Lecture 6

Intro to Node.js & Express.js

Today’s Contents

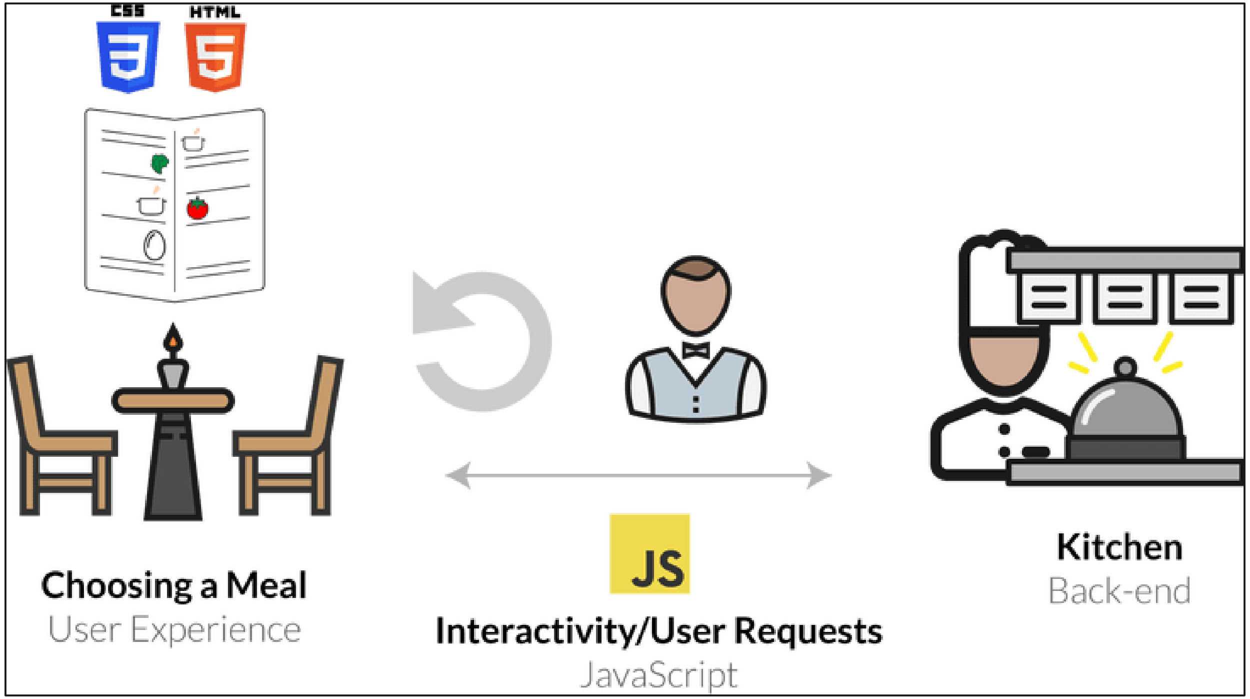
- Introduction to Node.js
- Parameters in Node
- Introducing Express.js
- HTTP Errors

Review: Web Service

Web service: software functionality that can be invoked through the Internet using common protocols (by contacting a program on a web server).

- Web services can be written in a variety of languages
- Many web services accept parameters and produce results
- Clients contact the server through the browser using XML over HTTP and/or AJAX Fetch code
- The service's output might be HTML but could be text, XML, JSON, or other content

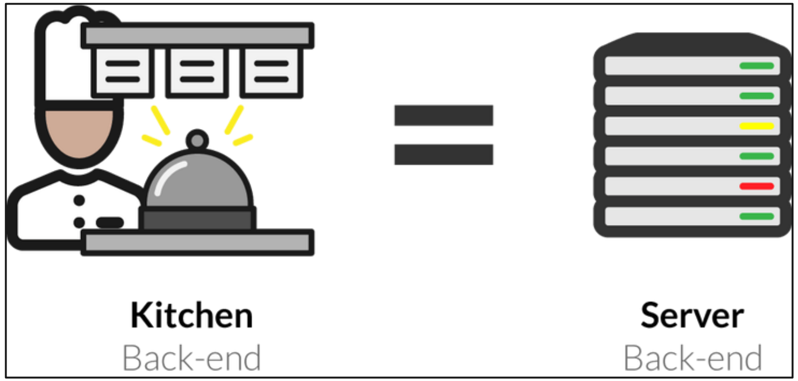
How Does a Web Service Respond to Requests?



