **File > New > Project**.

Step 2

In the template selector, select **iOS** as the platform, select the **App** template, and then click **Next**.

Step 3

Enter “Landmarks” as the Product Name, select **SwiftUI** for the user interface, enter “fr.eurecom” as Organization identifier, and kick **Next**. Choose a location to save the Landmarks project on your Mac.

Step 4

In the project navigator, select *ContentView.swift*. By default, SwiftUI view files declare two structures. The first structure conforms to the *View* protocol and describes the view's content and layout. The second structure declares a preview for that view.

Step 5

In the canvas, click **Resume** to display the preview.

Tip

If the canvas is not visible, select **Editor > Editor and Canvas** to show it.

Step 6

Inside the body property, change “Hello World” to a greeting for yourself. As you change the code in a view's body property, the previews updates to reflects your changes.

Section 2

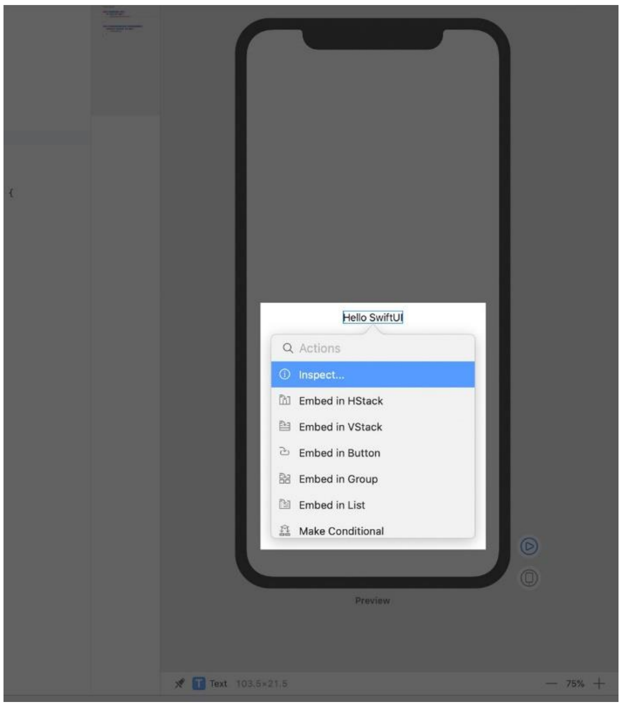
Customise the Text View

You can customise a view’s display by changing your code, or by using the inspector to discover what’s available and to help you write code.

As you build the Landmarks app, you can use any combination of editors: the source editor, the canvas or the inspectors. Your code stays updated, regardless of which tool you use.

Step 1

In the preview, *Command-click* the greeting to bring up the structured editing popover, and choose **Inspect**. The popover shows different attributes that you can customise, depending on the type of view you inspect.



Step 2

Use the inspector to change the text to “Turtle Rock”, the name of the first landmark you’ll show in your app.

Step 3

Change the Font modifier to **Title**. This applies the system font to the text so that it responds correctly to the user’s preferred font sizes and settings.

Tip

To customise a SwiftUI view, you call methods called *modifiers*. Modifiers wrap a view to change its display or other properties. Each modifiers returns a new view, so it's common to chain multiple modifiers, stacked vertically.

Step 4

Edit the code by hand to add the foreground *Color(.green)* modifier; this changes the text's color into green.

FYI

Your code is always the source of truth for the view. When you use the inspector to change or remove a modifier, Xcode updated your code immediately to match.

Step 5

This time open the inspector by *Command-clicking* on the *Text* declaration in the code editor, and then choose **Inspect** from the popover. Click the **Color** pop-up menu and choose **Inherited** to change the text color to black again.

Step 6

Xcode updates your code automatically to reflect the change, removing the *foreground Color(.green)* modifier.

