ATLS 4320: Advanced Mobile Application Development
Week 2: Tab Bar Controllers

Model-View-Controller(MVC) Review

It's important to understand the MVC architecture as it's used in both iOS and Android

- Model: holds the data and classes
  - Should be UI independent
- View: all items for the user interface(objects in IB)
- Controller: links the model and the view together. The backbone or brain of the app.
  - usually subclasses from UI frameworks that will allow the view and the model to interact
  - Controllers are usually paired with a single view, that's where ViewController comes from
- The goal of MVC is to have any object be in only one of these categories.
  - These categories should never overlap.
- Ensures reusability

Tab Bar Controllers

https://developer.apple.com/design/human-interface-guidelines/ios/bars/tab-bars/
The tab bar controller organizes information in a flat hierarchy of different sections of an app in a list of tabs. Provides access to several peer information categories or modes at once.
Tab bars should be used for navigation, not to perform actions (those are tool bars)

The tab bar controller is the root view controller and each tab points to a view with its own view controller
UITabBarController class https://developer.apple.com/documentation/uikit/uitabbarcontroller
Use when you want to present different perspectives of related data (music, clock, phone apps)
Data is not passed between views in a tab bar, it is just a way to organize categories of information.
UITabBarItem class https://developer.apple.com/documentation/uikit/uitabbaritem
Rule of thumb is to have 3-5 tabs on an iPhone(5 are shown in horizontal compact), a few more are acceptable on iPad.

- Too few tabs make the interface feel disconnected
- When there are more tabs than can be shown the tab bar controller will automatically display a "More" tab with the rest of the items. This requires additional taps, is a poor use of space, and should be avoided.

Pickers

https://developer.apple.com/design/human-interface-guidelines/ios/controls/pickers/
A picker is a scrollable list of distinct values (replaces a select list on the web)

- slot-machine looking UI element that is used when you have a list of values
- Date picker also available for date and/or time
- can have multiple components (columns) that are independent or dependent

UIPickerView class https://developer.apple.com/documentation/uikit/uipickerview
Use a picker when users are familiar with the entire set of values. Because many of the values are hidden when the wheel is stationary, it's best when users can predict what the values are.
If you need to provide a large set of choices that aren't well known to your users, use a table view.
Pickers don't hold any data themselves. They call methods on their data source and delegate to get the data they need to display.

The UIPickerViewDataSource protocol handles the data for the picker view and must be adopted
https://developer.apple.com/documentation/uikit/uipickerviewdatasource

The UIPickerViewDelegate protocol handles the view content and must be adopted
https://developer.apple.com/documentation/uikit/uipickerviewdelegate

Music
Create a new project called music and choose the Tabbed App template.
Look at the files created, the Tabbed Application template did a lot for us.
Main.storyboard already has a tab bar controller set up with three scenes.
Click on the tab bar controller. In the identity inspector you'll see that the class is UITabBarController. In the attributes inspector you'll see that Is Initial View Controller is checked. And in the Connections inspector you'll see that segues are already set up to the other two view controllers.
Click on the first view and look at the identity inspector. It's already set up to use the class FirstViewController and the swift file was created for us.
In the Connections inspector note that a relationship segue has already been set up from the tab bar controller.
You'll also notice that the tab bar item is set up to use an image already in the project called first.png as well as a title(click on the First Scene's tab bar button and go into the attributes inspector). We can change the image later.
Look at the second scene as well. This one uses the SecondViewController class, has a relationship segue, and its tab bar item uses an image called second.png.

Go ahead and run it and you'll see that the tab bar is already working and loads both views, so this template does a lot for us.

Single component Picker
In our first view we're going to add a single component picker where you choose a genre of music.
Go into the Storyboard and delete the two labels there.
Add a picker view(ignore the default data it shows)
Add a label above it that says Music Chooser.
Then add another label below the picker that we'll use for output.

