Anatomy of an HTTP Transaction

The purpose of this guide is to impart a solid understanding of the process of Node.js HTTP handling. We'll assume that you know, in a general sense, how HTTP requests work, regardless of language or programming environment. We'll also assume a bit of familiarity with Node.js `EventEmitters` and `Streams`. If you're not quite familiar with them, it's worth taking a quick read through the API docs for each of those.

Create the Server

Any node web server application will at some point have to create a web server object. This is done by using `createServer`.

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

const server = http.createServer((request, response) => {
  // magic happens here!
});
```

The function that's passed in to `createServer` is called once for every HTTP request that's made against that server, so it's called the request handler. In fact, the `Server` object returned by `createServer` is an `EventEmitter`, and what we have here is just shorthand for creating a `server` object and then adding the listener later.

```
const server = http.createServer();
server.on('request', (request, response) => {
  // the same kind of magic happens here!
});
```

When an HTTP request hits the server, node calls the request handler function with a few handy objects for dealing with the transaction, `request` and `response`. We'll get to those shortly.

In order to actually serve requests, the `listen` method needs to be called on the `server` object. In most cases, all you'll need to pass to `listen` is the port number you want the server to listen on. There are some other options too, so consult the API reference.

Method, URL and Headers

When handling a request, the first thing you'll probably want to do is look at the method and URL, so that appropriate actions can be taken. Node makes this relatively painless by putting handy properties onto the `request` object.

```
const { method, url } = request;
```

> **Note:** The `request` object is an instance of `IncomingMessage`.

The `method` here will always be a normal HTTP method/verb. The `url` is the full URL without the server, protocol or port. For a typical URL, this means everything after and including the third forward slash.

Headers are also not far away. They're in their own object on `request` called `headers`.

```
const { headers } = request;
const userAgent = headers['user-agent'];
```

It's important to note here that all headers are represented in lower-case only, regardless of how the client actually sent them. This simplifies the task of parsing headers for whatever purpose.

If some headers are repeated, then their values are overwritten or joined together as comma-separated strings, depending on the header. In some cases, this can be problematic, so `rawHeaders` is also available.

Request Body

When receiving a `POST` or `PUT` request, the request body might be important to your application. Getting at the body data is a little more involved than accessing request headers. The `request` object that's passed in to a handler implements the `ReadableStream` interface. This stream can be listened to or piped elsewhere just like any other stream. We can grab the data right out of the stream by listening to the stream's `'data'` and `'end'` events.

The chunk emitted in each `'data'` event is a `Buffer`. If you know it's going to be string data, the best thing to do is collect the data in an array, then at the `'end'`, concatenate and stringify it.

```
let body = [];
request.on('data', (chunk) => {
  body.push(chunk);
}).on('end', () => {
  body = Buffer.concat(body).toString();
  // at this point, `body` has the entire request body stored in it as a string
});
```

> **Note:** This may seem a tad tedious, and in many cases, it is. Luckily, there are modules like `concat-stream` and `body` on `npm` which can help hide away some of this logic. It's important to have a good understanding of what's going on before going down that road, and that's why you're here!

A Quick Thing About Errors

Since the `request` object is a `ReadableStream`, it's also an `EventEmitter` and behaves like one when an error happens.

An error in the `request` stream presents itself by emitting an `'error'` event on the stream. **If you don't have a listener for that event, the error will be *thrown*, which could crash your Node.js program.** You should therefore add an `'error'` listener on your request streams, even if you just log it and continue on your way. (Though it's probably best to send some kind of HTTP error response. More on that later.)

```
request.on('error', (err) => {
  // This prints the error message and stack trace to `stderr`.
  console.error(err.stack);
});
```

There are other ways of handling these errors such as other abstractions and tools, but always be aware that errors can and do happen, and you're going to have to deal with them.

What We've Got so Far

At this point, we've covered creating a server, and grabbing the method, URL, headers and body out of requests. When we put that all together, it might look something like this:

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

http.createServer((request, response) => {
  const { headers, method, url } = request;
  let body = [];
  request.on('error', (err) => {
    console.error(err);
  }).on('data', (chunk) => {
    body.push(chunk);
  }).on('end', () => {
    body = Buffer.concat(body).toString();
    // At this point, we have the headers, method, url and body, and can now
    // do whatever we need to in order to respond to this request.
  });
}).listen(8080); // Activates this server, listening on port 8080.
```

If we run this example, we'll be able to *receive* requests, but not *respond* to them. In fact, if you hit this example in a web browser, your request would time out, as nothing is being sent back to the client.

So far we haven't touched on the `response` object at all, which is an instance of `ServerResponse`, which is a `WritableStream`. It contains many useful methods for sending data back to the client. We'll cover that next.

HTTP Status Code

If you don't bother setting it, the HTTP status code on a response will always be 200. Of course, not every HTTP response warrants this, and at some point you'll definitely want to send a different status code. To do that, you can set the `statusCode` property.

```
response.statusCode = 404; // Tell the client that the resource wasn't found.
```

There are some other shortcuts to this, as we'll see soon.

Setting Response Headers

Headers are set through a convenient method called `setHeader`.

```
response.setHeader('Content-Type', 'application/json');
response.setHeader('X-Powered-By', 'bacon');
```

When setting the headers on a response, the case is insensitive on their names. If you set a header repeatedly, the last value you set is the value that gets sent.

