Task [Chapter 5]

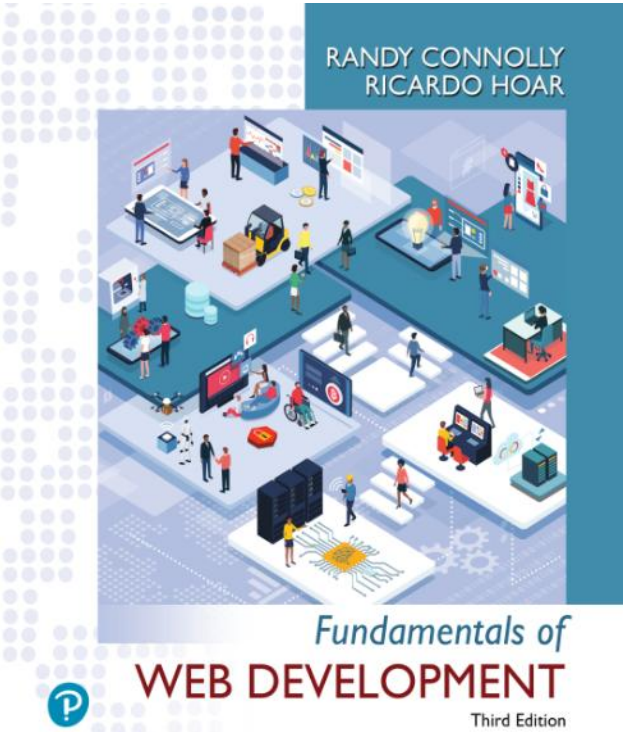
```
<html>
<head> </head>
<body>
<h1 id="myh1"> Show Table </h1>
<div> </div>
</body>
</html>
```

name	course	grade
abc	se371	A

Question:
When the user clicks on the "Show Table" heading, the table below should be displayed. Use DOM manipulation methods to achieve this.

Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar



Chapter 6

Server-Side:

Modules, Async, Fetch, Web APIs,
Node.js, Express.js

Introducing Node.js

Node can be used to build a server that generates HTML in response to HTTP requests, it uses JavaScript as its programming language.

Node Advantages

- JavaScript Everywhere
- Push Architectures (pub/sub, webSockets)
- Nonblocking Architectures
- Rich Ecosystem of Tools and Code
- Broad Adoption

https://youtu.be/uVwtVBpw7RQ?feature=shared (3.5mins)

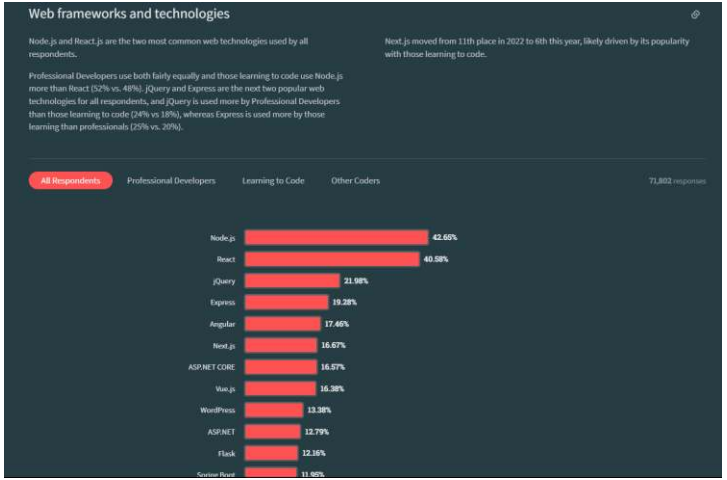
JavaScript Everywhere

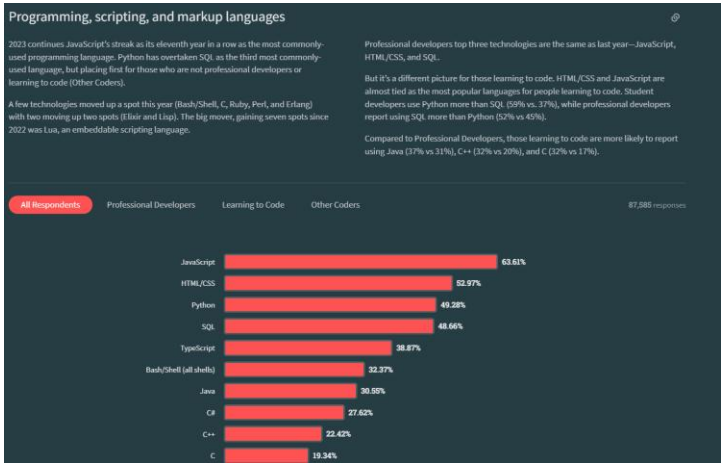
Using the same language on both the **client** and the **server** has multiple benefits:

1- Developer productivity is likely to be higher when there is only a **single language** to use for the entirety of a project.

2- With a single language, there are also **more opportunities for code sharing and reuse.**

Now, the most popular and widely used programming language in the world; this means hiring knowledgeable developers is likely to be easier and the hiring team only needs to test its potential applicants for knowledge with a single language.



