Web Servers

Prior Knowledge

- The anatomy of a url: (protocol, domain, path, queries).
- The definition of a parametric endpoint.
- Some basic status codes: (200: ok, 201: created, 404: not found, 500: internal server error).
- The different types of HTTP requests: (***GET***, ***POST***, ***PUT/PATCH*** and ***DELETE***).
- How to make an HTTP request with insomnia / browser.

Learning Objectives

- Understand how to use Node's built in `http` module to build a server.
- Understand why `http.createServer` requires a `responseHandler` callback.
- Understand how a server listens out for requests on a basic level.
- Understand the use of the request and response parameters in the `responseHandler` callback.
- Understand how to formulate and send a JSON formatted response.
- Understand how to assemble a body from packets of data sent over from the client side.

Clients and Servers

A client is an application making a request for external data - typically a browser.

A server is any program listening for requests on a computer port, that then processes these requests and sends responses accordingly.

For web development purposes, both request and response will usually follow the HTTP protocol.

Building a server from scratch

We can use the built-in node `http` library to build a server.

createServer()

The `.createServer` method is used to create the actual server itself, and takes a single argument which is a request/response handler - a callback function that is called each time that the server receives a request.

```
const server = http.createServer((request, response) => {
  console.log('request received');
});
```

listen()

In order for a server to be able to receive and handle requests, it must be **listening** for requests. To make an `http` server listen, we must invoke the server's `.listen` method with the port number that the server should be listening on.

```
const server = http.createServer((request, response) => {
  console.log(request);
});

server.listen(9090);
```

In order to get the server listening, the file must be run using `node`, e.g. `node server.js`.

The `listen` method also takes a second argument, which is a callback function that can be used to handle any errors when setting the server up to listen, e.g.:

```
server.listen(9090, (err) => {
  if (err) console.log(err);
  else console.log('Server listening on port: 9090');
});
```

Sending a response

Once the server is listening and receives a request from a client, the `createServer` request/response callback function will be invoked. And using the `response` parameter, a response can be sent back to the client by setting the appropriate headers (`response.setHeader()`), status code (`response.statusCode =`), and body (`response.write()` , which must be passed a string value):

```
const server = http.createServer((request, response) => {
  response.setHeader('Content-Type', 'application/json');
  response.statusCode = 200;
  response.write(JSON.stringify({ msg: 'hello world' }));
  response.end();
});

server.listen(9090, (err) => {
  if (err) console.log(err);
  else console.log('Server listening on port: 9090');
});
```

Once the response has been formed, `response.end()` needs to be invoked in order to send the response, otherwise the browser/insomnia will still be hanging: i.e. the request won't have ended!

Using the method and url

Within the callback function passed to `.createServer` we can add conditional logic to change what our server does based on different requests. For example changing the logic based on different request methods, url paths and queries.

```
const server = http.createServer((request, response) => {
  const { method, url } = request;
  if (url === '/' && method === 'GET') {
    response.setHeader('Content-Type', 'application/json');
    response.statusCode = 200;
    response.write(JSON.stringify({ msg: 'Server up and running' }));
    response.end();
  }
  if (url === '/users' && method === 'GET') {
    // logic to retrieve users data and send back to the client on the response body
  }
  if (url === '/users' && method === 'POST') {
```

v 0.1

```
      // logic to add the new user from the request body to the data and send the appropriate
    }
  });

  server.listen(9090, (err) => {
    if (err) console.log(err);
    else console.log('Server listening on port: 9090');
  });
```

Assembling the request body

When dealing with POST/PATCH/PUT requests that include information on the request body, the body of a request must be pieced together from the `request` object. This is similar to collecting the packets of data that form the body of a ***response*** when making requests using `https.request` .

```
 let body = '';
request.on('data', (packet) => {
  body += packet.toString();
});
request.on('end', () => {
  // logic to handle the body from the request and update the data
});
```

copy original markdown to clipboard

v 0.1