Seneca College

Web Programming

# WEB322 – Week 4 – Class 1

WEB322 H

## Introduction to Express.js

### What is express.js?

Taken directly from the express.js website:

*Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobil applications.*

What does that mean exactly?

Express is a node module that takes a lot of the leg work out of creating a framework to build a website. It is extremely popula node.js community with tens of thousands of developers using it to build websites. It is a proven way to build flexible web app quickly and easily.

The express website has very good documentation and we will refer to it frequently in this week's notes for examples and documentation.

### API docs and code examples

The current major version of Express is 4.x. The documentation is available here – they do a great job keeping it up to date an provide excellent code examples to explain the concepts.

In the lecture for this week we will go over most of the main features of express in the API, which will enable you to setup a ba server with node and express that can accept requests and send responses from a client for various url 'routes'. A url route is of the url that comes after the host. For example, in the url http://www.seneca.ca/courses/ the host is www.seneca.ca, the rou Express terms) is /courses.

#### THE APPLICATION OBJECT

Taken directly from the Express API documentation ( updated to use ES6 style syntax) – here is a very simple example on how web server on localhost, listening on port 8080 on the main 'root' route.

```
const express = require("express");
const app = express();

app.get("/", (req, res) => {
  res.send("hello world");
});

app.listen(8080);
```

The app variable in the example above represents the express main application object. It contains several methods for routing requests, setting up middleware, and managing html views or view engines.

In the above example we setup a route on the index of the host to handle GET requests to "/". This means any requests to localhost:8080/ will be sent to this function. The standard form of a route handler in express is to call the verb method that ma the type of request on the app object and pass it 2 parameters: a string representing the route you want this route to match a and a callback function to invoke when the route is matched. In the above example we want to handle GET requests (typically from the browser to load the page initially). In other words 'get' the data for this route and call the provided function when tha is hit with a GET request.

All the other common HTTP verbs are supported as well. POST, PUT, DELETE, etc. You can also use app.all() to cover all HTTP ve one function.

Here are some of the commonly used application properties and methods you will typically make use of.

| property or method | Description |
|---|---|
| app.locals | This property allows you to attach some local variables to the application that you can access inside your fun for routing |
| app.all() | This method will catch all HTTP requests for all verbs. IE: GET, PUT, POST, DELETE, etc. |
| app.delete() | Routes a HTTP DELETE request for a matching path to a callback function specified. |
| app.engine() | The engine method is used to map a specific view extension to a template engine. For example if you want to all html view templates to a specific engine you would register it here. |
| app.get() | Routes a HTTP GET request for a matching path to a callback function specified. |
| app.listen() | The function is typically called once to setup a port and optionally a host to listen on for requests. This is the method on the express application that helps setup a https server and allows it to listen on a specified port. H default port is 80, HTTPS default port is 443. |
| app.post() | Routes a HTTP POST request for a matching path to a callback function specified. |
| app.put() | Routes a HTTP PUT request for a matching path to a callback function specified. |
| app.use() | The **use** method is used to add middleware to your application. Middleware consists of functions (typically pl before the routing methods) that automatically execute either when a specified path is matched or globally b every request. This is very useful when you want to do something with every request like add properties to th request object or check if a user is logged in. |

### THE REQUEST OBJECT

The req object represents the object that contains all the information and metadata for a request to the server. When you see examples of the request object in use it will always typically be referred to as 'req' (short for request object).