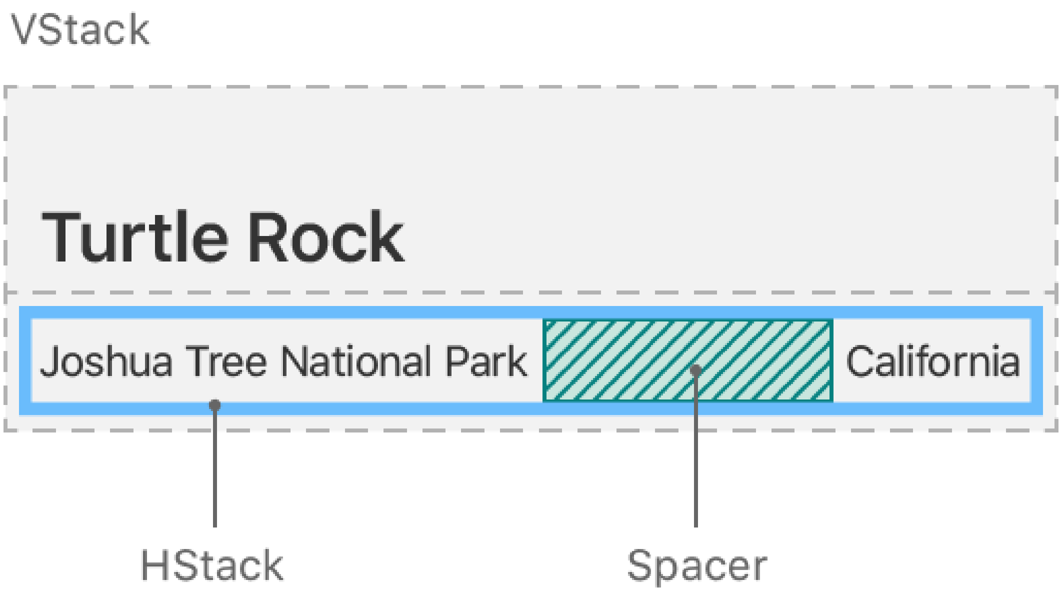
Section 3

Combine Views Using Stacks

Beyond the title view you created in the previous section, you'll add text views to contain details about the landmark, such as the name of the park and state it is in.

When creating a SwiftUI view, you describe its content, layout and behaviour in the view's body property; however, the body property only returns a single view. You can combine and embed multiple views in *stacks*, which group views together horizontally, vertically, or back-to-front.

In this section, you will use a vertical stack to place the title above a horizontal stack that contains details about the park.



You can use Xcode's structured editing support to embed a view in a container view, open an inspector, or help with other useful changes.

Step 1

Command-click the text view's *initialiser* to show the structured editing popover, and then choose **Embed in VStack**.

Step 2

You will add a text view to the stack by dragging a *textView* from the library. Open the library by clicking the plus button (+) at the top-right of the Xcode window, and then drag a Text view to the place in your code immediately after the *"Turtle Rock"* text view.

Step 3 & 4

Replace the Text view’s placeholder with **Joshua Tree National Park** and set the location’s font to *subheadline*.

Step 5

Edit the VStack initialiser to align the views by their leading edges. By default, stacks center their contents along their axis and provide context-appropriate spacing.

You will now add another text view to the right of the location, this for the parks state.

Step 6

In the canvas, Command-click **Joshua Tree National Park**, and choose **Embed in HStack**.

Step 7

Add a new text view after the location, change the placeholder text to the park’s state, and then set its font to subheadline.

Step 8

To direct the layout to use the full width of the device, separate the park and the state by adding a *Spacer* to the horizontal stack holding the two text views. A *Spacer* expands to make its containing view use all of the space of its parent view, instead of having its size defined only by its contents.

Step 9

Finally, use the *padding()* modifier method to give the landmark’s name and details a little more space.

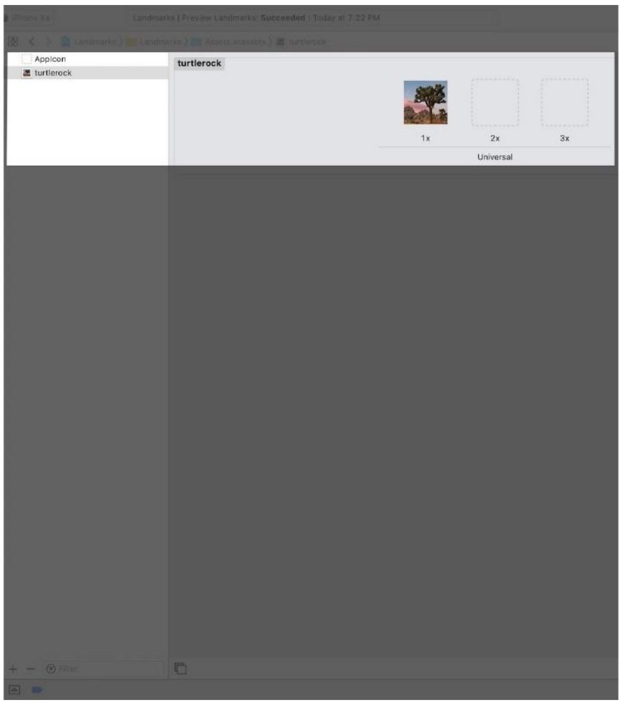
Section 4

Create a Custom Image View

With the name and location views all set, the next step is to add an image for the landmark. Instead of adding more code in this file, you'll create a custom view that applies a mask, border, and drop shadow to the image.