Create outlet connections for the picker and the 2nd label called musicPicker and choiceLabel. Make sure you're connecting to FirstViewController.swift.

With the picker selected go into the connections inspector and connect the dataSource and delegate to the view controller by clicking in the circle and dragging the connection to View Controller (white square in a yellow circle icon on top of the view). Now this picker knows that FirstViewController is its data source and delegate, and will ask it to supply the data to be displayed. (can also do this programmatically in viewDidLoad)

Go into FirstViewController .swift and add the protocol we're going to implement
```
class FirstViewController: UIViewController, UIPickerViewDelegate,
UIPickerViewDataSource
```
You will have an error until we implement the required methods for `UIPickerViewDataSource`
Look at the delegate classes.

Define an array that will hold the music genres.

```
let genre = ["Country", "Pop", "Rock", "Classical", "Alternative", "Hip
Hop", "Jazz"]
```

Now let's add the methods needed to implement the picker.

```
//Methods to implement the picker
//Required for the UIPickerViewDataSource protocol
func numberOfComponents(in pickerView: UIPickerView) -> Int {
        return 1 //number of components
}

//Required for the UIPickerViewDataSource protocol
func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent
component: Int) -> Int {
        return genre.count //returns number of rows of data
}
```

The delegate methods are optional, but you need to implement at least one and it's often this one.

```
//Picker Delegate methods
//Returns the title for a given row
func pickerView(_ pickerView: UIPickerView, titleForRow row: Int,
forComponent component: Int) -> String? {
    return genre[row]
}

//Called when a row is selected
func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int,
inComponent component: Int) {
    choiceLabel.text="You like \(genre[row])" //writes the string with the
row's content to the label
}
```

Run it in the simulator
Click and drag your mouse in the simulator to move the picker.
Use auto layout so the view looks good in portrait and landscape.
Use autoresizing feature or add constraints (can't use both in a view).
Assistant editor – use the jump bar to access preview mode.

Multi-component picker

Either modify the view we've been working in or use the second tab for a 2 component independent picker.
We're going to add a 2nd component that lists the decade for music.
Genre will be component 0
Decade will be component 1

Let's add another array to hold the decade.

```
let decade = ["1950s", "1960s", "1970s", "1980s", "1990s", "2000s", "2010s"]
```

Most of the changes will be in the data source methods.
This time we're going to return 2 in `pickerView(_:numberOfRowsInComponent:)` since we have 2 components.

And now we have to check with component is picked before we can return the row count.

```
func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent
component: Int) -> Int {
        if component==0 {
            return genre.count
        }
        else {
            return decade.count
        }
}
```

Then in our delegate method we also have to check which component was picked before we can return the value.

```
func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent
component: Int) -> Int {
        if component==0 {
            return genre[row]
        }
        else {
            return decade[row]
        }
}
```

Don't forget the delegate method to draw the title for each row

```
    //Picker Delegate methods
    //returns the title for the row
    func pickerView(_ pickerView: UIPickerView, titleForRow row: Int,
forComponent component: Int) -> String? {
        if component == 0 {
            return genre[row]
        } else {
            return decade[row]
        }

    }
```

Now we have to use both components when printing out our results.

```
func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int,
inComponent component: Int) {
    let genrerow = pickerView.selectedRow(inComponent: 0) //gets the
selected row for the genre
    let decaderow = pickerView.selectedRow(inComponent: 1) //gets the
selected row for the decade
    choiceLabel.text="You like \(genre[genrerow]) from the
\(decade[decaderow])"
}
```

You don't need to change anything in the storyboard because all the connections are there.
You might need to make the label field larger since our output is larger.

If you hover over code such as an if statement and click the command key the editor will show you the code block. A click then gives you code options.

Property Lists
Few applications use arrays with data hard coded so this time we're going to use the data stored in a property list which is a simple data file in XML.