There are several ways to send data from the client to the server. Four of the more common ways are: send a POST request w in the body of the request, send data in cookies, send the data in the url, or send the data in the query string.
Here is a table of the common properties and methods used on the request object (req).

| property or method | Description |
|---|---|
| req.body | The req.body property contains the data submitted from a request. It requires that you use a body parsing middleware first which will attach the parameters to req.body. If you post data in your request then this is hov access that data. See the example in the docs on how it's used. |
| req.cookies | The second way of sending data is through a cookie. If you are using a cookie and you use cookie parsing middleware, then the cookie data will be available on the req.cookie object. |
| req.params | The third way of sending data is in the url. For example, if you want to have a url in the pattern /user/:name w the :name is dynamic; ie: for 'Bob' it's /user/bob and for 'Mary' it's /user/mary. You can have one route which ca accept an infinite number of requests with different user names and dynamically handle / access each differin username through req.params (ie: request.params.user). |
| req.query | The fourth way of sending data from the client to the server is in the query string. If you send data for a user u url like /user?user=bob then you can access that username with req.query.user. 'query' will be an object with k value pairs matching the key value pairs sent in the query string of the request. |
| req.get() | req.get() is useful for checking what the values are of any of the headers sent with the request. If you need to c the 'content-type' for example or the 'referer' or 'user-agent' headers. |

### THE RESPONSE OBJECT

The response object represents the object that contains all the information and metadata for a response from the server. Whe see examples of the response object in use it will always typically be referred to as 'res' (short for response object). The data yo back from the server can one of several different formats -the most common are HTML, JSON, CSS, JS and plain files (.pdf, .txt,

Here is a table of the common properties and methods used on the response object (res).

| property or method | Description |
|---|---|
| res.cookie() | This allows you to set a cookie on the response with a name = value key pair. You can set the value of the n a string or an object in JSON notation and it will be attached to the response header of the response. |
| res.download() | This allows you to send a file back as a response. The user's browser will display a download prompt and th response will set the Content-Type and Content-Disposition headers for you. Use this to send files back to a |
| res.end() | Use this if you want to end a response immediately and send nothing back. You can also chain .end() off of to send a status code and end the response, ie: res.status(400).end(); |
| res.get() | You can use res.get() to lookup any header that is outgoing on the response. This can be useful to check if c headers are set to certain values and respond accordingly. |
| res.json() | You can use res.json() to send back a JSON object with key value pairs of data. The values can be any valid J type, even Arrays and other objects. This method also automatically "stringifies" a JavaScript object, so no conversion is needed before an object is provided. Additionally, The json() method will automatically set the correct Content-Type header. |
| res.redirect() | Use this to perform a redirect to another page on your site, go back to the previous page, or redirect to anc domain all together |

| | |
|---|---|
| | domain all together. |
| res.send() | This is the bread and butter response method to send back a response to the client. You can send a String, object, an Array, or even a Buffer object back to the client. The send() method will automatically set the Cor Type header for you based on the type of data sent. |
| res.status() | You can use this to send back a specific status code and appropriate response to go with it. status() is chair before a send() so you can do operations such as: res.status(404).send("Page Not Found"); to send a 404 re code for a resource that is not found or invalid. |

### Routing, static files

Now that we've gone over the three core objects related to express let's show some examples of them in use by writing a few f and building out a basic web server with a few routes and some statically served files.

Routing is one of the core features of express. A detailed guide can be found here in the Express documentation. It is the com that handles setting up routes on your server that can be accessed by the client. A 'route', as mentioned before, is a path on yo that can have requests sent to it, and respond from it. Each route gets registered with express 'routers' and has callback functi get called when a request to that route is made.

Requests to routes can also be processed by multiple route definitions if your application requires it. More on that in the midc section below.

For now let's create a simple web server with express that can handle requests to "/", "/headers", and anything else will be con a 404 or page not found.

```
const express = require("express");
const app = express();

const HTTP_PORT = process.env.PORT || 8080;

// call this function after the http server starts listening for requests
function onHttpStart() {
  console.log("Express http server listening on: " + HTTP_PORT);
}

// setup a route on the 'root' of the url
// IE: http://localhost:8080/
app.get("/", (req, res) => {
  res.send("<h1>Welcome to my simple website</h1><p>Be sure to visit the <a href='/headers'>he
});

// now add a route for the /headers page
// IE: http://localhost:8080/headers
app.get("/headers", (req, res) => {
  const headers = req.headers;
  res.send(headers);
});

// This use() will not allow requests to go beyond it
// so we place it at the end of the file, after the other routes
```

```
// so we place it at the end of the file, after the other routes.
// This function will catch all other requests that don't match
// any other route handlers declared before it.
// This means we can use it as a sort of 'catch all' when no route match is found.
// We use this function to handle 404 requests to pages that are not found.
app.use((req, res) => {
  res.status(404).send("Page Not Found");
});

// listen on port 8080. The default port for http is 80, https is 443. We use 8080 here
// because sometimes port 80 is in use by other applications on the machine
app.listen(HTTP_PORT, onHttpStart);
```