Step 1

Find *turtlerock.png* in the **Resources** folder; drag it into the asset catalog's editor. Xcode creates a new image set for the image.



Step 2

Choose **File > New > File** to open the template selector again. In the **User Interface** section, select **SwiftUI View** and click **Next**. Name the file *CircleImage.swift* and click **Create**.

Step 3

Replace the text view with the image of Turtle Rock by using the *Image(_:)* initialiser.

Step 4

Add a call to clip *Shape(Circle())* to apply the circular clipping shape to the image. The *Circle* type is a shape that you can use as a mask, or as a view by giving the circle a stroke or fill.

Step 5

Create another circle with a grey stroke, and then add it as an overlay to give the image a border.

Step 6

Add a shadow with a 10 point radius.

Step 7

Switch the border color to white. This completes the image view.

Section 5

Use UIKit and SwiftUI Views Together

Now you are ready to create a map view. You can use the *MKMapView* class from *MapKit* to render the map.

To use *UIView* subclasses from within SwiftUI, you wrap the other view in a SwiftUI view that conforms to the *UIViewRepresentable* protocol. SwiftUI includes similar protocols for WatchKit and AppKit views.

To get started, you will create a new custom view that can present an *MKMapView.*

Step 1

Choose **File > New > File**, select **iOS** as the platform, select the **SwiftUI View** template, and click **Next**.

Name the new file *MapView.swift* and click **Create**.

Step 2

Add an import statement for MapKit, and declare UIView Representable conformance for the MapView type.

Note

There will be an error, but don’t panic, you will fix it in the next steps.

Step 3

Replace the body property with a make *UIView(context:)* method that creates and returns an empty *MKMapView*.

Step 4

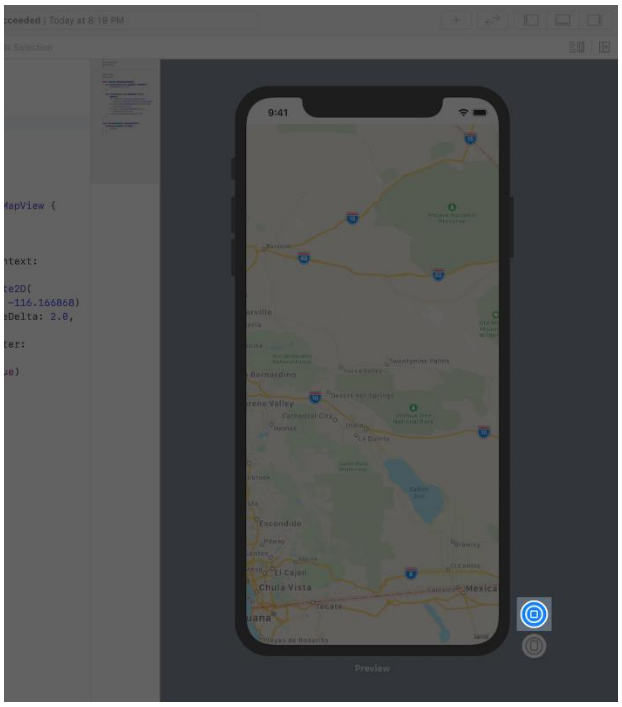
Create an update *UIView(_:context:)* method that sets the map view’s region to the correct coordinates to center the map on Turtle Rock.

Step 5

Tip

The previews are in static mode, they only fully render SwiftUI views. Because MKMapView is a UIView subclass, you will need to switch to a live preview to see the map

Click the **Live Preview** button to switch the preview to live mode. You might need to click the **Try Again** or **Resume** button above your preview. You will then be able to see a map of Joshua Tree National Park, home of Turtle Rock.