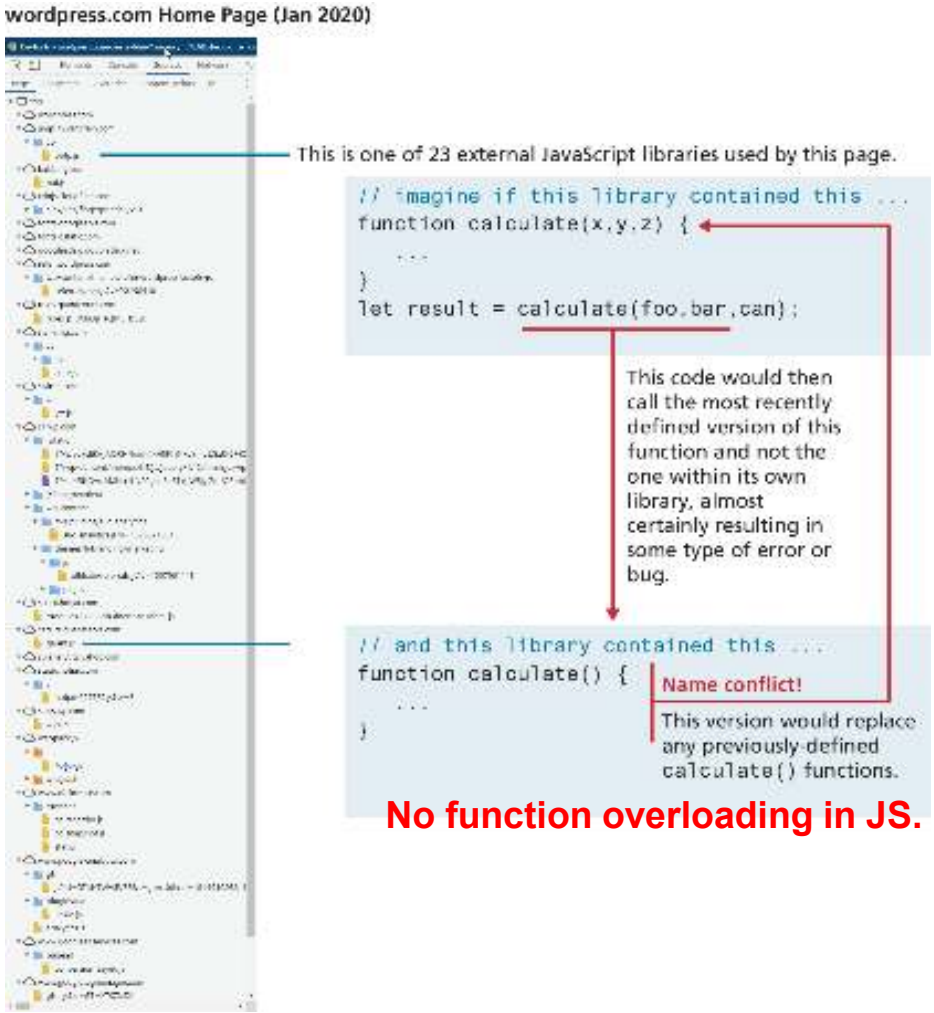


https://survey.stackoverflow.co/2023/#section-admired-and-desired-web-frameworks-and-technologies

Modules: Name Conflicts In JavaScript

Complex contemporary JavaScript applications might contain hundreds of literals defined in dozens of .js files, so some way of preventing name conflicts is needed.

```
let product = (x, y) => x*y;
let product = (x, y, z) => x*y*z;

console.log(product(2, 3)); // NaN
```



Modules

Literals defined within a module are scoped to that module.

In Node, a **module** is simply a file that contains JavaScript. Unlike a regular JavaScript external file (on the browser/client-side).

On the client-side, you do have to tell the browser that a JavaScript file is a module and not just a regular external JavaScript file within the <script> element. This is achieved via the type attribute as shown in the following:

```
<script src="art.js" type="module"></script>
```

In Node, to make content in the module file available to other scripts outside the module, you have to make use of the **export** keyword, then import it using the **require** keyword.

Running a node application

If you have installed Node, you will need to open a command/terminal window, navigate to the folder of your application, and enter the following command:

- node app.js

You can also just type:

- node app

Then to stop it, type: ctrl-c

Node.js **version 22.x**(The latest version: https://nodejs.org/en/download/package-manager/current)

Import/Export Modules in JavaScript

File: app.js

```
// CommonJS, every file is module (by default)
// Modules - Encapsulated Code (only share minimum)

const names = require('./mod-names');

const selem = require('./mod-utils');


selem (names.first_name);

selem(names.secret);
```

mod-names.js

```
// local
const secret = 'SECRET PHRASE'

// share
const first_name = 'salah'
const last_name = 'abid'

module.exports =
        {  first_name, last_name }
// here we export an object
```

mod-utils.js

```
const saySelem = (name) => {
    console.log(`Selem Mr ${name}`)
}
// export default
module.exports = saySelem;
// here we export a function
```

To run the app

- node app

Selem Mr salah

Selem **Undefined**

Source Code Example: chapter06/01-modules

Node core modules

- Node has many included modules (os, path, fs, http, url, etc):

Core Module	Description
http	http module includes classes, methods and events to create Node.js http server.
url	url module includes methods for URL resolution and parsing.
querystring	querystring module includes methods to deal with query string.
path	path module includes methods to deal with file paths.
fs	fs module includes classes, methods, and events to work with file I/O.
util	util module includes utility functions useful for programmers.

