The HTTP Module

Let's move forward and learn the HTTP Module which helps you create Web Servers.

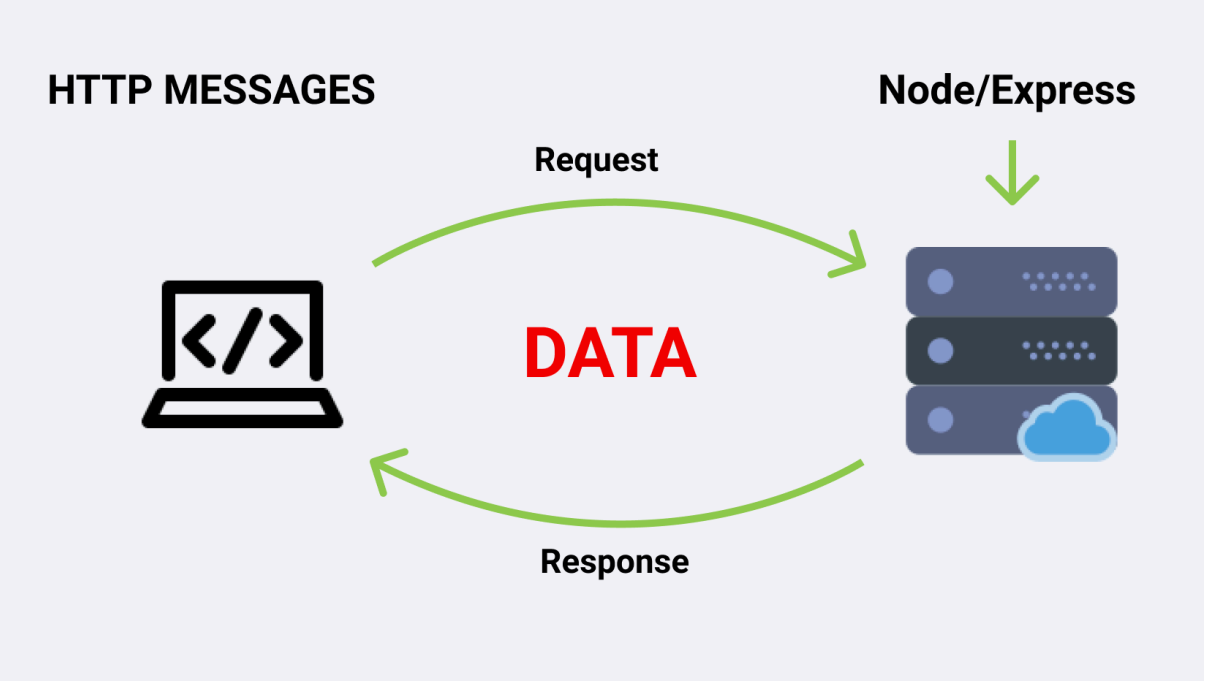
HTTP stands for Hypertext Transfer Protocol. It is used to transfer data over the internet which allows communication between clients and servers.

Suppose you want to watch some YouTube videos, you go to your web browser and type in: https://youtube.com, and then YouTube's home page gets displayed on your screen. This entire process happened because of communication between your machine (client) and YouTube's Server. The client, in this case, your machine requested for YouTube's home page and the server sent back the HTML, CSS and JS Files as the response.

The client sends a request to the server in the form of a URL with some additional information, such as headers and query parameters.

The server processes the request, performs necessary operations, and sends a response back to the client. The

response contains a status code, headers, and the response body with the requested data



