Introduction to Web services

You may have heard the term web services or possibly had discussions about a service-oriented architecture or maybe a company is even moving towards this kind of architecture, but what exactly does it mean? It can be hard to pin down this term, but for the purposes of this course, we want to understand how to test web services. So let's define it as a function that we can access over the web. Now let's look at the implications that this definition has for testing. To do that, I want to take a step back into something that I learned in grade school when the concept of mathematical functions were introduced to me.

And don't worry, we're not going to get into any deep math here or anything, but we can think about it this way. A function takes in an input and performs an action on the input and then produces an output. Therefore, we can think of a web service as this box here in the middle. We give it inputs. It uses those inputs to decide on calculations or other actions and then it produces an output. These inputs and outputs are what we call an API.

Input > [Service] > Output

API as a definition of

what commands a service will accept from us and

what kind of things it will produce for us.

So let's try to make this just a little more concrete. Let's think about it with a simplified example for a minute. So we'll look at this service here called Math.js

http://api.mathjs.org

and this is a service that takes in a math equation and produces the result for us. So we can see down here an example of it. We can give it this expression and if we do that, it will produce the answer for that expression. So let's just click on this and we can see the answer for that is eight.

So this is just a very simplified example of course that shows what a web service is.

A web service takes in a command, a formatted URL,

and does some calculation on it

and produces an output for us.

So in the way that I've been talking about this, **the service itself is a black box**.

We don't know how this Math.js service is implemented. And when we issue these commands, we don't know what it's doing to figure out the answer for us, but we just know that

**given a certain command, it should produce a certain output.**

And so in this course, we're going to be focusing on this kind of approach to testing web services and APIs. We'll treat the services that we're looking at as black boxes and then we can use a lot of what we can call black box testing techniques to figure out how to do this. So let's move forward with that.

**Understanding API terminology**

As we get into the details of API testing there's some terminology that it'd be good to understand. These are things that are pretty commonly used terms and they help us in the way that we think about and test APIs. So you're probably pretty familiar with some of the terminology around rest APIs. There are the standard actions that are sometimes called verbs that define what you want to do.

API Verbs

get,

post,

put

delete.

**GET** action tells the service that you want to retrieve some information.

This action's very easy to use, so for example we can take a look at the GitHub API.

URL into our browser

api.github.com/users/octocat

and if we hit enter here,

we get back data in this format it's called **json**.

{

"login": "octocat",

"id": 583231,

"node\_id": "MDQ6VXNlcjU4MzIzMQ==",

"avatar\_url": "https://avatars3.githubusercontent.com/u/583231?v=4",

"gravatar\_id": "",

"url": "https://api.github.com/users/octocat",

"html\_url": "https://github.com/octocat",

"followers\_url": "https://api.github.com/users/octocat/followers",

"following\_url": "https://api.github.com/users/octocat/following{/other\_user}",

"gists\_url": "https://api.github.com/users/octocat/gists{/gist\_id}",

"starred\_url": "https://api.github.com/users/octocat/starred{/owner}{/repo}",

"subscriptions\_url": "https://api.github.com/users/octocat/subscriptions",

"organizations\_url": "https://api.github.com/users/octocat/orgs",

"repos\_url": "https://api.github.com/users/octocat/repos",

"events\_url": "https://api.github.com/users/octocat/events{/privacy}",

"received\_events\_url": "https://api.github.com/users/octocat/received\_events",

"type": "User",

"site\_admin": false,

"name": "The Octocat",

"company": "GitHub",

"blog": "http://www.github.com/blog",

"location": "San Francisco",

"email": null,

"hireable": null,

"bio": null,

"public\_repos": 8,

"public\_gists": 8,

"followers": 2546,

"following": 9,

"created\_at": "2011-01-25T18:44:36Z",

"updated\_at": "2019-02-22T15:27:33Z"

}

made our first API request.

The browser automatically interprets this URL as a get request and then returns the actions for us.

**a POST action**

you're telling the server that you want to create something new.

need to specify rather than just a URL and an action

some parameters that define the object that we're trying to create.

I.e. create a new user instead of just get information about a user,

parameters with your request.

what login it needs,

what the name is,

what company it has

So you'd include these parameters.

We can't do a command like this in the address bar in the browser since there's no place to tell the browser that we're trying to do a post, and it doesn't really give us an easy way to do that. But we'll talk a little more later on in the course about some ways that you can test this.

the **PUT** action

is similar to the post except that it acts on an already existing object

need those parameters and stuff but it's an already existing object.

the **DELETE** action

tells the server that you want to delete the object that you give it.

**API Terminology**

**idempotency**

an action that will always result in the same server state, no matter how many times we call it.

a **DELETE** action.

a **PUT** action

a silly example - a bookshelf full of books.

action - you come along and **take all the books off the second shelf**

You can perform the same action again and again but the state of the bookshelf will stay the same. Once you've removed those books (***DELETE*** action), if you come back and try to remove them again you won't change the state of the bookshelf.

If we want to repeat one of them over and over the state on the server shouldn't change.

**Safety**

the action doesn't have any effect on the state.

a ***GET*** action.

I.e. bookshelf example, a safe action would be something **like reading one of the titles on the spine of the book (a GET action).**

nothing on the bookshelf has changed - not affected the state of it at all.

they should not change anything on the server.

So these two terms might be a little more difficult to understand than some of the verbs and we'll get into some of these terms more throughout the course but for now, they should give you a good grasp of at least some of the basic terminology that you might hear when working with APIs and hopefully they give you an understanding of how things might work.

**Types of API’s**

a very common way of creating APIs called ***REST***.

make a REST API, you're following a set of standards laid out in the doctoral thesis of a guy named Roy Fielding.

These are a set of principles that help make API development consistent and accessible. Probably most APIs created today are RESTful in nature, and if you aren't sure what kind of API you're working with, a good guess is that it's RESTful. This course will be primarily geared towards testing and understanding REST APIs,

a few other kinds of APIs that you might see, so that you know what you're looking at.

**REST** - **REpresentation State Transfer**.

a **REST**ful API is one that consistently applies the

actions (verbs)

GET, POST, PUT and DELETE,

to resources (nouns)

usually a URL that may have some parameters.

We've got verbs, and nouns, working together in a RESTful API.

**SOAP** - Simple Object Access Protocol approach.

a SOAP API is more standardized.

It needs to follow a stricter set of rules than a REST API.

These rules are known as the **Web Services Description Language (**WSDL)

and they define what form a SOAP message should take,

what the response should look like.

advantages, compared to REST API - it's self documenting.

So if you give someone the XML that has the rules for a SOAP API, you can build up the entire API, and there's tools that will do this automatically.

disadvantages, in that any changes to the specification, require all users of the API to rebuild the entire thing.

So it helps keep things more controlled, but it also means that there's less independence between the server and the client.

**GraphQL**

is kind of trying to bridge the gap between REST and SOAP in some ways,

so it's a little bit more standardized rules,

but it still has some of the freedom of REST APIs.

**Hypermedia**

describes a way of using RESTful APIs, in a way that the API itself tells you what commands are available to use.

when we send a request to the server, we get back the actions that are available to you, for the object that you're looking at.

i.e. going to a web page.

You ask for a resource, you get back the page, and on that page, there are links that allow you to go to other places on the web.

the same thing, except at an API level.

So, doing this helps make APIs work a little bit better, a little bit more web like, in the way that they work, and so many people suggest that if you're working with REST APIs, you should use a hypermedia approach.

**Challenge**

map out the application you're testing.

map your application

a line diagram, linking different parts of the application together,

a mind map,

some lists of different parts of the application.

Try to figure out

what services it has,

what APIs are available,

what kind of APIs are they,

how do they relate to each other,

what responsibilities do they have,

ask some questions about your application

and figure out how to map it out.

have an understanding of how they're being used and for what purpose.

Now, maybe the way that I put this exercise is leaving you feeling overwhelmed.

So let me give you a couple of pointers to help you on your way.

**the math.js API documentation page**

there's some documentation that this page has that shows us how to use the API. Now, this is a key resource you can use in understanding the layout of an API in your application.

If it's a public API, reading through the documentation can help you understand it, and even if it's just an internal API, you'll probably find that there's some internal developer documentation somewhere that you can look up and use, so documentation is a great source of information when you're looking at testing and mapping out an API.

**talk to the architect or developer who designed the app.**

So in this case, if we look at this, we can see there's this GitHub, and we can go to the GitHub repo, and we can see here's the author for this, so if we were trying to map this out, we might want to contact the author in this case. In your own company, you're going to go talk to the developers who are working on the API, or the architects who helped design it. So, when you ask and talk to these people, you're probably going to get a lot of information that you don't fully understand, and that's okay.

***https://github.com/josdejong/mathjs***

You're starting to slowly get a picture of how things work, and as you discover more information, and learn more in this course, the picture will become less and less fuzzy, and you'll get a better idea of how things are laid out, and how you can effectively test them.

So if you feel particularly brave, you could even

**look through some of the source code**

so in this example, we can see, we have the source code here, we could look at some of the different source code, we could even go

**look at some of the tests, and see how these tests use the code**

and so these are some ways, some strategies, that are going to help you get a picture.

It’s going to be an imperfect picture, it's going to be incomplete information, you're not going to be able to understand everything, but the reality is, that learning something often requires this period of, kind of struggle, and confusion, but if you stick with it, if you do your best, keep poking at it, you'll find that things become less and less confusing, and before you know it, you'll have a firm grasp of the subject.

