ATLS 4320: Advanced Mobile Application Development
Week 6: Split Views

In the beginning there was the iPhone. Then in 2010 the first iPad was released and it became clear that having a single table view take up the entire screen on the larger screen wasted space.

In iOS 3.2 the split view controller was introduced so two views could be displayed side-by-side on the larger iPad in landscape orientation. As larger iPhones were released, starting in iOS8 these side-by-side views are displayed on all devices when the width has a regular size class.

- Width of regular size class has the navigation column fixed position on the left, with the content of the selected item displayed on the right in a split-view
- All other size classes will display one controller at a time with a navigation button to go from the secondary view back to the primary view
- Apple apps that use a split view include settings and mail

Split Views
A split view manages the presentation and navigation of hierarchical content in an app when there are multiple levels of content. The two column presentation has been updated in iOS 14 to allow for double or triple columns. https://developer.apple.com/design/human-interface-guidelines/ios/views/split-views/

- Double column: primary/secondary views
- Triple column: primary/supplementary/secondary views

Changing one controller drives changes in the content of another

- The primary view often has persistent content and can act as a sidebar-based interface
- Supplementary and/or secondary views show details about the selected content
    - Make sure users stay oriented and discern the relationship between the panes

You can specify custom widths for the primary and supplementary columns in a split view

- .**preferredPrimaryColumnWidthFraction** for the relative width of the primary view
    - The primary pane should be narrower than the secondary pane
- .**preferredSupplementaryColumnWidthFraction** for the relative width of the primary view
- Default is .**automaticDimension** and the system determines the appropriate behavior based on the available space.
- Avoid creating a secondary pane that's narrower than the primary pane

A **UISplitViewController** is a container that manages child view controllers in a hierarchical interface. https://developer.apple.com/documentation/uikit/uisplitviewcontroller

- Split view controllers should be the root view controller for an app and can't be pushed onto a navigation stack.
- Primary pane (primary): persistent information
- Secondary pane (detail): related information to the selected item in the primary pane
- The panes in a split view can contain any other type of controller.
- The **viewControllers** property is an array that stores the two controllers.

The **UISplitViewControllerDelegate** protocol manages the presentation of the child view controllers. https://developer.apple.com/documentation/uikit/uisplitviewcontrollerdelegate

Display Mode

A split view controller's current display mode represents the visual arrangement of its child view controllers. It determines how many of its child view controllers are shown, and how they're positioned in relation to each other.

This is controlled by the displayMode property of type UISplitViewControllerDisplayMode (enum) https://developer.apple.com/documentation/uikit/uisplitviewcontroller/displaymode

- The display mode will be set automatically or you can change the preferredDisplayMode
- The split view controller will try to respect the display mode you specify but may not be able to accommodate that mode because of space constraints.
    - Split view controllers use the size class to decide how to arrange its child view controllers
    - Regular width: tries to display both panes side by side
        - iPad and iPhone plus/max landscape orientation
    - Compact width: only the secondary pane is displayed. A navigation button reveals and hides the primary pane. The primary pane is layered on top of the secondary pane when visible.
        - All other iPhones and iPad and iPhone plus/max in portrait orientation
- The split view controller includes a bar button item in the navigation bar for changing the display mode
- The split view controller also installs a gesture recognizer so the user can change the display mode by swiping

Build for iPad, WWDC 2020 1:41- 6:57 New multi-column split view in iOS 14
https://developer.apple.com/videos/play/wwdc2020/10105/

Popovers
A popover is a temporary view that appears above other content onscreen when you tap a control or in an area such as the bar button item in a split view. https://developer.apple.com/design/human-interface-guidelines/ios/views/popovers/

- Popovers can contain any view, but in a split view regular width size class (portrait) it will contain the hidden left pane.
- The share functionality we implemented in our last app uses a popover
- Use a popover to show options or information related to the content onscreen.
- When a popover is visible, interactions with other views are normally disabled until the popover is dismissed. https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/modality/
    - A nonmodal popover is dismissed by tapping another part of the screen or a button on the popover.
    - A modal popover is dismissed by tapping a Cancel or other button on the popover.
    - You can also dismiss the popover programatically
- UIPopoverPresentationController class (added in iOS8) https://developer.apple.com/documentation/uikit/uipopoverpresentationcontroller
    - In nearly all cases, you use this class as-is and do not create instances of it directly.
    - UIKit creates an instance of this class automatically when you present a view controller using the UIModalPresentationPopover style.
    - Can also use the UIModalPresentationPopover style to configure a UIViewController as a popover in a horizontally regular environment
- UIPopoverPresentationControllerDelegate (added in iOS8) https://developer.apple.com/documentation/uikit/uipopoverpresentationcontrollerdelegate
    - Lets you customize the behavior of popovers

Multitasking
Multitasking lets users quickly switch from one app to another at any time through a multitasking interface on an iOS device, or by using a multifinger gesture on an iPad.
https://developer.apple.com/design/human-interface-guidelines/ios/system-capabilities/multitasking/

- Slide-over lets users overlay an app to briefly interact with it without leaving the app they're currently using.
- Split-view lets users display and use two apps side-by-side
    - Users can view, resize, and interact with both apps
- Picture in picture lets users watch a video while working in another app.