Simple Node Application: the fs module

File: app.js

```
const { readFileSync, writeFileSync } = require('fs');
console.log('start');

const first =
  readFileSync('./content/first.txt', 'utf8');
const second =
  readFileSync('./content/second.txt', 'utf8');

writeFileSync(
        './content/result.txt',
        `Here is the result : ${first},
                        ${second}` + "\n",
        { flag: 'a' }  // a for appending
)

console.log('Task completed!')
```

first.txt

Selem this is the first text file

second.txt

Selem this is the second text file

Tu run the app

```
> node app

start
Task completed!
```

result.txt

Here is the result : Selem this is the first text file, Selem this is the second text file

Source ***Code Example: chapter06/02-files***

Simple Node Application: the http module

```
// make use of the http module
const http = require('http');

// configure HTTP server to respond with simple message to all requests
const server = http.createServer(function (request, response) {
                response.writeHead(200, {"Content-Type": "text/plain"});
                response.write("Hello this is our first node.js HTTP server");
                response.end();
});

// Listen on port 8080 on localhost
const port = 8080;
server.listen(port);

// display a message on the terminal
console.log("Server running at port=" + port);
```

Simplest http server

File: simplest-server.js

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

const server = http.createServer(
    (req, res) => {
        res.write("This is my response to your request!");
        res.end();
        return;
     }
);

server.listen(5000);
console.log("Listening on port 5000");
```

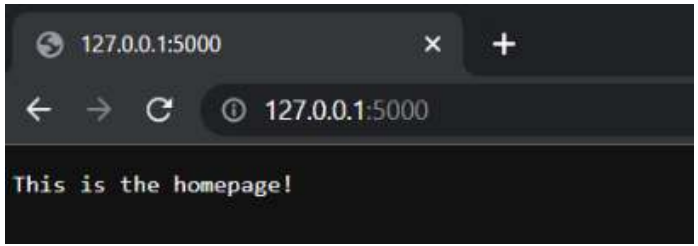
Run using > node simplest-server
- Now you can access http://127.0.0.1:5000 from your browser.
- Note that even if you try to access http://127.0.0.1:5000/ffdfgf/gdfg , you will get the same response.

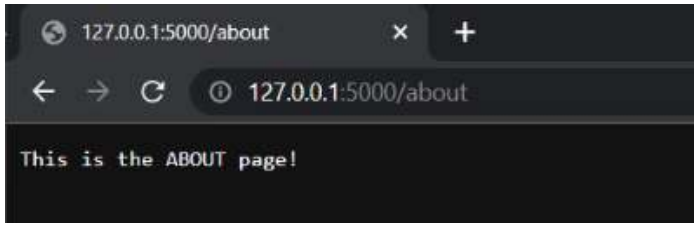
Source Code Example: chapter06/031-simple-http-server

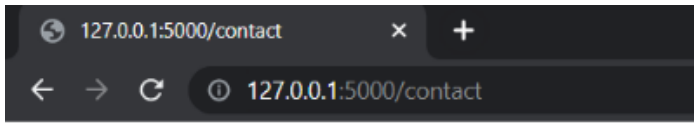
http server: handling different requests

File: simplest-server-2.js

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

const server = http.createServer(
  (req, res) => {
    if( req.url === '/' ){
      res.end("This is the homepage!");
    } else if( req.url === '/about' ){
      res.end("This is the ABOUT page!");
    } else
    res.end(`<h1> Page not Found! <h1>
                <p><a href="/">Homepage</a></p>`);
    return;
  }
);

server.listen(5000);
console.log("Listening on port 5000");
```







Page not Found!

Homepage

Source Code Example: chapter06/032-simple-http-server

Static file server example: the http and path modules

fileserver.js

```
const http = require("http");
const url = require("url");
const path = require("path");
const fs = require("fs");

// our HTTP server now returns requested files
const server = http.createServer(function (request, response) {

    // get the filename from the URL
    let requestedFile = url.parse(request.url).pathname;
    // now turn that into a file system file name by adding the current
    // local folder path in front of the filename
    let filename = path.join(process.cwd(), requestedFile);

    // check if it exists on the computer
    fs.exists(filename, function(exists) {
        // if it doesn't exist, then return a 404 response
        if (! exists) {
            response.writeHead(404, {"Content-Type": "text/html"});
            response.write("<h1>404 Error</h1>\n");
            response.write("The requested file isn't on this machine\n");
            response.end();
            return;
        }

        // if file exists then read it in and send its
        // contents to requestor
        fs.readFile(filename, "binary", function(err, file) {
            // maybe something went wrong (e.g., permission error)
            if (err) {
                response.writeHead(500, {"Content-Type": "text/html"});
                response.write("<h1>500 Error</h1>\n");
                response.write(err + "\n");
                response.end();
                return;
            }
            // ... everything is fine so return contents of file
            response.writeHead(200);
            response.write(file, "binary");
            response.end();
        });
    });
});

// we don't have to use port 8080; here we are using 7000
server.listen(7000, "localhost");
console.log("Server running at http://127.0.0.1:7000/");
```