**Risk of using Services and APIs**

Using APIs and web services gives you a lot of benefits.

A web service allows you to isolate parts of your application,

and

using an API to get data from it allows you hand off some of the work, which lets you

focus on presentation in the UI and   
 not needing to worry about business logic.

There are a lot of benefits that can come from using APIs and web services, but they also come with some **risks**.

encourage you to think about other ones that might be applicable to the application that you're testing.

API changes,

availability,

timing of responses,

performance,

security

these are all different types of risks that we can face in API testing.

**API changes**

**API version changes** could affect the way that we

use it,

interact with it.

**API structure (the schema of the API)** changes

could change the way that different parts of the API work together.

**Server calculation changes**

something changed on the server code so that the same action gives a different result now.

i.e. fixing a bug or something else. But the server can sometimes change the calculations and give us different results from the same action.

**Data Format changes**

**other things**.

So there a lot of ways that an API can change, especially when we're dealing with a live system, so we need to be aware of those things, and we need to think about the risks that come from testing those things.

**Availability.**

**Network issues**

web APIs - APIs that interact over the network.

i.e. disconnection from the internet

even just a slow network

can have an effect on how the API works.

**Permissions**

are we allowed to do the commands that we're giving?

What if permissions change?

So to think about the availability of the resources that you're looking for and using on the server, and how your application handles it when those resources are not there is another important risk to think about in testing APIs.

**Timing**

the client / the front-end - sending the commands

the back end / the server - doing the work

**Out of order calls**

So when we send the commands to the server what happens if those **commands come there in a different order** than we expected? Just because we sent them in a certain order doesn't mean they are received in that order.

could have issues on the network, where those commands are coming in a different order than we expected.

So we need to think about the risks of that, what happens when those commands come out of order.

**Slow calls**

what happens when the network is slow and those commands take a long time to calculate or to come back,

it's just a heavyweight calculation on the server.

How does our application handle it when a call is slow?

Does it time out, or

Are there other issues that happen?

**Concurrent changes**

what if two people are trying to change the same resource at the same time?

What happens in the application?

How does it handle it?

**general risks in software** development but they're especially things that we need to think about **in API testing as well.**

**Performance.**

**Programmatic Access through an API**

So if you're interacting with something through the UI, you can really only make commands happen as fast as you can click, or as fast as you as a user can interact with the user interface.

But if doing it through an API, you can write a script that will send those commands.

We've provided **a programmatic interface**, an interface that programs can use, and so we need to think about the performance implications of that as well.

What happens if someone is sending thousands of requests a second, and

how are we going to deal with that?

**Security.**

So, again, we can do these things programmatically, so there's a lot of things that people can do security-wise that we need to think about as well in API testing.

So all these things apply obviously to software testing in general, but these are things that we need to especially consider and think about in API testing. And some of them are maybe not totally specific to the application you're working on, and maybe there's other ones that you should think about with yours.

So take some time. Think about how these things and other ones that might be affecting you should be considered in the APIs that you're testing.

**Getting Started with API Testing**

**First tool: Postman**

if we want to test APIs, we're going to need to use a tool that will let us easily send and receive commands to the APIs that we're testing.

introduce you to **Postman** as one of the tools that we use.

a very intuitive and easy to use tool when it comes to API testing.

It's free to use and it has a large active user base,

so there's many resources if you find yourself Googling for answers.

www.getpostman.com

open and this is where you can go.

You can download it and follow the installation instructions to set it up. And once you've done that, you can just open Postman and get started right away. So this first splash screen gives you a bunch of information.

look at the **Request**

example and we'll make a request name. call it ***test***

create a collection. call it ***collection1***

for now. And we'll close that. So we've got a request here that sets up how we can interact with an API.

So if you remember, we have the verbs. So here's a list of verbs. And there's quite a big list here, but like I said, we're only going to consider get, post, put, and delete as the common ones that we use. And then here's your URL. So this is where you put in the URL resource. So we have our verb, we have our noun, and then once we have that we can just send it. And then there's a couple other things. So if you remember, when we do posting, we need to specify some parameters. So we can specify those in Postman here with the body.

And there's a couple different ways we can do it. We can either put in a key value pair. So if we had a parameter like username, we could put that in here and we could give it a value that we want to have as our username. Or we could also specify it using the raw format here. And we could put in some text or, quite commonly what we'll do is JSON. So if we wanted to specify it as JSON, we could do the same thing as we did earlier and we could put in username and then give it some value that we wanted it to have.

So if we wanted our username to be some kind of value we could give it that value. And then we could send that and Postman will take care of formatting that in the way that you need to send it as JSON to the server. So those are kind of the basic things that you're going to need. Another thing that you'll probably run into at some point in your testing is authorization. And Postman also makes that fairly easy and straightforward to interact with. So we can see here we've got this authorization tab and there's a bunch of different authorization options that we can choose from.

We'll talk a little bit more about some of these different options and how you can use them later on in the course. But for now, you can see that these options are here, they're available, and if you follow through and read through the documentation you can figure out some of those things as well if you really need to. So this is just a really quick overview of this tool. Like I said, this course isn't about Postman as such. It's about testing APIs. But it's difficult to test an API without a tool. So we'll use Postman in a few different videos throughout this course. So it would be good to download.

It would be good to get a little bit familiar with it and be able to use it at least in a basic way. So, I would suggest download it, poke around a bit, have some fun, and hopefully you'll find that this is a helpful first tool to get you started with API testing.

**Install and setup API challenges**

git clone

<https://github.com/djwester/api-testing-foundations.git>

<http://desktop.github.com>

Pkg manager

<https://www.npmjs.com/get-npm>

click **[Download Node.js and npm]**

which is ***https://nodejs.org/en/***

his package has installed:

* Node.js v10.15.1 to /usr/local/bin/node
* npm v6.4.1 to /usr/local/bin/npm

Make sure that /usr/local/bin is in your $PATH.

cd /Users/mohansrinivasan/Documents/GitHub/api-testing-foundations

Mohans-MacBook-Pro:api-testing-foundations

Mohans-MacBook-Pro:api-testing-foundations mohansrinivasan$ pwd

/Users/mohansrinivasan/Documents/GitHub/api-testing-foundations

Mohans-MacBook-Pro:api-testing-foundations mohansrinivasan$ npm install

npm WARN api-testing-foundations No repository field.

npm WARN api-testing-foundations No license field.

added 231 packages from 132 contributors and audited 441 packages in 10.873s

found 0 vulnerabilities

Mohans-MacBook-Pro:api-testing-foundations mohansrinivasan$

**Exploring an API**

A big part of testing is discovering what there is to test. You could say that a tester is an explorer. And for this video I want you to put on your explorer hat. So let's say you have an API that you need to test. Maybe it's a fairly new API or maybe it's an existing API that's getting changed in some way. In either case, in order to test it you need to know how it works and what options it has. We've already talked about speaking to developers, reading documentation, maybe even looking at the code to even help you get a grasp of how an API is laid out and what it does. But now, we'll talk a little bit more about how you might explore an API once you've got some of those basics down.

look at the Star Wars API

https://www.swapi.co

And let's see what we can figure out. So this API does have full documentation available that shows us what options there are. But we'll pretend that the only information we have is this little hint text here that shows us a couple of URLs.

So the first request we have here is ***people/1*** and let's try that out.

{

"name": "Luke Skywalker",

"height": "172",

"mass": "77",

"hair\_color": "blond",

"skin\_color": "fair",

"eye\_color": "blue",

"birth\_year": "19BBY",

"gender": "male",

"homeworld": "https://www.swapi.co/api/planets/1/",

"films": [

"https://www.swapi.co/api/films/2/",

"https://www.swapi.co/api/films/6/",

"https://www.swapi.co/api/films/3/",

"https://www.swapi.co/api/films/1/",

"https://www.swapi.co/api/films/7/"

],

"species": [

"https://www.swapi.co/api/species/1/"

],

"vehicles": [

"https://www.swapi.co/api/vehicles/14/",

"https://www.swapi.co/api/vehicles/30/"

],

"starships": [

"https://www.swapi.co/api/starships/12/",

"https://www.swapi.co/api/starships/22/"

],

"created": "2014-12-09T13:50:51.644000Z",

"edited": "2014-12-20T21:17:56.891000Z",

"url": "https://www.swapi.co/api/people/1/"

}

So, we'll request it and if we scroll down here we can see the result of that request displayed nicely for us here.

And this gives us some information about structure and layout of this API. So we can see, that we've got these various links here that point us to other resources that the API has. So there's home world and it has a link that points to planets/1. So, clearly planets is one of the resources that we can look at in this API. We also have the films that this person is in and the species that he is, the vehicles, the star ships.

So we can see just with this request we've learned a lot about this API and about the layout and structure and available resources on this API.

Now, not every API is going to be, as well laid out as this one. And this is using something that we call **Hypermedia APIs** that give us some pretty clear guidance on what other things are available. But, most of the time there will be at least some hints in the data that you can gather from one call that will tell you about some other calls that might be available.

Alright, so let's look around and see what else we can learn about this API. Let's go back up here to the request field. And let's just call, **people** by itself without any number and see what we get back.