Components Of Request-Response

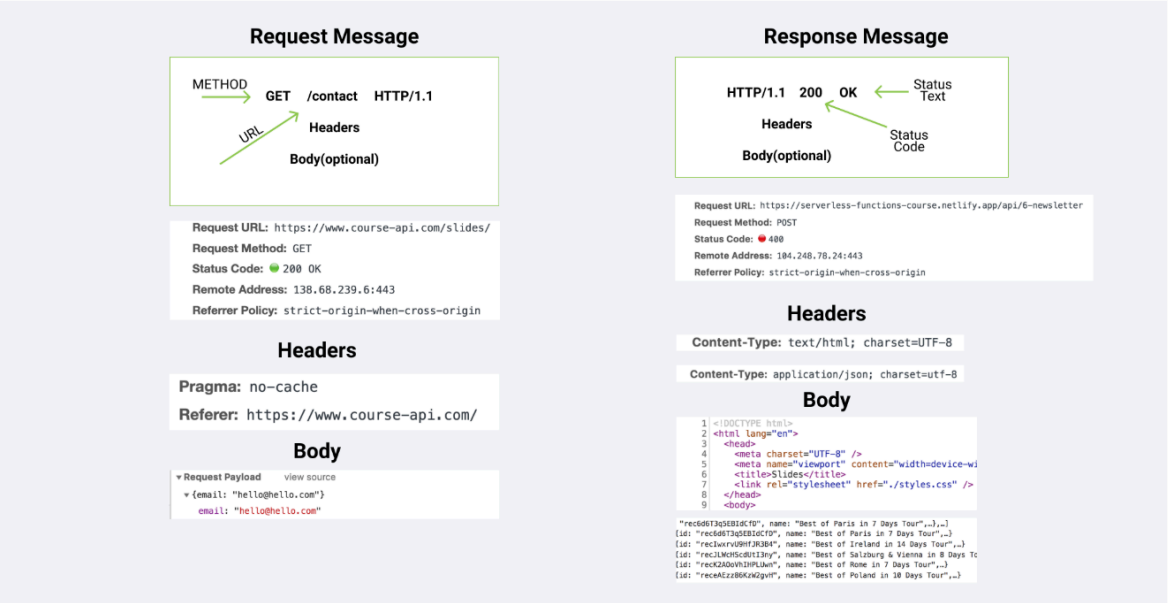
Both the Request (sent by client to the server) and the Response (sent by server to the client) comprises of 3 parts:

1. **The Status Line**: This is the first line of the request or response. It contains information about the message, such as the method used, URL, protocol version, and so on.
2. **The Header**: This is a collection of key-value pairs, separated by colon.

The headers include additional information about the message such as the content type, content length, caching information, and so on.

3. **The Body**: The Body contains the actual data being sent or received. In the case of requests, it might contain form data or query parameters. In the case of responses, it could be HTML, JSON, XML, or any other data format.

The 3 components of a Request and Response are described in much more detail in the below image:



What are HTTP Methods?

HTTP methods, also known as HTTP verbs, are actions that a Client can perform on a Server. The 4 HTTP Methods are:

- GET: Retrieves a resource from the server
- POST: Inserts a resource in the server
- PUT: Updates an existing resource in the server
- DELETE: Deletes a resource from the server

This might sound complicated, but let's try to understand these methods with the help of an example:

1. **GET:** Retrieves a resource from the server
   When you enter `http://www.google.com` in your web browser's address bar and press enter, your browser sends a HTTP GET request to the Google server asking for the HTML content of the Google homepage. That's then rendered and displayed by your browser.
2. **POST**: Inserts a resource in the server
   Imagine you're filling out a registration form to create an account on Google. When you submit the form, your browser sends a POST request to Google's server with

