Foundations in App Development

Blake Hall

January 3, 2022

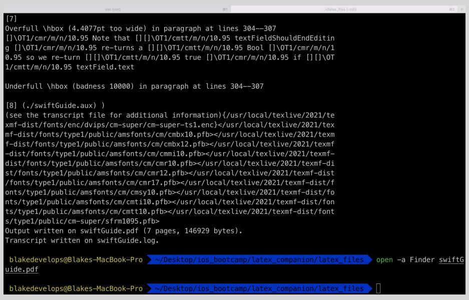
1 Introduction

Hello! My name is Blake Hall and I live in San Diego. I have my Bachelors Degree in Mathematics (focus in Comp. Sci.) from Reed College in Portland, OR. I started programming in roughly 2009. Enough about me. The intention of this book is to teach you (well, me, really) how to develop apps. For money.

2 Console, vim, git, and version control

The Terminal and iTerm

So, most programming books and courses start you off slowly. They introduce you to coding through an IDE (integrated development environment), blah blah blah. Well I'm going to throw you in the deep end knowing that you'll be just fine. We're going to learn how to develop using a tool called the Terminal. On your Mac, press Command-Space and type in Terminal and press enter.



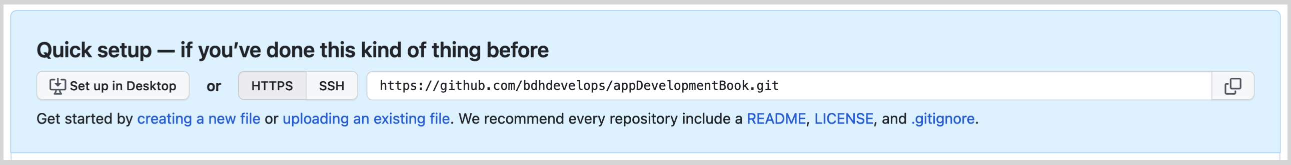
```
[7]
Overfull \hbox (4.4077pt too wide) in paragraph at lines 304--307
[]\OT1/cmr/m/n/10.95 Note that [][]\OT1/cmtt/m/n/10.95 textFieldShouldEndEditin
g []\OT1/cmr/m/n/10.95 re-turns a [][]\OT1/cmtt/m/n/10.95 Bool []\OT1/cmr/m/n/1
0.95 so we re-turn [][]\OT1/cmtt/m/n/10.95 true []\OT1/cmr/m/n/10.95 if [][]\OT
1/cmtt/m/n/10.95 textField.text

Underfull \hbox (badness 10000) in paragraph at lines 304--307

[8] (./swiftGuide.aux) )
(see the transcript file for additional information){/usr/local/texlive/2021/te
xmf-dist/fonts/enc/dvips/cm-super/cm-super-ts1.enc}</usr/local/texlive/2021/tex
mf-dist/fonts/type1/public/amsfonts/cm/cmbx10.pfb></usr/local/texlive/2021/texm
f-dist/fonts/type1/public/amsfonts/cm/cmbx12.pfb></usr/local/texlive/2021/texmf
-dist/fonts/type1/public/amsfonts/cm/cmmi10.pfb></usr/local/texlive/2021/texmf-
dist/fonts/type1/public/amsfonts/cm/cmr10.pfb></usr/local/texlive/2021/texmf-di
st/fonts/type1/public/amsfonts/cm/cmr12.pfb></usr/local/texlive/2021/texmf-dist
/fonts/type1/public/amsfonts/cm/cmr17.pfb></usr/local/texlive/2021/texmf-dist/f
onts/type1/public/amsfonts/cm/cmsy10.pfb></usr/local/texlive/2021/texmf-dist/fo
nts/type1/public/amsfonts/cm/cmti10.pfb></usr/local/texlive/2021/texmf-dist/fon
ts/type1/public/amsfonts/cm/cmtt10.pfb></usr/local/texlive/2021/texmf-dist/font
s/type1/public/cm-super/sfrm1095.pfb>
Output written on swiftGuide.pdf (7 pages, 146929 bytes).
Transcript written on swiftGuide.log.

 blakedevelops@Blakes-MacBook-Pro  ~/Desktop/ios_bootcamp/latex_companion/latex_files  open -a Finder swiftG
uide.pdf

 blakedevelops@Blakes-MacBook-Pro  ~/Desktop/ios_bootcamp/latex_companion/latex_files 
```

Figure 1: "The powerful yet elusive Terminal."

