

iOS Dev Accelerator

Week3 Day1

- Static Table View
- HTTP & Rest API
- NSURLSession
- Swift 1.2 Optional Binding
- Structs & Value Types
- UISearchBar

The MVC layout of our Week 3 App

Model Layer

Repo

User

RepoJSON
Parser

UserJSONP
arser

GithubService

Controller Layer

Login
View
Controller

Menu
View
Controller

RepoSearch
ViewController

UserSearch
ViewController

MyRepos
ViewController

Web
View
Controller

Profile
ViewController

Animation
Controller

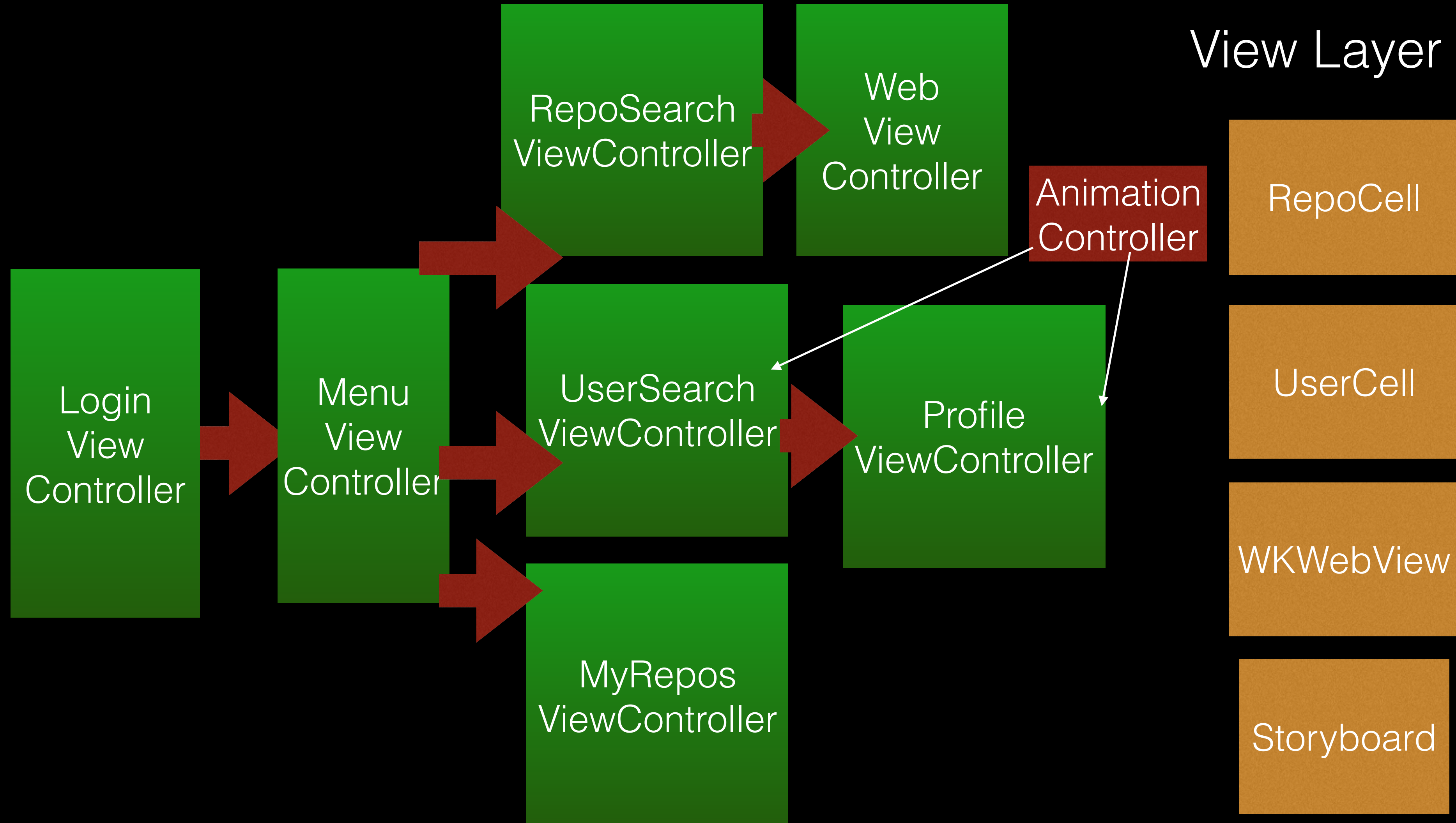
View Layer

RepoCell

UserCell

WKWebView

Storyboard



Static Table View

Static Table Views

- Static Table Views were a new feature that debuted with storyboards in iOS 5
- Cant do them in Nibs
- Instead of the table view dynamically laying out the cells by querying the datasource, the table view relies strictly on the layout (and number) of cells on the storyboard
- **Must be laid out in a UITableViewController or subclass of it**

Why Static?