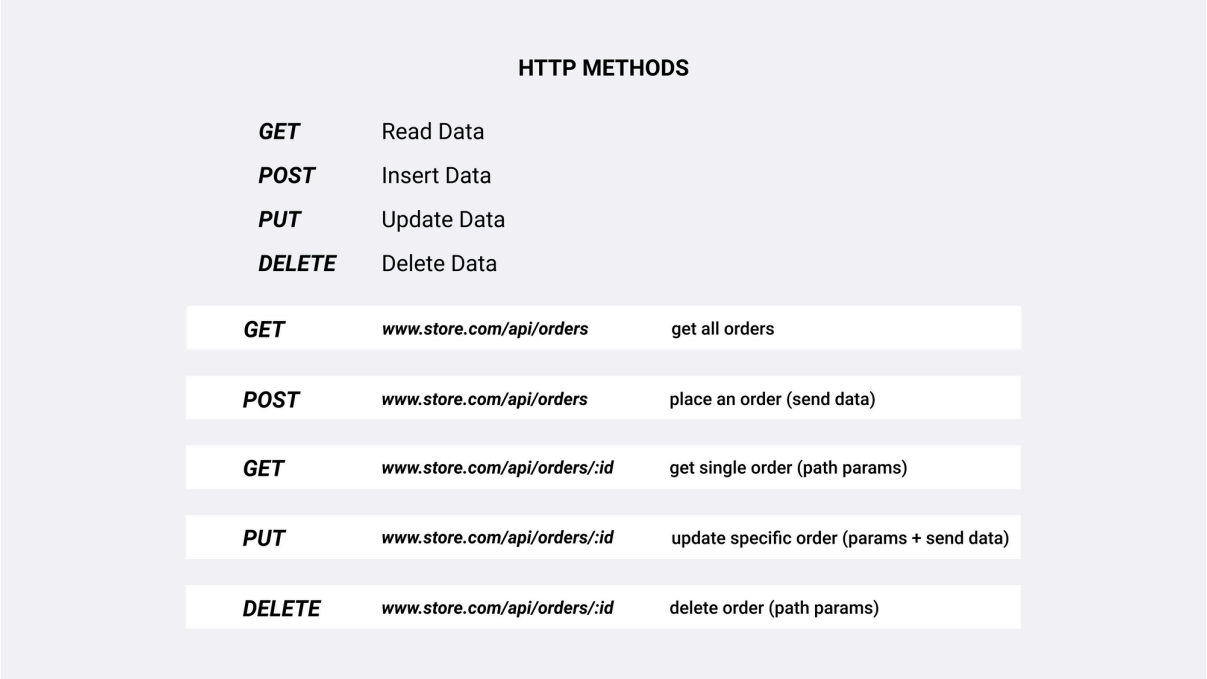
the data you typed in the form fields like: Username, Age, Birthdate, Address, Phone Number, Email, Gender and so on.

The server will then create a new user account in its database storing all the information sent to it using the POST Request. So a POST request is used to add/insert a resource in the server.

3. **PUT**: Updates an existing resource in the server
   Now imagine you want to update your Google account's password. You would send a PUT request to the server with the new password. The server would then update your user account in its database with the new password.
4. **DELETE**: Deletes a resource from the server
   Finally, imagine you want to delete your Google user account. You would send a DELETE request to the server indicating that you want your account to be deleted. The server would then delete your user account from its database.

Note that these are just examples. The actual requests and their purposes may vary.

To see more examples of HTTP Methods, you can refer this image:



What is a Status Code?

HTTP status codes are three-digit numbers that indicate the status of a HTTP request made to a server. They are server responses that provide information about the request's

outcome. Here are some of the most common HTTP status codes and what they represent:

Status Code	Meaning	Description
200	OK	The request has succeeded, and the server has returned the requested data.
201	Created	The request has been fulfilled, and a new resource has been created.
204	No Content	The server has successfully processed the request, but there is no data to return.
400	Bad Request	The server couldn't understand the request (e.g. missing required parameters).
401	Unauthorized	The server requires authentication before it can respond to the request.
403	Forbidden	The server understands the request, but the client does not have permission to access the requested resource.
404	Not Found	The server could not find the requested resource.
500	Internal Server Error	The server encountered an error while processing the request.
503	Service Unavailable	The server is currently unable to handle the request due to maintenance or overload.

Let's Create a Server

Finally let's move to the good part 🥳🔥 and learn how to create a Web Server using the `http` module:

Step 1: Import the `http` module like this:

```
const http = require('http');
```

Step 2: The `http` module provides you with `http.createServer()` function which helps you create a server. This function accepts a callback function with 2 parameters – `req` (which stores the incoming request object) and `res` which stands for the response to be sent by the server. This callback function gets executed every time someone hits the server.

This is how we can create a server using the `createServer()` function:

```
const http = require('http');

const server = http.createServer((req, res) => {
    res.end('Hello World');
})
```

Note: `res.send()` is a function attached on the `res` object using which we can send some data back to the client. Here

once we are done setting up the server, you will see a `Hello World` message in your web browser.

Step 3: Listening the server at some port using the `listen()` method.

The `listen()` function in Node.js `http` module is used to start a server that listens for incoming requests. It takes a port number as an argument and binds the server to that port number so that it can receive incoming requests on that port.