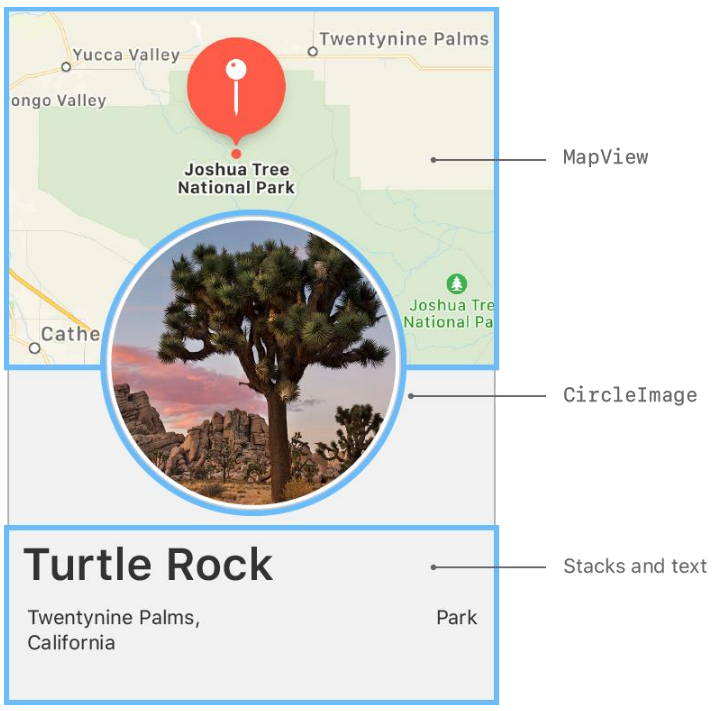


Section 6

Compose the Detail View

You now have all of the components you need - the name and place, a circular image and a map for the location.

With the tools you've used so far, combine your custom views to create the final design for the landmark detail view.



In this section you will be on your own - based on the previous step, the image above represents the final result you should obtain. As you can see it consists of a **VStack** embedded in another **VStack** where you will add the elements you previously created in order to have them all together in one view.

Once you are done with that, please write all the steps you followed in order to reach the final result, below or in another text document, attached to the final code.

After you are done with that there are 4 short questions in the next page, in order to check your understandings.

Section 7

Check Your Understanding

Question 1 of 4

When creating a custom SwiftUI view, where do you declare the view's layout?

☐ In the view's initialiser.

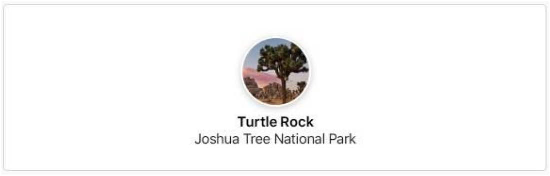
☐ In the body property.

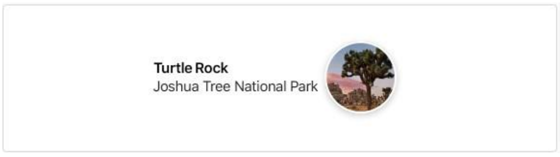
☐ In the layoutSubviews() method

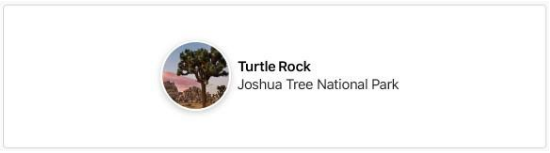
Question 2 of 4

Which layout renders from the following view code?

```
var body: some View {
    HStack {
        CircleImage()
        VStack(alignment: .leading) {
            Text("Turtle Rock")
                .font(.title)
            Text("Joshua Tree National Park")
        }
    }
}
```

☐ 

☐ 

☐ 

Question 3 of 4

Which of these is a correct way to return three views from a custom view's body property?

☐

```
VStack {
    Text("Turtle Rock")
        .font(.title)
    Divider()
    Text("Joshua Tree National Park")
}
```

☐

```
[
    Text("Turtle Rock").font(.title),
    Divider(),
    Text("Joshua Tree National Park")
]
```

☐

```
Text("Turtle Rock")
    + Divider()
    + Text("Joshua Tree National Park")
```

Question 4 of 4

Which is the correct way to use modifier methods to configure a view?

☐

```
var text = Text("Hello world!")
text.font(.title)
text.foregroundColor(.purple)
return text
```

☐

```
var text = Text("Hello world!")
text.font = .title
text.foregroundColor = .purple
return text
```

☐

```
Text("Hello world!")
    .font(.title)
    .foregroundColor(.purple)
```