- Static table view are great for creating menus or settings screens
- Specifically when the number of rows does not have to be dynamic or change over time
- Way easier to create distinct cells vs all the cells looking the same in a dynamic table view
- Obviously, way less code to write

Demo

HTTP & Client/Server Model

HTTP (Hyper Text Transfer Protocol)

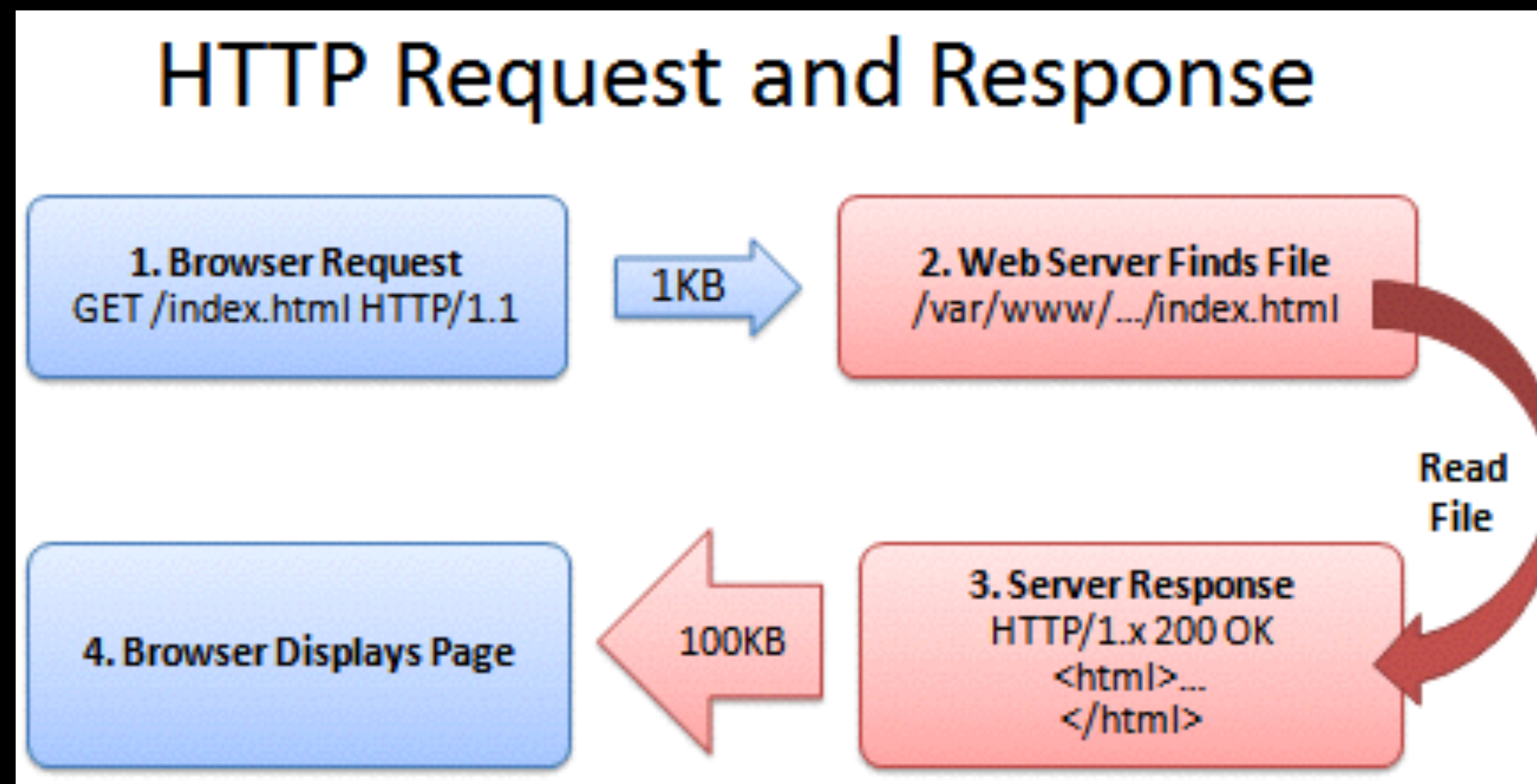
- **HTTP** is a protocol that computers on the internet uses to format and transmit resources (pages,files, data) on the web.
- It defines how web servers and clients should respond to various commands.
- When entering a URL into your browser, you are just sending an HTTP command to a web server to retrieve a resource.

HTTP & URL

- A **resource** is anything that can be identified by a URL. It is the R in URL.
- So your browser is considered an *HTTP Client*, and it sends requests to *HTTP(s) Servers*.
- The default port for HTTP servers to listen on is port 80, but they can use any port.
- It is a stateless protocol, because each request does not know about any of the previous requests.

HTTP & Client-Server

- HTTP uses a client-server model.
- A client sends a request to a server, and then the server returns a response message, which most often will contain the resource that was requested.



Request and Response Format

- The format of a request and response are very similar.
- They are considered 'English-oriented' and human readable.
- Both start off with an initial line.
- The initial line is where the main differences are between requests and responses.
- After the initial line, there can be zero or more header lines.
- After the header line is a blank line
- And finally you have the optional message body.

Request Example

```
GET /doc/test.html HTTP/1.1
```

```
Host: www.test101.com
```

```
Accept: image/gif, image/jpeg, */*
```

```
Accept-Language: en-us
```

```
Accept-Encoding: gzip, deflate
```

```
User-Agent: Mozilla/4.0
```

```
Content-Length: 35
```

```
bookId=12345&author=Tan+Ah+Teck
```

Request Line

Request Headers

Request
Message
Header

A blank line separates header & body

Request Message Body

Response Example

HTTP/1.1 200 OK

Date: Sun, 08 Feb xxxx 01:11:12 GMT

Server: Apache/1.3.29 (Win32)

Last-Modified: Sat, 07 Feb xxxx

ETag: "0-23-4024c3a5"

Accept-Ranges: bytes

Content-Length: 35

Connection: close

Content-Type: text/html

<h1>My Home page</h1>

Status Line

Response Headers

Response
Message
Header

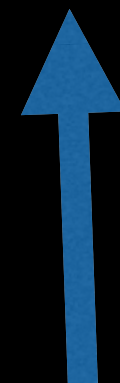
A blank line separates header & body

Response Message Body

Request Initial Line

- A request line has three parts and they are **separated by spaces**.
- The first part is the method name (more on this in a bit)
- The second part is the local path of the request resource
- and the last part is the version of HTTP being used
- Example:

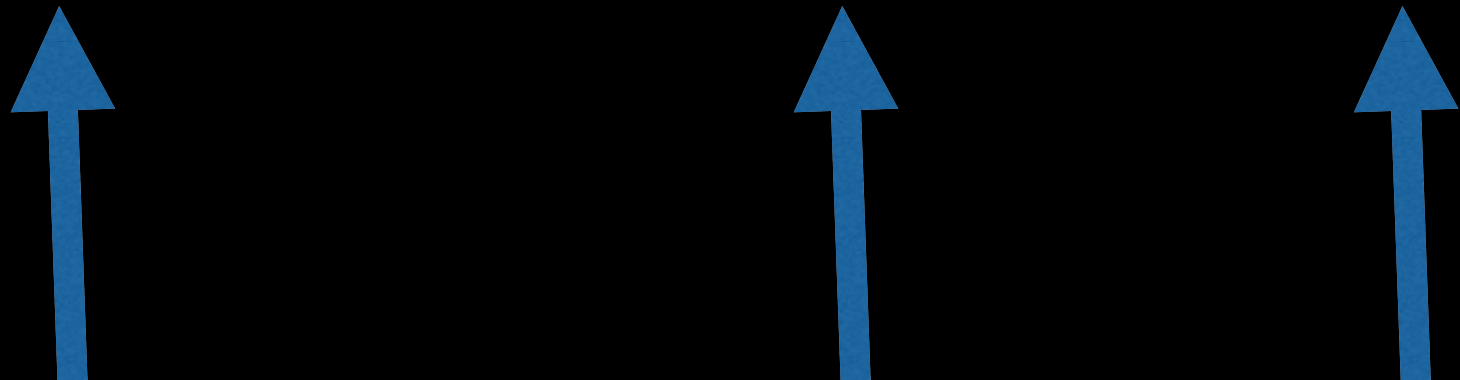
GET /NFL/seahawks/tickets.html HTTP/1.1



Response Initial Line, aka the Status Line

- A status line has three parts and they are **separated by spaces**.
- The first part is the version of HTTP being used
- The second part is a response status code
- and the last part is an english reason phrase describing the status
- Example:

HTTP/1.1 404 Not Found



Most common HTTP Methods (Verbs)

- GET - Most common HTTP method. Retrieves whatever resource is at the URL location. Your browser history is just a history of all the GET requested you have made.
- POST - Method used to request that the server accept data enclosed in the request. When you tweet, you are using POST to create a new record.
- DELETE - Used to delete a resource.
- PUT - Similar to POST, but instead of creating a new record, you are updating a preexisting one.

HTTP Status Codes

- For the most part only a few status codes that mobile apps need to check for
- 200 OK - standard response for a successful HTTP request
- 400 Bad - bad request, most likely syntax error
- 401 Unauthorized - authentication was required but not provided or incorrect in request
- 403 Forbidden - Request was valid, but the server refuses to respond.
- 404 Not found - The requested resource was not found
- 429 Too Many Requests - rate limited
- 5xx Server Error - not your app's fault!

Header lines

- Header lines are one line per header and they take the format of:
 - “Header-Name: value”
- The header name is not case-sensitive
- You can have as many spaces or tabs between the : and the value
- Header lines that begin with space or tab are a part of the last header line for easy multi-line reading.

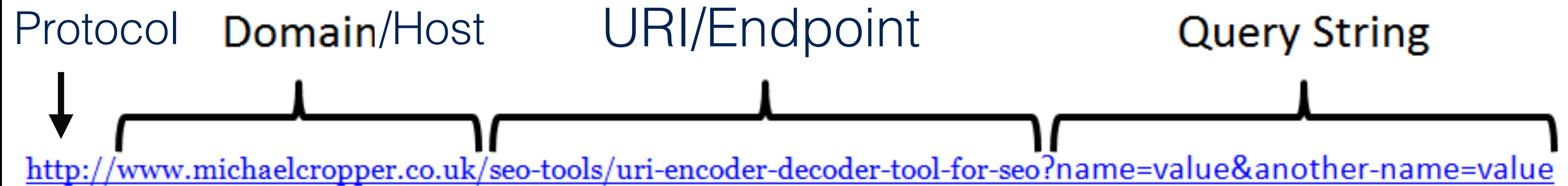
Specific Headers

- While the HTTP protocol itself does not require any header fields to function, API services require certain header fields to be set. Good API's will tell you exactly which header fields you need. A lot of them won't :(
- Content-Type header field is used to specify the nature of data in the body of the request. **You usually need to manually set this if you are doing a POST/PUT call**
- Authorization header field can be used to pass credentials for protected resources. Usually an OAuth token (OAuth tomorrow)
- The Apple provided classes we use to make network calls will fill out most of the headers we need to make our requests (yay)

The Message Body

- Any HTTP message may have a body after the header lines.
- In a request, this is where the appropriate data or files are placed in a POST.
- In a response, this is where the requested resource is (HTML, JSON, XML, etc)
- Whenever there is a body, Content-Type and Content-Length are usually included in the header lines so the client can make sure everything came over the wire as intended

URL



Domain:	The physical server where your website is hosted
URI:	The identifier which maps to files on your server
Query String:	Part of a GET request to easily pass in values to customise the output

* Note: URI stands for Uniform Resource Identifier

HTTP GET Example

- Lets say I wanted to get the box score for a specific mariners game. Heres the URL I would use for my GET request:
 - ★ <http://scores.espn.go.com/mlb/boxscore?gameId=340718103>
- The protocol is http://
- the domain/host is scores.espn.go.com
- the endpoint/URI is mlb/boxscore
- the query string is gameId=340718103. The beginning of the query string is always marked by a question mark. You can add multiple parameters by appending them with a & (gameID=43&playerID=24543&playID=334252354)

Foundation Network Objects

- NSData - an object oriented wrapper for byte buffers (binary data).
- NSData has a number of methods to create data objects from raw bytes.
- `let path = "/u/smith/myFile.txt"`
- `let myData = NSData(contentsOfFile:path)`
- Can also go the other way:
- `let image = person.Image`
- `let imgData = UIImagePNGRepresentation(image) as NSData`

Foundation Network Objects

- NSURL : object that represents a URL that can contain the location for a resource on a remote server or a path of a file on disk.
- Can examine the URL .scheme (http), .host (www.example.com), .path (/scripts), and .query string (name=value) via properties on the NSURL object.
- Usually instantiated with a string.
- NSURLRequest is a class used to make a simple URL HTTP Request.
- Instantiate with an NSURL object.
- Has many properties for specifying the attributes of the HTTP request (http method, body, header fields), if you need to set these things use NSMutableURLRequest

Demo

Web APIs

- Now that you Learned The Internet, lets look at how our iOS apps can communicate with web apps/sites
- API, or Application programming Interface, is way for parts of software to interface with other software.
- Web APIs, what you will be working with, are defined as a set of HTTP requests and response messages usually in JSON or occasionally in XML.
- Most apps on your iPhone are just clients for a web api (Facebook, Twitter, Instagram, Spotify, etc)

REST APIs

- REST is an acronym for **RE**presentational **S**tate **T**ransfer. It is not a protocol, it is just a architectural style.
- You will see the term REST API a lot during your job search. It's a bit of a buzz word now, but it is essentially an API that follows certain constraints:
- Uniform Interface: Resourced based Endpoints that are consistent
- Stateless: The required state to handle the request is all contained within in the request itself. The server doesn't need to keep track of communication histories.

REST APIs

- Cacheable: Responses must defines themselves as cacheable or not
- Client-Server: Client and Server concerns are separated
- Layered System: Client doesn't know(or care) if they are connected to the main server or a load balancing server.
- Code on Demand (optional): Servers can temporarily extend or customize functionality of a client by transferring logic to them.

Web API Client workflow

1. Client makes a request to the server at a specific endpoint
2. Server receives request, does some server magic (queries DB/ generates data), and then sends back response
3. Client checks the response http status code
4. If its in the 200 range (everything is good), client parses the JSON response into model objects
5. Client updates UI and state with newly acquired model objects

NSURLSession

NSURLSession

- The NSURLSession class and related classes provide an API for making HTTP requests.
- NSURLSession is highly asynchronous.
- NSURLSession has two ways to use it, with completion handlers (closures) or delegation.
- We will focus on the closure aka callback way

NSURLSession Setup

- NSURLSession is initialized with a NSURLSessionConfiguration.
- NSURLSessionConfiguration has 3 types:
 1. Default session - disk based cache
 2. Ephemeral session - no disk based caching, everything in memory only
 3. Background session - similar to default, except a separate process handles all data transfers
- Configurations have many properties for customization. For example you can set the maximum number of connections per host, timeout intervals, cellular access or wifi only, and http header fields.
- NSURLSession also has a sharedInstance singleton available to use based on a default session. We will use this one.

NSURLSession Setup

```
var configuration =  
NSURLSessionConfiguration.ephemeralSessionConfigur  
ation()  
  
self.urlSession = NSURLSession(configuration:  
configuration)
```

NSURLSession

- All http requests made with NSURLSession are considered 'Tasks'. Think of them as little minions for your session
- A task must run on a session (NSURLSession class)
- Three types of tasks:
 1. Data tasks - receive and send data using NSData objects in memory
 2. Upload tasks - send files w/ background support
 3. Download tasks - download w/ background support

NSURLSession Data Task

```
var request = NSMutableURLRequest(URL: NSURL(string: "https://api.github.com/search/repositories?q=\n(string)"))
    request.HTTPMethod = "GET"

    let repoDataTask = self.urlSession.dataTaskWithRequest(request, completionHandler: {(data,\nresponse, error) in

        if error {
            //do something for general error
        }
        else {
            //do switch statement on response code and do something with the data
        }
    })
    repoDataTask.resume()
```

 Don't forget this line!!!

Demo

JSON parsing with Swift

1.2

Swift 1.2 Optional Binding

- Swift 1.2 has a much improved feature of optional binding: simultaneous optional bindings!
- This helps us avoid the pyramid of doom style if-lets we saw in week 1 with our twitter JSON parsing
- The syntax is simple, just use commas to separate each binding.