In the below code, we use the `listen()` function to start the server and bind it to port 5000. The second argument to the `listen()` function is a callback function that is executed when the server starts listening on the specified port. We are using this callback function just to display a success message in the console.

```
const http = require('http');

const server = http.createServer((req, res) => {
    res.end('Hello World');
})

server.listen(5000, () => {
    console.log('Server listening at port 5000');
})
```

You are likely to see a `Hello World` message when you visit this URL: http://localhost:5000/.

If you try to visit some other port like 5001 (http://localhost:5001/), which is not bound to your server, you won't see any response because your server is not listening on that port. You will likely receive an error message stating that the connection to the server could not be established.

At this point, we've made a server that renders a simple `Hello World` message every time someone tries to access it. This is quite good but there is a problem....

The problem is that for every route, the server sends the same message. For example, if I try to access the about page

or the contact page, still the server shows the same message:

- http://localhost:5000/ -> Hello World
- http://localhost:5000/about -> Hello World
- http://localhost:5000/contact -> Hello World

There is a simple way to fix this: there's a property called `url` in the `req` object which gives the URL of the request or in other words it tells you about the resource the client is trying to access.

Suppose if I type in: **http://localhost:5000/about** in my web browser's search bar, this means I am performing a GET Request on the server and I am trying to access the `/about` page. So In this case the value of `req.url` will be `/about`.

Similarly for the below requests, the value of `req.url` will be:

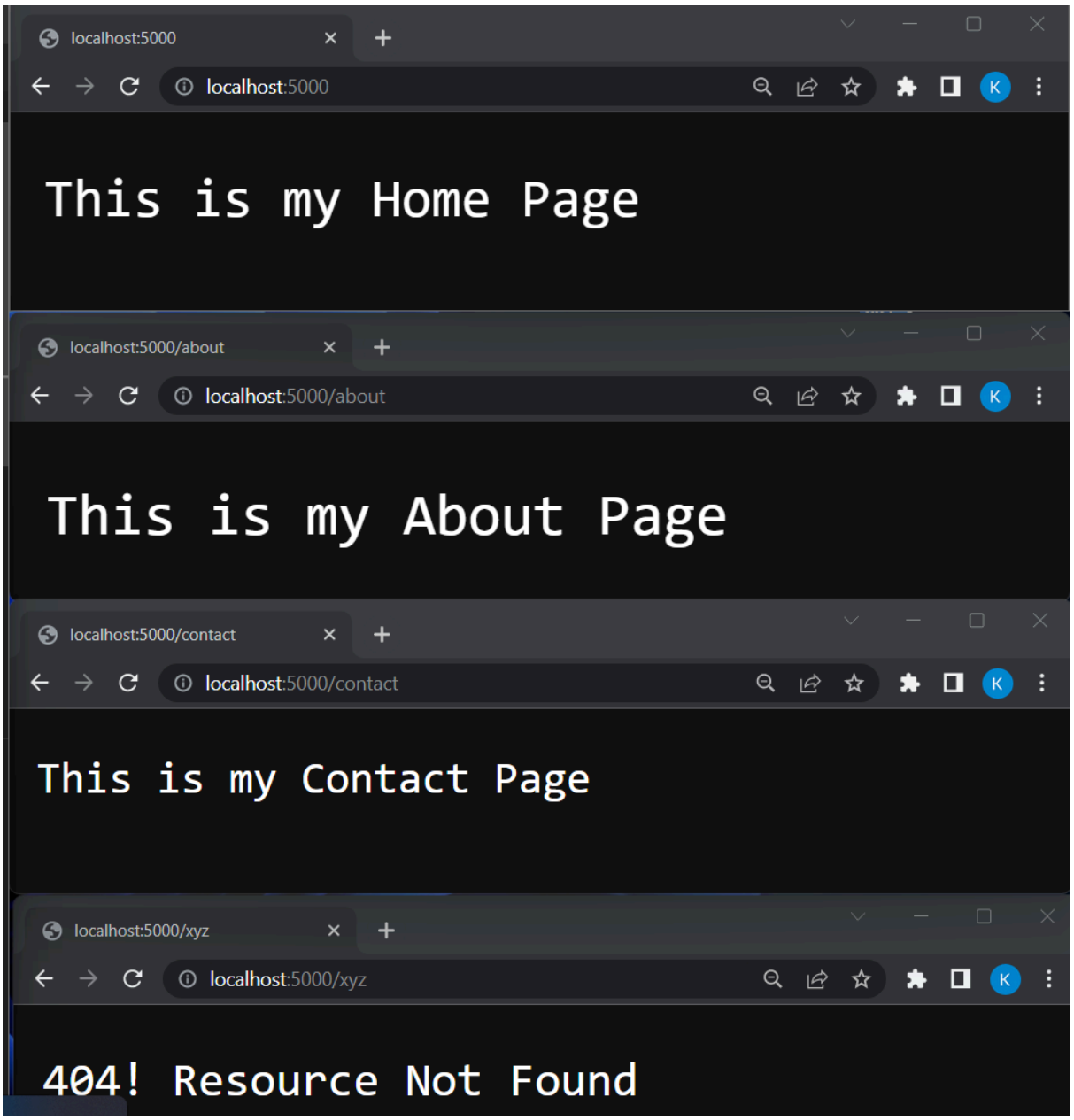
URL	`REQ.URL`
http://localhost:5000	`/`
http://localhost:5000/about	`/about`
http://localhost:5000/contact	`/contact`
http://localhost:5000/error	`/error`