{

"count": 87,

"next": "https://www.swapi.co/api/people/?page=2",

"previous": null,

"results": [

{

"name": "Luke Skywalker",

"height": "172",

"mass": "77",

"hair\_color": "blond",

"skin\_color": "fair",

"eye\_color": "blue",

"birth\_year": "19BBY",

"gender": "male",

"homeworld": "https://www.swapi.co/api/planets/1/",

"films": [

"https://www.swapi.co/api/films/2/",

"https://www.swapi.co/api/films/6/",

"https://www.swapi.co/api/films/3/",

"https://www.swapi.co/api/films/1/",

"https://www.swapi.co/api/films/7/"

],

"species": [

"https://www.swapi.co/api/species/1/"

],

"vehicles": [

"https://www.swapi.co/api/vehicles/14/",

"https://www.swapi.co/api/vehicles/30/"

],

"starships": [

"https://www.swapi.co/api/starships/12/",

"https://www.swapi.co/api/starships/22/"

],

"created": "2014-12-09T13:50:51.644000Z",

"edited": "2014-12-20T21:17:56.891000Z",

"url": "https://www.swapi.co/api/people/1/"

},

{

"name": "C-3PO",

"height": "167",

"mass": "75",

"hair\_color": "n/a",

"skin\_color": "gold",

"eye\_color": "yellow",

"birth\_year": "112BBY",

"gender": "n/a",

"homeworld": "https://www.swapi.co/api/planets/1/",

"films": [

"https://www.swapi.co/api/films/2/",

"https://www.swapi.co/api/films/5/",

"https://www.swapi.co/api/films/4/",

"https://www.swapi.co/api/films/6/",

"https://www.swapi.co/api/films/3/",

"https://www.swapi.co/api/films/1/"

],

"species": [

"https://www.swapi.co/api/species/2/"

],

"vehicles": [],

"starships": [],

"created": "2014-12-10T15:10:51.357000Z",

"edited": "2014-12-20T21:17:50.309000Z",

"url": "https://www.swapi.co/api/people/2/"

},

{

"name": "R2-D2",

"height": "96",

"mass": "32",

"hair\_color": "n/a",

"skin\_color": "white, blue",

"eye\_color": "red",

"birth\_year": "33BBY",

"gender": "n/a",

"homeworld": "https://www.swapi.co/api/planets/8/",

"films": [

"https://www.swapi.co/api/films/2/",

"https://www.swapi.co/api/films/5/",

"https://www.swapi.co/api/films/4/",

"https://www.swapi.co/api/films/6/",

"https://www.swapi.co/api/films/3/",

"https://www.swapi.co/api/films/1/",

"https://www.swapi.co/api/films/7/"

],

"species": [

"https://www.swapi.co/api/species/2/"

],

"vehicles": [],

"starships": [],

"created": "2014-12-10T15:11:50.376000Z",

"edited": "2014-12-20T21:17:50.311000Z",

"url": "https://www.swapi.co/api/people/3/"

},

{

"name": "Darth Vader",

"height": "202",

"mass": "136",

"hair\_color": "none",

"skin\_color": "white",

"eye\_color": "yellow",

"birth\_year": "41.9BBY",

"gender": "male",

"homeworld": "https://www.swapi.co/api/planets/1/",

"films": [

"https://www.swapi.co/api/films/2/",

"https://www.swapi.co/api/films/6/",

"https://www.swapi.co/api/films/3/",

"https://www.swapi.co/api/films/1/"

],

"species": [

"https://www.swapi.co/api/species/1/"

],

"vehicles": [],

"starships": [

"https://www.swapi.co/api/starships/13/"

],

"created": "2014-12-10T15:18:20.704000Z",

"edited": "2014-12-20T21:17:50.313000Z",

"url": "https://www.swapi.co/api/people/4/"

},

{

"name": "Leia Organa",

"height": "150",

"mass": "49",

"hair\_color": "brown",

"skin\_color": "light",

"eye\_color": "brown",

"birth\_year": "19BBY",

"gender": "female",

"homeworld": "https://www.swapi.co/api/planets/2/",

"films": [

"https://www.swapi.co/api/films/2/",

"https://www.swapi.co/api/films/6/",

"https://www.swapi.co/api/films/3/",

"https://www.swapi.co/api/films/1/",

"https://www.swapi.co/api/films/7/"

],

"species": [

"https://www.swapi.co/api/species/1/"

],

"vehicles": [

"https://www.swapi.co/api/vehicles/30/"

],

"starships": [],

"created": "2014-12-10T15:20:09.791000Z",

"edited": "2014-12-20T21:17:50.315000Z",

"url": "https://www.swapi.co/api/people/5/"

},

{

"name": "Owen Lars",

"height": "178",

"mass": "120",

"hair\_color": "brown, grey",

"skin\_color": "light",

"eye\_color": "blue",

"birth\_year": "52BBY",

"gender": "male",

"homeworld": "https://www.swapi.co/api/planets/1/",

"films": [

"https://www.swapi.co/api/films/5/",

"https://www.swapi.co/api/films/6/",

"https://www.swapi.co/api/films/1/"

],

"species": [

"https://www.swapi.co/api/species/1/"

],

"vehicles": [],

"starships": [],

"created": "2014-12-10T15:52:14.024000Z",

"edited": "2014-12-20T21:17:50.317000Z",

"url": "https://www.swapi.co/api/people/6/"

},

{

"name": "Beru Whitesun lars",

"height": "165",

"mass": "75",

"hair\_color": "brown",

"skin\_color": "light",

"eye\_color": "blue",

"birth\_year": "47BBY",

"gender": "female",

"homeworld": "https://www.swapi.co/api/planets/1/",

"films": [

"https://www.swapi.co/api/films/5/",

"https://www.swapi.co/api/films/6/",

"https://www.swapi.co/api/films/1/"

],

"species": [

"https://www.swapi.co/api/species/1/"

],

"vehicles": [],

"starships": [],

"created": "2014-12-10T15:53:41.121000Z",

"edited": "2014-12-20T21:17:50.319000Z",

"url": "https://www.swapi.co/api/people/7/"

},

{

"name": "R5-D4",

"height": "97",

"mass": "32",

"hair\_color": "n/a",

"skin\_color": "white, red",

"eye\_color": "red",

"birth\_year": "unknown",

"gender": "n/a",

"homeworld": "https://www.swapi.co/api/planets/1/",

"films": [

"https://www.swapi.co/api/films/1/"

],

"species": [

"https://www.swapi.co/api/species/2/"

],

"vehicles": [],

"starships": [],

"created": "2014-12-10T15:57:50.959000Z",

"edited": "2014-12-20T21:17:50.321000Z",

"url": "https://www.swapi.co/api/people/8/"

},

{

"name": "Biggs Darklighter",

"height": "183",

"mass": "84",

"hair\_color": "black",

"skin\_color": "light",

"eye\_color": "brown",

"birth\_year": "24BBY",

"gender": "male",

"homeworld": "https://www.swapi.co/api/planets/1/",

"films": [

"https://www.swapi.co/api/films/1/"

],

"species": [

"https://www.swapi.co/api/species/1/"

],

"vehicles": [],

"starships": [

"https://www.swapi.co/api/starships/12/"

],

"created": "2014-12-10T15:59:50.509000Z",

"edited": "2014-12-20T21:17:50.323000Z",

"url": "https://www.swapi.co/api/people/9/"

},

{

"name": "Obi-Wan Kenobi",

"height": "182",

"mass": "77",

"hair\_color": "auburn, white",

"skin\_color": "fair",

"eye\_color": "blue-gray",

"birth\_year": "57BBY",

"gender": "male",

"homeworld": "https://www.swapi.co/api/planets/20/",

"films": [

"https://www.swapi.co/api/films/2/",

"https://www.swapi.co/api/films/5/",

"https://www.swapi.co/api/films/4/",

"https://www.swapi.co/api/films/6/",

"https://www.swapi.co/api/films/3/",

"https://www.swapi.co/api/films/1/"

],

"species": [

"https://www.swapi.co/api/species/1/"

],

"vehicles": [

"https://www.swapi.co/api/vehicles/38/"

],

"starships": [

"https://www.swapi.co/api/starships/48/",

"https://www.swapi.co/api/starships/59/",

"https://www.swapi.co/api/starships/64/",

"https://www.swapi.co/api/starships/65/",

"https://www.swapi.co/api/starships/74/"

],

"created": "2014-12-10T16:16:29.192000Z",

"edited": "2014-12-20T21:17:50.325000Z",

"url": "https://www.swapi.co/api/people/10/"

}

]

}

So, if we do, people. We can see that we got back our result here that has a list of different people that are available in this API. You can see there's 87 people in this list and we also see something interesting here.

Where we have the next thing. And this gives us a URL that has this **funny little question mark** in it. That question mark is a parameter. So, **the ?page option** is available because there is 87 people in this list. And they're only displaying some of them on each page. So, this I think has the first 10 people displayed here and then if we want to see the next 10. We could add ?page=2 to see he next 10 resources in that list.

So, that is one more thing that we can add to out list to test, right?

**We want to test that this ?page option, works as expected**.

So, if there's an option like this there are probably other options available. So, let's try some out and see if we can find anything that would work.

let's try ***?q*** for query people use ?q

***people/?q=Luke***

nothing seem to happen so that's probably not right.

search for Luke.