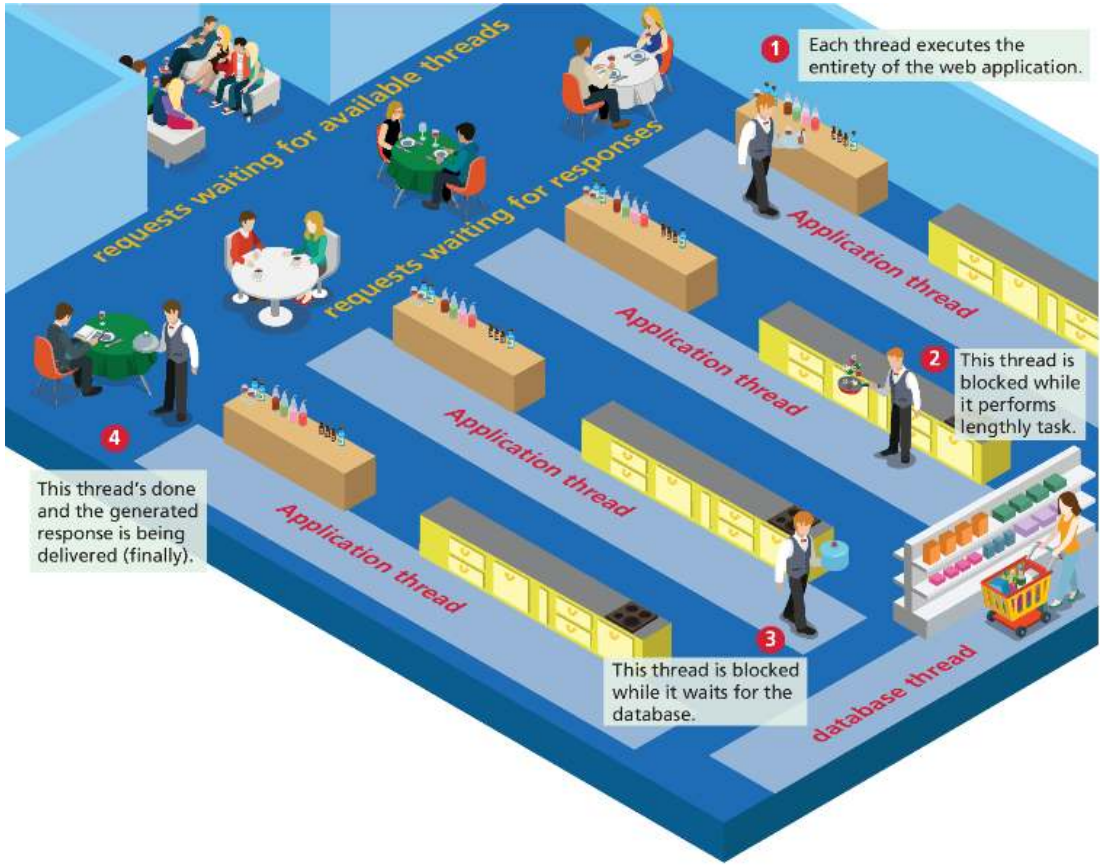
Using three new modules in this example that process URL paths and read/write local files.

Source Code Example: chapter06/033-simple-file-server

Blocking Architectures

Apache runs applications like JEE or PHP using either a blocking multiprocessing or multithreaded model.

Using a restaurant analogy, it would be as though a single person had to handle all the tasks required for each table.