- Use the Bundle(previously NSBundle) class to access the plist
    - An Bundle object represents a location in the file system that groups code and resources that can be used in a program
    - We can use a bundle is to get access to the files in our app
- Property lists can be created using the Property List Editor application (*/Developer/Applications/Utilities/Property List Editor.app*) or directly in Xcode
- You can use the new PropertyListDecoder class to decode data from a plist and assign the data to an array or dictionaries(depending on plist structure).

Dependent multi-component picker
Now let's add a third tab to our tab bar. In this tab we're going to have a 2 component picker but this time the components are going to be dependent on each other.
Let's also look at how you add more tabs.
Go back into Main.Storyboard and drag a View Controller from the object library onto the canvas.
Make the connection from the tab bar controller to the new view controller by cntrl-click and drag from the tab bar controller to the new view controller and choose Relationship segue view controllers. This will add a third tab bar button that is hooked up to the new view controller.

Now we need a new class to control this view.
File | New | File
iOS Source | Cocoa Touch class
Call it ThirdViewController and make sure it's a subclass of UIViewController.
Uncheck Also create xib file and make sure the language is Swift.
Make sure it's being saved into your project folder and the music target is checked
This should create ThirdViewController.swift

In the Storyboard click on the third ViewController and in the identity inspector change its class to ThirdViewController.

Go into the Storyboard and create a similar interface as before - a picker and two labels (one that says Artist Picker and another label that's empty that we'll use for output). Same constraints will work.
Create outlet connections for the 2nd label and the picker called choiceLabel and artistPicker. Make sure you're making the connections to ThirdViewController.
With the picker selected go into the connections inspector and connect the dataSource and delegate to the View Controller icon.

Grab the artistalbums2 plist from my github repo and drag it into your project.
Make sure you choose Copy items if needed and that the project target is checked.
Look at the plist. Note that it's an array of dictionaries, with the key name, value a string, and key albums, value an array of strings.

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

We'll create a struct to represent this data model.

```
struct ArtistAlbums: Decodable {
    let name : String
    let albums : [String]
}
```

Starting in Swift 4.1 the ability to encode and decode custom types became much easier through the Encodable and Decodable protocols (and the Codable type alias for both). By adopting these protocols for your custom types you can implement the Encoder and Decoder protocols which will encode or decode your data to and from an external representation such as JSON or property list.
The property names in your custom type MUST match the property names in your data. In the case of a plist the names of the keys must be the same as the names of the properties in your struct for the decoding to work automatically.

Go into ThirdViewController.swift and add the protocols like last time.

```
class ThirdViewController: UIViewController, UIPickerViewDataSource,
UIPickerViewDelegate
```

Let's define 2 constants for the component numbers since we'll be using them a lot.

```
    let artistComponent = 0
    let albumComponent = 1
```

Now we're going to define an array of ArtistAlbums to load our plist into and 2 arrays to hold the artists names and album names.

```
    var artistAlbums = [ArtistAlbums]()
    var artists = [String]()
    var albums = [String]()
```

viewDidLoad() is a good place to load the data into our dictionary and arrays.

```
    override func viewDidLoad() {
        super.viewDidLoad()
        // URL for our plist
        if let pathURL = Bundle.main.url(forResource: "artistalbums2",
withExtension: "plist"){
            //creates a property list decoder object
            let plistdecoder = PropertyListDecoder()
            do {
                let data = try Data(contentsOf: pathURL)
                //decodes the property list
                artistAlbums = try plistdecoder.decode([ArtistAlbums].self,
from: data)
                for artist in artistAlbums{
                    artists.append(artist.name)
                }
                albums = artistAlbums[0].albums
            } catch {
                // handle error
                print(error)
            }
        }
    }
```

