An Introduction To

HTTP://

THE BASICS

HTTP://

INTRODUCTION

Web developers commonly work with the HTTP protocol and it is important to understand the whole request and response cycle. In this guide we will dive into looking at the different types of responses, methods, status codes and what we get back from a server when we send a request.

We will use Node.js, Express and Postman to test some of these concepts to understand the theory behind the requests. Even though we may not be familiar with Express or Node.js, this will be perfectly fine as we are not focusing on the specific server framework/language.

HTTP://

WHAT IS HTTP?

- Hyper Text Transfer Protocol
- Communication between web servers & clients
- HTTP Requests / Responses
- Loading pages, form submit, Ajax Calls

HTTP is basically responsible for communication between web servers and clients i.e. the protocol of the web. Every time we open up a browser and visit a webpage, submit a web form or click a button on a webpage that sends some kind of Ajax/fetch request etc. we are using HTTP and going through what is called the request and response cycle. We make a request and we get a response back that has something called headers and body (we will dive into this in more detail later).

HTTP://

HTTP STATELESS?

- Every request is completely **independent**
- Similar to transactions
- Programming, Local Storage, Cookies, Sessions are used to create enhanced user experiences

It is important to understand that HTTP is stateless, meaning that every request is completely independent. When we make one request visiting a webpage, navigate to another page after that or reload a page – it does not remember anything about/from the previous page. Each request is a single transaction. We can utilise tools for instance to hold login data and make a more enhanced user experience such as local storage, cookies, sessions etc. HTTP at its core is completely stateless.

HTTP://

WHAT IS HTTPS?

- Hyper Text Transfer Protocol Secure
- Data sent is encrypted
- SSL / TLS
- Install certificate on web host

HTTPS is basically where all data that is sent back and forth is encrypted by something called SSL (Secure Socket Layer) or TLS (Transport Layer Security). Anytime we have users sending sensitive information it should always be over a HTTPS, especially for debit/credit card data or any social security data, we would want a high level of security. We can do this by installing a SSL certificate on our web host and there are different level of security and certificates.

HTTP://

WHAT IS HTTPS?

GET	Retrieves data from the server
POST	Submits data to the server
PUT	Updates data to the server
DELETE	Deletes data from the server

When a request is made to a server it has some kind of method attached to it and there are many methods. The four methods above are the main methods we will work with, but it is important to note that there are a couple more methods.

Every time we visit a page we are making a get request to the server via HTTP to retrieve HTML, CSS, JavaScript, images, videos, JSON, XML etc.

POST is when we add resources to the server and this occurs typically when we submit a form. We are sending data to the server and typically that data will be stored in a database somewhere.

We can have forms that may get requests but it is less secure because the things we send in the form is actually going to be visible in the URL. So typically we do not want to use a GET request with a form unless it is some kind of search form where all we are doing is filtering data that is coming back from the server.

The PUT request is used for updating data that is already on the server while the DELETE request of course just deletes data from the server.

HTTP://

HTTP HEADER FIELDS

General:

Request URL

Request Method

Status Code

Remote Address

Referrer Policy

Response:

Server

Set-Cookie

Content-Type

Content-Length

Date

Request:

Cookies

Accept-xxx

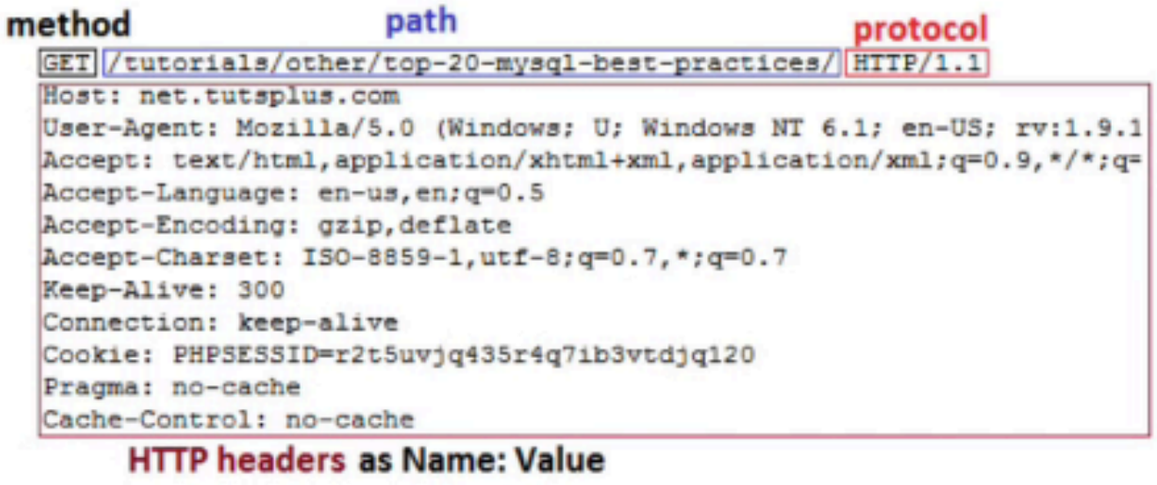
Content-Type

Content-Length

Authorisation

User-Agent

Referrer



With each request and response using HTTP, we have something called the header and something called the body.