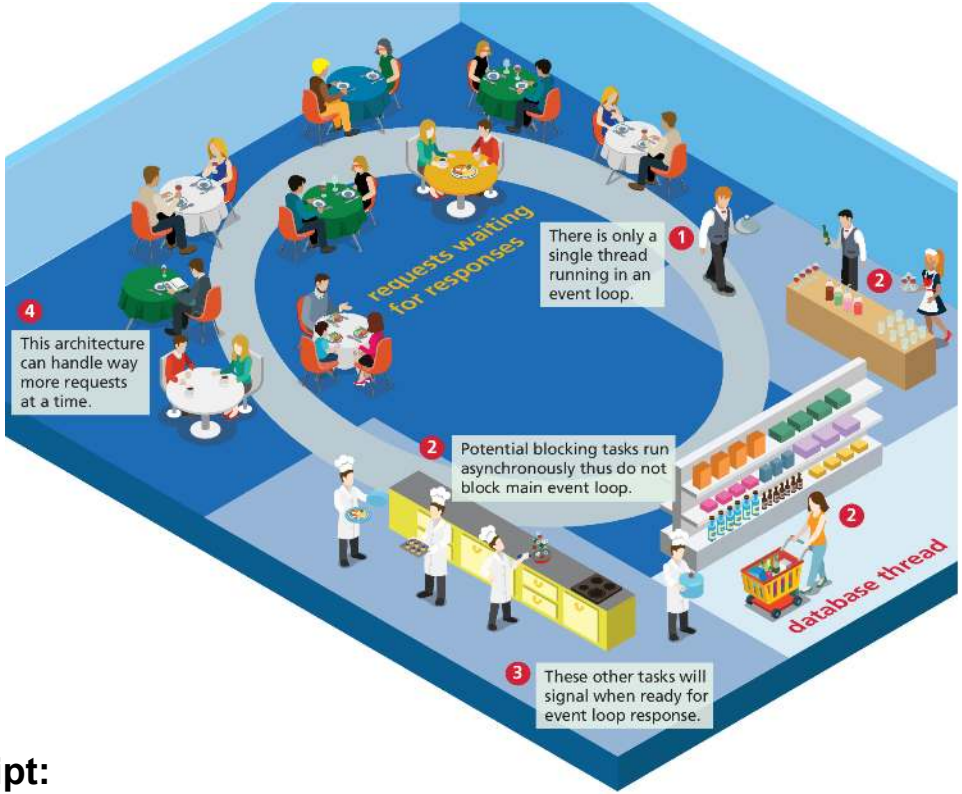


Non-blocking Architectures

Here a single worker is servicing all the requests in a single event loop thread.

This worker can only be doing a single thing at a time. But other tasks are delegated to other agents.

Node uses a **nonblocking**, asynchronous, single-threaded architecture.



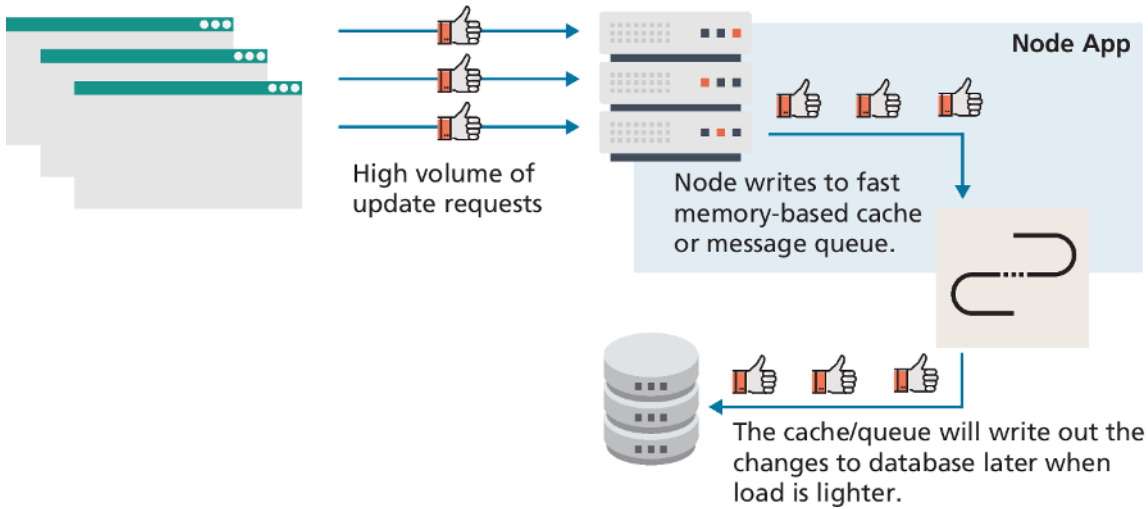
Nice videos on the event loop in Javascript:
https://www.youtube.com/watch?v=8aGhZQkoFbQ

Handling high volume data changes in Node

Node is well suited to developing data-intensive real-time applications that need to interact with distributed computers and whose data sources are noSQL databases.

Consider a simple Like button. It would need to handle a massive number of concurrent data writes.

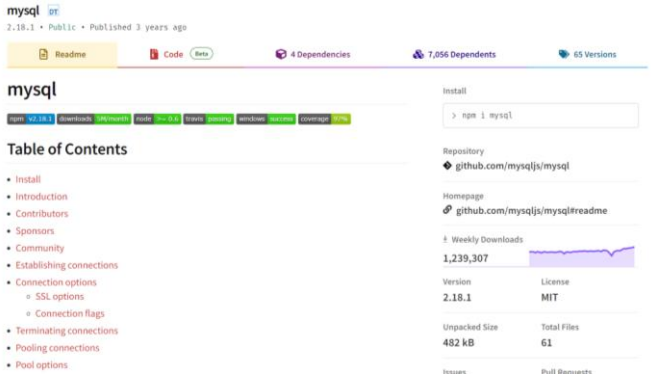
A Node-based system could make use of a memory-based message queueing system that would keep a record of all data changes, and then those changes could eventually be persisted



Node Disadvantages

Node isn’t ideal for all web-based applications.