Why Do We Need a Server to Handle Web Service Requests?

Servers are dedicated computers for processing data efficiently and delegating requests sent from many clients (often at once).

These tasks are not possible (or appropriate) in the client's browser.



Our (New) Server-Side Language: JS (Node)

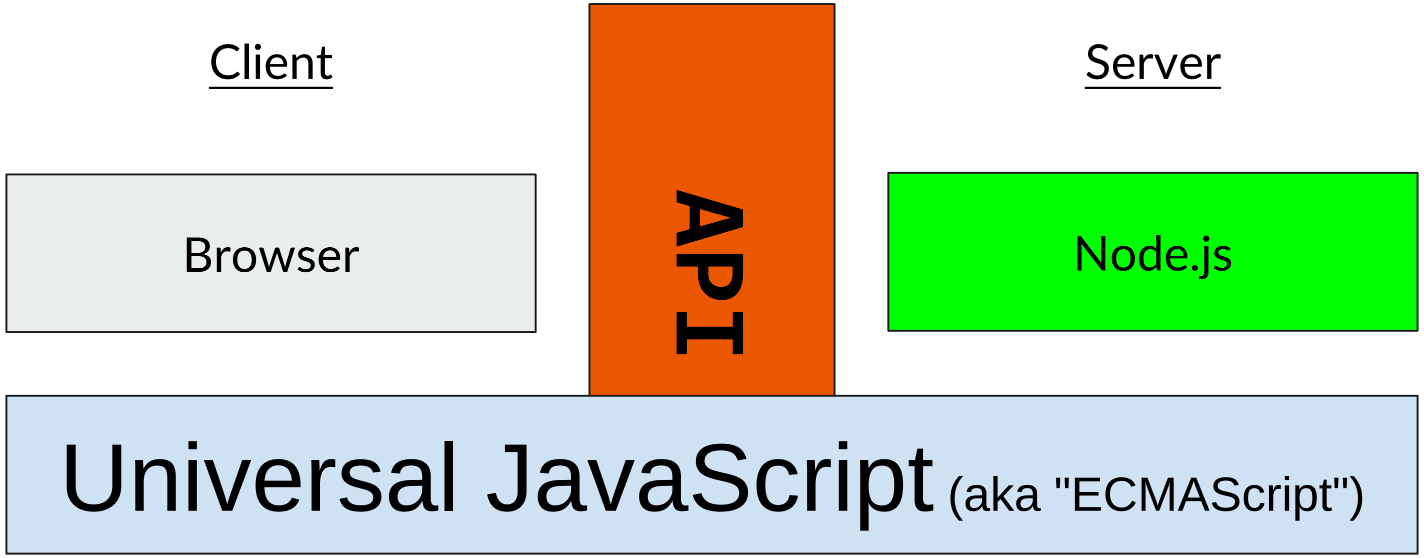
- Open-source with an active developer community
- Flourishing package ecosystem
- Designed for efficient, asynchronous server-side programming
- You can explore other server-side languages after this course!

Languages for Server-Side Programming

Server-side programs are written using programming languages/frameworks such as PHP, Java/JSP, Ruby on Rails, ASP.NET, Python, Perl, JS (Node.js), and many more...

Web servers contain software to run those programs and send back their output.

Client-side vs. Server-side JavaScript



What is Client-Side JS?

- So far, we have used JS on the browser (client) to add interactivity to our web pages
- "Under the hood", your browser requests the JS (and other files) from a URL resource, loads the text file of the JS, and interprets it realtime in order to define how the web page behaves.
- In Chrome, it does this using the V8 JavaScript engine, which is an open-source JS interpreter made by Google. Other browsers have different JS engines (e.g. Firefox uses SpiderMonkey).
- Besides the standard JS language features, you also have access to the DOM when running JS on the browser - this includes the `window` and `document` objects.

Node.js: Server-side JS

- Node.js uses the same open-source V8 JavaScript engine as Chrome
- Node.js is a runtime environment for running JS programs using the same core language features, but outside of the browser.
- **When using Node, you do not have access to the browser objects/functions (e.g. document, window, addEventListener, DOM nodes).**
- Instead, you have access to functionality for managing HTTP requests, file i/o, and database interaction.
- This functionality is key to building REST APIs!