***people/?search=Luke***

{

"count": 1,

"next": null,

"previous": null,

"results": [

{

"name": "Luke Skywalker",

"height": "172",

"mass": "77",

"hair\_color": "blond",

"skin\_color": "fair",

"eye\_color": "blue",

"birth\_year": "19BBY",

"gender": "male",

"homeworld": "https://www.swapi.co/api/planets/1/",

"films": [

"https://www.swapi.co/api/films/2/",

"https://www.swapi.co/api/films/6/",

"https://www.swapi.co/api/films/3/",

"https://www.swapi.co/api/films/1/",

"https://www.swapi.co/api/films/7/"

],

"species": [

"https://www.swapi.co/api/species/1/"

],

"vehicles": [

"https://www.swapi.co/api/vehicles/14/",

"https://www.swapi.co/api/vehicles/30/"

],

"starships": [

"https://www.swapi.co/api/starships/12/",

"https://www.swapi.co/api/starships/22/"

],

"created": "2014-12-09T13:50:51.644000Z",

"edited": "2014-12-20T21:17:56.891000Z",

"url": "https://www.swapi.co/api/people/1/"

}

]

}

Now we can see we got back a count of one. There's only one resource in this list. So that worked. That seems like we found another parameter in our API. We can search our API using the search parameter.

And here we've returned back, Luke Skywalker as a result of our search for Luke. So, at this point we could try out some other parameters maybe we could try a

?sort

or something else to see if there's any other options that the API might have. And we're not going to do that.

But, this illustrates to us the fact that **there are ways to poke around and discover things in an API.**

You can do that by looking at the result of the API. You can do that by guessing some of the things and clearly a guess in the dark approach shouldn't be the only way you try to figure out what an API does.

But, it does show us that if you pay close attention to the little hints that the API is giving you. You can find out a lot about it. Just by poking around. Just by asking some questions and being a little bit curious about it. And that's what exploration is all about. It's about using the information that you have where you are to try and help and figure out next steps and where you might want to go. So, pay attention as you go and you'll be surprised at what you can find out about the way an API works.

**Challenge**

In an earlier video I already challenged you to try and map out the API of an app that you might be workin' on. You probably ran into some difficulties while doing this, so in this challenge I want to help you get a feel for some of the things that you can use to help figure out the layout and structure of an API.

For this challenge, I want you to go through this Dog API here at

***https://dog.ceo/dog-api***

figure out what you think is interesting or important in this API.

So the idea of this challenge is to

try and map out all the different paths that you might want to test in this API.

write them down in any way you want,

a mind map

just a list,

create a fairly comprehensive summary of the API paths and commands that should be tested.

As I mentioned, there are a few different sources of information that can help you with this.

the documentation that is provided on this site,

some of the values returned by various calls that you use.

And often by seein' the structure of something you can make an educated guess as to what other options might be available.

About All Sort By Breed

\ | /

All Sub Breeds ——- —- Breeds ——-— Browse Breeds List

| \

Random Image Submit dog

About - from Stanford Dogs Dataset

<http://vision.stanford.edu/aditya86/ImageNetDogs/>

Projects made using Dog API

All Breeds

https://dog.ceo/api/breeds/list/all

Random Image

<https://dog.ceo/api/breeds/image/random>

Sort By Breed

https://dog.ceo/api/breed/hound/images

All Sub Breeds

https://dog.ceo/api/breed/hound/list

https://dog.ceo/api/breed/hound/afghan/images

<https://dog.ceo/api/breed/hound/afghan/images/random>

https://dog.ceo/api/breed/hound/afghan/images/random/***nbr-of-pics***

https://dog.ceo/api/breed/hound/afghan/images/random/***3***

Browse Breeds List

<https://dog.ceo/api/breed/> ***input-specific-breed*** /images/random

Submit your dog - post pic on GitHub

<https://github.com/jigsawpieces/dog-api-images#dog-api-images>

Solution

explore an API and hopefully you've taken a little time to do that on you're own. But now let's take a look at how I approached it. So we are start on the main page here, and there's this URL down here so I'll just copy this and I'm going to put it over in Postman to try this out, we'll send that and we can see that we get back down here a message that includes a link to an image.

So I wonder if we can use this, let's just copy this and let's put it in here and see what happens if we do that call. And there we go, we got back the image, so this is another path that we can use to get information from our API. All right so now let's go back to the documentation. So I'm going to take this URL here and let's do that one next, so let's copy it and put it into Postman.

***https://dog.ceo/api/breeds/list/all***

We will send that and you can see here in Postman we've gotten back a whole long list of different breeds of dogs that are available. One thing that's interesting to note here as we scroll through this list is that some of these dogs have sub-species, so here we have hound and then we have a list of different kinds of hounds, so that's probably another interesting thing that I would want to write down and note as I'm building out my API map, is that some dog species can have sub-species.

And so I would wonder how that fits into resources that might be available and the things that I could do in the API. All right so let's actually try this out, let's see what happens. So maybe if I replace the all here with hounds I can get a list of all the breeds of hounds.

***https://dog.ceo/api/breeds/list/hounds***

So let's send that and see what happens. **No, We get a 404 error.**

***404 Error, page not found. API documentation is located at <https://dog.ceo/dog-api>***

So this is not an available piece of functionality. It seemed intuitively to me that it should be.

That I should get do hounds and get a list of the different breeds of hounds that are available. So that might be something that I'd want to write down and if this was an API that I was testing, maybe discuss with the developer or the designer. Because my intuition was that this should work, but it didn't, so that's some interesting information that we might want to consider.

All right so let's go back to the documentation and see what else we have. Maybe if we click on the by breed link here we can see that we've got another URL, so let's copy this URL and let's bring it over to Postman.

***https://dog.ceo/api/breed/hound/images***

Send that and we've got back a nice long list of different kinds of hound images. Now we can see here that because we've sent the breed hound and it has sub-species so the links here show sub-species so let's modify this link here. And let's put in hound blood.

***https://dog.ceo/api/breed/hound-blood/images***

See if we get blood hounds back. So we send that and it looks like we indeed do filter it down. So that seems to be the way if we want to get sub-species, we have to put the sub-species after the original species.

https://dog.ceo/api/breed/***dogspecies-dogsubspecies***/images

Alright so that's another interesting thing that we'd want to record and write down and know about how the API works.

Okay so let's go back to the documentation and see what else there is. There's a link here for by sub-breed, so let's click on that and see what it does. And let's scroll down a little here to see, if we want to get a sub-species here, this is interesting because it shows that we do hound/afgan or in the example that we did it would be hound/blood, but we did hound-blood.

So that's another interesting thing to note, there's actually two different ways to get the same resource.

***https://dog.ceo/api/breed/hound-blood/images***

***https://dog.ceo/api/breed/hound/blood/images***

So this highlights some of the ways that we can go about mapping out an API and figuring out how an API works and looking at it, but there's more exploration that we could do here, but hopefully this is enough to get you thinking about the possibilities. I mean we've just spent a couple of minutes here, we've barely scratched the surface of this API. And even in that we've already managed to build out a fairly rich map of the API. We found a number of different paths, we've found some interesting things that we might want to talk to a developer about.

That we might want to talk to designers about and that we might want to consider going forward.

**Overview of Authorization and Authentication**

One of the more difficult parts of working with an API is **security**.

Since APIs can be used programmatically, they can be attacked in many different ways. We need to be careful in designing them. And so there's a lot of thought and effort that goes into making sure that they're secure. This is good, but sometimes it can make it a little more difficult to work with APIs that need authentication.

There's a number of different protocols that can be used to secure an API,

The difference between **authorization** and **authentication**.

So in security you hear these two terms, and it's sometimes hard to distinguish exactly what they are. But it's good for us to understand the difference between them.

So in essence,

**authentication** - verifying **who you are**

**authorization** - verifying **what you can do.**

a real-life example - at a restaurant and want to order an alcoholic beverage.

They ask to see your ID.

Your ID allows them to both authenticate and authorize you.

**Authentication**

are you who you say you are?

I.e. matching up the picture and the name on an ID. are you Dave Westerveld?

Your ID has a picture and a name that allows them to authenticate that you are who you say you are.

**Authorization**.

Are you allowed to do what you're asking to do?

I.e. looking at your ID and seeing the date of birth and saying, are you old enough (authorized) to be served alcohol?

Now, much the same as with your ID,

in API security,

combine the authorization and the authentication together. you only need to **give one token** or **one ID badge** (if we think of it as a ID)

that can be used to validate both your authentication and your authorization in one step.

**Using OAuth tokens**

OAuth 2 has become a very common way to secure API calls. sign in with Google, Facebook, Twitter options on an application.

use OAuth 2 workflows to authorize the application.

A basic understanding of OAuth 2, 3 different pieces involved.

1st piece the **application** that you're using.

Once you tell the application that you want to use OAuth, it will send a request to the Auth server.

this request says to the server,

"Hey, I'm Dave, and here's my password."

The **server sends back a token** that can be used to authenticate and authorize you.

So **this token can then be used in API calls**

to the server itself or

to the web service.

Essentially what you're doing at that point is saying

"Hey, here's my token that says that I am who I say I am, and that allows me to do"

"the actions that it allows me to,"

"and I want to do this action.

Am I allowed to?"

the server will run checks to ensure that the given token is allowed to perform the action and

if it is, it will return the response for the given action.

if not, it will return an error code that tells you that you're Not allowed to perform that action.