The body typically with a response is going to be the HTML page that we are trying to load i.e. whatever is being sent from the server such as the JSON data. When we make a request, we can also send a request body for example when sending a form, the form fields are part of the request body.

When it comes to the header, we also have a request headers and a response headers and something called the general header. It is basically divided into three parts and there are different fields in each part. The header will look something like the above picture.
We would make a method to a path/URL and with a protocol (in the above HTTP 1.1) and then we would have various different header fields. A lot of the fields we do not really need to care about but it is good to know some of the more common fields do and what they are (especially the general part of it).

HTTP://

HTTP STATUS CODE

1xx : Informational	Request received / processing
2xx : Success	Successfully received, understood and accepted
3xx : Redirect	Further action must be taken / redirect
4xx : Client Error	Request does not have what it needs
5xx : Server Error	Server failed to fulfil an apparent valid request

HTTP status codes are really important to understand (at least the most common codes).

Status codes are through a range of 100 to 500. To the right are common Status Codes that we should try to memorise and understand.

200 – OK
201 – OK created
301 – Moved to new URL
304 – Not modified (Cached Version)
400 – Bad request
401 – Unauthorised
404 – Not found
500 – Internal server error

HTTP://

HTTP/2

- Major revision of HTTP
- Under the hood changes
- Respond with more data
- Reduce latency by enabling full request and response multiplexing
- Fast efficient & secure

We have been dealing with version 1.1 for a long time, but all the changes to version 2 are under the hood meaning we do not need to go change the way our applications work, everything is exactly the same but it is much faster, efficient and secure.

HTTP EXAMPLE

We can now jump into Express so that we can look at some examples on dealing with requests headers and body, sending status codes, etc.

We can navigate to any website and within the dev tools (Chrome shortcut on a MacOS is cmd + option + j) network tab, if we reload the page it will make all of the request it needs and get all the files from the server to render the page and this will be shown in this tab. If we click on a file we can view the response/preview tab which shows the body e.g. HTML file. In the Headers tab, we can actually see all the different header fields which are grouped in the three categories of General, Response Headers and Request Headers. We can narrow the documents using the filter such as viewing all the CSS files or JS files, or XHR (Ajax, Fetch API, Axios request) etc.
As a developer we should get familiar with the network tab.

HTTP://

POSTMAN

Postman is a client API Development Environment which is especially useful if we are building API's. We can get it at https://www.getpostman.com/

We can make any type of request to any URL and this will show everything we would get in the browser without rendering the page. This is a useful graphical interface to look at HTTP request and view the status codes, headers, body, cookies etc.
Again, this is a fantastic tool for testing APIs we are building.

HTTP://

EXPRESS CODE EXAMPLE

Server.js File:

```
const express = require('express');
const path = require('path');

const app = express( );

app.use(express.json( ));
app.use(express.urlencoded({ extended: false }) );

app.get('/', (req, res) => {
   res.send('Hello From Express');
});

app.listen(5000, ( ) => console.log(`Server started on 5000`));
```

The above code is bringing in Express, initialising it and setting it to listen on port 5000 and we have one endpoint. The endpoint meaning that if we make a GET request to forward slash (/) i.e. the index page, it is going to run the function that has access to the request and response object and with our response object we can call .send which will just send whatever we put in our parentheses to the client – in the above is a string of 'Hello From Express'.

With the Express server running within the terminal and using Postman, if we make a GET request to the URL of: http://localhost:5000/ we should receive Hello From Express in the Body area of Postman and we can also view the request header and other details within Postman.
Note: we can also run the URL in the browser normally and see the string printed to the screen and use the developer tools to see the response.

With Postman we can send all kinds of things such as request body, headers, etc. So we have a lot more freedom to interact with the request and response cycle.

In Express the res.send will detect the content type as best as it can, so a string will show up as a text/html while a JSON will show up as a content type of application/json and so on.

Note: there is a res.json( ) which we should use if we are sending a JSON object to the client.

We can also get our header values/fields if we want – below code example:

```
app.get('/', (req, res) => {
   res.send(req.header('host'));
});
```

We can take our request object and use the .header( ) method and within the parentheses pass in any header value we want to display, in the above example we want to display the host. If we save this change and go back to Postman and send the request again we would get a response with our host i.e. localhost:5000. If we changed host to user-agent, this will return us PostmanRuntime/7.6.0 because we are using postman as the client and will be different to what we would normally see which is the browser version of Mozilla, Chrome and Safari along with the client operating system if we used our browser. We can get all the rawHeaders using the command below:

```
   res.send(req.rawHeaders);
```

We can send data to the server in the request body. When we send a request we can attach data to the body and the way we can access that is using the req object along with the .body method.

```
app.post('/contact', (req, res) => {
    res.send(req.body);
});
```

The req.body will not work with JSON data unless we add the app.use(express.json( )) middleware into our code file and it will not work for form data unless we add app.use(express.urlencoded({ extended: false })) middleware. This is an express requirement. If we save the code and test the post request to the URL http://localhost:5000/contact in Postman, we would get back an empty object. This is because req.body is empty. If we want to send any data, we can click on the Body tab in postman (typically we would do this in our front end JavaScript whether React or Angular) we can click on x-www-form-urlencoded radio button which will automatically add a content type in our header (because we would also need to send this along to the client) and in

the body we could put a key:value pair (simulating a form from a website). If we now send the request in Postman we would now see the key:value pairs as a JSON object in our body because we sent that as the req.body.

