ATLS 4320: Advanced Mobile Application Development
Week 5: Split Views

Having a single table view take up the entire screen on larger screens wastes space. Instead when the iPad or iPhone plus/max is in landscape mode, the navigation column stays at a fixed position on the left, with the content of the selected item displayed on the right in what's known as a split-view, and applications built this way are called master-detail applications.

Split Views
A split view manages the presentation of two side-by-side panes of content in a master-detail relationship, with persistent content in the primary pane and related information in the secondary pane.
https://developer.apple.com/design/human-interface-guidelines/ios/views/split-views/

- By default, a split view devotes a third of the screen to the primary pane and two-thirds to the secondary pane. You can choose a different split appropriate for your content, just make sure the panes don't look unbalanced. The primary pane should be narrower than the secondary pane.
- Keep the active selection in the primary pane highlighted so users can easily understand the relationship between the panes.
- Restrict navigation to one side of the split view so users stay oriented and discern the relationship between the two panes.
- If the primary pane may be off screen make sure you provide a button, typically in the nav bar, to reveal the pane.
- Before iOS 8 split view controllers were only available on the iPad, now they are on devices when the width is a regular size class.

UISplitViewController class 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 (master): 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
By default, a split view devotes a third of the screen to the primary pane and two-thirds to the secondary pane and appears only in landscape orientation.

- The width of the split in side by side is configurable
- Avoid creating a secondary pane that's narrower than the primary pane.

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
- 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

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.

Modes:

- automatic
  - chooses the best display mode
- allVisible
  - both view controllers are displayed on the screen, primary on the left typically narrower than the secondary on the right.
- primaryHidden
  - the secondary view controller is displayed, the primary view controller is hidden
- primaryOverlay
  - the secondary view controller is onscreen and the primary view controller is layered on top of it

UISplitViewController has a displayModeButtonItem property that is a preconfigured bar button item that changes the display mode

- When tapped it tells the split view controller to change its current display mode to the value returned by the delegate's targetDisplayModeForAction(in:) method which returns a display mode. This method is also called during rotation. Implement this method if you want to customize what display mode is returned, otherwise it will be automatic.

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
  - Pre iOS8 the class was UIPopoverController
- UIPopoverPresentationControllerDelegate (added in iOS8) https://developer.apple.com/documentation/uikit/uipopoverpresentationcontrollerdelegate
  - Lets you customize the behavior of popovers

Multitasking
Multitasking(iOS 9) 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.

- Slide-over lets users overlay an app to briefly interact with it without leaving the app they're currently using.
- https://developer.apple.com/design/human-interface-guidelines/ios/system-capabilities/multitasking/
- 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)

HarryPotter
(harrypotter)
File | New Project
Master-Detail Application template
HarryPotter

Initial Setup
Run the app in an iPad simulator to see all that has been done already. Make sure you switch between landscape and portrait mode so you can see the split view.

Look at the Storyboard to see everything that was created.

- A split view controller as the root view controller
- A navigation controller and table view controller for the master view controller (left side)
- A navigation controller and detail view controller (right side)
- Relationship segues from the split view controller to the master and detail view controllers
- A showDetail segue from the cells in the master view controller to the detail view controller
    - A showDetail(Replace) segue is used in a splitViewController to replace the detail view with content based on a selection in the master view.
    - Different from a show(push) segue as the master is sometimes still visible (iPad landscape) and there's no ability to navigate back
- The datasource and delegate have also been set for the table view

In AppDelegate.swift a lot of work has been done for us

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool
```

- Get access to the root view controller which is the split view controller
- Grabs the last object in the splitviewcontrollers array of controllers which is the navigation controller of our detail view controller
- Assigns the displayModeButtonItem of the split view controller to the left bar button in the navigation bar of the detail view controller
- Assigns the split view's delegate (for a split view controller you can't do this in the storyboard, book gives a detailed explanation)