If for some reason you want to opt out of allowing your app from being presented in slide over or split view configurations you need to set the UIRequiresFullScreen key to YES in your targets plist file (or check the Requires full screen box in the General tab of the target settings)

potter
File | New Project
iOS App
Product Name: potter
Team: None
Org identifier: ATLAS (can be anything, will be used in the bundle identifier)
Interface: Storyboard
Life Cycle: UIKit App Delegate
Language: Swift
Uncheck core data and include tests.
Uncheck create local git repo

Storyboard
Delete the initial view controller scene in the storyboard.
Add a split view controller from the object library and check Is Initial View Controller.
This adds several scenes.

- A split view controller as the container view controller
    - Make sure you've made this your initial view controller
    - For style you can chose double or triple column, we'll be using double
        - Unspecified can be used to support previous versions (not recommended)
    - Includes a left bar button in the navigation bar which will hide/show the primary view controller
- A navigation controller with a relationship segue to a table view controller for the primary view controller (left side)
- A secondary view controller (right side)
- Relationship segues from the split view controller to the navigation controller and secondary view controllers
- The datasource and delegate have also been set for the table view

Build and run the app on an iPad simulator. Rotate your simulator to landscape. You should see an empty split view controller.

Now run it on any iPhone simulator except a plus-sized phone. You'll see that it starts showing the secondary view in full screen. It'll also allow you to tap the back button on the navigation bar to pop back to the primary view controller.

On iPhones other than the large-sized Plus or Max devices in landscape, a split view controller will use a navigation controller pushing and popping a primary view controller and secondary view controller back and forth. This is built-in functionality and requires very little extra configuration.

View Controllers
Now we need view controller classes to control these views.
Delete ViewController.swift (and move to trash)
Create a new Cocoa Touch class called PrimaryViewController and subclass UITableViewController.
Create a new Cocoa Touch class called SecondaryViewController and subclass UIViewController.

Back in the storyboard in the identity inspector change the class of the primary table view scene to PrimaryViewController.
Also select the table view cell and give it the reuse identifier "Cell".
Change the primary view controller title to "Characters" instead of it saying Root View Controller.
Select the detail view controller scene and in the identity inspector change the class to SecondaryViewController.

Data
Drag in the harrypotter2.plist file we'll be using and remember to check Copy items if needed.
Look at the plist to understand what's in it and what the key/value pairs look like.

Now let's add a model class to represent this data.
File | New | File | Swift File
Character
Make sure the potter target is checked.

We'll create a struct to represent this data model and make it conform to the Decodable protocol so we can use a PropertyListDecoder instance to decode the plist.

```
struct Character: Decodable{
    let name : String
    let url : String
}
```

Now we'll add a class to load our data and handle an instance of our data model. This is similar to our previous apps.

```
class DataHandler{
    var allData = [Character]()

    func loadData(filename: String){
        if let pathURL = Bundle.main.url(forResource: filename,
withExtension: "plist"){
            //creates a property list decoder object
            let plistdecoder = PropertyListDecoder()
            do {
```

```
                let data = try Data(contentsOf: pathURL)
                //decodes the property list
                allData = try plistdecoder.decode([Character].self, from:
data)
            } catch {
                // handle error
                print(error)
            }
        }
    }

    func getCharacters() -> [String]{
        var characters = [String]()
        for character in allData{
            characters.append(character.name)
        }
        return characters
    }

    func getURL(index:Int) -> String {
        return allData[index].url
    }
}
```

In PrimaryViewController we need an instance of the DataHandler class to load and access our data, an array for the list of characters, and a constant for our file name.

```
var characters = [String]()
var characterData = DataHandler()
let dataFile = "harrypotter2"
```

In viewDidLoad we need to load our data and get the list of characters.

```
        characterData.loadData(filename: dataFile)
        characters=characterData.getCharacters()
```

Now let's update the table view data source methods.

```
    override func numberOfSections(in tableView: UITableView) -> Int {
        // #warning Incomplete implementation, return the number of sections
        return 1
    }

    override func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
        // #warning Incomplete implementation, return the number of rows
        return characters.count
    }
```

Uncomment and update

```
    override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
```

```
        let cell = tableView.dequeueReusableCell(withIdentifier: "Cell",
for: indexPath)
        cell.textLabel!.text = characters[indexPath.row]
        return cell
    }
```

Run the app on an iPad and rotate to landscape and you'll see the characters in the table view in the primary view. In portrait orientation or on an iPhone you can tap the Characters button to see the list of characters (in a popover on iPad)

Secondary View

In the storyboard select the cell in the Character table and connect it to the secondary view controller with a Selection Segue – Show Detail segue.