Old way

```
class func tweetsFromJSONData(data : NSData) -> [Tweet] {

    var tweets = [Tweet]()
    var error : NSError?
    if let jsonObject = NSJSONSerialization.JSONObjectWithData(data,
        options: nil, error:&error) as? [[String : AnyObject]] {

        for object in jsonObject {
            var text : String
            if let text = object["text"] as? String {
                if let userInfo = object["user"] as? [String : AnyObject] {
                    {
                        if let id = object["id_str"] as? String {
                            if let username = userInfo["name"] as? String {
                                if let profileImageURL =
                                    userInfo["profile_image_url"] as? String {
                                    let tweet = Tweet(text: text, username: username,
                                        id: id, profileImageURL: profileImageURL)
                                    tweets.append(tweet)
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return tweets
}
```


New way

```
class func tweetsFromJSONData(data : NSData) -> [Tweet] {  
  
    var tweets = [Tweet]()  
    var error : NSError?  
    if let jsonObject = NSJSONSerialization.JSONObjectWithData(data,  
        options: nil, error:&error) as? [[String : AnyObject]] {  
  
        for object in jsonObject {  
  
            if let  
                text = object["text"] as? String,  
                userInfo = object["user"] as? [String : AnyObject],  
                id = object["id_str"] as? String,  
                username = userInfo["name"] as? String,  
                profileImageURL = userInfo["profile_image_url"] as? String  
  
            {  
                let tweet = Tweet(text: text, username: username, id: id,  
                    profileImageURL: profileImageURL)  
                tweets.append(tweet)  
            }  
        }  
    }  
    return tweets  
}
```


Where clause

- You can use a where clause, which acts like an expression in a traditional if statement.
- The where clause is evaluated after all the bindings are evaluated:

```
if let a = a, b = b, c = c where c != 0 {  
    println("(a + b) / c = \((a + b) / c)")    // (a + b) / c = 5  
}
```

Demo

Swift: Value and Reference Types

*Sourced from Apple's official Swift blog

Swift Types

- Types in Swift fall into two categories: value types and reference types
- Value types: each instance keeps a unique copy of its data. passed by copy
- Reference types: instances share a single copy of data. usually defined as a class.
- So what are the benefits of each and how do you choose the right type?

Value types example

```
// Value type example
struct S { var data: Int = -1 }
var a = S()
var b = a // a is copied to b
a.data = 42 // Changes a, not b
println("\$(a.data), \$(b.data)") // prints "42, -1"
```

Reference types example

```
// Reference type example
class C { var data: Int = -1 }
var x = C()
var y = x // x is copied to y
x.data = 42 // changes the instance referred to by x (and y)
println("\$(x.data), \$(y.data)") // prints "42, 42"
```

Safety & Mutation

- One primary reason of choosing value types over reference types is ease of mind
- If you always get a unique, copied instance, you never have to worry about another part of your app changing the data under the covers
- Huge benefit when dealing with multi-threaded environments, as a different thread altering your data while another thread uses that data causes really nasty bugs that can be hard to debug

Classes and Immutability

- Classes can achieve the same level of safety as value types by simply having all of its properties as immutable types
- In addition to the properties, the class should only expose methods that don't alter its data
- NSURL is an example of a class constructed like this

How to choose

- Use a value type when
 - You want copies to have independent state
 - The data will be used in code across multiple threads
 - The data properties you are storing are relatively simple, and are value types as well
 - You don't need inheritance
- Use a reference type when
 - You have relatively complex data
 - You want to create shared, mutable state

Swift Structs

- Structs in Swift are a value type
- Structs have an advantage in performance over classes. They are allocated on the stack instead of the heap (more in this later in the course)
- Use less memory due to no isa and refcount flags (more on this later in the course)
- Structs do not support inheritance

Swift Structs

- Structs are defined in the same way as classes are:

```
struct Resolution {  
    var width = 0  
    var height = 0  
}
```

- Structs can have properties and methods, just like a Class

Swift Structs

- Choose structs instead of a class when:
 - The structure's primary purpose is to encapsulate simple data values (strings, numbers, other structs)
 - Passing by copy will not prevent you from implementing features
 - Properties of the structure are themselves structs
 - The structure does not need to inherit properties or behaviors from another existing type
 - The type will be manipulated by multiple threads/queues

Demo

UISearchBar

UISearchBar

- The UISearchBar class is a text field based control.
- The search bar provides a text field for text input, a search button, a bookmark button, and a cancel button.
- It relies on its delegate to perform searches when one of its buttons is pressed.
- You can also embed a search bar into a tableview by dragging it into the tableview on storyboard.

Demo