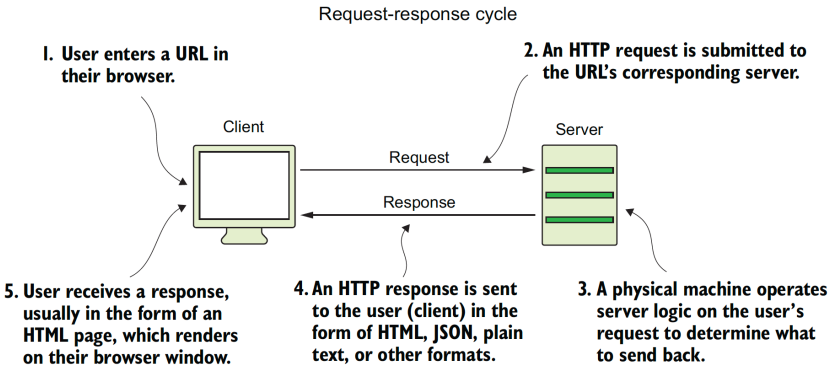
COSI 152A

Web Application Development

Web Servers

Web servers

- Web servers are the foundation of most Node.js web applications.
  - They allow you to load images and HTML web pages to users of your app.
- What happens when you visit https://www.google.com ?
  - HTTP request-response relationship:



Request Command

- Ways to send a request from a browser tab to the server:
  - Type url (GET)
  - Form (GET, POST.. etc)

- Request methods:
  - GET: only has header (parameter are sent in the header, NO body)
  - POST: has header and body (parameters are sent in the body)

HTTP GET vs. POST requests

- HTTP is a protocol for contacting the server and thereby gaining access to all the resources on the server.
- GET : asks a server for a page or data
  - request parameters, if any, are sent in the URL as a query string
  - URLs cannot contain special characters without encoding
  - Private data in a URL can be seen or modified by users
- POST : submits data to a web server
  - parameters are embedded in the HTTP request body, not the URL

Generating a Web Server

- Create a new directory called simple_server.
- Initialize the project.
- In the project folder create a new file called main.js and save it.
  - This file will serve as the core application file.
- Within the project directory in terminal, run `npm i http-status-codes -S`
  - Saves the http-status-codes package as an application dependency.
  - The package provide constants for use where HTTP status codes are needed in your application's responses.

Generating a Web Server

main.js file

Require the http and http-status-codes modules.

Create the server with request and response parameters.

```
const port = 3000,
  http = require("http"),
  httpStatus = require("http-status-codes"),
  app = http.createServer((request, response) => {
    console.log("Received an incoming request!");
    response.writeHead(httpStatus.OK, {
      "Content-Type": "text/html"
    });

    let responseMessage = "<h1>Hello, Universe!</h1>";
    response.write(responseMessage);
    response.end();
    console.log(`Sent a response : ${responseMessage}`);
  });

app.listen(port);
console.log(`The server has started and is listening on port number:
➥ ${port}`);
```

Write the response to the client.

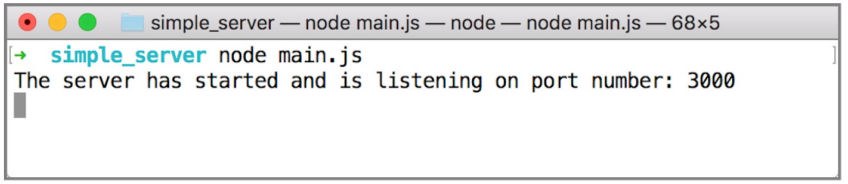
Tell the application server to listen on port 3000.

- Note: 200 is the HTTP status code for OK. For more HTTP status codes enter http.STATUS_CODES in the Node.js REPL shell.

Run Your Application!

- Run `node main` in your terminal
- Open a browser tab and visit localhost:3000

Congratulations, your first web server is responding!!!!!!



- You see a message indicating that the server has started.
- Press **Ctrl-C** in your terminal window to STOP your application

Reworking Your Server Code

- Create a new project, second_server
- Initialize your application and add a new main.js file.
- Modify your code to the following this:

```
const port = 3000,
  http = require("http"),
  httpStatus = require("http-status-codes"),
  app = http.createServer();

app.on("request", (req, res) => {
  res.writeHead(httpStatus.OK, {
    "Content-Type": "text/html"
  });

  let responseMessage = "<h1>This will show on the screen.</h1>";
  res.end(responseMessage);
});

app.listen(port);
console.log(`The server has started and is listening on port number:
➥ ${port}`);
```

Listen for requests.

Prepare a response.

Respond with HTML.

Analyzing Request Data

- The server can modify the content based on the type of request it gets.
  - If the user is visiting the contact page or submitting a form they filled out, for example, they'll want to see different content on the screen.
- The first step is determining which HTTP method and URL were in the headers of the request.

Analyzing Request Data

- Each request object has a url property. You can view which URL the client requested with req.url.
- Add the following code to the app.on("request") code block.

```
console.log(req.method);      ← Log the HTTP method used.
console.log(req.url);         ← Log the request URL.
console.log(req.headers);     ← Log request headers.
```

- Convert the objects to more-readable strings by using JSON.stringify within your own custom wrapper function, getJSONString.

```
const getJSONString = obj => {
  return JSON.stringify(obj, null, 2);   ← Convert JavaScript object to string.
};
```

Analyzing Request Data

- The request object can also listen for events, similarly to the server.
- The request listens for a specific data event.
  - req.on("data") is triggered when data is received for a specific request.
  - posted data comes into the http server via chunks.
- To collect all the posted data with a server, listen for each piece of data received and arrange the data.

Analyzing Request Data

- Within the app.on("request") code block, add the new request event handlers

Listen for requests.

Create an array to hold chunk contents.

Process it in another callback function.

Add received data to the body array.

Run code when data transmission ends.

Convert the body array to a String of text.

Log the request's contents to your console.

```
app.on("request", (req, res) => {
  var body = [];
  req.on("data", (bodyData) => {
    body.push(bodyData);
  });
  req.on("end", () => {
    body = Buffer.concat(body).toString();
    console.log(`Request Body Contents: ${body}`);
  });

  console.log(`Method: ${getJSONString(req.method)}`);
  console.log(`URL: ${getJSONString(req.url)}`);
  console.log(`Headers: ${getJSONString(req.headers)}`);


   res.writeHead(httpStatus.OK, {
     "Content-Type": "text/html"
   });

   let responseMessage = "<h1>This will show on the screen.</h1>";
   res.end(responseMessage);
});
app.listen(port);
console.log(`The server has started and is listening on port number:
➥ ${port}`);
```

POST Request with CURL

- CURL is a simple way of mimicking a browser's request to a server.
- Using the curl keyword, you can use different flags, such as –data, to send information to a server via a POST request.
- Because you haven't built a form yet, you can use a curl command. Follow these steps:
  - With your web server running in one terminal window, open a new terminal window.
  - In the new window. run the following command: curl --data "username=Jon&password=secret" http://localhost:3000

Adding Routes to a Web Application

- Routing is a way for your application to determine how to respond to a requesting client.
- An application should route a request to the home page differently from a request to submit login information.
- Web applications uses routes alongside its servers to ensure that users get to see what they specifically requested.
- Routes can be designed by matching the URL in the request object.

Adding Routes to a Web Application

- The next step is checking the client's request and basing the response body on that request's contents.
- This structure is otherwise known as application routing.
- Routes identify specific URL paths, which can be targeted in the application logic, and which allow you to specify the information to be sent to the client.

Adding Routes to a Web Application

- Duplicate the simple_server project folder with a new name: simple_routes.
- Then add a few routes to the main.js file:

```
const routeResponseMap = {
  "/info": "<h1>Info Page</h1>",
  "/contact": "<h1>Contact Us</h1>",
  "/about": "<h1>Learn More About Us.</h1>",
  "/hello": "<h1>Say hello by emailing us here</h1>",
  "/error": "<h1>Sorry the page you are looking for is not here.</h1>"
};

const port = 3000,
  http = require("http"),
  httpStatus = require("http-status-codes"),
  app = http.createServer((req, res) => {
    res.writeHead(200, {
      "Content-Type": "text/html"
    });

    if (routeResponseMap[req.url]) {
      res.end(routeResponseMap[req.url]);
    } else {

      res.end("<h1>Welcome!</h1>");
    }
  });

app.listen(port);
console.log(`The server has started and is listening on port number:
➥ ${port}`);
```

Define mapping of routes with responses.

Check whether a request route is defined in the map.

Respond with default HTML.

Thank You!