Explicitly Sending Header Data

The methods of setting the headers and status code that we've already discussed assume that you're using "implicit headers". This means you're counting on node to send the headers for you at the correct time before you start sending body data.

If you want, you can *explicitly* write the headers to the response stream. To do this, there's a method called `writeHead`, which writes the status code and the headers to the stream.

```
response.writeHead(200, {
  'Content-Type': 'application/json',
  'X-Powered-By': 'bacon'
});
```

Once you've set the headers (either implicitly or explicitly), you're ready to start sending response data.

Sending Response Body

Since the `response` object is a `WritableStream`, writing a response body out to the client is just a matter of using the usual stream methods.

```
response.write('<html>');
response.write('<body>');
response.write('<h1>Hello, World!</h1>');
response.write('</body>');
response.write('</html>');
response.end();
```

The `end` function on streams can also take in some optional data to send as the last bit of data on the stream, so we can simplify the example above as follows.

```
response.end('<html><body><h1>Hello, World!</h1></body></html>');
```

> **Note:** It's important to set the status and headers *before* you start writing chunks of data to the body. This makes sense, since headers come before the body in HTTP responses.

Another Quick Thing About Errors

The `response` stream can also emit `'error'` events, and at some point you're going to have to deal with that as well. All of the advice for `request` stream errors still applies here.

Put It All Together

Now that we've learned about making HTTP responses, let's put it all together. Building on the earlier example, we're going to make a server that sends back all of the data that was sent to us by the user. We'll format that data as JSON using `JSON.stringify`.

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

http.createServer((request, response) => {
  const { headers, method, url } = request;
  let body = [];
  request.on('error', (err) => {
    console.error(err);
  }).on('data', (chunk) => {
    body.push(chunk);
  }).on('end', () => {
    body = Buffer.concat(body).toString();
    // BEGINNING OF NEW STUFF

    response.on('error', (err) => {
      console.error(err);
    });

    response.statusCode = 200;
    response.setHeader('Content-Type', 'application/json');
    // Note: the 2 lines above could be replaced with this next one:
    // response.writeHead(200, {'Content-Type': 'application/json'})

    const responseBody = { headers, method, url, body };

    response.write(JSON.stringify(responseBody));
    response.end();
    // Note: the 2 lines above could be replaced with this next one:
    // response.end(JSON.stringify(responseBody))
    // END OF NEW STUFF
  });
}).listen(8080);
```

Echo Server Example

Let's simplify the previous example to make a simple echo server, which just sends whatever data is received in the request right back in the response. All we need to do is grab the data from the request stream and write that data to the response stream, similar to what we did previously.

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

http.createServer((request, response) => {
  let body = [];
  request.on('data', (chunk) => {
    body.push(chunk);
  }).on('end', () => {
    body = Buffer.concat(body).toString();
    response.end(body);
  });
}).listen(8080);
```

Now let's tweak this. We want to only send an echo under the following conditions:

- The request method is POST.
- The URL is `/echo`.

In any other case, we want to simply respond with a 404.

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

http.createServer((request, response) => {
  if (request.method === 'POST' && request.url === '/echo') {
    let body = [];
    request.on('data', (chunk) => {
      body.push(chunk);
    }).on('end', () => {
      body = Buffer.concat(body).toString();
      response.end(body);
    });
  } else {
    response.statusCode = 404;
    response.end();
  }
}).listen(8080);
```

> **Note:** By checking the URL in this way, we're doing a form of "routing". Other forms of routing can be as simple as `switch` statements or as complex as whole frameworks like `express`. If you're looking for something that does routing and nothing else, try `router`.

Great! Now let's take a stab at simplifying this. Remember, the `request` object is a `ReadableStream` and the `response` object is a `WritableStream`. That means we can use `pipe` to direct data from one to the other. That's exactly what we want for an echo server!

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

http.createServer((request, response) => {
  if (request.method === 'POST' && request.url === '/echo') {
    request.pipe(response);
  } else {
    response.statusCode = 404;
    response.end();
  }
}).listen(8080);
```

Yay streams!

We're not quite done yet though. As mentioned multiple times in this guide, errors can and do happen, and we need to deal with them.

To handle errors on the request stream, we'll log the error to `stderr` and send a 400 status code to indicate a `Bad Request`. In a real-world application, though, we'd want to inspect the error to figure out what the correct status code and message would be. As usual with errors, you should consult the `Error` documentation.

On the response, we'll just log the error to `stderr`.

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

http.createServer((request, response) => {
  request.on('error', (err) => {
    console.error(err);
    response.statusCode = 400;
    response.end();
  });
```

```
    response.on('error', (err) => {
      console.error(err);
    });
    if (request.method === 'POST' && request.url === '/echo') {
      request.pipe(response);
    } else {
      response.statusCode = 404;
      response.end();
    }
  }).listen(8080);
```

We've now covered most of the basics of handling HTTP requests. At this point, you should be able to:

- Instantiate an HTTP server with a request handler function, and have it listen on a port.
- Get headers, URL, method and body data from `request` objects.
- Make routing decisions based on URL and/or other data in `request` objects.
- Send headers, HTTP status codes and body data via `response` objects.
- Pipe data from `request` objects and to `response` objects.
- Handle stream errors in both the `request` and `response` streams.

From these basics, Node.js HTTP servers for many typical use cases can be constructed. There are plenty of other things these APIs provide, so be sure to read through the API docs for `EventEmitters`, `Streams`, and `HTTP`.