Here's the nitty-gritty of it. You enter a command into the Terminal, and it gives you a response. But I like to imagine it like an old school text RPG. You can move around, you can check your inventory, you can fight monsters (call functions!), etc., etc. It's the same thing. Well, what are the commands that are available to you?

1. Move from one directory to another: `cd targetDirectory`
2. List the contents of the current directory: `ls`
3. Create a new folder in the current directory: `mkdir newDirectory`
4. Create a new file: `touch newFile`

5. Delete a file: `rm fileToRemove`
6. Move a file: `mv fileToMove targetDirectory`


Quick setup — if you've done this kind of thing before

```
https://github.com/bdhdevelops/appDevelopmentBook.git
```

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

Figure 2: "The URL from your new initilized GitHub repository you should copy from GitHub."

vim

git

1. `gh repo create`
2. `git init`
3. `git add *`
4. `git commit -m "Changes I've made since last push"`
5. Visit GitHub and copy the source link for your repository. We will call this `URL`.
6. `git remote add origin URL`
7. `git push -u origin master`

Version control

3 HTML and CSS

Now that you're a pro at navigating the Terminal (or iTerm, whichever path you've chosen), we're going to get to building apps. And we're going to jump right in by learning how to build attractive website with just HTML and CSS. A good way to think about these two languages is like a house: HTML is the wood and beams and foundation, CSS is the paint and decorations and all the things that make a house nice and cozy. Take a look at the HTML file below.

Basic HTML File

I'm going to tell you right now there's a heck of a lot of stuff you can ignore here. In fact, here's another copy with all the extra fluff hidden:

Lets work inside out. The string `"This is a paragraph!"` lives inside the `<p>` element (where `p` stands for paragraph). The `<p>` element lives inside the `<body>` element. If you examine the hierarchy of the elements, you'll see that `<body>` doesn't live in `<head>` and `<head>` doesn't live in `<body>`, but `<head>` lives on top of `<body>`. So on your webpage, the head will appear above the body. Inside the `head` element lives the `<title>` element. Finally, you'll see that head and body live inside the biggest element, `<HTML>`.

```
index.html
----------------------------------------------------------------------
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="device-width", initial-scale="1">
        <title> My Webpage </title>
    </head>
    <body>
        <p> This is a paragraph! </p>
    </body>
</html>
----------------------------------------------------------------------
```

Figure 3: Boiler plate HTML.

```
index.html
----------------------------------------------------------------------
<html>
    <head>
        <title> My Webpage </title>
    </head>
    <body>
        <p> This is a paragraph! </p>
    </body>
</html>
----------------------------------------------------------------------
```

Figure 4: All the HTML you need to worry about right now.

CSS is very, very simple. Just like it's easier to paint a house than it is to build it, CSS is that much easier than HTML. Let's say we want to paint the text inside the paragraph white. With code, all we need to do is have some way to select the paragraph we want to paint, and then "paint it". Two step process. One way we can select the paragraph is by giving it an `id`. How do we do that? Simple. Alter the code for the paragraph to look like this:

```
<p id="paragraphToPaint"> This is a paragraph! </p>
```

Now, our parapgraph has an `id` of *paragraphToPaint*. Okay, now that our paragraph has a name we can point to, we can paint it. The CSS code, again, is very simple:

Here, the octothorpe (#) means we're talking about an `id`, `paragraphToPaint` is the value stored in our `id`, `color` is the selector for text color, and `white` is a premade color we're allowed to use. Here's the catch with `ID`s. You can only use 'em once.