Client-side vs. Server-side JavaScript

Client-side

- Adheres (mostly) to the ECMAScript standards
- Parsed and run by a browser and therefore has access to `document`, `window`, and more
- Calls APIs using `fetch`, which sends HTTP requests to a server
- Runs only while the web page is open
- Handles "events" like users clicking on a button

Server-side

- Adheres (mostly) to the ECMAScript standards
- Parsed and run by Node.js and therefore has access to File I/O, databases, and more
- Handles requests from clients, sending HTTP responses back
- Runs as long as Node is running and listening
- Handles HTTP requests (GET/POST etc)

JavaScript Template Literals

- Also known as *template strings* and *back-tics syntax*
- Template literals use back-ticks (` `) rather than quotes ("") to define strings
- Example:

```
let s = `We call him the "Black Bear"
         of the forest.`;
```

- Quotes can be used inside this string
- Multi-line strings are possible with template literals

JavaScript Template Literals

- Template literals provide an easy way to embed *variables* and *expressions* into a string
- With this syntax: `${<variable/expression>}`

```
let firstName = "John";
let lastName = "Doe";
let str = `Welcome ${firstName}, ${lastName}!`;
```

```
let price = 10;
let VAT = 0.25;
let total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;
```

Nesting Template Literals

- You can next template literals inside a template literal.
- See the following example:

```
const classes = `header ${
    isLargeScreen() ? "" : `icon-${
        item.isCollapsed ? "expander" : "collapser"
    }`
}`;
```

HTML Templates

- You can create long HTML strings in your JavaScript code and embed variables and expressions in them.
- This often proves useful when manipulating HTML with JavaScript.

```
const html = `<div id="user-panel">
    <div class="mini-avatar">
        <img src="${user.avatarURL}" alt="${user.name}" />
    </div>
    <div class="user-name">
        <a href="/profile">${user.name}</a>
    </div>
</div>`;
document.querySelector("#user-area").innerHTML = html;
```

Getting started with Node.js

When you have Node installed, you can run it immediately in the command line.

1. Start an interactive REPL with `node` (no arguments). This REPL is much like the Chrome browser's JS console tab.
   a. ("REPL" stands for "Read Evaluate Print Loop". It's a kind of tool that allows you to run one line of code at a time. REPL-style tools exist in most languages.)
2. Execute a JS program in the current directory with `node file.js`

Getting started with Node.js

node command

Running node without a filename runs a REPL loop

- Similar to the JavaScript console in Chrome, or when you run "python“

```
$ node
> let x = 5;
undefined
> x++
5
> x
6
```

Getting started with Node.js

Node can be used for executing scripts written in JavaScript, completely unrelated to servers

simple-script.js

```
function printPoem() {
console.log('Roses are red,');
console.log('Violets are blue,');
console.log('Sugar is sweet,');
console.log('And so are you.');
console.log();
}
printPoem();
printPoem();
```

The node command can be used to execute a JS file:

```
$ node fileName
```

```
$ node simple-script.js
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.

Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.
```

Node for servers

Here is a very basic server written for NodeJS:

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

const server = http.createServer();

server.on('request', (req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/html');
  res.end('<h1>Hello, World!</h1>');
});

const PORT = 3000;
server.listen(PORT, () => {
  console.log(`Server running at
http://localhost:${PORT}/`);
});
```

WARNING: We really **wouldn't** write servers like this!!!
We'll be using ExpressJS to help, which we'll cover for you later.

Require()

```
const http = require('http');
const server = http.createServer();
```

The NodeJS **require()** statement loads a module, similar to **import** in Java, python or include in C++.

- We can **require()** modules included with NodeJS, or modules we've written ourselves.
- In this example, 'http' is referring to the HTTP NodeJS module

The **http** variable returned by **require('http')** can be used to make calls to the **HTTP API:**

- **http.createServer()** creates a Server object

Emitter .on

```
server.on('request', (req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type',
'text/html');
  res.end('<h1>Hello, World!</h1>');
});
```

The **on()** function is the NodeJS equivalent of **addEventListener.**

The **request** event is emitted each time there is a new HTTP request for the NodeJS program to process.

Emitter .on