The below code will send only the name.

```
app.post('/contact', (req, res) => {
    res.send(req.body.name);
});
```

If we want to grab the content-type of our post request, we would use the code below:

```
app.post('/contact', (req, res) => {
    res.send(req.header('Content-Type'));
});
```

This will return the content type of application/x-www-form-urlencoded

We can send our own statuses using conditionals to create different statuses based on the criteria. For example, if our route actually need to have a name ad if there is no

name we want to send back a 400 response which means it is a bad request. Below is the code demonstrating this:

```
app.post('/contact', (req, res) => {
   if (!req.body.name) {
         return res.status(400).send('Name is required');
   }
   // DATABASE STUFF
   res.status(201).send(`Thank you ${req.body.name}`);
});
```

We can use res.status to send back a status code number if the condition is true. We can use .json or .send to return a response back to the client describing the error. Below the if statement we can add the success response, and typically we would perform some database action, in the above we would send a 201 status of 'OK and created' and send the name back to the client.

Note: we do not need to add the return keyword to the last res.status in our code.

A lot of the times when working with full stack applications, we use tokens or JSON web tokens for authentication. We can either send it in the authorisation or in a value called x-auth-token. Below is a simple simulation of a login request.

```
app.post('/login', (req, res) => {
  if (!req.header('x-auth-token')) {
       return res.status(400).send('No Token');
  }
  if (req.header('x-auth-token') !== '123456' ) {
       return res.status(401).send('No Authorised');
  }
  res.send(`Logged in`);
});
```

If the x-auth-token is missing then this should send a status 400 and message, if there is a x-auth-token but does not match 123456, then this will send a 401 error message and if all passes then this will send back a logged in. We can use Postman to test this route.

Below is an example code to simulate a PUT request:

```
app.put('/post/:id', (req, res) => {
   // DATABASE STUFF
   res.json({
         id: req.params.id,
         title: req.body.title
   });
});
```

Usually we would have to identify the post we are updating and we would do that within the URL i.e. using the colon (:) followed by a name such as id and this acts as a placeholder which will take whatever value after post/ from the URL. Usually we would would perform some database actions and then send back a response. To get the id that was passed into the URL we can use the req object and call .params. followed by the placeholder name. The req.params will access the URL value while req.body will access the form/JSON data value we send in the body.

If we were to use Postman to simulate the request and use the PUT request with URL http://localhost:5000/post/100 and use a Header value of content-type: application/json to send a raw JSON and in the body we send { “title”: “Blog Post” } and send this request we would see a JSON response back of:
{ “id” : “100”, “title” : “Blog Post” }

Typically we would do some kind of database update using the post id. The DELETE request would basically be the same as the PUT request but using delete, perform a delete operation in the database and then returning a response with a message rather than the deleted object. Example below:

```
app.delete('/post/:id', (req, res) => {
   // DATABASE STUFF
   res.json({ msg: `Post ${req.params.id} deleted` });
});
```

We can use Postman to test the delete request without using any body or headers sent.

In Express, we can set a static folder and this pertains to if we just have a static HTML website where we have a bunch of HTML, CSS, JavaScript files, etc. In Express we can set a specific folder to be our static folder by using express.static( ) and passing in the location of the static folder, example below:

```
const express = require('express');
const path = require('path');

const app = express( );

app.use(express.json( ));
app.use(express.urlencoded({ extended: false }) );
app.use(express.static('public') );

app.listen(5000, ( ) => console.log(`Server started on 5000`));
```

Even though we do not have a route for the index.html file located in the public folder, it should still load all files from that public static folder when we visit localhost:5000.

CONCLUSION

This should hopefully provide a solid introduction into how HTTP works and some understanding to headers, body, content-type and statuses and how it works with Express or any other server side code when dealing with requests and responses.

Keep in mind when we deploy a Node.js and Express app, we are not going to serve it from localhost:5000, instead we are going to use something like NGINX/reverse proxy and there is a whole bunch of configuration that goes into deploying online, but as far as how HTTP works, it is exactly the same i.e we send and receive data in exactly the same way.

Below are some useful materials for further reading in order to build on our knowledge on the HTTP protocol.

HTTP:

https://httpwg.org/specs/

https://www.w3schools.com/whatis/whatis_http.asp

https://developer.mozilla.org/en-US/docs/Web/HTTP

SSL/TLS:

https://info.ssl.com/article.aspx?id=10241

https://www.globalsign.com/en/ssl-information-center/what-is-ssl/

https://www.globalsign.com/en/blog/ssl-vs-tls-difference/

https://nodejs.org/api/tls.html

Node/Express:

https://expressjs.com/

https://nodejs.org/en/

Reverse Proxy:

https://www.nginx.com/