For example, if we had three paragraphs and we wanted to paint them all white, we couldn't assign them all the `id paragraphToPaint`. Instead, we use what's called a `class`. We can use as many of those as we

```
#paragraphToPaint {
    color: white;
}
```

```
<p class="paragraphsToPaint"> This is a paragraph! </p>
```

```
.paragraphToPaint {
    color: white;
}
```

like. The only two differences are: one, we use a `.` instead of a `#`; and two, we use the term `class` instead of `id`. Here's the code:
This will ensure that all of the paragraph receive a fresh coat of white paint. We can assign classes and IDs to just about any HTML element, allowing us a wide variety of avenues to take when designing an app. Additionally, CSS has a incredible array of selectors that allow you to make a website look like anything you can dream of. It's super easy to change background colors, fonts, borders, margins, the list goes on and on. Literally. You can find a list of all CSS properties here (check out the bar on the left):

`https://www.w3schools.com/css/default.asp`

4 JavaScript, HTML, CSS, and Web Development

5 Xcode, Swift 5, and iOS Development

Section 1: Getting Started with iOS Development and Swift 5

15: The I Am Rich App

1st Module Game Plane

Very simple app that sold for $999. It had no functionality. Eight people bought it.

This module will cover the following:

1. How to create and set up a new iOS project from scratch.
2. Get an overview of Xcode, the software for creating iOS apps.
3. How to design your app in Xcode using iOS components.
4. How to incorporate your own image assests into your app.
5. How to design and create a custom app icon.
6. How to run your app on a Simulator and the iPhone.

17: Let's Create a Brand New Xcode Project

Creating and setting up a new iOS project from scratch

Open Xcode. Click **Create a new Xcode project**. Create an iOS ⇒ App. Set **Interface** to *Storyboard*. Set **Organization Identifier** to *com.yourNameHere*. The other defaults are fine: **Team** can stay *none*; Life

Cycle can stay *UI Kit App Delegate*; **Language** is *Swift*; and the three checkboxes at the bottom remain unchecked. Click **Next** and save your file anywhere with any name.

Note! Xcode constantly saves the changes you make to your code.

If you ever want to find out where any of your files live, you can right click on any file in the file tree at the top right and click **Show in Finder**. This will open a Finder window containing your file. You can always navigate to that folder and open the project by double-clicking on the *.project* file.
Next we will familiarize ourselves with Xcode's layout.

Getting an overview of Xcode

Expand Xcode as much as possible and open your project. The first screen to pop up is the General tab. We can change our minimum supported iOS version here. We can choose if we want it to run on iPhone and iPad or just one. We can limit our app to only selected orientations/rotations. We can choose our Status Bar style. We do most of our important work within the .swift and .storyboard files.

The Xcode layout is split up into four main areas:

1. the **Status Bar** at the top,
2. the **Navigator Bar** at the left,
3. the **Main Storyboard** at the center (when *Main.storyboard* is selected in the **Navigator Bar**)
4. the **Inspector Bar** at the right.

In the **Inspector Bar**, there is a tab called the **Size Inspector**. First drag and drop an element onto the story board from the **Object Library** (plus shaped button at the upper right of the storyboard view. Select the element. Then, at the top of the **Inspector Bar**, navigate to the **Size Inspector**. Here, you can set the *x* and *y* coordinates of the center of the element; additionally you can set the *height* and *width* properties of the element.

While a storyboard is selected, there is a bar visible just to the right of the **Navigator Bar** called the **Document Outline**. If this were PhotoShop, this would be where the *layers*. Finally, the very bottom pane is the **Debug Pane**.

Note! All of these bars and windows can be toggled on and off.

Xcode has Light and Dark Mode capability – head to **Xcode** ⇒ **Preferences...** ⇒ **Appearance** to select which one you like.

19: Let's Design the User Interface!

Design your app in Xcode using iOS components

Ensure *Main.storyboard* is selected in the **Navigator Bar**. At the bottom of the storyboard view there is a tiny icon of an iPhone next to the text“iPhone 11". By clicking on either, we can selected which phone Xcode will emulate while running our program. Different iPhones have different aspect ratios, resolutions, etc. and will display the same app differently.

In the storyboard view, if things are ever wonky, you can click on *View Controller Scene* in the **Document Outline** to bring things back to center. When dragging an object from the **Object Library** onto the storyboard, there are guidelines for it to snap to.

Example: drag a label onto a storyboard. Navigate to **Attributes Inspector** and change the text of the label. Change the color to white. Change the font size. We can change the *background color* of a *View*.