```
 server.on('request', (req, res) => {
   res.statusCode = 200;
   res.setHeader('Content-Type',
 'text/html');
   res.end('<h1>Hello, World!</h1>');
 });
```

The **req** parameter gives information about the incoming request, and the **res** parameter is the response parameter that we write to via method calls.

statusCode: Sets the HTTP status code.

setHeader(): Sets the HTTP headers.

end(): Writes the message to the response body then signals to the server that the message is complete

Template literals

Template literals allow you to embed expressions in JavaScript strings:

```
const PORT = 3000;
server.listen(PORT, () => {
  console.log('Server running at http://localhost:' +
              PORT + '/');
});
```

```
const PORT = 3000;
server.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}/`);
});
```

Listen()

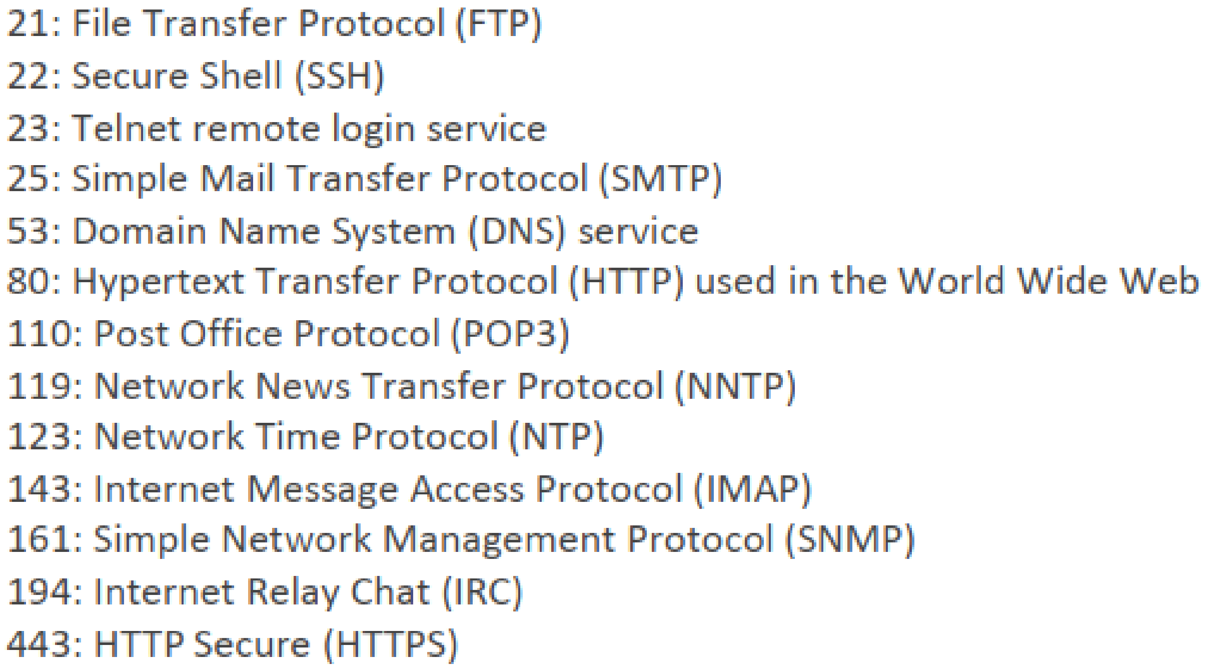
```
const PORT = 3000;
server.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}/`);
});
```

listen(): Starts the server and listens for requests on the specified port.

You can think of the **port** as a house number that, together with the **server's IP address**, uniquely identifies **the server's address**.

Popular default ports

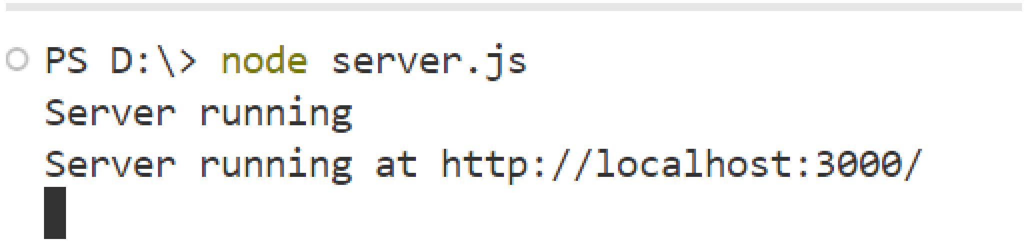
There are many well-known ports, i.e. the ports that will be used by default for particular protocols



21: File Transfer Protocol (FTP)
22: Secure Shell (SSH)
23: Telnet remote login service
25: Simple Mail Transfer Protocol (SMTP)
53: Domain Name System (DNS) service
80: Hypertext Transfer Protocol (HTTP) used in the World Wide Web
110: Post Office Protocol (POP3)
119: Network News Transfer Protocol (NNTP)
123: Network Time Protocol (NTP)
143: Internet Message Access Protocol (IMAP)
161: Simple Network Management Protocol (SNMP)
194: Internet Relay Chat (IRC)
443: HTTP Secure (HTTPS)

Running the server

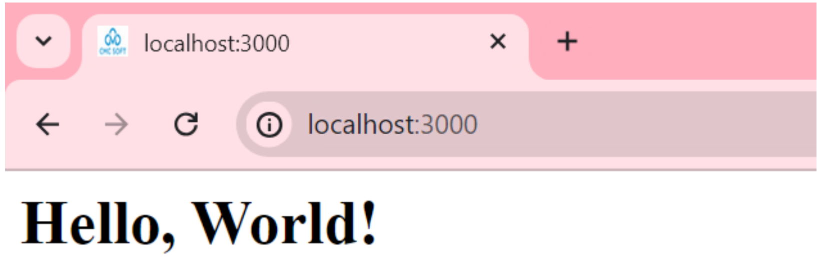
When we run node server.js in the terminal, we see the following:



```
PS D:\> node server.js
Server running
Server running at http://localhost:3000/
```

The process does not end after we run the command, as it is now waiting for HTTP requests on port 3000.

Server response →



Hello, World!

What is REST architecture?

1. RESTful or REST stands for **REpresentational State Transfer**. REST is a well known software architectural style. It defines how the architecture of a web application should behave. It is a resource-based architecture where everything that the REST server hosts, (a file, an image, or a row in a table of a database), is a resource, having many representations. REST was first introduced by *Roy Fielding in 2000.*

2. A REST Server provides access to resources and REST client accesses and modifies the resources using HTTP protocol. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML but JSON is the most popular one.

HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

1. **GET Method**

The purpose of the GET operation is to retrieve an existing resource on the server and return its XML/JSON representation as the response. It corresponds to the READ part in the CRUD (CREATE, RETRIEVE, UPDATE and DELETE) term.

Example:

Retrieve all users from the database.
Retrieve details of a specific user by their ID.

HTTP methods

Example of the **GET method** using Node.js to implement.

```
const server = http.createServer((req, res) => {
    if (req.method === 'GET' && req.url === '/users') {
        res.writeHead(200, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify(users));
    }
});

server.listen(3000, () => {
    console.log('Server running on port 3000');
});
```

The server checks if the incoming request is a GET method and matches the /users endpoint.

HTTP methods

2. POST Method

The POST verb in the HTTP request indicates that a new resource is to be created on the server. It corresponds to the CREATE operation in the CRUD term.

Some applications of POST request:

- Creating a New User Account
- Submitting a Contact Form
- Placing an Order in an E-commerce Application
- Posting a New Blog Article
- ....

HTTP methods

Example of the **POST method** using Node.js to implement.

```
const server = http.createServer((req, res) => {
    if (req.method === 'POST' && req.url === '/users') {
        let body = '';
        req.on('data', chunk => {
            body += chunk.toString();
        });
        req.on('end', () => {
            const { name, email } = JSON.parse(body);
            const newUser = {
                id: users.length ? users[users.length - 1].id + 1 : 1,
                name,
                email
            };
            users.push(newUser);
            res.writeHead(201, { 'Content-Type': 'application/json' });
            res.end(JSON.stringify(newUser));
        });
    }
});
```

HTTP methods

3. PUT Method

- PUT is used to send data to a server to create/update a resource.
- The difference between POST and PUT is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly have side effects of creating the same resource multiple times.

- Example:
  - Updating a User’s Information

HTTP methods

Example of the **PUT method** using Node.js to implement.