We can use some conditionals `if...else` along with the `req.url` property to make our server respond to different requests differently. This is how we can achieve this:

```
const http = require('http');

const server = http.createServer((req, res) => {
    if(req.url === '/'){
        res.end('This is my Home Page');
    } else if(req.url === '/about'){
        res.end('This is my About Page');
    } else if(req.url === '/contact'){
        res.end('This is my Contact Page');
    } else {
        res.end('404, Resource Not Found');
    }
})

server.listen(5000, () => {
    console.log('Server listening at port 5000');
})
```



Now we have a perfect server that responds to different requests differently. We are sending back responses using a method called `res.end()`. But there is an even better way of sending back a response in which we can add on 2 more methods along with `res.end()`:

1. `res.writeHead()` – This method is used to send the response headers to the client. The status code and headers like `content-type` can be set using this method.
2. `res.write()` – This method is used to send the response body to the client.
3. `res.end()` – This method is used to end the response process.

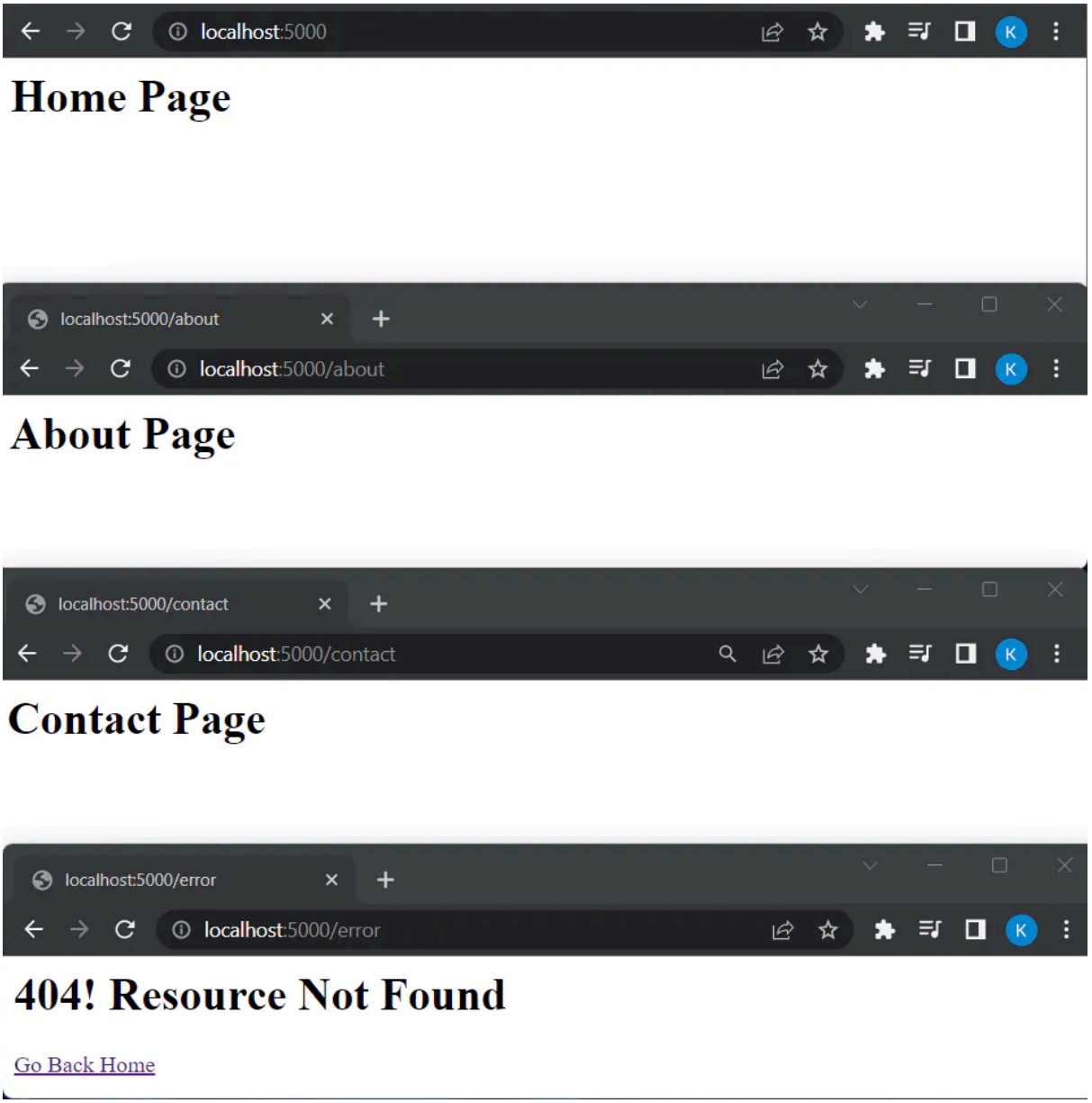
Below is the modified code where we added the `writeHead()` and `write()` methods along with `end()` method:

```
const http = require('http');

const server = http.createServer((req, res) => {
    if(req.url === '/'){
        res.writeHead(200, {'content-type': 'text/html'});
        res.write('<h1>Home Page</h1>');
        res.end();
    } else if(req.url === '/about'){
        res.writeHead(200, {'content-type': 'text/html'});
        res.write('<h1>About Page</h1>');
        res.end();
    } else if(req.url === '/contact'){
        res.writeHead(200, {'content-type': 'text/html'});
        res.write('<h1>Contact Page</h1>');
        res.end();
    } else {
        res.writeHead(404, {'content-type': 'text/html'});