Section 13: Networking, JSON Parsing, APIs, and Core Location

142: What You'll Make

A beautiful weather app that can get live weather data for the phone's GPS location or search for your own location. The app is also Dark Mode enabled. Here are some core concepts:

1. How to create a dark-mode enabled app and use vector assets.
2. Learn to use the *UITextField* to get user input.
3. Learn about **Swift Protocals** and the **Delegate Design Pattern.**
4. Learn to work with APIs by making HTTP requests with *URLSession*.
5. Parse JSON with the native JSONDecoder.
6. Learn to use computed properties, closures, and extensions.
7. Learn to use Core Location to get the GPS data.

143: Dark Mode and Working with Vector Assets

Setting Up Dark Mode Capability for Our App

As of iOS 13, Dark Mode is available to the entire operating system. An explanation on how colors work with Dark Mode. The weather app uses SF Symbols, Apple's proprietary bundle of symbols for use in iOS development.

144: Learn to Use the UITextField

The *Text Field* object allows the user to input some data into a text field using iOS's keyboard. You can set many properties of the UITextField in the *Inspector Bar*, including one called *Secure Text Entry* which obscures the user's input (as if they were putting in a password). Additionally, one can change the value of the return key. For example, instead of having the return key have text "return", you can have it say "go".

Let's set up *Main.storyboard*. Create IBOutlets for the image, the temperature label, city label, and for the search field. It should look like this when you're done.

```
WeatherViewController.swift
---------------------------

import UIKit

class WeatherViewController: UIViewController {
    @IBOutlet weak var conditionImageView: UIImageView!
    @IBOutlet weak var temperatureLabel: UILabel!
    @IBOutlet weak var cityLabel: UILabel!
    @IBOutlet weak var searchTextField: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()
    }
    // @IBActions go below me
    // ...
}
```

We now wish to add an `@IBAction` below our `viewDidLoad` function by Option-dragging from the search bar to the space between the last two curly braces. Name the `@IBAction` `searchPressed` and set the *Type* to `UIButton`. The idea: user taps on the search field, types in the name of a city, presses the search button, and then we'll be able to access `searchTextField` to see what the user typed.

We can access the value of the `searchTextField` using the `text` method and print it to the console:

```
    @IBAction func searchPressed(_ sender: UIButton) {
        print(searchTextField.text)
    }
```

This is not to be confused with *Placeholder* text, which is text that just sits in the text field until the user interacts with it.

Because `searchTextField.text` has the possibility of being `nil`, it is an optional type and must be unwrapped. One way we can do this is by adding an exclamation point:

```
        print(searchTextField.text!)
```

If you're using the iPhone simulator to run your apps and the keyboard is nowhere to be found, the shortcut is Command-K ti bring it back.

Try running the app and typing something into the search field. Press the return button on the iOS keyboard. And... nothing happens. We can't use an `IBAction`, so we need to think of another way to get the button to do what we want it to do. And that's by using a `delegate`.

Start by adding a comma and `UITextFieldDelegate` after `UIViewController` in *WeatherViewController.swift*:

```
class WeatherViewController: UIViewController, UITextFieldDelegate {
    ...
}
```

Inside `viewDidLoad()` we are going to initialize the `delegate` as `self`.

```
        searchTextField.delegate = self
```

When the interacts with `searchTextField`, the search field will notify `WeatherViewController` about what happened. For example, we are going to use the premade function `textFieldShouldReturn` which asks the delegate (the weather view controller) if the text field should process the pressing of the return button.

The code in the following function executes exactly when the return button is pressed on the keyboard.

```
    func textFieldShouldReturn(_ textField: UITextField) -> Bool {
        print(searchField.text!)
        return true
    }
```

One problem. The keyboard won't dismiss itself after the return key is pressed. We can resolve this by using the `endEditing` method. We're going to add it to both `searchPressed` and `textFieldShouldReturn`:

```
    @IBAction func searchPressed(_ sender: UIButton) {
        searchTextField.endEditing(true)
        print(searchTextField.text!)
    }

    func textFieldShouldReturn(_ textField: UITextField) -> Bool {
        searchTextField.endEditing(true)
        print(searchTextField.text!)
        return true
    }
```

Now the keyboard should disappear upon either a press of the search button or of the return key on the keyboard. We now want to clear the search field after we're done editing it. Again, there's a `delegate` method for that.

```
    func textFieldDidEndEditing(_ textField: UITextField) {
        searchTextField.text = ""
    }
```

This code will run any time a text field is done being editing, as long as the `endEditing` method is included in the function. Finally, we have `textFieldShouldEndEditing`, where we can define functionality for when the user tries to get out of the keyboard screen. Under what circumstances might we want to lock a user into the keyboard screen? Well, we can check to make sure that the user is inputting what we want them to.