~~For sites whose data is in a traditional relational database such as MySQL, accessing that data in Node was **is** a complex programming task.~~



The single-thread nonblocking architecture of Node is also not ideal for computationally heavy tasks, such as video processing or scientific computing

Asynchronous Coding with JavaScript

asynchronous code is code that is doing (or seemingly doing) multiple things at once. In multi-tasking operating systems, asynchronous execution is often achieved via **threads**: each thread can do only one task at a time, but the operating system switches between threads

Many contemporary web sites make use of asynchronous JavaScript data requests of Web APIs, thereby allowing a page to be dynamically updated without requiring additional HTTP requests.

(A **web API** is simply a web resource that returns data instead of HTML, CSS, JavaScript, or images).

Simple Node Application: fs module, async version

File: app.js

```
const { readFile, writeFile } = require('fs');
console.log(Starting task A...');

readFile('./content/first.txt', 'utf8', (err, result) => {
  if (err) {
    console.log(err);
    return;
  }

  writeFile(
      './content/result-async.txt',
      `Here is the Async result : ${result}`,
      (err, result) => {
        if (err) {
          console.log(err);
          return;
        }
        console.log('Task A completed!');
      }
  )
})
console.log('starting next task...')
```

first.txt

Selem this is the first text file

Tu run the app

```
> node app

Starting task A...
starting next task...    // notice the order here
Task A completed!        // notice the order here
```

result-async.txt

Here is the Async result : Selem this is the first text file

Note: readFile and writeFile are built as asynchronous functions.

Source Code Example: 04-files-async

Simple Node Application: fs module, async version (2)

File: app.js (with two files to read from)

```
const { readFile, writeFile } =
                                require('fs');

console.log('Starting Task A…');

readFile('./content/first.txt', 'utf8',
    (err, result) => {
        if (err) {
            console.log(err);
            return;
        }
        const first = result;
```

```
readFile('./content/second.txt', 'utf8', (err, result) =>
    {
        if (err) {
            console.log(err);
            return;
        }
        const second = result;
        writeFile(
            './content/result-async.txt',
            `Here is the result : ${first}, ${second}`,
            (err, result) => {
                if (err) {
                    console.log(err);
                    return;
                }
                console.log('done with this task');
            }
        )
    })
})
console.log('starting next task');
```

Tu run the app

```
> node app

Starting task A...
starting next task...
Task A completed!
```

result-async.txt

Here is the result : Selem this is the first text file, Selem this is the second text file

Notice that the second readFile is called by the callback function that is passed as the third argument of the first readFile, so that it has access to the read result.
-> **Callback HELL!!!**

So what are Promises?

Solution to the Callback Hell.

A **Promise** object represents the eventual completion (or failure) of an **asynchronous** operation and its resulting value.

Probably, that promise will be completed and we will receive the **data,** or it won't, and we will get an **error** instead.

```
// declaration and instantiation
const promiseObj = new Promise( (resolve, reject) => {
    doWork();
    if (someCondition)
        resolve(someValue);      // works like as a return
    else
        reject(someMessage);    // also works like as a throw err
 });

// call to promise
promiseObj
    .then( someValue => {
        // success, promise was achieved!
    })
    .catch( someMessage => {
        // oh no, promise was not satisfied!!
    });
```

Creating a Promise

Creating a promise is quite simple: you simply instantiate a Promise object.
For promises to make some sense, we must first understand that the handler function passed to the Promise constructor must take two parameters: a **resolve**() function and a **reject**() function.

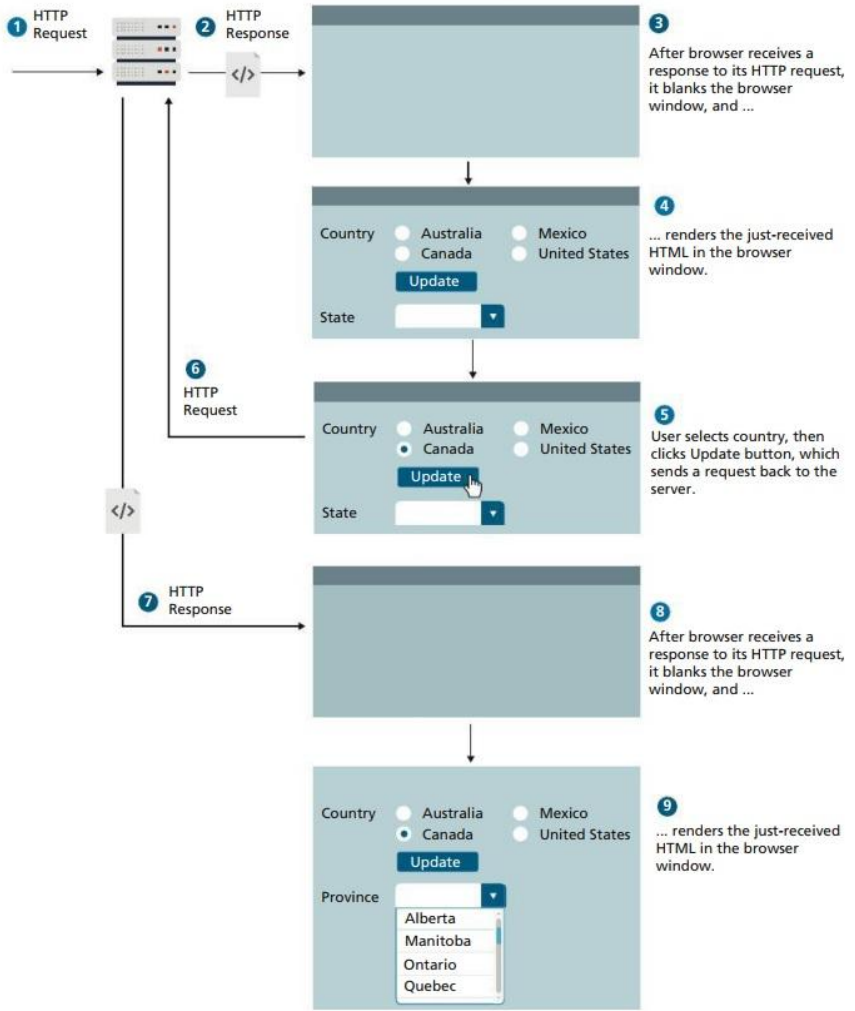
File: app.js

```
const { readFile, writeFile } = require('fs');

console.log('start');

const read = (path) => {
  return new Promise((resolve, reject)=>{
    readFile(path, 'utf8', (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
};

read('./content/first.txt')
  .then( (data) => console.log(data) )
  .catch( (err) => console.log(err) );
```