Application

\/

request

\/

Auth Server

Application

/\

token

/\

Auth Server

Application

\/

token

\/

Web service or Server itself

Application

/\

response Or error code

/\

Web service or Server itself

So this is kind of abstract, high-level way of looking at how this works, but how does this data actually get sent around?

**What you do with the tokens once you have them.**

Postman and take a look at how we can do this in real life.

So Postman actually abstracts a lot of the details away for us, which makes it really easy to set up.

But we can kind of peek under the hood a little bit and get an idea of how this works under the hood.

an example call to the Github API.

a test repo I have, we're going to try to delete it.

**DELETE \/** https://api.github.com/repos/djw-test/test2

TYPE

**No Auth \/**

Click [Send]

we get a message back.

{

"message": "Must have admin rights to Repository.",

"documentation\_url": "https://developer.github.com/v3/repos/#delete-a-repository"

}

So we failed the authorization because we didn't present the correct token.

**How to make API calls with bearer tokens,**

switch TYPE OAuth 2.0

put in the Access Token

Click [Send]

We get a successful request, so it worked.

Click **Headers** tab

**KEY VALUE**

*Authorization* header *Bearer* followed by our API key.

*Bearer* lets the server know that this is an *OAuth 2 bearer token*  so it knows what to do with the token that we gave it.

Postman takes care of all of this in the background for us

what it's actually doing is adding this authorization header into the call that it's sending to the server,

so that it can authorize that we're allowed to do the action that we're requesting to do.

So once you have this token, it's pretty straightforward to use it.

- just send it in our header along with our API request

**Finding and Using Bearer Tokens**

Where you can get them from. Many sites will give you the option to generate tokens, usually these tokens can be scoped to authorize certain actions. So let's take a look at a real life example of this. I have a test github account open

settings

developer settings

<https://github.com/settings/tokens>

click personal access tokens.

click [Generate a new token]

give it Token description

**Select scopes**

Click scopes this token has

**what things is this user allowed to do?**

**what is this user authorized for?**

Once we've done that, we could generate this token and then copy it and use it in Postman or another API testing tool to allow us to do what we need to do. If your application has this capability, that's probably one of the simplest ways to get a token.

**Generate tokens right in Postman**

authorization TYPE set to **OAuth 2.0**

Get New Access Token window pops up

Token Name *Token Name*

Grant Type *Password Credential*

Access Token URL *OATH 2.0 Auth server url*

Username

Password

Client ID *username of client*

Client Secret *password of client*

Scope what I’m allowed to do

the layout and structure of an OAuth 2.0 application

The top part is specifying the authentication,

are you who you say you are,

The bottom part is specifying the authorization.

what are you allowed to do?

So once we have this information we can fill this form out, send it off and Postman would get a token back for us to use.

**API’s special API route to get tokens**

some APIs will provide a path that takes in parameters like your username and password and it will generate a token for you that you can then use; similar to what Postman is doing here but sometimes that can be set up directly on your API as well.

**Developer Consoles**

Another option to find tokens.

if your application passes tokens between different parts of it you may be able to access those tokens through the developer console.

chat to a developer or team member

who has an understanding of how the OAuth workflow, or how the authorization workflow in general, works in your application.

Most modern APIs depend on tokens in some way and so you should be able to find them if you just ask someone how to access them and what ways your company has, or your API has available for this.

**Additional API authentication options**

Authenticating yourself on an API options.

***Basic Auth***  Basic authentication method (only Username and Password)

Not all APIs offer this method, but if the API you're working with does provide it, it's pretty straightforward to use.

essentially all it's doing is asking us for a username and password.

In Postman:

TYPE *Basic Auth*

Username box

Password box

Click [Preview Request]

Click Headers

KEY

*Authorization* header

VALUE

Basic *basic token id*

another tool like *Python* or *cURL*

provide your username and password if your application was using this type of authentication.

Under TYPE dropdown, additional authorization and authentication options available

you're able to access most APIs with

*Basic Auth*

*OAuth2.0*

*Bearer Token*

working with an AWS application

*AWS Signature*

resources that are available

people on your team who know how to use this and have set the APIs up.

the Postman documentation to see how they set it up and use it.

So hopefully with the basic ones that we've covered, you have enough to get you started on testing in most API applications that you're going to work on.

**Hands-On-API Testing**

**Testing GET Calls**

I want to get into some specific strategies for API testing. So let's start with get requests. It seems like this would be pretty easy to test, right? You just send a request to the server and check that you get back what you asked for.

But right there, it's one of the challenges already,

How do you know what you should get back?

What if you get back something unexpected?

Did you send the request in the wrong format?

Or did the service or API do something wrong?

Well, there are a number of **heuristics** that we can use when exploring something through an API.

For example, let's take another look at the Star Wars API.

<https://swapi.co/api>/people/1

As we can see here we've got Luke Skywalker's results up and his home world is planets/1.

<https://swapi.co/api>/planets/1

So let's put that URL in and see what we get. Planets/1, and we'll request that. So, here in the result for this we get back a bunch of information and some of that information includes the residents of that planet. And you can see here that the first resident is people/1, and if you remember that was the URL that we had for Luke Skywalker.

So what we've done here is **check for consistency.**

We've checked that

people/1, Luke Skywalker has a relationship to planets/1

and

then we verified that planets/1 also has the relationship to people/1.

So, what we're doing is checking that if there's a relationship between two objects is that relationship defined consistently on both of the objects.

We could also do the same thing for the films, or species, et cetera, and make sure that relationships between each of these and Luke or any other person in the system is correctly defined.

Let's look at another example of **checking for consistency.** If we request /people, so we'll do that. We'll just request people.

***<https://swapi.co/api>/people***

And this gives us the full list of people and we can see here that we've got a **count of 87.** - So there's 87 people in this list.

Well, let's try something. Let's go people/88 and see what happens.

***<https://swapi.co/api>/people/88***

So we request 88 and that's interesting. We actually get a result.

So, we would expect there would only be 87 people in the system because that count told us there was 87. But it appears that there is 88 or there's somebody who's missing somewhere between one and 87. If we were testing this API we'd certainly want to dig into this a little more and see what's going on. But for now, we'll just move on. Now, this is an open API, so there's no authentication or hidden items in this API, so we can't actually see this in action, but

**one important consideration in API testing is to make sure that hidden resources aren't accidentally revealed.**

So, for example this API allows us to search for things. So we will be put in a search term. We'll say search and we'll search for hidden. We'll search for the term hidden.

***<https://swapi.co/api>/people/?search=hidden***

And we'll send that request. And we get back zero. So there's no results that have hidden in them, but this shows you something.

What if there was a result that was supposed to be hidden and you, if you went directly to the URL it would forbid you, or it would say this resource is not there, but **if you search for it it shows up in the search results.** So this is an example of another thing that we want to be careful of in APIs.

**schema**

look at the intended structure of the data in the Star Wars API - search for people; put in **schema** and we'll send that request,

***<https://swapi.co/api>/people/schema***

and we get back here the **schema** - basically the data layout.

this is what data is supposed to be there for anybody who is an object of people.

So these properties,

have the properties’ name, height, films, URL, birth year, eye color, created, and so on and so forth.

the schema defines what the objects need to look like,

if we were testing this we could go through and verify the various people in the system and make sure that they conform properly to this schema that we've been given.

As with most things in testing, you're only limited by your imagination. So try these get calls out, think like an explorer. Put on your explorer hat and you might be surprised at what you can find out about your API.

**Challenge "Find the secret."**

And let's navigate into the directory where we have our repo cloned.

***cd Documents/GitHub/api-testing-foundations***

So we'll navigate into here. And now we can start up the challenge. So we'll do that by calling node, and then we'll use the find the secret server, and just hit enter to start that.

***node find\_the\_secret\_server.js***

And you can see that this has started up a server for us that we can use, and the request that we want to make, so the API request, can be made using this base URL. So we'll copy that and we'll put that in Postman.

So that's our base URL.

http://localhost:3000/

But then we need the actual paths, we need to know the actual paths that we can use in the API. So for that we can see there's some documentation available on GitHub.

https://lnkd.in/gM6gMtP

<https://github.com/djwester/api-testing-foundations#secret-profile>

So the API testing foundations project, or repo on GitHub has documentation for these challenges that I have. So here's the documentation for the secret profile and it shows you what routes are available. So you can do GET on blog posts, or an individual blog post with a particular ID, and same for comments and profiles.

And then we also have these relationships, they're called, or parameters, so we can do this "question mark, underscore, embed, equals comments" and that will give children resources, so that will give the comments that belong to this blog post or we can do "expand" on comments and then we'll get blog posts that are referenced by that comment. So this documentation is available for you on GitHub to look through and look at. And the challenge itself is, if we go back to Postman, is to look at this, so we look at "profiles, slash 1”

http://localhost:3000/profiles/1

and we'll send that.

And we get an unauthorized. So this says "You're not allowed," "you're not authorized to see this profile." The challenge is find out a way to see that profile anyways. So for this challenge, don't look at the server code or anything like that. Just use the API with the commands that were given in the documentation and see if you can figure out a way to get at this hidden profile information. Have fun.

**Solution**

http://localhost:3000/blogposts/1?\_embed=comments

{

"id": 1,

"title": "My First Blog Post",

"author": "Janice Jones",

"comments": [

{

"id": 1,

"body": "some comment",

"blogpostId": 1

}

]

}

it has embedded the comment with the blog post so it's returned the information about the blog post and then it has also returned the information about any comments that referenced that blog post.

<http://localhost:3000/blogposts/1?_embed=profiles>

{

"id": 1,

"title": "My First Blog Post",

"author": "Janice Jones",

"profiles": [

{

"id": 1,

"name": "secret\_name",

"blogpostId": 1

}

]

}

We've gotten back some information about this profile. So we know the secret name of this profile. So even though we were forbidden access to it when we tried to directly go to that profile, we could still work around that restriction by asking it to **embed** something.

So when we go through this path, it's not doing the security check that we need it to.

So this example shows us why it's important to figure out different paths in the API. Sometimes things are implemented only assuming one path and so we want to **make sure that as testers, we're considering all the different ways that we might be able to get at the information.**

**Testing POST Calls**

We'll be looking at POST calls in this video, and for these, we are going to look at something that creates new material in our service, so creates a new object in our service.

This gives a whole new set of testing challenges. We are now giving API users the ability to modify something in our service, and so we need to make sure that they cannot do so in a harmful way.

There are two main ways that people can harm our app

through bad API calls, or through bad POST calls.

1 malicious users

2 users accidentally doing the wrong thing.

send data in the wrong format,

or

try to overwrite data that's already in the system?

There are many different ways that things could go wrong with POST calls,

so let's take a **look at one of them.**

Switch over to Postman here, and we'll be using a local application that I have that can demonstrate how we can vary inputs in a field.

So we'll look at a local host, three thousand, which is just a local application that I'm hosting, and for this application we're pretending that it's a form of some sort, so we can create blog posts. Let's take a look first at what we have, so let's do blogposts/1, and let's just take a look, do a GET on that, and see what we have.

***http://localhost:3000/blogposts/1***

So this is a blog post that's in the system.

*{*

*"id": 1,*

*"title": "My First Blog Post",*

*"author": "Janice Jones”*

*“profileId”: 1*

*}*

It's got a title, it has a body, there's a profile that's associated with it, and then it has its own internal ID that it uses.

So for this let's do a POST now, let's just copy this so we can use that in the body of our post, cause we need to provide parameters. So we'll do a POST here, and we'll change that to blogposts. For the body we'll use raw, and we'll switch it to json, because the data we have here is in json format, and we don't need to specify the ID. So let's do this... And let's also, let's try something, so let's take this title out.

Okay. And then let's just change this wording, we'll just say this is another blog post. And let's send that and see what happens. Alright so we can scroll down here, and we can see that it has created this for us, but it did not give us a title.

{

"body": "This is Another Blog Post",

"profileId": 1,

"id": 2

}

So this might be okay, but this might be something that we need to look at.

If title is a required field, if there's a business reason that we need that there, then we would need to make sure that the API doesn't let us create objects that are missing the required field.

So let's go ahead, we'll put title back in here. So we'll say title as a parameter that we want to give it, and say my title, and one of the things that you noticed in the POST is that it has an ID. So, let's see what happens when we send a call that has the ID specified, so here we did not specify the ID and you can see it created something with, automatically assigned an ID to it, but let's try what happens if we give it an ID, so ID, and let's say ID five and see what happens.

So we'll send that. And we got an error because I did not correctly put a comma after that so that this is json. That one needs a comma as well. So we'll send that again, and

{

"id":5,

"title":"My Title",

"body": "This is another Blog Post",

"profileId": 1

}

we can see that it has indeed created an object for us, that has an ID of 5 like we specified.

{

"id": 5,

"title": "My Title",

"body": "This is another Blog Post",

"profileId": 1

}

So now we know that we can specify the ID itself, so I wonder what happens if we play around with that value a bit. Let's try some interesting values and see what happens.

**double max -** the maximum value that can be stored in a double variable type,

1e to the 308, *double max*, and let's send that and see what happens.

{

"id":***1e308***,

"title":"My Title",

"body": "This is another Blog Post",

"profileId": 1

}

So we send that, and it seems to have worked okay.

{

"id": ***1e+308,***

"title": "My Title",

"body": "This is another Blog Post",

"profileId": 1

}

So I wonder, what if we increase it, what if we go higher than double max, let's change that to ***1e to the 309***, and send that, and see what happens.

{

"id":***1e309***,

"title":"My Title",

"body": "This is another Blog Post",

"profileId": 1

}

Now this is interesting. So it created it,

{

"id": null,

"title": "My Title",

"body": "This is another Blog Post",

"profileId": 1

}

it gave us a status here of ***201 Created***, but the ID is null. So that seems to be interesting. At this point we could start exploring some more issues that might come related to having big numbers like this but, I'll leave the example here for now, because there are so many other things to consider in POST testing.

For example, we just looked at some problems that might come from if you

put in a bad float input,

put weird characters in a string

or any number of other kinds of bad inputs?

security and performance

things like making sure that calculated values are recalculated correctly,

and so many other considerations that come into posting.

And so the point of what I'm trying to do here is to show you a glimpse of what can be done. As with all parts of testing, you need to explore and investigate and continue to learn when you're looking at POST testing.

**Testing PUT Calls**

Testing PUT calls is similar in a lot of ways to testing a POST call.

**a PUT call is meant to edit an existing object** rather than create a new one.

So for example things like what kinds of data you might try to send or how you would authenticate the calls might be the same between PUT and POST. But there are a few things we should consider that are specific to doing PUT calls. One of the most important things is looking at which fields we allow to change. Most databases have an ID field that is used as a reference point for building all queries and so we often don't want to be able to change that once its been created.

So if we take a look here at our sample application and try to do a PUT call on blog post one, so this is the blog post that has ID one, and we'll do a PUT call on it but we'll try to change the ID, so let's say I want to change the ID to five, let's see if it will let us do that.

{

"id":5,

"title":"My Title",

"body": "This is another Blog Post",

"profileId": 1

}

So we send that, it says it's okay so it let us do it in a way, but we can see that the ID came back still as one.

{

"id": 1,

"title": "My Title",

"body": "This is another Blog Post",

"profileId": 1

}

And if we do a GET on that we can verify that indeed the ID has not changed.

{

"id": 1,

"title": "My Title",

"body": "This is another Blog Post",

"profileId": 1

}

So in this case we **can't modify the ID of something in a PUT call.**

**how associated data might change when we modify an object.**

So let's look at an example to understand what I'm talking about here. So in our sample database we have a comment that is associated with a POST and so we can see that comment by calling a GET on the blog post, so we'll make a GET here and we'll add a parameter to embed the comments.

*{*

*"id":1,*

*"title":"The FIRST blog post",*

*"body": "This is my first blog Post",*

*"profileId": 1*

*}*

And let's send that. And we see that we get back the comments embedded with the blog post.

*{*

*"id": 1,*

*"title": "My Title",*

*"body": "This is another Blog Post",*

*"profileId": 1,*

*"comments": [*

*{*

*"id": 1,*

*"body": "some comment",*

*"blogpostId": 1*

*}*

*]*

*}*

But what happens if we do a PUT on this comment, so comment one, what happens if we do a PUT on there, does it correctly update. So let's take a look at that. Let's change this to comments/1

http://localhost:3000/comments/1

and we'll do a PUT on it,

let's just copy this so that we can use that as the body of our PUT. We'll put it in here and let's just change the body of our comment so we've changed something in it.

{

"id": 1,

"body": "some comment - changed“,

"blogpostId": 1

}

And now let's send this PUT command.

And so we can see that our comment has indeed been updated

{

"id": 1,

"body": "some comment - changed",

"blogpostId": 1

}

but let's go back and check and see on blog posts if we embed the comments does this request get updated as well.

So we send that and, oops I did a PUT there, let's see, let's do a GET. Alright so we send that GET

http://localhost:3000/blogposts/1?\_embed=comments

and we can see this actually changed my, when I did the PUT on the blog post, it changed that as well, but we can see that the comment has been changed.

{

"id": 1,

"title": "My Title",

"body": "This is another Blog Post",

"profileId": 1,

"comments": [

{

"id": 1,

"body": "some comment - changed",

"blogpostId": 1

}

]

}

So that's good, that's what we wanted to see was that updating something in one spot in the system caused it to be updated in other spots in the system as well.

**Security related -what happens when we try to do a PUT on to an object that we're not authenticated for:**

a simple example here that doesn't allow you, as the current user, to access a comment with ID 2.

So let's just try that out for a sec and see. So let's PUT and let's take this comment info here for our body again

{

"id": 2,

"body": "some comment - changed",

"blogpostId": 1,

"profileId": 1

}

and we'll say comments/2 and we will change the ID here to 2.

http://localhost:3000/comments/2

Okay and let's see if we can change the value of this comment.

So we'll send this PUT request. And if we look down here we get an unauthorized return. So the status here is **401 Unauthorized**, so we've been forbidden from accessing this object through a PUT request.

And this might seem reasonable right, because this is actually what happened, we're imagining that **this comment was posted by someone else, we're not allowed to modify it, so we're forbidden access to it.**

But the problem is there's actually a little minor security flaw here. By returning this unauthorized error code we've given the user some information about the system that they really don't need to know. We've told them there is a resource that exists here, and if you're a bad actor, if you're trying to hack in to the system in some way, that can be valuable information.

So in general it's usually better in this case to just return a **404**, which means that **the object isn't there**.

Try out new things, build your own series of testing ideas for PUT requests.

**Testing DELETE Calls**

Delete calls allow you to delete existing resources from the service.

many things that you want to carefully test when it comes to deleting things.

don't want to delete the wrong things.

authentication and authorization.

make sure that

1 we have all the right checks in place to prevent things from being deleted that should not be.

2 you can't delete another user's content,

only users with certain permission levels can delete things.

take the form of data types.

Only certain things in your service that are allowed to be deleted.

have some shared resources that you don't want to allow people to delete,

or

you might not allow deleting of an object until child objects under it have been removed.

an example in POSTMAN

GET

*http://localhost:3000/profiles/1?\_embed=blogposts*

TYPE

Inherit auth from parent

a simple service,

User 1 - *profiles/1* I've embedded the blog posts that this user has created.

*?\_embed=blogposts*

do a **GET** on this and send that command

, and we can see here that this user has two blog posts that they've created.

Let's also take a look at what comments this user might have created,

*http://localhost:3000/profiles/1?\_embed=comments*

so we'll send that, and we can see that this user has also got two comments that they've created as well.

So now let's change this to a delete. And let's delete that user's profile, so we're going to delete the profile of User One.

DELETE

*<http://localhost:3000/profiles/1>*

So we send that command and now we've deleted that profile,

but let's take a look at the blog posts, and let's see what's happened to those, so we'll look at all the blog posts here. We'll change this to GET

GET

*http://localhost:3000/blogposts*

we'll send that, and we can see there's no blog posts left, so the user's blog posts have been deleted.

**{}**

And let's do the same thing for comments; So if we do a GET on the comments,

GET

*http://localhost:3000/comments*

we can see that they too are empty.

**{}**

So when we deleted this user, the system went ahead and deleted all the blog posts and comments that belonged to that user as well.

**Deleting depends on the business case for this application**

Now that might be what we want it to do, but it might also not be what we want.

so perhaps when we delete a user, we would want to

keep the comments and blog posts in the system so that other people could still see what they have,

or perhaps this is a full data removal, and need to remove all the comments related to a particular user.

So **whatever the business case is** would depend on whether or not this is the right thing, but these are the kinds of things we want to check carefully when we check deleting.

We want to make sure that if we delete an object that has other objects that depend on it, that **it's doing the right thing.**

Is it removing them when it should?

Is it leaving them when it should?

Those are the kind of questions we want to ask when we're deleting things.

So at this point we've covered all the major REST API actions.

GET, POST, PUT, and DELETE

other options available - Not as often used in practice

understand how to think about testing these four that were covered, you're well on your way to being an API testing professional.

**the Find the 500 challenge**

find some 500 errors that I've hidden in the sample API. To successfully solve this challenge, you're going to need to

use POST, PUT, DELETE, and GET calls as you investigate this.

In

api\_testing\_foundations, start it by calling

***cd Documents/GitHub/api-testing-foundations***

***node find\_the\_500\_server.js***

hit **Enter** key to start the server up for you.

JSON Server is running. Requests can be accessed at http://localhost:3000/

And then you can take this URL and that's your base URL for the testing challenge.

***http://localhost:3000/***

So you can use that in Postman or whatever tool you're looking at to test the API. There's also some documentation available on Git Hub. So the API testing foundation's project on Git Hub, under my username here, has documentation for Find the 500 Errors.

***https://github.com/djwester/api-testing-foundations#find-the-500-errors***

And it explains a little bit about the challenge and then shows you what routes are available. So there's a bunch of git routes here. You can get blog posts, individual blog posts, same for comments, profiles, and then this thing called avatars. Which is just kind of a summary of, it's meant to simulate images that you might have associated with your profile. And then there's also post and put commands available as well. So you can post new blog posts, comments, profiles, avatars, and you can modify existing objects in the system with put as well.

There are delete commands. You shouldn't need to use them too much to solve this challenge to get to the 500 errors that I know about. There may be additional ones in there, but the delete commands will work as well if you're trying things out. And then we have these relationships. So there's embed which allows you to include children resources. So if you have a blog post that has comments associated with it, you can embed those comments in the blog posts. And the same for expand. It kind of works in reverse. So if you go to a child object, you can expand out the parents of that child object.

So here's an example of doing that with comments. You can expand the blog post that reference that comment.

This challenge will probably stretch you a bit and I would encourage you to let it do that. Struggle with it for a bit and see what you can figure out on your own. It really is one of the best ways to learn something as challenging as API testing is. And just one other note is that this service is designed to have bugs in it. So you might run into some issues while testing it.

So if you get to the point where you're kind of stuck and things just aren't making sense and it's a mess, you might want to reset everything to start over.

And the easiest way to do that is actually just to open up Git Hub desktop and see your changed files. So here's my json file. This is kind of the database that we're using for this service. And you can see that I've got it, you know I've modified something here for demonstration purposes. But if you see this file as very changed, you might just

right-click ***discard changes***.

And say yes I want to discard the changes. And that will get rid of any changes that you've made.

go back to Terminal where I have your service running,

hit control C

to kill the service and then you can start it up again.

And it will be reset back to the starting point.

So if you do get stuck on things and you think something weird is happening, you can just reset everything and start over and try it out again if you want to. And if you get stuck you can look at the solution video and get some hints, but I hope you have fun figuring this out on your own. Happy testing.

My Answer for 500 errors:

DELETE

http://localhost:3000/blogposts/1?\_embed=comments

GET

<http://localhost:3000/comments?_expand=blogpost>

DELETE

<http://localhost:3000/comments/1?_expand=blogpost>

Instructor Answers for 500 Internal Server Error:

Try to POST to an already existing id

POST

***http://localhost:3000/blogposts***

{

“id”: 1,

“title”: “The FIRST blog post”,

“body”: “This is my first blog post”,

“profileId”: 1

}

Error: Insert failed, duplicate id

GET

***http://localhost:3000/avatars/1***

{

"id": 1,

"description": "A Public avatar",

"url": "www.example.com/avatar.jpg",

"profileId": 1,

"commentId": 1,

"blogpostId": 1

}

PUT

***http://localhost:3000/avatars/1***

{

"id": 1,

"description": "A Public avatar",

"url": "www.example.com/avatar.jpg",

"profileId": 1,

"commentId": 1,

}

GET

***http://localhost:3000/avatars/1?\_expand=blogpost***

{

"id": 1,

"description": "A Public avatar",

"url": "www.example.com/avatar.jpg",

"profileId": 1,

"commentId": 1,

**"blogpostId": null**

}

Results in

*500 Internal Server Error*

*TypeError: Cannot read property ‘toString’ of null*

**Using mocks, stubs, and fakes in API testing**

terms like mocks, stubs or fakes in relation to unit testing

help you understand what they mean and also how you can effectively use them in your API testing.

definitions.

**test doubles** - use to stand in for some real part of the system.

i.e. mocks, stubs, fakes or spies

things that we create to take the place of something else in the system,

Why use them:

Isolate the server

if you want to see how your application works with POST calls, but you don't want to actually keep creating new objects in the database or the server.

Server Not available

you can't send calls over the network during testing,

that isn't available;

the network's not available;

the server itself isn't yet fully completed.

So in either of those cases,

want to mock out the server,

create a test double that takes the place of the server.

In Postman,

easily create a mock server.

New > Mock Server

Need an account

sign in to a Postman account

or

create a Postman account to be able to access this functionality.

Once you've done that, you can create a mock server. And here we define the method, so we'll leave this as GET, and let's make a path, let's call it test, and then we can tell what response code, so we'll leave this as 200.

And then a response body. So in this case, let's just have it respond back, got test. So we know that this server is working. We'll click Next here. Let's give it a name, just call it Test Server. And let's go ahead and create that server. So now, it's created that server, and it's made this mock URL. So this is a URL that it's created for us, that we can use as our base URL for our API calls. So I'll just click on this and Postman will load that for me, so we've got this URL which references that mock server for us.

https://aa7e9b12-1d02-4746-8c20-48bdfd562df2.mock.pstmn.io

And then, let's just do /test, because that is the call that we put in there. And let's send that. So we send that request, and we get back, got test. So every time we send a GET to this URL, to the server, the mock server, with the path test, we'll get back this response, **got test.** So in this case, you don't actually need to run a bunch of calculations on the server, or do anything like that. You don't need to create any objects on the server, you just send the request, you get back the response, and you're done.

And this is kind of the power of test doubles. They let you create something fake for part of the system, so that you can focus on testing other parts of the system.

There are some even lower tech ways to create test doubles as well.

created JSON data in text files,

then organized them into folders that fits the URL structure of the API.

just reference them in that way.

So the idea of test doubles really is a powerful tool that we can use. I showed you how to do it in Postman, that's just one tool that expresses it, but there are

many ways that you can create these.

through text files,

different tools and implement them in different ways.

But the point of it, is that you want to be able to set these things up in ways that help you.

be careful with these as well.

Just like any testing tool, things can go wrong with them.

By creating test doubles,

testing with something that's not actually the real thing,

need to be careful because that means we can miss things.

So we want to do additional checks to make sure that we are still testing the end to end system, we're still testing with the real data at some point.

But at the end of the day, these are powerful tools that can help you when testing an API.

don't be afraid of them, use them, be careful with them

to help you achieve your testing goals.