Think of when you sign up for a new website and we want to lock the user into the keyboard screen until the input a password that has a number, an uppercase character, and a special character. Below we make sure that `textField.text` isn't an empty string or else we use the text field's *placeholder* value to display a message to the user:

```
func textFieldShouldEndEditing(_ textField: UITextField) -> Bool {
    if textField.text != "" {
        return true
    } else {
        textField.placeholder = "Type something..."
        return false
    }
}
```

Note that `textFieldShouldEndEditing` returns a `Bool` so we return `true` if `textField.text` is not an empty string (and therefore the user entered something, so yes, the text Field Should End Editing). And we return `false` in the other case and keep the user just where they are.

Finally, we wish to access whatever text the user inputted into the search field so that we can use it to search for weather. We're going to utilize our `textFieldDidEndEditing` for this (which again, triggers when the user exits the keyboard screen).

```
func textFieldDidEndEditing(_ textField: UITextField) {
    // Use searchTextField.text to get the weather for the city.
    searchTextField.text = ""
}
```

145 Swift Protocals

What is a Swift Protocol and what does it allow us to do? Think of it as a certification in your resume. Imagine a CPR certification: doctors need it, paramedics need it, etc. A protocol defines a set of requirements. Then, a class or struct adopts a protocol and from there on it needs to meet the requirements set forth in the protocol:

```
protocol MyProtocol {
    // Define requirements.
}

struct MyStruct: MyProtocol {}
ckass MyClass: MyProtocol {}
```

Create a new **macOS ⇒ Command Line Tool** in Xcode and name it Protocols Demo, hit Finish, and save it anywhere. Clear out *main.swft*. Let's define a class for birds. For now, our bird can fly and lay an egg if it's female. Additionally, let's create an Eagle class from our Bird class, but let's say eagles can Soar in addition to Fly. Finally, we'll create a Penguin class, and we'll say Penguins can swim too.

```
main.swift
---------------------------------------------------------------------
---
class Bird {
    var isFemale = true
    func layEgg() {
        if isFemale() {
            print("The bird lays an egg.")
        }
    }
    func fly() {
       print("The bird flies.")
    }
}

class Eagle: Bird {
    func soar() {
        print("The eagle soars.")
    }

class Penguin: Bird {
    func swim() {
        print("The penguin swims.")
}
-----------------------------------------------------------------------------
---
```

Alls well and good here. Our birds and eagles can fly, lay eggs, soar. Neato. But what's the issue? Well, what about penguins? Penguins can swim and lay eggs, but they can't fly or soar. We'd like to create a Penguin class from our Bird class, but we can't have our penguins be flying around. Let's say we have a museum and they want to demonstrate the flying capabilities of different birds.

We'll create a struct for the museum and it will have a method `flyingDemo` that takes as input only objects that can fly (here, birds). Next, we'll instantiate an Eagle object and have our Mueseum demonstrate it's flying capabilities.

```
struct Museum(flyingObject: Bird) {
    func flyingDemo(flyingObject: Bird) {
        flyingObject.fly()
    }
}

myEagle = Eagle()
Museum.flyingDemo(flyingObject: myEagle)
```

Run the program, and the console should log `"The bird flies."` Remember, this works because the Eagle class inherited the `fly` method from the `Bird` class. Now, if we made a Penguin object and passed it to tbe Museum's `flyingDemo` method, we'd get the same results. But penguins can't fly! We'll deal with that later.

Let's say our museum wants to demonstrate other types of flight, like airplanes and helicopters. We could create classes for these inherting from `Bird` and override the `fly` method, but then our airplanes and helocp[ters would be able to lay eggs too. Rather, we want museum to be able to check if an object can fly irregardless if it's a bird or not, then demonstrate it's flying capabilities. We do this by using `protocols`.

We'll create a `protocol CanFly` at the beginning of our *main.swift*. Next, we'll define the fly requirement by using the header of a function definition (not the body of the function!). Next, we assign the `protocol` to all classes that can fly by adding "`, CanFly`" after our class names. Finally, we ensure each class marked with the `protocol` actually has a `fly` method. If you mark a class with `CanFly` and don't give it `f;y` method, you will get an error! Note that we don't mark penguin, because penguins can't fly darnit!

```
main.swift
-----------------------------------------------------------------------------
---
protocol CanFly {
     func fly()
     // Notice how we're not defining any fly function right here
}

class Bird {
     var isFemale = true
     func layEgg() {
         print("The bird lays an egg.")
     }
}

class Eagle, CanFly {
     func fly() {
         print("The eagle flies.")
     }
     func soar() {
         print("The eagle soars.")
     }
}

class Penginuin {
     func swim() {
         print("The penguin swims.")
     }
}

class Airplane {
     func fly() {
         print("The airplane flies.")
     }
}
-----------------------------------------------------------------------------
---
```

Finally, we can refactor our `Museum` struct definition so that it only accepts objects with a `fly` methods by using our `CanFly protocol` as the datatype:

```
struct Museum {
    func FlyingDemo(flyingObject: CanFly) {
        flyingObject.fly()
    }
}
```

Note that you can define and use multiple `protocols` defining your class our struct like so: `class MyClass: Class`
and etc.

AJAX and API's in JavaScript

In this section, we're going to learn how to make requests using JavaScript code. It's a concept called **AJAX**: making requests behind the scenes will the user is interacting with the web app. **AJAC** stands for Asynchronous JavaScript and XML (even though instead of XML we use JSON). For example, if we're scrolling down Reddit we want the app to continuously offer new posts. Or, if we have a search bar and we want to present live search results as the user types.

When you use a browser to navigate to a website, you are a making a request. Usually, this request returns a mess of HTML which your browser then constructs into a website you can use. With **AJAX**, we want to request a stripped down object filled with information that we can manupulate easily. The name of this type of object is **JSON**. Here's an example of a request you can make that will return one of these stripped down objects:

api.cryptonator.com/api/ticker/btc-usd

If you follow this link your browser will return an object like the followng:

```
api.cryptonator.com/api/ticker/btc-used in your browser
```

```
{
      "ticker":  {
            "base":  "BTC",
            "target":  "USD",
            "price":  "46318.38591953",
            "volume":  "16442.72301906",
            "change":  "249.44780355"
      },
      "timestamp":  1641249902,
       "success":  true,
       "error":  ""
}
```

An **API** is an application programming interfrace; these are portals into other applications that allow us to gather information, etc. A lot of more useful APIs cost money to use. Lots of APIs work with a data format called **JSON**, or JavaScript Object Notation. We like to use it because it is uniform and universal. It looks like a JavaScript object, but with a few key differences. Each key must by surrounded by quotes in the definition.

The types of data a JSON object can hold include: objects, arrays, strings, numbers, booleans, and null. Now, when we first get our JSON object, it's just a string. It needs to be parsed into a JavaSCript object. Luckily there's an easy JavaScript funciton for that: `JSON.parse`. For now, let's save a dummy JSON object in a variable, then call our parse method on it:

```
const data = "{"key1":value1,"key2":value2,"key3":value3}"
const parsedData = JSON.parse(data)
```

Now we can access `value1`, `value2`, `value3` with `parsedData.key1`, `parsedData.key2`, etc. etc. Additionally, we can go the other way. We can turn a JavaScript object into a string for API usage: `JSON.string`.

Working with Postman

The most basic thing to do with Postman: enter a URL and press Send. If the URL is a webpage, it will return HTML. If it's and API link, it will return a JSON object. There are many different types of requests that can be made, but for now we'll be using the **GET** request, veruses the **POST** reqhest, which will send some to the server.

A JSON object consists of a few different parts: the body (which contains the code we've been working with above), the status (which provides information about the success or failure of our request), and headers (key/value pairs that provide additional information about the object like a timestamp, content-type, etc.).