The next 3 methods haven't changed much, we're just using the constants we defined.

```
//Required for the UIPickerViewDataSource protocol
    func numberOfComponents(in pickerView: UIPickerView) -> Int {
        return 2
    }

    func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent
component: Int) -> Int {
        if component == artistComponent {
        return artists.count
    } else {
        return albums.count
        }
    }


//Picker Delegate methods
//Returns the title for a given row
    func pickerView(_ pickerView: UIPickerView, titleForRow row: Int,
forComponent component: Int) -> String? {
        //checks which component was picked and returns the value for the
requested component
        if component==artistComponent {
            return artists[row]
        }
        else {
            return albums[row]
        }
    }
```

We need to change this method so when the artist is changed the list of albums changes as well.

```
//Called when a row is selected
    func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int,
inComponent component: Int) {
        //checks which component was picked
        if component == artistComponent {
            albums = artistAlbums[row].albums //gets the albums for the
selected artist
            artistPicker.reloadComponent(albumComponent) //reload the album
component
            artistPicker.selectRow(0, inComponent: albumComponent, animated:
true) //set the album component back to 0
        }
        let artistrow = pickerView.selectedRow(inComponent: artistComponent)
//gets the selected row for the artist
        let albumrow = pickerView.selectedRow(inComponent: albumComponent)
//gets the selected row for the album
        choiceLabel.text = "You like \(albums[albumrow]) by
\(artists[artistrow])"
    }
```

If you find your text being cut off in the label, set the attribute Autoshrink to Minimum font scale or size so it will automatically adjust.

Icons

You can customize the text and image for each tab
You can use a standard image or add a custom one

- About 25x25 pixels for 1x (max 48x32)
- png format
- Colors are ignored, alpha values from 0 (completely invisible) to 1 (completely visible) are used.
- Different versions for unselected and selected(filled in)

Let's set the tab bar name and image. Drag the png files into Assets.xcassets that you want to use. Then in the Storyboard click on the tab bar item for the first scene and in the attributes inspector you can change its title and image, or remove the title if you just want an image. (65-note, 120-headphones, 194-note-2, 31-ipod, 66-microphone) (you can download glyphish icons to use)
For app icons you need 120x120 for the 2x and 180x180 for 3x.
You should also set up a launch screen and its constraints.

App access

Now let's add a fourth tab to our tab bar. In this tab we're going to access other apps.
Add a new view controller into your storyboard and connect the tab bar controller with a relationship segue.
Then add a new Cocoa Touch class, subclass UIViewController and call it FourthViewController.
Make this the class for your new view controller.
Add a new class to control this view and call it FourthViewController and make sure it's a subclass of UIViewController.
This should create FourthViewController.swift
In the Storyboard click on the fourth ViewController and in the identity inspector change its class to FourthViewController.

(these steps are the same as what we did when we added the third view controller)

Now lets get our fourth view set up.
Let's make this a view where a button will take the user to Spotify, the iTunes music library or if that's not on their device(simulator) then it opens itunes in Safari.
And add a button that says Listen and connect it as an action to a method called gotomusic.
Since this is the fourth scene we must connect to FourthViewController.swift.
Add Missing Constraints for all views in the fourth view controller and fix as needed.
Change the tab bar button image(ipod)

Now let's go into FourthViewController.swift and implement gotomusic().

```
    @IBAction func gotomusic(_ sender: UIButton) {
        //check to see if there's an app installed to handle this URL scheme
        if(UIApplication.shared.canOpenURL(URL(string: "spotify://")!)){
            //open the app with this URL scheme
            UIApplication.shared.open(URL(string: "spotify://")!, options:
[:], completionHandler: nil)
        }else {
            if(UIApplication.shared.canOpenURL(URL(string: "music://")!)){
                UIApplication.shared.open(URL(string: "music://")!, options:
```

```
[:], completionHandler: nil)
            } else {
                UIApplication.shared.open(URL(string:
"http://www.apple.com/music/")!, options: [:], completionHandler: nil)
            }
        }
    }
```

For a Swift app, iOS creates a UIApplication object to set up the app's runtime environment: its use of the display, its ability to handle touches and rotation events, etc. This object is also how we can interact with the rest of the iOS system. We're using the shared.canOpenURL(_:) method to see if there's an app available to handle that string(returns a Boolean).