Source Code Example: 05-fs-with-promise

Another use case for promises: (on the client-side)

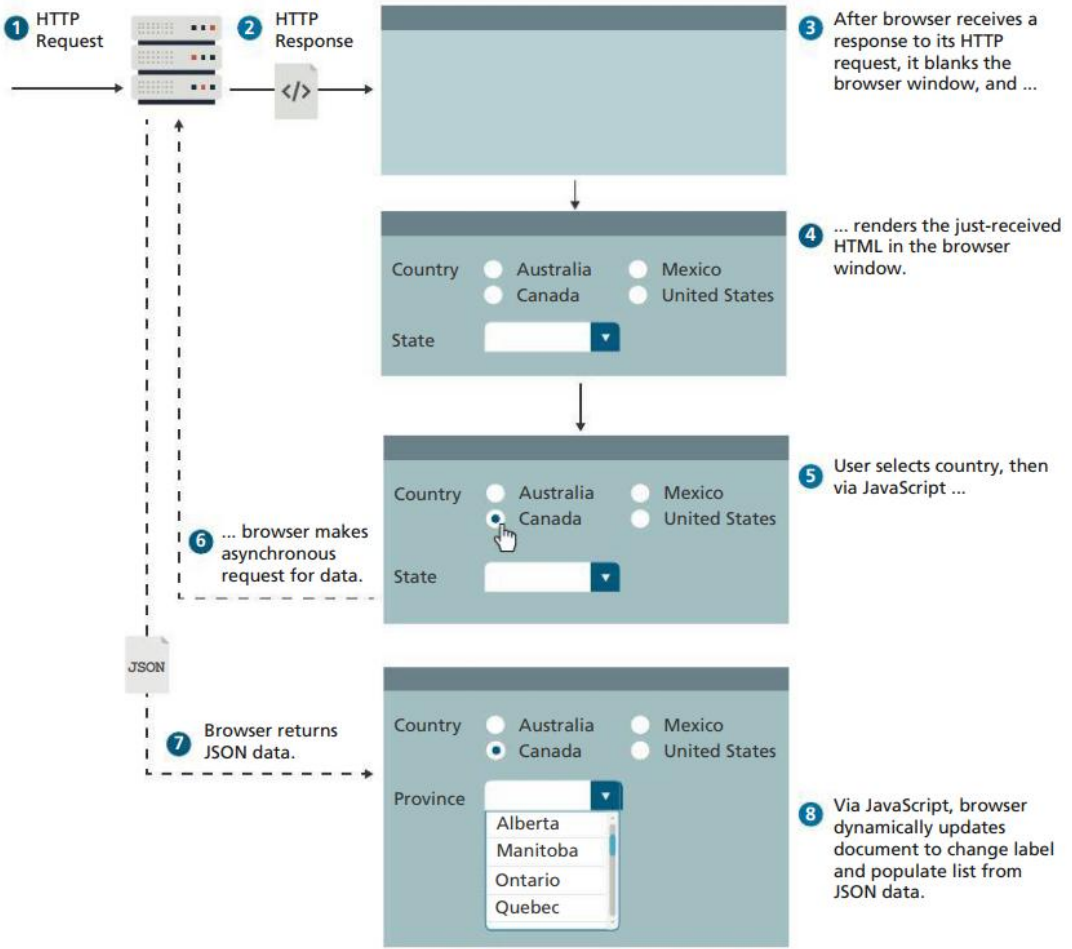
Let's look at the **normal** (synchronous) request-response loop:



Asynchronous data requests

Note: ***fetch*** is done from the client-side to get data asynchronously. It replaces an older technology (**AJAX**).

```
let cities = fetch('/api/cities?country=italy');
```



Fetching Data from a Web API

So what does **cities** contain after this call? You might expect it to contain the requested JSON data. But it doesn't. Why? Because it will take time for this service to execute and respond to our request.