```
    func splitViewController(_ splitViewController:
UISplitViewController, collapseSecondary
secondaryViewController:UIViewController, onto
primaryViewController:UIViewController) -> Bool
```

- This method performs the tasks related to the transition to a collapsed interface
- Return values
    - "false" tells the split view controller to use its default behavior to try and incorporate the secondary view controller into the collapsed interface
    - "true" tells the split view controller not to apply the default behavior, that you do not want the split view controller to do anything with the secondary view controller
- Adjusts the primary view controller and incorporate the secondary view controller into the collapsed interface
- A compact horizontal size class (iPhone) will collapse the split view
- When the split view collapses what happens to the second view controller is up to the second view controller (return false)
- Removes the second view controller and have the first view controller as the only child of the split view controller(return true). This case is so the master view controller is shown on the iPhone when the app launches.

This template created a MasterViewController class and a DetailViewController class. These represent, respectively, the views that will appear on the left and right sides of the split view.

MasterViewController.swift
MasterViewController is a subclass of UITableViewController and defines an instance for the detail view controller.

viewDidLoad()
adding the edit and + buttons to the navigation bar.
Setting the detailviewcontroller

viewWillAppear()
The value of `clearsSelectionOnViewWillAppear` is based on whether or not the split view is collapsed.
By default, UITableViewController is set up to deselect all rows each time it's displayed. That may be OK in an iPhone app, where each table view is usually displayed on its own; however, in an iPad app featuring a split view, you probably don't want that selection to disappear.

prepare(for:sender:) handles the segue.

All the needed table view delegate methods are implemented.

DetailViewController.swift
DetailViewController is a subclass of UIViewController
Defines the outlet detailDescriptionLabel for the label in the storyboard.
Creates a variable called detailItem where the view controller stores its reference to the object that the user selected in the master view controller. The code in the didSet block is called after its value has been changed and calls configureView(), another method that's generated for us. configureView() updates the label with the detail item. configureView() is also called from viewDidLoad()

Wow, a lot of the setup was done for us. Now let's get going on our app.

Data
Drag in the harrypotter2.plist file we're 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 harrypotter 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 control our data model (same file or separate). This is similar to our previous apps.

```
class CharacterDataModelController{
    var allData = [Character]()
    let fileName = "harrypotter2"

    func loadData(){
        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 MasterViewController we need an instance of the CharacterDataModelController class to load and access our data and an array for the list of characters (you can change the one called object, we won't need it. This will cause errors that we'll fix next).

```
var characters = [String]()
var characterData = CharacterDataModelController()
```

In viewDidLoad remove(or comment out) the code that's in there to allow edits since we won't be doing that. Then we need to load our data and get the list of characters.

```
        characterData.loadData()
        characters=characterData.getCharacters()
```

We also need to remove some table view delegate and data source methods.
Delete/comment out these whole methods:

```
func insertNewObject(_ sender: AnyObject)

func tableView(tableView: UITableView, commitEditingStyle editingStyle:
UITableViewCellEditingStyle, forRowAtIndexPath indexPath: NSIndexPath)
```

Now let's update the other table view data source methods (stubs)
We have 1 section so `numberOfSections(in tableView:)` is fine

```
 func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
        return characters.count
    }
```

Update `func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) -> Bool {`

```
     return false
    }
```

And now configure the appearance of the cell

```
    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
    }
```

Before we can pass the URL to the detail view controller go into DetailViewController.swift and change detailItem from `NSDate?` to `String?`

Back in MasterViewController.swift we have to update prepare(for segue:) to pass the URL to the detail view controller.

```
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "showDetail" {
            if let indexPath = self.tableView.indexPathForSelectedRow {
                let url = characterData.getURL(index: indexPath.row)
                let name = characters[indexPath.row]
                let controller = (segue.destination as!
UINavigationController).topViewController as! DetailViewController
                controller.detailItem = url
                controller.title = name
                controller.navigationItem.leftBarButtonItem =
self.splitViewController?.displayModeButtonItem
                controller.navigationItem.leftItemsSupplementBackButton =
true
            }
        }
    }
```

When you run it tap the Master button in the upper-left corner to bring up a popover with a list of characters. Tap a character's name to display their Wikipedia page URL in the detail view.

Rotate to landscape and you'll see the characters in the table view in the master 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

Detail View

In the storyboard change the Master view controller title to "Characters" instead of it saying Master.
Now let's get the detail view working.
Remove the title 'Detail' so that doesn't show when the app first starts.
Go into the storyboard and move the existing label to the top just under the navigation bar and update its constraints.(replace center Y constraint with top space to top layout 0)
Change it to say "Select a Harry Potter character". We will use this label to show the URL but before the user selects anything it will provide instructions.
Look in the connection inspector to see that it's already set up as an outlet called detailDescriptionLabel.
Then 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 below, and not covering, the label.

In DetailViewController.swift you need to import WebKit.

```
import WebKit
```

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 configureView

```
    func configureView() {
        // Update the user interface for the detail item.
        if let detail = self.detailItem {
            if let label = self.detailDescriptionLabel {
                label.text = detail.description
                loadWebPage(detail.description)
            }
        }
    }
```

You should now be able to select a character and see their Wikipedia page in the detail.
To change the title of the back button for the popover in portrait mode change the Title for the master view controller in the Storyboard in its navigation controller.

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 it gray 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 DetailViewController .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.