UIApplication also has the method shared.open(_: options: completionHandler:), which will launch other applications and have them deal with the URL.
mailto:, facetime:, and tel: open the Mail, FaceTime, and Phone apps, respectively.

UIApplication also has a UIApplicationDelegate object, which is informed of major life-cycle events that affect the app. This is the AppDelegate class that Xcode gave us to start with. When all the app setup is done, the application(_: didFinishLaunchingWithOptions:) method gets called. From our point of view, this is where the app "starts," although a bunch of stuff has already been done for us by this point.

Starting in iOS9 you have to declare any URL schemes of non-Apple apps you want your app to be able to call canOpenURL(_:). This does not affect open(_: options: completionHandler:).
Go into Info.plist and add LSApplicationQueriesSchemes as an Array. Click on the arrow to the left of LSApplicationQueriesSchemes to open it and then hit the + to add an item to the array called "spotify".

Audio
The iOS SDK has multiple multimedia frameworks to access iOS's audio capabilities.
Frameworks for audio from easiest to use to harder

- System Sound Services - plays user-interface sound effects, or to invoke vibration
    - supports caf, aif, or wav formats and must be less than 30 secs.
- Media Player framework - plays songs, audio books, or audio podcasts from a user's iPod library.
- AVFoundation framework - plays and records audio
    - We'll be using this to record and play back sounds in our app
- Audio Toolbox framework - plays audio with synchronization capabilities, access packets of incoming audio, parse audio streams, convert audio formats, and record audio with access to individual packets.
- Audio Unit Framework - connect to and use audio processing plug-ins
- OpenAL framework - provides positional audio playback and lets you mix sounds
    - Best choice for games
    - OpenAL gives you more control of audio but is more complicated.

The **AVFoundation** framework
The **AVAudioSession** class acts as an intermediary between your app and the system's media services
https://developer.apple.com/documentation/avfoundation/avaudiosession

- Configure your audio session

- Configure audio settings such as sample rate, I/O buffer duration, and number of channels
- Handle audio route changes
    - Events such as a phone call
    - Audio use by another app
- Audio session category
    - how your audio session interacts with others

The **AVAudioPlayer** class provides playback of audio data from a file or memory
https://developer.apple.com/documentation/avfoundation/avaudioplayer

- Play sounds of any duration from files or memory
- Configure and control playback
- Manage audio level metering

The **AVAudioPlayerDelegate** protocol has optional methods that are called when the audio file finishes playing, if there are interruptions or if there's an error.
https://developer.apple.com/documentation/avfoundation/avaudioplayerdelegate
The **AVAudioRecorder** class provides recording of audio data
https://developer.apple.com/documentation/avfoundation/avaudiorecorder

- Record until the user stops the recording
- Record for a specified duration
- Pause and resume a recording
- Obtain input audio-level data for level metering
- In iOS, the audio being recorded comes from the device connected by the user such as the built-in microphone or a headset microphone.
- Configurable setttings include bit depth, bit rate, and sample rate conversion quality
  https://developer.apple.com/documentation/avfoundation/avaudioplayer/1389359-settings

The **AVAudioRecorderDelegate** protocol has optional methods that are called when the recording completes, if there are interruptions or if there's an error
https://developer.apple.com/documentation/avfoundation/avaudiorecorderdelegate

Add a fifth tab as before, along with a FifthViewController class for the new view controller and set the class for the new view in Interface Builder.
(same steps as before)

Click on the Target and go into the Build Phases tab.
Open Link Binary with Libraries and click the + to add the AVFoundation framework under iOS 11.2.

In the storyboard add 3 buttons for Record, Play, and Stop. (add needed constraints)
Connect these as outlets called recordButton, playButton, and stopButton. We need these so we can enable and disable these buttons as needed.
Also connect these as actions called recordAudio, playAudio, and stopAudio.

You must request permission before your app can access the microphone.
In info.plist you must add the entry "Privacy - Microphone Usage Description" and give it a value that will be used in the permission request to the user.

In FifthViewController.swift we need to import the AVFoundation framework and adopt the AVAudioPlayerDelegate and AVAudioRecorderDelegate.

```
import UIKit
```

```
import AVFoundation

class FifthViewController: UIViewController, AVAudioPlayerDelegate,
AVAudioRecorderDelegate
```

Create an instance variable for our audioplayer and audiorecorder

```
    var audioPlayer: AVAudioPlayer?
    var audioRecorder: AVAudioRecorder?
```

Create a constant for the name of the file where the audio will be saved

```
     let fileName = "audio.m4a"
```

viewDidLoad is a good place to do our setup and initialization

```
    override func viewDidLoad() {
        super.viewDidLoad()

     //disable buttons since no audio has been recorded
        playButton.isEnabled = false;
        stopButton.isEnabled = false;

        //get path for the audio file
        let dirPath = FileManager.default.urls(for: .documentDirectory, in:
.userDomainMask)
        let docDir = dirPath[0] //documents directory
        let audioFileURL = docDir.appendingPathComponent(fileName)
        print(audioFileURL)

      //the shared audio session instance
        let audioSession = AVAudioSession.sharedInstance()
        do {
           //sets he category for recording and playback of audio
           try audioSession.setCategory(AVAudioSessionCategoryPlayAndRecord)
        } catch {
            print("audio session error: \(error.localizedDescription)")
        }

        //recorder settings
        let settings = [
            AVFormatIDKey: Int(kAudioFormatMPEG4AAC), //specifies audio
codec
            AVSampleRateKey: 12000, //sample rate in hertz
            AVNumberOfChannelsKey: 1, //number of channels
            AVEncoderAudioQualityKey: AVAudioQuality.high.rawValue //audio
bit rate
        ]

        do {
            //create the AVAudioRecorder instance
            audioRecorder = try AVAudioRecorder(url: audioFileURL, settings:
settings)
            audioRecorder?.prepareToRecord()
            print("audio recorder ready")
        } catch {
            print("audio recorder error: \(error.localizedDescription)")
```

```
        }
    }
```

Then we'll implement our methods to record, stop, and play.

```
    @IBAction func recordAudio(_ sender: UIButton) {
        //if not already recording, start recording
        if audioRecorder?.isRecording == false{
            playButton.isEnabled = false
            stopButton.isEnabled = true
            audioRecorder?.delegate = self
            audioRecorder?.record()
        }
    }

    @IBAction func stopAudio(_ sender: UIButton) {
        stopButton.isEnabled = false
        playButton.isEnabled = true
        recordButton.isEnabled = true
        //stop recording or playing
        if audioRecorder?.isRecording == true {
           audioRecorder?.stop()
        } else {
           audioPlayer?.stop()
        }
    }

    @IBAction func playAudio(_ sender: UIButton) {
        //if not recording play audio file
        if audioRecorder?.isRecording == false{
            stopButton.isEnabled = true
            recordButton.isEnabled = false

            do {
                try audioPlayer = AVAudioPlayer(contentsOf:
                    (audioRecorder?.url)!)
                audioPlayer!.delegate = self
                audioPlayer!.prepareToPlay()//prepares the audio player for
playback by preloading its buffers
                audioPlayer!.play() //plays audio file
            } catch let error {
                print("audioPlayer error: \(error.localizedDescription)")
            }
        }
    }
```

Both delegate protocols have optional methods. Since you don't tap a button when the audio playing ends, let's implement that one to change the buttons as needed.

```
    //AVAudioPlayerDelegate method
     //Called when a recording is stopped or has finished due to reaching
its time limit
    func audioPlayerDidFinishPlaying(_ player: AVAudioPlayer, successfully
```

```
flag: Bool) {
        recordButton.isEnabled = true
        stopButton.isEnabled = false
    }
```