- A show Detail segue is used in a splitViewController to replace the secondary view with content based on a selection in the primary view.
- Different from a show segue as the primary is sometimes still visible (iPad landscape) and there's no ability to navigate back

Select your new segue and in the attributes inspector give it the identifier detailsegue.
We're not going to set the title for this view in the storyboard we'll do it programmatically.

Right now the secondary view controller is embedded in the primary view controller's navigation controller, which we don't want. So select the secondary view controller and embed it in a navigation controller to address this. This gets rid of the back button because it's putting the secondary view controller in its own stack.

Now let's get the detail view working.
In the storyboard add a webkit view that fills up the rest of the view and make an outlet connection called webView. (make sure you don't use the deprecated web view)
In the attributes inspector make sure the View Content Mode is set to Scale to Fill.
Use constraints so the webkit view fills up the whole view.

Web Views

Web views load and display web content embedded in your app. This is useful when you want to briefly access web content within your app without having users sent to Safari. If you need the full functionality of Safari however, you should open Safari with the web content.
https://developer.apple.com/design/human-interface-guidelines/ios/views/web-views/

In iOS8 Apple introduced a new class, WKWebView, to display web content (the older UIWebView has been deprecated).
WKWebView is run in a separate process from your app so that it can draw on native Safari JavaScript optimizations. This means WKWebView loads web pages faster and more efficiently than UIWebView, and also doesn't have as much memory overhead.
https://developer.apple.com/documentation/webkit/wkwebview

- must import WebKit (not part of UIKit)
- load(_:) loads a URLRequest
- stopLoading() stops loading all resources on the current page
- the isLoading property returns a Boolean value indicating whether the view is currently loading content

- navigation is disable by default but can be enabled to navigate the history using buttons or gestures

The WKUIDelegate class provides methods for presenting native user interface elements in a web page.

The WKNavigationDelegate protocol handles behaviors triggered during a web page request
https://developer.apple.com/documentation/webkit/wknavigationdelegate
- webView(_:didStartProvisionalNavigation:) is called when a web page starts to load
- webView(_:didFinish:) is called called when a web page finishes
- methods for handling errors during loading a web page or navigation
- WKWebView navigationDelegate property

The UIActivityIndicatorView is a spinner that indicates a task in in process
https://developer.apple.com/documentation/uikit/uiactivityindicatorview
- startAnimating() starts the animation
- stopAnimating() stops the animating

In SecondaryViewController.swift you need to import WebKit.
```
import WebKit
```

We need a variable to store the url for the Wikipedia page passed to this view.
```
var webpage : String?
```

We need a method to load the web page.
```
    func loadWebPage(_ urlString: String){
        //the urlString should be a propery formed url
        //creates a URL object
        let myurl = URL(string: urlString)
        //create a URLRequest object
        let request = URLRequest(url: myurl!)
        //load the URLRequest object in our web view
        webView.load(request)
    }
```

Call loadWebPage() from viewWillAppear()
```
    override func viewWillAppear(_ animated: Bool) {
        if let url = webpage {
            loadWebPage(url)
        }
    }
```

In PrimaryViewController.swift we will need to implement prepare(for segue:) to pass the URL to the detail view controller.

```
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "detailsegue" {
            if let indexPath = self.tableView.indexPathForSelectedRow {
                let url = characterData.getURL(index: indexPath.row)
                let name = characters[indexPath.row]
                let detailVC = (segue.destination as!
```

```
UINavigationController).topViewController as! SecondaryViewController
                detailVC.webpage = url
                detailVC.title = name
            }
        }
    }
```

You should now be able to select a character and see their Wikipedia page in the detail view.

I didn't like how the secondary view started out blank, especially when the width was the compact size class so I decided to load the Wiki page for the Harry Potter novels as the default by updating viewDidLoad().

```
loadWebPage("https://en.wikipedia.org/wiki/Harry_Potter")
```

Activity Indicator

It's good practice to always provide an indicator to the user when something is happening.
Add an activity indicator in the middle of the web view and connect it as an outlet called webSpinner
Make the style medium and check Hide When Stopped and uncheck Animating (we'll control it programmatically).
For the activity indicator set constrains to align horizontal and vertical centers.
Make sure in the document outline that the activity spinner is below the web view or it will be hidden behind it.
Go into SecondaryViewController.swift and add the `WKNavigationDelegate` protocol.
We need to set the navigation delegate, we can do that in viewDidLoad()

```
webView.navigationDelegate = self
```

Then add the two delegate methods that get called when a web page starts and finishes loading, that's where we'll start and stop the activity indicator.

```
 //WKNavigationDelegate method that is called when a web page begins to load
    func webView(_ webView: WKWebView, didStartProvisionalNavigation
navigation: WKNavigation!) {
        webSpinner.startAnimating()
    }

//WKNavigationDelegate method that is called when a web page loads
successfully
    func webView(_ webView: WKWebView, didFinish navigation: WKNavigation!)
{
        webSpinner.stopAnimating()
    }
```

Another useful delegate method is `webView(WKWebView, didFailProvisionalNavigation: WKNavigation!, withError: Error)` which is called if the web page doesn't load (such as no internet connection). We won't be implementing it in this app.