```
const server = http.createServer((req, res) => {
    if (req.method === 'PUT' && req.url.startsWith('/users/')) {
        const id = req.url.split('/')[2];
        let body = …….
         req.on('end', () => {
            const { name, email } = JSON.parse(body);
            const user = users.find(user => user.id === parseInt(id));

            if (user) {
                user.name = name || user.name;
                user.email = email || user.email;
                res.writeHead(200, { 'Content-Type': 'application/json' });
                res.end(JSON.stringify(user));
            } else {
                res.writeHead(404, { 'Content-Type': 'application/json' });
                res.end(JSON.stringify({ message: 'User not found' }));
            }
        })}
});
```

HTTP methods

4. DELETE Method

- The DELETE method deletes the specified resource.
- Example: Deleting a User by ID

```
const server = http.createServer((req, res) => {
    if (req.method === 'DELETE' && req.url.startsWith('/users/')) {
        const id = req.url.split('/')[2];
        const userIndex = users.findIndex(user => user.id === parseInt(id));
         if (userIndex !== -1) {
            users.splice(userIndex, 1);
            res.writeHead(200, { 'Content-Type': 'application/json' });
            res.end(JSON.stringify({ message: 'User deleted successfully'
}));
        } else {
            res.writeHead(404, { 'Content-Type': 'application/json' });
            res.end(JSON.stringify({ message: 'User not found' }));
        }}});
```

Node for servers

```
const http = require('http');
const server = http.createServer();

server.on('request', (req, res) => {
 res.statusCode = 200;
 res.setHeader('Content-Type', 'text/html');
 res.end('Hello, World!\n');
});

server.on('listening',function(){
    console.log("Server running");
})
const PORT = 3000;
server.listen(PORT, () => {
 console.log(`Server running at
http://localhost:${PORT}/`);
});
```

The NodeJS server APIs are actually **pretty lowlevel**:

- You build the **request manually**
- You write the **response manually**
- There's **a lot of tedious** processing code

Starting a Node.js Project

There are a few steps to starting a Node.js application, but luckily most projects will follow the same structure.

When using Node.js, you will mostly be using the command line

1. Start a new project directory (e.g. `node-practice`)
2. Inside the directory, run `npm init` to initialize a `package.json` configuration file (you can keep pressing Enter to use defaults)
3. Install any modules with `npm install <package-name>`
4. Write your Node.js file! (e.g. `app.js`)
5. Include any front-end files in a `public` directory within the project.

Along the way, a tool called `npm` will help install and manage packages that are useful in your Node app.

Starting a Node.js Project (ExpressJS)

Run `npm init` to create `package.json`

Run `npm install express` to install the Express.js package

Or better, run `npm install -s express` to save the `express` package into the `dependencies` list under `package.json`

Starting `app.js`

Build your basic app:

```
"use strict";

const express = require('express');
const app = express();

app.get('/posts', function (req, res) {
  res.type("text").send("Hello World");
});

app.listen(8080);
```

Node.js Modules

When you run a `.js` file using Node.js, you have access to default functions in JS (e.g. `console.log`)

In order to get functionality like file i/o or handling network requests, you need to import that functionality from modules - this is similar to the `import` keyword you have used in Java or Python.

In Node.js, you do this by using the `require()` function, passing the string name of the module you want to import.

For example, the module we'll use to respond to HTTP requests in Node.js is called `express`. You can import it like this:

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

Quick reminder on `const` Keyword

Using `const` to declare a variable inside of JS just means that you can never change what that variable references. We've used this to represent "program constants" indicated by `ALL_UPPERCASE` naming conventions

For example, the following code would not work:

```
const specialNumber = 1;
specialNumber = 2; // TypeError: Assignment to constant variable.
```

When we store modules in Node programs, it is conventional to use `const` instead of `let` to avoid accidentally overwriting the module.

Unlike the program constants we define with `const` (e.g. `BASE_URL`), we use `camelCase` naming instead for module objects.

Basic Routing in Express

- Routes are used to define endpoints in your web service
- Express supports different HTTP requests - we will learn GET and POST
- Express will try to match routes in the order they are defined in your code

Adding Routes in Express.js