        res.write('<h1>404, Resource Not Found <a href="/">Go Back Home</a></h1>');
        res.end();
    }
})

server.listen(5000, () => {
    console.log('Server listening at port 5000');
})
```

The below image shows what the server sends back as response when we visit multiple URL's:



Let's break down what's happening in the above code:

1. Here we are responding differently to different incoming requests by using the `req.url` property.
2. In every response, we are doing 3 things:
   – Setting the response header using the `res.writeHead()` method. Here we provide 2 parameters to `res.writeHead()`: the status code and an object which has the `content-type` property set to `text/html`
   – Setting the response body using the `res.write()` method. Note that instead of sending simple messages, we are actually sending some HTML code in this case,
   – And closing the response process using the `res.end()` method.
3. In case of resources like: `/`, `/about` and `/contact` the status code is set to `200` which means that the request to access a resource was successful. But if the client tries to access some other resource, they simply get back an error message and the status code is set to `404`.
4. Here the `'content-type': 'text/html'` is a way of telling the browser how it should interpret and display the response. In this case, we are telling the browser to interpret the response as some HTML code. There are different `content-type`'s for different types of responses:
   – To send back JSON data as a response, we need to

set the `content-type` to `application/json`

– To send back CSS as a response, the `content-type` should be `text/css`

– To send back JavaScript code as a response, the `content-type` should be `text/javascript`, and so on...

Setting the content type is very important as it determines how the web browser interprets the response. For example: if we just change the content-type from `text/html` to `text/plain`, this is how the response will be displayed in the web browser: