ATLS 4320: Advanced Mobile Application Development
Week 6: JSON

JSON

JavaScript Object Notation(JSON) is a language independent data format used to store and exchange data. http://json.org/
Supported by every major modern programming language including JavaScript, Swift, and Java
JSON is built on two structures

- A collection of name/value pairs stored as an object, record, struct, dictionary, hash table, keyed list, or associative array in various languages
  - An object in in curly brackets { }
  - Format: name : value
  - Name/value pairs are separated by a comma ,
- An ordered list of values are stored as an array, vector, list, or sequence in various languages
  - An array is in square brackets [ ]
  - Values are separated by a comma

XML: https://api.whitehouse.gov/v1/petitions
JSON: https://api.whitehouse.gov/v1/petitions.json?limit=100
To display the JSON formatted in Chrome you'll need to install an extension.

iOS Networking

There are three main classes you need to know about in order to handle networking in iOS:

1. URLRequest encapsulates information about a URL request.
   a. Used by URLSession to send the request
2. URLSession coordinates the set of requests and responses that make up a HTTP session https://developer.apple.com/documentation/foundation/urlsession
- URLSession has a singleton shared session for basic requests https://developer.apple.com/documentation/foundation/urlsession/1409000-shared
- You can create a URLSessionTask to retrieve the contents of a URL using URLSession.shared.dataTask(with:completionHandler:)
https://developer.apple.com/documentation/foundation/urlsession/1407613-datatask
   a. Requests a URLRequest
   b. Completion handler to call when the request is complete and successful
      i. Data – holds the downloaded data if successful
      ii. Response - An object that provides response metadata, such as HTTP headers and status code. If you are making an HTTP or HTTPS request, the returned object is actually an HTTPURLResponse object.
         1. The HTTP status code is stored in the HTTPURLResponse statusCode property
         2. HTTP status codes
            a. https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
            b. 200 is OK
      iii. Error – an error object if the request fails, data will be nil
   c. After you create the task, you must start it by calling its resume method
   d. Returns the new session data task.
3. URLSessionTask performs the actual transfer of data through the URLSession instance. It has different subclasses for different types of tasks. URLSessionDataTask is used for the contents of an URL.

So, making an HTTP request in iOS boils down to:

- create and configure an instance of URLSession, to make one or more HTTP requests
- optionally create and configure an instance URLRequest for your requests. Needed only if you need some specific parameters.
- start a URLSessionDataTask through the URLSession instance.

JSON in iOS

Once the JSON has been downloaded successfully we need to parse the data and save it in our data model

- The DispatchQueue class manages the execution of work items. Each work item submitted to a queue is processed on a pool of threads managed by the system.
- The DispatchQueue.main.async{} method will submit requests to be run asynchronously on another threads. You should use this to parse the JSON asynchronously. It's really important to only use the main queue for the UI, otherwise the other tasks make the app unresponsive and slow as it's waiting on the other tasks.
- Instead of the PropertyListDecoder we've been using for plists, we'll use the JSONDecoder which is the JSON equivalent. Just as with property lists, we need the property names of our struct to match the key name in the JSON file.

petitions

(petitions basic)
File | New Project
Master-Detail App

Setup

Go into the Main storyboard and in the table view change the table view cell to the style subtitle.
Change the title of the master scene to White House Petitions and Back Button to Petitions.

In MasterViewController.swift delete the insertNewObject() method entirely, delete everything from viewDidLoad() except the call to super.viewDidLoad(), delete or comment out the table view's commitEditingStyle and canEditRowAtIndexPath methods, and finally delete the as! NSDate text from the prepareForSegue() and cellForRowAtIndexPath methods – not the whole line, just the bit that says as! NSDate.

Make sure your app builds at this point.

Create Model

Look at JSON data https://api.whitehouse.gov/v1/petitions.json?limit=100
It's easier to read in XML https://api.whitehouse.gov/v1/petitions
Look at the items in results, they are key-value pairs
We're going to use a struct to represent a petition. We'll pick out the data items in the value of results that we're interested in.
File | New | File | Swift File
Petition

```
struct Petition: Decodable{
    let title: String
    let sigCount : Int
```

```
    let url : String
}
```

To use a JSONDecoder we need a data structure with a property named results and the value an array.

```
struct PetitionData: Decodable {
    var results = [Petition]()
}
```

Load JSON file

In MasterViewController change objects to store these key-value pairs using our Petition struct.

```
var petitions = [Petition]()
```

Create a method to download the json file.

```
    func loadjson(){
        let urlPath =
"https://api.whitehouse.gov/v1/petitions.json?limit=50"
        guard let url = URL(string: urlPath)
            else {
                print("url error")
                return
            }

        let session = URLSession.shared.dataTask(with: url,
completionHandler: {(data, response, error) in
            let httpResponse = response as! HTTPURLResponse
            let statusCode = httpResponse.statusCode
            guard statusCode == 200
                else {
                    print("file download error")
                    return
            }
            //download successful
            print("download complete")
            //parse json asynchronously
           //DispatchQueue.main.async {self.parsejson(data!)}
        })
        //must call resume to run session
        session.resume()
    }
```

Update viewDidLoad() to call this method.

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

If you run it now you should get the download successful message.

Parse JSON file

Now we'll create a method to get the json data and parse it so we can use it.

```
    func parsejson(_ data: Data){
        do
        {
            let api = try JSONDecoder().decode(PetitionData.self, from:
data)
            //print(api)
            for petition in api.results
            {
                petitions.append(petition)
            }
        }
        catch let jsonErr
        {
            print(jsonErr.localizedDescription)
            return
        }
        //reload the table data after the json data has been downloaded
        tableView.reloadData()
    }
```

Call this asynchronously from loadjson() right after the print statement that the download was successful.

```
DispatchQueue.main.async {self.parsejson(data!)}
```

Load table data

Update the following method to use our petitions array.

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

Now we're ready to load the petition data in our table. Update the following table view delegate method

```
    override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "Cell",
for: indexPath)
        let petition = petitions[indexPath.row]
        cell.textLabel!.text = petition.title
        cell.detailTextLabel!.text = String(petition.sigCount) + "
signatures"
        return cell
    }
```

Detail View

In the main storyboard go into the detail view and remove the label (check that this removes its connection as well) and replace it with a webKit view that fills up the whole view.
Add an activity indicator on top of the webKit view. (it must be below the web view in the document hierarchy). In the attributes inspector check Hides When Stopped.
Connect the web view and activity indicator as webView and webSpinner.
Add needed constraints.

Before leaving the storyboard go to the Master view and change the accessory on the cell to a disclosure indicator to give the user the visual cue that selecting the row will lead to more information.

In DetailViewController remove detailDescriptionLabel.
import WebKit and adopt the WKNavigationDelegate protocol for the class.

```
import WebKit
class DetailViewController: UIViewController, WKNavigationDelegate
```

Set up the web view's navigation delegate in viewDidLoad()

```
webView.navigationDelegate = self
```

Write a method to load a web page.

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

Update detailItem and configureView() as follows:

```
    var detailItem: String?

    func configureView(){
        if let url = detailItem{
            if url != "null"{
                loadWebPage(url)
            }
        }
    }
```

Implement the two delegate methods that are called when the web page starts and stops loading.

```
    //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()
    }
```

In MasterViewController update prepareForSegue() to send the detail view the data it needs.

```
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "showDetail" {
```

```
            if let indexPath = self.tableView.indexPathForSelectedRow {
                let petition = objects[indexPath.row]
                let title = petition["title"]
                let url = petition["url"]
                let controller = (segue.destination as!
UINavigationController).topViewController as! DetailViewController
                controller.detailItem = url
                controller.title = title
                controller.navigationItem.leftBarButtonItem =
self.splitViewController?.displayModeButtonItem
                controller.navigationItem.leftItemsSupplementBackButton =
true
            }
        }
    }
```

When you run the app if there's a space about the web view you need to update the top constraint to go all the way to the top of the view, not just to the bottom of the navigation bar.

Data Model Controller
Since making the network request and parsing the JSON are not part of controlling the view, they really don't belong in our MasterViewController class. But we can't populate the table until we've downloaded and parsed the JSON, so it adds a bit of complexity.