```
app.get(path, (req, res) => {
  ...
});
```

- `app.get` allows us to create a GET endpoint. It takes two arguments: The endpoint URL path, and a callback function for modifying/sending the response.
- `req` is the request object, and holds items like the request parameters.
- `res` is the response object, and has methods to send data to the client.
- `res.set(...)` sets header data, like "content-type". Always set either "text/plain" or "application/json" with your response.
- `res.send(response)` returns the response as HTML text to the client.
- `res.json(response)` does the same, but with a JSON object.

When adding a route to the path, you will retrieve information from the request, and send back a response using res (e.g. setting the status code, content-type, etc.)

If the visited endpoint has no matching route in your Express app, the response will be a 404 (resource not found)

Useful Request Properties/Methods

Name	Description
`req.params`	Array of parameters from the request’s path
`req.query`	Array of parameters from the request’s querystring

Useful Response Properties/Methods

Name	Description
`res.write(data)`	Writes data in the response without ending the communication
`res.send()`	Sends information back (default text with HTML content type)
`res.json()`	Sends information back as JSON content type
`res.set()`	Sets header information, such as "Content-type"
`res.type()`	A convenience function to set content type (use "text" for "text/plain", use "json" for "application/json")
`res.status()`	Sets the response status code
`res.sendStatus()`	Sets the response status code with the default status text, and sends response back

Setting the Content Type

By default, the content type of a response is HTML - we will only be sending plain text or JSON responses though in our web services

To change the content type, you can use the `res.set` function, which is used to set response header information (e.g. content type).

You can alternatively uses `res.type("text")` and `res.type("json")` which are equivalent to setting `text/plain` and `application/json` Content-Type headers, respectively.

```
app.get('/hello', function (req, res) {
  // res.set("Content-Type", "text/plain");
  res.type("text"); // same as above
  res.send('Hello World!');
});
```

```
app.get('/hello', function (req, res) {
  res.set("Content-Type", "application/json");
  res.send(JSON.stringify({ "msg" : "Hello world!" }));
  // can also do res.json({ "msg" : "Hello world!"});
  // which also sets the content type to application/json
});
```

Request Parameters: Path Parameters

Act as wildcards in routes, letting a user pass in "variables" to an endpoint
Define a route parameter with `:param`

```
Route path: /states/:state/cities/:city
Request URL: http://localhost:8000/states/wa/cities/Seattle
req.params: { "state": "wa", "city": "Seattle" }
```

These are attached to the request object and can be accessed with `req.params`

```
app.get("/states/:state/cities/:city", function (req, res) {
  res.type("text");
  res.send("You sent a request for " + req.params.city + ", "
    + req.params.state);
});
```

Request Parameters: Query Parameters

You can also use query parameters in Express using the `req.query` object, though they are more useful for optional parameters.

```
Route path: /cityInfo
Request URL: http://localhost:8000/cityInfo?state=wa&city=Seattle
req.query: { "state": "wa", "city": "Seattle" }
```

```
app.get("/cityInfo", function (req, res) {
  let state = req.query.state; // wa
  let city = req.query.city;   // Seattle
  // do something with variables in the response
});
```

Unlike path parameters, these are not included in the path string (which are matched using Express routes) and we can't be certain that the accessed query key exists.

If the route requires the parameter but is missing, you should send an error to the client in the response.

Servers, clients, localhost

- Client: The browser. Wants a website, HTML
- Network: Protocols like HTTP communicate between client and server
- Server: Has the website, can "serve" the HTML (and other files)
- localhost: the friendly name for the IP address of the local computer

Servers, clients, localhost, node...

What if we're making food for ourselves/friends/family at home? I'm giving the order, taking the order, prepping it, serving it, everything.

This is just like what we're looking at with Node on our localhost. So, previously we talked about why. Let's look at what and how.

Choosing Error Codes

Use 400 (Invalid Requests) for client-specific errors.

- Invalid parameter format (e.g. "Seattle" instead of a required 5-digit zipcode)
- Missing a required query parameter
- Requesting an item that doesn't exist

Use 500 (Server error) status codes for errors that are independent of any client input.

- Errors caught in Node modules that are not related to any request parameters
- SQL Database connection errors (next week!)
- Other mysterious server errors...

Setting Errors

The Response object has a `status` function which takes a status code as an argument.

The 400 status code is what we'll use to send back an error indicating to the client that they made an invalid request.

A helpful message should always be sent with the error.

```
app.get("/cityInfo", function (req, res) {
  let state = req.query.state;
  let city = req.query.city;
  if (!(state && city)) {
    res.status(400).send("Error: Missing required city and state
query parameters.");
  } else {
    res.send("You sent a request for " + city + ", " + state);
  }
});
```