Try out this code in a new node application and run it from the command line. You will need to have express installed using 'np install express' since we have not created a package.json file for this application.

Now try visiting **http://localhost:8080** in your browser to view your website. Click on the headers link and you should see a jso that represents the headers object on the request. Have a look at the key 'user-agent' and it's value. Try opening **http://localhost:8080/headers** in another browser and see how it differs.

Try visiting a route that is not the home route "/" or "/headers". You should see the 404 Page Not Found message to indicate th is not valid for this website.

Now that we have covered handling basic routes for a page and how to accept get requests that can respond with data; the ne question is: "how do we respond to requests for things like images, or css files?"

**SERVING STATIC FILES**

Static file serving refers to responding back to a client for a request to a static resource. A static resource is a resource (file) tha going to change and is required as part of the content for the site. Perfect examples are images and css files that need to be re from the site and loaded on the page. We can't make a route to handle loading each image for the website -that would be tedi impossible to maintain. Instead, it would be better to just specify a folder that contains all those static resources and allow exp handle sending those files back to the client when they are requested.

Check out the guide on Serving static files in Express on their website for the full information. We will do a simple example bel add an image to the home page and serve it statically from express

```
// setup the static folder that static resources can load from
// like images, css files, etc.
app.use(express.static("static"));
```

Add this code to the previous example just before the first app.get route handler. The "static" string refers to the name of the directory, relative to where the server is running from, that contains the static content. The "static" string does not become pa URL in this example, it is just specifying to express where to server static resources from.

Now we can put images or css files in the /static folder, and link to them from our website. In week 6 we will cover templating will come in handy when we start working with html templates.
Try adding an image to the static folder, and including it in an <img> element for the main route response.

## Middleware

Middleware in Express refers to functions that can execute in the 'middle' of a request/response cycle typically before a match function is executed.

Taken directly from the Express documentation:

> Middleware functions are functions that have access to the request object (req), the response object (res), and the next function in the application's request-response cycle. The next() function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.

There are 5 types of middleware:

| Type | Description |
|---|---|
| Application middleware | Application level middleware is bound to your entire application and can run when every request comes in or when it matches a specified route. Examples in the documentation. |
| Router middleware | Router level middleware works the same way as application middleware but is attached to a separate router i which means it can be applied only to a subset of all the routes in your application that are under that router. |
| Error handling middleware | Error handling middleware is defined with 4 parameters in the callback function. (err, req, res, next) is the sign You must specify all 4 parameters so that express can differentiate it from a regular middleware function in wl case next would be undefined. Error handling middleware gets called when a regular middleware function call next(err) instead of next(). Passing next an error object will invoke the error handlers instead of the next route handlers. |
| Built in middleware | Starting with Expres 4.x there is no more built in middleware except the static middleware for serving static fil mentioned how to use the express.static() middleware above in the serving static files section. |
| Third party middleware | With express 4.x, previously included middleware that did common things like handle cookies, parse the body form submission, or handle file uploads, have been moved to third party middleware packages. This way you them as you need them and reduce bloat in your system. If you're never going to handle file uploads, why hav middleware loaded? A great list of some of the most popular third party middleware is available here. |

