CS275
Fall 2020

iOS: Web Services

BNRG CHAPTER 20

Photorama: Chapters 20-23

- This is a complex app
- Chapter 20: web services, JSON, threads
- Chapter 21: collection views
- Chapters 22-23: persistence, through Core Data

Structure of Client/Server Apps

The client (the front end) makes requests to a web server (the back end)

- using HTTP: a protocol (standardized format) for information exchange

The request can be static in nature ("give me the page www.uvm.edu")

- or can include values based on input from the user
- in this case, the server responds with customized data

Client/Server Communication

The server (the back end) then responds to the request from a client

- by producing and sending data

The data can be simple or complex

- and might be structured as raw text or JSON or xml

JSON: JavaScript Object Notation

JSON is a human-readable encoding of data

- data can be numbers, strings, arrays, or dictionaries
- arrays and dictionaries can contain numbers, strings, arrays, and dictionaries
- advantage: easy to parse; but we have to know what to expect in order to parse it

Example:

```
{
     "name": "Christian",
     "friends": ["Stacy", "Mikey"],
     "job": {
          "company": "Big Nerd Ranch",
          "title": "Senior Nerd"
     }
 }
```

RESTful API

REST = Representational State Transfer

Framework for enabling a client application to make queries to a server for information

- that the server then provides to the client
- in a way that does not require the server to save state

Each query from the client is independent

A RESTful API completely decouples the front end from the back end

`URLComponents, URLQueryItem`

These are structs from the Apple Foundation framework

`URLComponents`

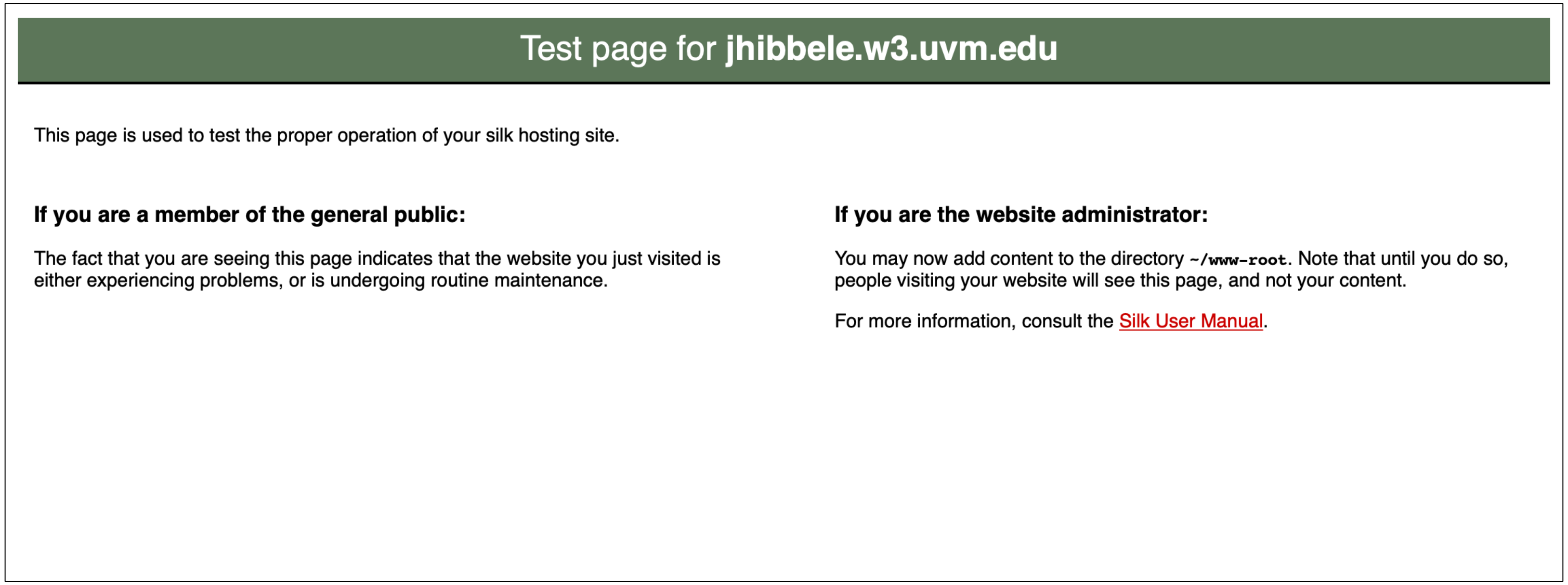
- structure that builds a URL from its parts (or parses a URL into its parts)

`URLQueryItem`

- a single name-value pair from the query portion of a URL

Simple HTTP Query

Here's a static web page



Test page for **jhibbele.w3.uvm.edu**

This page is used to test the proper operation of your silk hosting site.