**API Automation**

APIs can often run more quickly and tend to lend themselves well to automated checking it's often beneficial to use them to run test automation.

think of test automation as merely an extension of exploratory testing that we do and certainly the exploratory testing gives us insight in what we need to automate.

But exploration and automation are usually trying to achieve different things and so we need to think about them differently.

Exploration

can lay the foundation for us for the automated test that we want.

is about discovery,

it's about finding new things,

it's about understanding what additional things we might need to investigate or dig into in the system.

Automation is more about repeating things.

Automation and exploration,

are not the same.

are similar, they're related.

Automation

relies on exploration in order to be implemented.

But they're not quite the same thing.

**need to repeat things in automation, what is it that we need to repeat?**

a few things that we should consider.

1 things that do not change.

If we automate these then we know that any changes in those,

any failures in our automation are things that we should investigate and look into more.

2 things that you want to know about if they do change.

I.e. a third-party API.

you don't have control over what happens in that API.

you don't have control over those changes but you would want to know if something changed because you might need to change your application in response to it.

So that would be another great candidate for things that you could automate.

**API Automation Approaches.**

**1 data driven testing.**

this kind of testing what we're looking at is basically checking every endpoint or combination.

what data is available in the system,

what things does it make sense for us to iterate over or repeat again and again.

potential challenges

i,e, how do we share data in this way?

If we have 50 different calls or 500 different calls through the system, **How do we share data?**

**What things are common between them?**

**How do we share that across tests?**

2 **workflow driven**

making a series of API calls in sequence,

that would mimic the kind of workflow that a user might do.

we're trying it out a little bit more like a customer would.

following a workflow that a customer might follow.

potential challenges

how do we pass data around?

So if were making an API call at one point,

we get some data, and then we might need to use that data in the next API call.

**Automating:**

**Approach it a little more like coding.**

think carefully about sharing data between different objects.

how data is effectively passed around the system.

when we explore, we can do that mentally, we can do that in our head.

With automation, we have to explicitly encode it.

And so we need to be careful that we're thinking about it a little bit more like coding. And

**Don't automate everything.**

There are a lot of things in the system that it just doesn't make sense to automate. It's because automation is more like coding than it is like exploring.

If we try to automate everything, we try to automate things that are very hard for computers to do because we've got these things encoded in our minds and we have to try to put them into the system. **Be careful that you're not trying to automate things that are the wrong things to automate.**

we need to think carefully about

what we're automating?

why we're automating it?

what the goal is of it?

So much of this course has been focused on figuring out what to test in an API and how to effectively find problems. Now these are very valuable skills to have but they're more exploratory focus skills. So if you want to automate, take some time, learn the skills that you need so that you can do a great job with the automation. I've seen far too many automation initiatives fail due to the people trying to automate things in the same way that they approach the exploration. So take some time, learn this stuff, and use the things that you've learned in this course to help you. Make sure that you're automating the right things.

**Performance Testing**

When you're interested in performance characteristics of a system, an API can be a helpful way to get you the answers that you're looking for. They can allow you to programmatically access certain functionality in the system, and yet the include enough of the system to give you a realistic feeling for how it could actually be used in real life. So APIs are a great way when you're looking at performance testing to help you out with that, but when we talk about performance testing, we could be talking actually about a number of different things. So API testing can actually help you with most of these.

There's many different ways that they can help.

load testing.

speed testing - how fast a page responds.

how many requests per minute a server can handle,

how many items a page can deal with.

stress testing

We could be looking at a number of different performance characteristics of our system.

2 examples for performance testing in an API

**1 using an API to help you out with load or stress testing on parts of your application.**

GitHub user 3 repositories that this user

https://github.com/djw-test?tab=repositories

Well, let's imagine we wanted to stress test this page. Maybe we want to see what happens when there's 100 repositories that this user has. Now we could going to through and we could click on this button 100 times, and type in a repository name and create 100 repositories manually like that,

**use an API to drive that - Postman**

I've got it set up here with the API call that we would need to do,

***https://api.github.com/user/repos***

we can POST to this API to create repos for that user.

in Postman click **Runner**, **collection runner** that we can set up.

Choose a collection - has the requests that we're interested in.

***GitHub Testing***

let's do 100 iterations

iterations ***100***

And then we could run this GitHub Testing, and it would run 100 iterations for us. So it would run that command 100 times for us and create those repos. So this is one example of how we can use an API to help us with performance testing.

**Speed testing.**

another kind of performance testing.

how fast does a particular command execute?

I.e. in POSTMAN, how long it takes when I click [New] button

until that response has come back and we're able to do something

with it in the UI.

could even set up a collection runner and run 100 tests that send that request over and over again.

here's the time 1.269 seconds to send that request.

if we had sent say 100 requests, compare all those times,

look at them and kind of get a feeling for how fast that command on average takes to execute.

**Artificially throttle your network; see those calls; what the performance is like on a slower network.**

And there's numerous other strategies that you could take with API testing to help you out with performance tests. By using strategies like this you can more easily automate things, and you can quickly get some useful information about where there might be areas that you need to look into in more detail.

As I said before, there are other ways that you can do performance testing on an API, so don't let these example limit you. Hopefully, though, they have given you some ideas for things that you could look at when you're investigating performance characteristics in an API.

**Security Testing**

We all know how important security is in the modern world of web development.

designing and testing APIs. Sometimes we can think that security is all about the

**Auth service**

making sure that people can't get your credentials or can't get around the authorization APIs.

vitally important; it's just the tip of the iceberg when it comes to security testing in an API.

**One of the most important rules of authentication and authorization**

**Use a standard auth protocol**

don't try to implement your own version of it.

Unless you're a big company like Google, that has the resources to take the time to do it right, you'll likely make mistakes, and this is an area you don't want to make mistakes.

don't try to reinvent the wheel in this area.

**Vulnerabilities:**

**1 standard security risks**

if you are using a good auth service, attack factors

sequel injection

cross site scripting.

the reality is that these vulnerabilities can be found in APIs as well.

APIs lend themselves well to automation;

when you're trying to hack a site, being able to automate, being able to try different things is helpful for the hacker and so, often, they will try to access these vulnerabilities through the API.

So you should definitely check for these standard security risks that you would anywhere, but in an API, there's also some additional things that you should really be careful of and think about.

**2 areas of responsibility - the API or client level**

vulnerabilities can happen in multiple places and it's not clear what part of the application ought to be responsible for that.

I.e. **cross site scripting**

can be blocked at the API

or it can be blocked at the client level.

And it's not always clear in every application which part of the application is responsible for it.

as testers to make sure that somebody is taking responsibility for it.

the API or the client needs to,

somebody is taking responsibility for blocking these kind of requests.

**3 check that all validation is correctly done.**

Since APIs can be run programmatically, doing

fuzzing attacks - just send random inputs to the API

or even

cycling through known lists of vulnerabilities can be done quite easily.

in API testing.

an area where we want to be careful of It's an area that's important to check.

If the API expects a resource to be an integer,

need to make sure that it only accepts integers, otherwise, we could end up with problems.

So security testing is a pretty big specialty in it's own right and we can't dig into all the details of it here, but with this quick overview, and with a little bit of Googling, I'm sure you can set up some stuff that helps you poke away at it a bit. So don't be afraid. Try a few things out like this. See how your API fairs.

**Microservices and IoT (Internet of Things) Testing**

A lot of work and thought has been going into using microservices as the architectural framework for many applications. We also hear a lot about the Internet of Things and smart devices of every sort are becoming more and more common. We have devices ranging from smart light bulbs to smart refrigerators and TVs and everything in between. This course has laid out a foundation for thinking about API testing and has given you the mental tools that you need for this. The truth is, if you've been following along with this, you're well prepared for testing microservices and the Internet of Things.

microservices are API-driven.

we use microservices architecture, we're breaking applications up into many small services.

these services talk to each other through APIs

one piece over that does something,

a piece over there that does something,

a piece over here, and each piece has its own little task.

The Internet of Things.

API-driven.

No UI

smart devices all need to be able to take in inputs

and sometimes they need to provide us with certain outputs.

in many cases, they don't even have user interfaces.

i.e. a smart light bulb,

how can we communicate with them? via APIs.

Future of Computing

cloud-based, distributed computing

leveraging the internet and the cloud more and more in the future, it seems to me the future will be one that has many things running in distributed ways.

API-driven

Although there are many different vendors and differences between their APIs, much of it is common stuff that we've actually learned in this course.

I.e. devicehub.net

<https://devicehub.net/developers/api-documentation>

a site that helps make it easier to manage smart devices.

And if we look here, this is the developer documentation for their API, what do we see? Well, we look down and we see stuff that looks pretty familiar to us.

We see JSON data here, we see that it has REST endpoints. So once again, we're looking at an API. The kind of API that we've seen over and over in this course. The kind of API that we've learned how to test. And so, as we look at this kind of approach, as we see a future that has microservices and a future that has many internet-connected smart devices, we can see a future that is going to rely heavily on APIs for these things to be able to talk to each other.

And this is a future that should be familiar to you right now. You're ready for this. You're ready for the future.

**Conclusion**

API testing - other tools

SoapUI

JMeter

scripting tools

Ruby

Python

Katrina The Tester

https://lnkd.in/gSuX2dm

<https://katrinatester.blogspot.com/2015/09/api-web-services-microservices-testing.html>

POSTMAN

<https://github.com/DannyDainton/All-Things-Postman>

<https://github.com/djwester>