## Error Handling

Error handling in Express is typically done through middleware. As mentioned above, when you write a middleware function w parameters, it will be interpreted by express as an error handler function. You can write multiple error handlers for handling c states of errors and use them in the same way you use middleware. By calling next(), it will invoke the next error handler in th and by sending a response it will terminate the chain. The error handling page on Express's website has great examples.
A common errorHandling procedure would be to log the error in the server console or database, return a 500 status code to t and send them to a 500 page or something else appropriate for your app.

```
function handleClientError(err, req, res, next) {
  // log the error to the DB with a utility method to log errors
  logError(err);

  // if the request was an xhr request respond with a 500 status and JSON message
  // otherwise respond with a string message
```

```
    if (req.xhr) {
      res.status(500).send({ message: 'There was an error processing your request' })
    } else {
      res.status(500).send('Something went wrong processing your request')
    }
  }
```

### Debugging in Visual Studio Code

Visual Studio Code has great debugging support for the node.js/express ecosystem. There is a tab on the left side of VS Code t the debugger panel.

Clicking the debug button will open the panel and you will see a bar along the top of the panel for launching the debugger and configuring it.

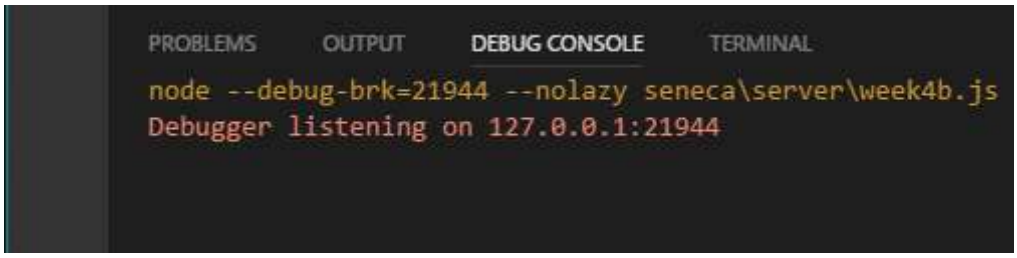

The launcher bar has a green play button to start debugging. However, to use the debugger you will first have to configure the launch.json file by clicking the gear icon and make the *program* property point to the main file you use to run your application

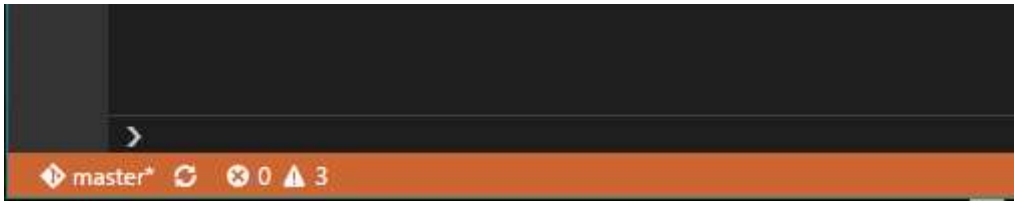
Here is an example of a configuration for the debugger when the main file to run the server is called week4b.js