If you are a member of the general public:

The fact that you are seeing this page indicates that the website you just visited is either experiencing problems, or is undergoing routine maintenance.

If you are the website administrator:

You may now add content to the directory `~/www-root`. Note that until you do so, people visiting your website will see this page, and not your content.

For more information, consult the Silk User Manual.

Simple HTTP Query

Here's a simple query to this static web page

```
static func URLTest() {
    let session: URLSession = {
        let config = URLSessionConfiguration.default
        return URLSession(configuration: config)
    }()
    let components = URLComponents(string: "https://jhibbele.w3.uvm.edu")!
    let request = URLRequest(url: components.url!)
    let task = session.dataTask(with: request) {
        (data, response, error) -> Void in
        if let data = data {
            if let string = String(data: data, encoding: .utf8) {
                print("result is \(string)")
            }
        }
    }
    task.resume()
}
```

initialize a session: kind of like the mothership for doing HTTP requests

create a task to query the specified URL (the `URLRequest`) and and return the data from the query

tasks are created in a suspended state; this starts the task

Query Result

Here's the result:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
          <head>
                    <title>Test Page for the Apache HTTP Server on Red Hat Enterprise Linux</title>
                    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
                    <style type="text/css">
                              /*<![CDATA[*/
                              body {
                                        background-color: #fff;
                                        color: #000;
                                        font-size: 0.9em;
                                        font-family: sans-serif,helvetica;
                                        margin: 0;
                                        padding: 0;
                              }
                              :link {
                                        color: #c00;
                              }
                              :visited {
                                        color: #c00;
                              }
                              a:hover {
                                        color: #f50;
                              }
                              h1 {
                                        text-align: center;
                                        margin: 0;
                                        padding: 0.6em 2em 0.4em;
                                        background-color: #5c7659;
                                        color: #fff;
                                        font-weight: normal;
                                        font-size: 1.75em;
                                        border-bottom: 2px solid #000;
                              }
```

etc.

URLRequest and URLSession

URLRequest

- encapsulates information about the communication from the client to the server
- contains the URL itself
- also other metadata (timeout interval, etc.)

URLSession

- an API that contains classes that use a request to communicate with a server
- serves as a factory to produce an instance of URLSessionTask
- URLSessionTask: a particular kind of communication with a server

`URLSessionTask`

Embodies a particular kind of communication with a server

- `URLSessionDataTask`: retrieves data from the server and returns it as `Data`
- `URLSessionDownloadTask`: retrieves data from the server and returns it as a file
- `URLSessionUploadTask`: sends data to the server

a `Data` instance holds some number of bytes of binary data

Creating a `URLSession`

```
class PhotoStore {
    private let session: URLSession = {
        let config = URLSessionConfiguration.default
        return URLSession(configuration: config)
    }()
}
```

It's possible to create different URL sessions with different configurations

- this uses the default configuration

`URLSession` and `URLRequest`

`URLSession` acts as a factory to create a `URLRequest`

- The `URLRequest` then gets created as a new task, which performs the actual HTTP request

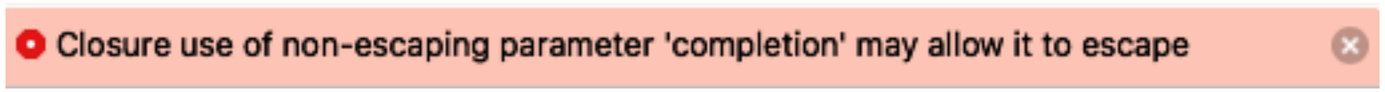
But sending a HTTP request is an asynchronous process

- hopefully, at some point in the future, the HTTP request will be satisfied, and we'll get a response
- we must provide a callback function (technically, a closure) that will be called when the request has been satisfied
- the callback will also handle error situations

Escaping Closures

If you pass a closure as a parameter to a method, and the closure can be invoked after the method returns, then that closure is escaping

- unless you mark the closure as `@escaping`, the compiler will complain:



```
Closure use of non-escaping parameter 'completion' may allow it to escape
```

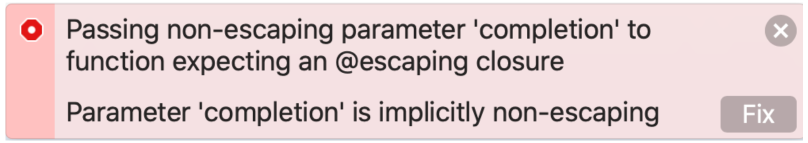
https://cocoacasts.com/what-do-escaping-and-noescape-mean-in-swift-3

If a closure is passed as an argument to a function and it is invoked after the function returns, the closure is escaping.

Escaping Closures

Example:

```
static func URLtest2(completion: (Data?, URLResponse?, Error?) -> Void) {
    let session: URLSession = {
        let config = URLSessionConfiguration.default
        return URLSession(configuration: config)
    }()
    let components = URLComponents(string: "https://jhibbele.w3.uvm.edu")!
    let request = URLRequest(url: components.url!)
    let task = session.dataTask(with: request, completionHandler: completion)
    task.resume()
}
```



Passing non-escaping parameter 'completion' to function expecting an @escaping closure

Parameter 'completion' is implicitly non-escaping

Escaping Closures

Here's the fix:

```
static func URLtest2(completion: @escaping (Data?, URLResponse?, Error?) -> Void) {
    let session: URLSession = {
        let config = URLSessionConfiguration.default
        return URLSession(configuration: config)
    }()
    let components = URLComponents(string: "https://jhibbele.w3.uvm.edu")!
    let request = URLRequest(url: components.url!)
    let task = session.dataTask(with: request, completionHandler: completion)
    task.resume()
}
```

New Project: Photorama

Uses flickr's RESTful API to make requests for photos

- example query – key-value pairs:

https://api.flickr.com/services/rest/?method=flickr.photos.getRecent&api_key=sdf89dsf8dfdf9&extras=url_h,date_taken&format=json&nojsoncallback=1

- `method` specifies the endpoint—like a function that is called on the server side
- `api_key` is a key that you obtain that "authenticates" you to flickr's web server
- `extras` specify other data you want
- `format` specifies that you want JSON
- `nojsoncallback` specifes that you want raw JSON

Encapsulation

Good practice: put all of the code for interacting with the flickr API into some kind of container

- here the container will be a struct: FlickrAPI
- this will contain all the details of creating URL requests
- and also parsing the JSON replies

FlickrURL code

```
struct FlickrAPI {
    private static let baseURLString = "https://api.flickr.com/services/rest"
    private static let apiKey = "44b38344ff198991fc9"

    static var interestingPhotosURL: URL {
        return flickrURL(method: .interestingPhotos, parameters: ["extras": "url_h,date_taken"])
    }

    // and private static func flickrURL(method:parameters:) is on the next slide
}
```

FlickrURL code

```
private static func flickrURL(method: Method, parameters: [String:String]?) -> URL {
    //return URL(string: "")!
    var components = URLComponents(string: baseURLString)!
    var queryItems = [URLQueryItem]()

    let baseParams = [
        "method": method.rawValue,
        "format": "json",
        "nojsoncallback": "1",
        "api_key": apiKey]

    for (key, value) in baseParams {
        let item = URLQueryItem(name: key, value: value)
        queryItems.append(item)
    }

    if let additionalParams = parameters {
        for (key,value) in additionalParams {
            let item = URLQueryItem(name: key, value: value)
            queryItems.append(item)
        }
    }

    components.queryItems = queryItems
    return components.url!
}
```

The Actual Fetch

`URLSessionTask` performs a URL request as an asynchronous task

```
func fetchInterestingPhotos() {
    let url = FlickrAPI.interestingPhotosURL
    let request = URLRequest(url: url)
    let task = session.dataTask(with: request) {
        (data, response, error) -> Void in
        if let jsonData = data {
            if let jsonString = String(data: jsonData, encoding: .utf8) {
                print(jsonString)
            }
        } else if let requestError = error {
            print("Error fetching interesting photos: \(requestError)")
        } else {
            print("Unexpected error with the request")
        }
    }
    task.resume()
}
```

Essentially, this defines a piece of work to do and then creates a "task" to execute the work

When the work is done, the completion handler (here, a closure) is called

The `resume()` call causes the task to execute

The `URLSessionTask` executes asynchronously

Modeling a Photo

Here's how to keep track of a photo received from flickr:

```
class Photo {
    let title: String
    let remoteURL: URL
    let photoID: String
    let dateTaken: Date
}
```

JSONSerialization

This is a built-in class for parsing JSON data

- it unwraps JSON data and peels off the strings, numbers, arrays, dictionaries
- the flickr JSON data has this structure:

```
"photos": array
    "photo": dictionary
        "url_h": string
        "title": string
    "photo": dictionary
        "url_h": string
        "title": string
```

JSON is a great way to encode data
- but we have to know what the format of the data is
- i.e., what to expect for the keys and the values

Parsing JSON

`JSONSerialization` is one way to parse JSON data

SwiftyJSON is a much simpler way

- you can get this from github, and there are "how to" guides on the web

Threads

All of the processing we've done until now in the course runs on the main thread in the app

- this is also called the UI thread
- it's the thread that controls the UI interaction

`URLSession` tasks run on a different thread

- so our downloading tasks run on a different thread than the UI thread
- we can force a completion handler to run on the main thread

```
OperationQueue.main.addOperation {
    completion(result)
}
```

https://developer.apple.com/documentation/foundation/operationqueue