There are three ways to pass data from a model to a controller:
https://medium.com/@stasost/ios-three-ways-to-pass-data-from-model-to-controller-b47cc72a4336

- Callbacks
- Delegation
- Notifications

Callbacks are a way to call some code, like a function, *after* another piece of code has finished.
Callbacks can be functions, or properties, that take the form of a closure.
A closure is basically a self-contained blocks of functionality that can be passed around. It can be a function but is usually unnamed (similar to anonymous functions or lambdas).

Let's create a new data model controller class
File | New | File | Swift File
PetitionDataController

```
class PetitionDataController {
    var petitionData = PetitionData()
    //property with a closure as its value
    //closure takes an array of Petition as its parameter and Void as its
return type
    var onDataUpdate: ((_ data: [Petition]) -> Void)?

    func loadjson(){
        let urlPath =
"https://api.whitehouse.gov/v1/petitions.json?limit=50"
        guard let url = URL(string: urlPath)
            else {
                print("url error")
```

```
                return
        }

        let session = URLSession.shared.dataTask(with: url,
completionHandler: {(data, response, error) in
            let httpResponse = response as! HTTPURLResponse
            let statusCode = httpResponse.statusCode
            guard statusCode == 200
                else {
                    print("file download error")
                    return
            }
            //download successful
            print("download complete")
            //parse json asynchronously
            DispatchQueue.main.async {self.parsejson(data!)}
        })
        //must call resume to run session
        session.resume()
    }

    func parsejson(_ data: Data){
        do
        {
            let api = try JSONDecoder().decode(PetitionData.self, from:
data)
            //print(api)
            for petition in api.results
            {
                petitionData.results.append(petition)
            }
        }
        catch let jsonErr
        {
            print("json error")
            print(jsonErr.localizedDescription)
            return
        }
        print("parsejson done")
        //passing the results to the onDataUpdate closure
        onDataUpdate?(petitionData.results)
    }

    func getPetitions()->[Petition]{
        return petitionData.results
    }
}
```

onDataUpdate is a callback defined as a property and once all the JSON data has been downloaded and parsed we pass this callback the results.

Then in MasterViewController add an instance for our PetitionDataController class.

```
var petitionData = PetitionDataController()
```

The loadjson() and parsejson() methods have been moved to our PetitionDataController class so delete those. We'll add a method that will get the list of petitions and reload the table data.

```
    func render(){
        petitions=petitionData.getPetitions()
        //reload the table data
        tableView.reloadData()
    }
```

Then update viewDidLoad().

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

        //assign the closure with the method we want called to the
onDataUpdate closure
        petitionData.onDataUpdate = {[weak self] (data:[Petition]) in
self?.render()}
        petitionData.loadjson()
    }
```

Here we're assigning the onDataUpdate callback in the PetitionDataController class the method we want it to call, which is our render method. So after all the JSON data is downloaded and parsed, we'll retrieve the list of petitions and reload the table so we can draw all the cells.

Although a bit more complex, it's a better structure since it better separates the model data and the view.

Automatic Reference Counting
Swift uses Automatic Reference Counting (ARC) to track and manage your app's memory usage. In most cases, this means that memory management "just works" in Swift, and you do not need to think about memory management yourself. ARC automatically frees up the memory used by class instances when those instances are no longer needed.
When you create a new instance of a class, ARC allocates a chunk of memory to store information about that instance. This memory holds information about the type of the instance, together with the values of any stored properties associated with that instance. Whenever you assign a class instance to a property, constant, or variable, that property, constant, or variable makes a *strong reference* to the instance. A "strong" reference does not allow it to be deallocated for as long as that strong reference remains. Once there are no strong references the memory will be deallocated(freed).
A weak reference does not keep a strong hold on the instance it refers to, and so does not stop ARC from disposing of the referenced instance. (outlet variables are a good example of this)
Closures are reference types so when you assign a closure to a property, you are assigning a strong reference to that closure. We don't want to create another strong reference to the closure because the class already has a strong reference it and that would create a strong reference cycle. So we use a capture list (it's an array as there could be more than one pairing) to define it as a weak reference since the property is an optional and may become `nil` at some point in the future. Weak references are always of an optional type, and automatically become `nil` when the instance the reference is deallocated.