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "Launch Program",
            "program": "${workspaceRoot}/week4b.js"
        }
    ]
}
```

In the example setup above the most important settings are the 'type' and 'program' properties. Using ${workspaceRoot} will r the root of the folder that you opened in Visual Studio code.

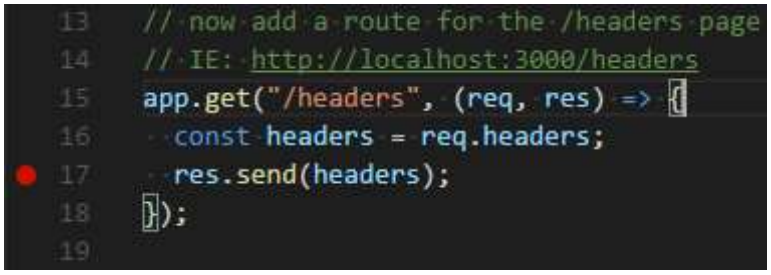
When the debugger is running you will see an orange bar in the bottom of VS Code and it will output info to the debug consol



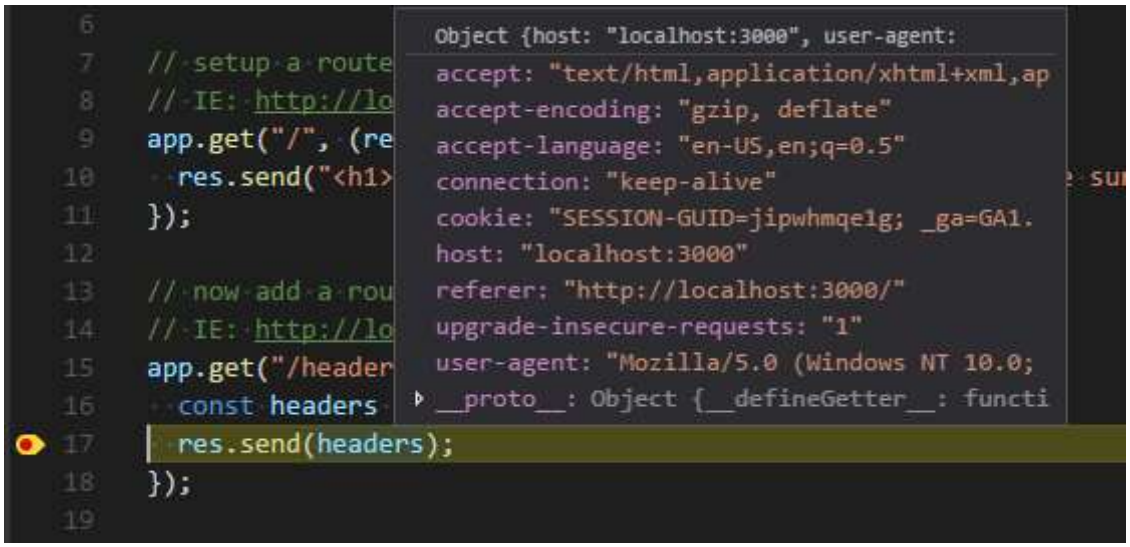



Once running, any breakpoints will be active and as the code executes it will stop on the breakpoints and allow you to inspect variables and state.