Query Strings and Headers

Example: tmaze.com/api. Query string:

```
/search/shows?q=:query
http://api.tvmaze.com/search/shows?q=girls
```

TVMaze will return multiple results in a JSON object if there are multiple matches. Show lookup endpoint:

```
/lookup/shows?imdb=:imdbid
http://api.tvmaze.com/lookup/shows?imdb=tt0944947
```

Query strings are information we can append to URLs that will impact what the API returns.

Episode by TVMaze id, season and number. Demonstrate how we can pass multiple values through the URL:

```
/shows/:id/episodebynumber?season=:season&number=:number
http://api.tvmaze.com/shows/1/episodebynumber?season=1&number=1
```

We can use Postman to build query string URLs.

Openweathermap API. You can pass in a city, or a latitude and longitute.

Icanhazdadjoke. Demonstrating how to give a header to an API that requires it. In Postman, enter https://icanhazdadjoke.com into the bar. Next, click the Headers tab and add key `Accept` with value `application/json`. Pressing Send, and we'll get a JSON object with an `id`, a random joke, and a status.

Making XHR's, or XMLHttpRequests

Horrible, old way to do this. Just for appreciation of newer methods. First we make an XMLttoRequest object. Next we define `onload` and `onerror` callbacks. Then we open the request, then we send the request. Example:

```
app.js
--------------------------------------------------------------------------------
const req = new XMLHttpRequest();
req,onload = function() {
   console.log("Request succeeded.");
   console.log(this);
}
req.onerror = function() {
   console.log("Request failed.");
   console.log(this);
}

req.open('GET', 'https://api.cryptonator.com/api/ticker/btc-usd');
req.send();
--------------------------------------------------------------------------------
```

We receive back a request object with a `responseText` property that includes the data we wish to parse. We'll save it as a variable `data` in our `onload` function and log the ticker price:

```
        const data = JSON.parse(this.responseText);
        console.log(data.ticker.price);
```

That's all for XMLHttpRequests.

5.0.1 The Fetch API

Written is response to how sucky XMLHttpRequests are. Fetch does the same thing, but it uses promises, async, and await. Here's the syntax; the result is a Promise.

```
        fetch('https://api.cryptonator.com/api/ticker/btc-usd')
            .then(res =>
                console.log("RESPONSE:", res);
            })
            .catch((e => {
                console.log("ERROR:", e);
            })
```

Fetch's `Promise` result will resolve as soon as it has received the object's headers. So, inside `.then`, there is still a possibility the request will fail. We will add the function `res.json()` which returns a `Promise` that resolves when all of the JSON object is loaded. We'll then return `res.json()` and chain another `.then` which will run once `res.json` has resolved:

```
            .then(res => {
                console.log("Waiting for JSON object parson to finish."
                return res.json()
            })
            .then(data => {
                console.log("Data parsed:", data);
                console.log(data.ticker.price);
            })
```

And now using `async` with error handling:

```
const fetBitCoinPrice = async () => {
    try {
        const res = await fetch('https://api.cryptonator.com/api/ticker/btc-usd');
        const data = await res.json();
        console.log(data.ticker.price);
     catch (e) {
        console.log("Something went wrong.")
     }
}
```

Intro to Axios: A Library for Making HTTP Requests

Axios is a JavaScript library which must be included in our HTML in order for it to work:

```
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js">
</script>
```

```
app.js
---------------------------------------------------------------------------------
axios.get('https://api.cryptonator.com/api/ticker/btc-usd')
    .then(res => {
        console.log(res.data.ticker.price);
    })
    .catch(err => {
        console.log("Error:", err);
    })

const fetchBitcoinPrice = async () => {
    try {
        const res =
            await axios.get('https://api.cryptonator.com/api/ticker/btc-usd')
            console.log(res.data.ticker.price);
    } catch (e) {
        console.log("Error: ", e);
    }
```

Returns a Promise containing data that's already been parsed! `axios.get` only resolves once the entire JSON object has been downloaded, unlike `fetch` which resolves after the headers has been received. Next we'll learn how to configure headers using axios.

```
const getDadJoke = async () => {
    const config = { headers: { Accept: 'application/json' } };
    const res = await axios.get('https://icanhazdadjokes', config);
    console.log(res.data.joke);
}
```

6 References

Syntax Highlight