To place a breakpoint click on a line number in a file and a red circle will appear indicating the code execution will stop at this l when the debugger is running.




When the debugger is run and that line of code is hit, you will be able to inspect the state of your application while execution i paused.

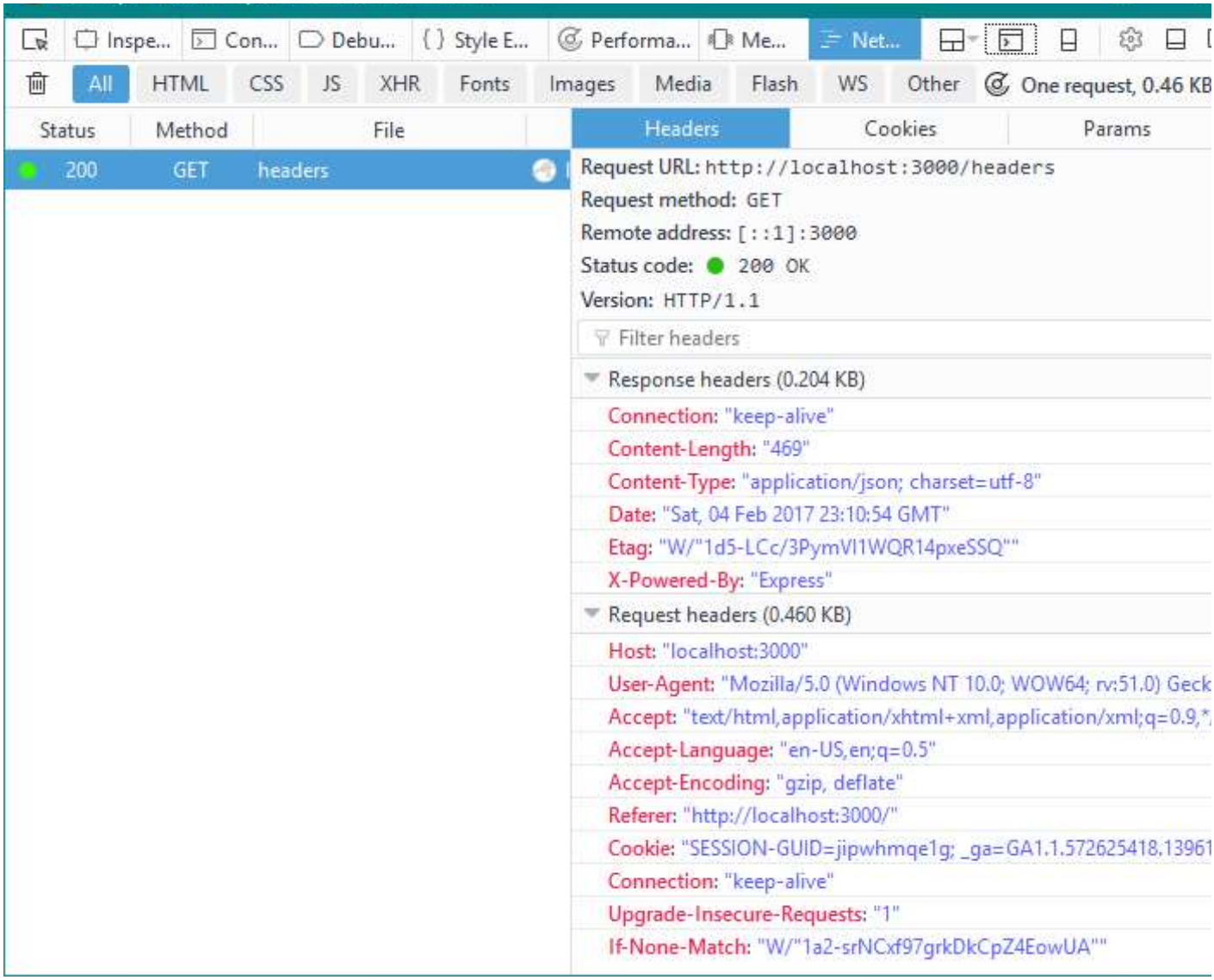



When you are finished inspecting, you can press play to continue with execution to the next breakpoint, or stop the program v stop button on the debugger control bar.

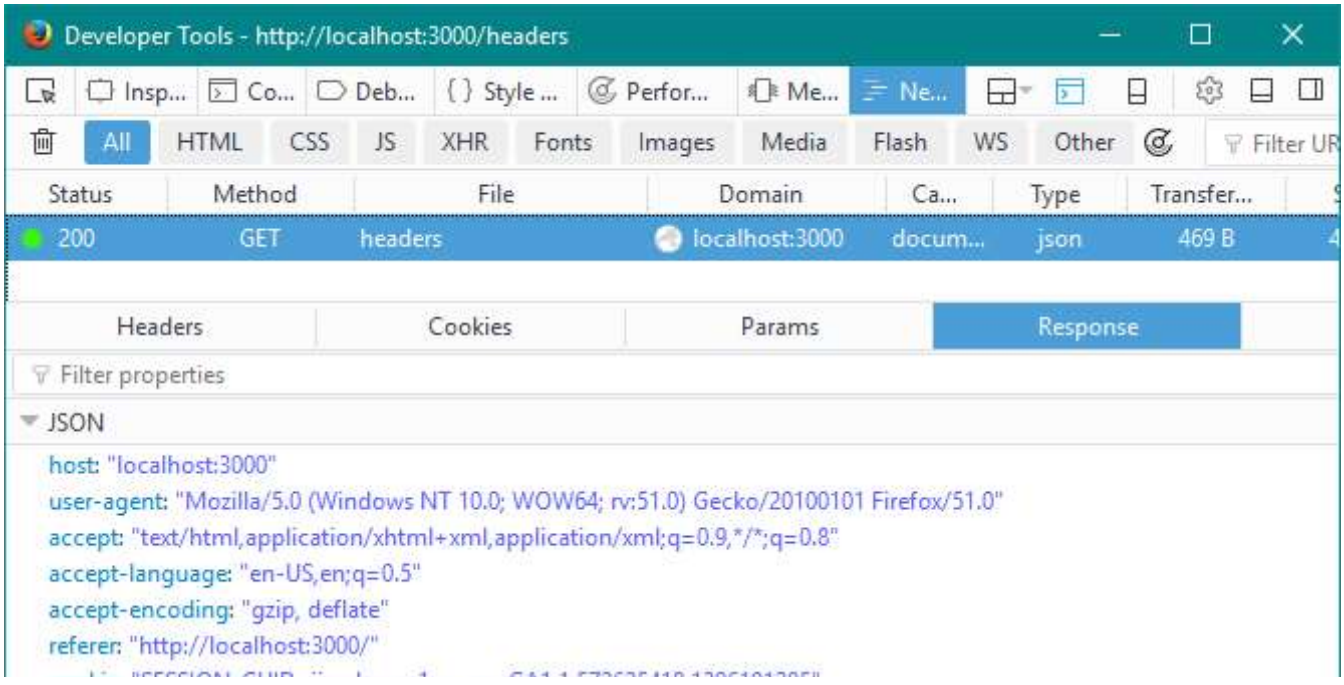
### The browser network tab

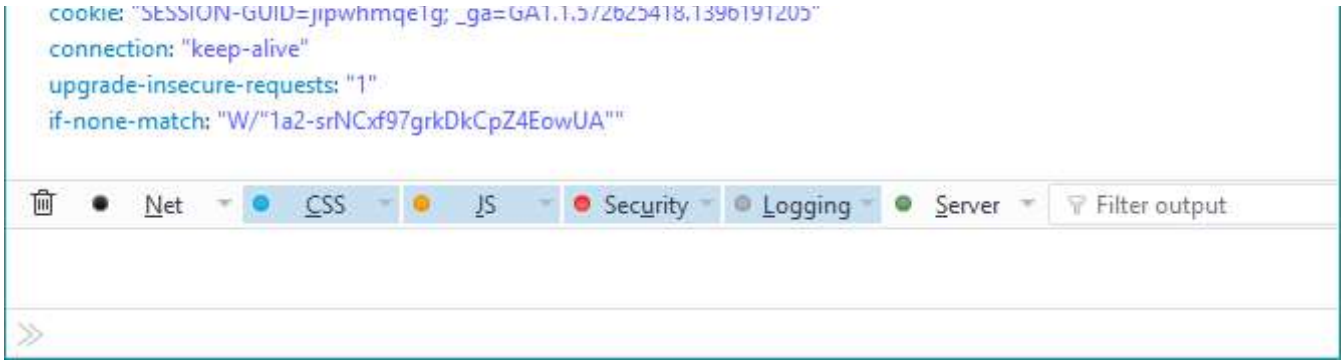
Chrome, Firefox, and IE all include developer tool and each of those tools has a network tab. The network tab of the dev tools you to see all requests going out from the browser to another server and the responses you get back from them.

In Firefox, you can inspect the headers from the headers tab:




And the response from the server in the response tab. The Google Chrome network tools are slightly different, however all in-developer toolbars essentially make the same information available.






The network tab in the dev tools is one of the core tools for debugging on the client side. You can check the returned status cc all the api requests being made, filter by just XHR requests, examine the cookies involved in the request and response, see rec and response headers, and examine the data returned from a request.

---