What the above fetch will return instead is a special **Promise** object.

Promises in JavaScript are usually handled by invoking two methods: then() for responding to the successful arrival of data, and catch() for responding to an unsuccessful completion.

```
fetch('/api/cities?country=italy')
  .then( (response) => {
      // do something with this data
      console.warn('response received!!!');
  })
  .catch( (err) => {
     console.log(err);
  )
```

Example of asynchronous request using fetch … and promises

How to run:
npm install json-server
1- npx json-server data.json
2- run index.html

The request returns JSON data in the following format:

```
[
 {"iso":"AT","name":"Austria", ...},
 {"iso":"CA","name":"Canada", ...},
 ...
]
```

```
<select id="countries">
    <option value=0>Select a country</option>
</select>
<script>
document.addEventListener("DOMContentLoaded", function() {
  const apiURL = 'api/countries.php';

  const countryList = document.querySelector('#countries');

  fetch(apiURL)
    .then( response => response.json() )
    .then( data => {

        // populate list with this JSON country data
        data.forEach( c => {
            const opt = document.createElement('option');
            opt.setAttribute('value', c.iso);
            opt.textContent = c.name;
            countryList.appendChild(opt);
        });
    })
    .catch( error => { console.error(error) } );
});
</script>
```

1. Make the fetch request.
2. Pass the function that will execute when the HTTP response is received.
3. Pass the function that will execute when the JSON data is extracted from that response.
4. Handle any error that might occur with the fetch.

Create a new <option> element using the fetched JSON data.

Sample generated markup from this code:

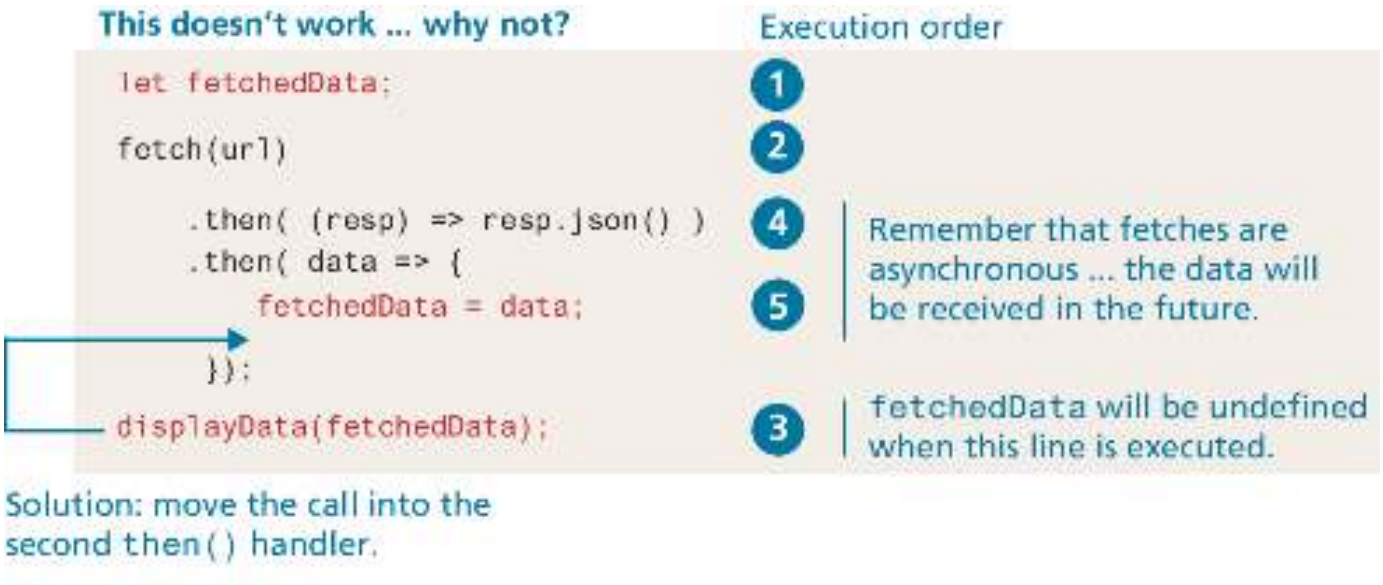
```
<select id="countries">
    <option value=0>Select a country</option>
    <option value="AT">Austria</option>
    <option value="CA">Canada</option>
    ...
</select>
```

Source Code Example: chapter06/12-fetch

Common Mistakes with Fetch

We often commit some version of the mistake shown below:

Multiple nested fetches can be problematic,



Npm, package.json and dependencies

Note: we need **Express**, a node package that can help us with **routing** HTTP requests. To install it in your project we start by creating a package.json to our project, it contains general information and dependencies:

(npm = Node Package Manager)

- **npm init –y**

// creates a basic package.json (-y to skip questionnaire)

- **npm install express**

// installs express in node-modules folder and adds it to the dependencies of the project in package.json

Source Code Example: 07-npm-demo

package.json

```
{
  "name": "05-npm-demo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2“
  }
}
```

Cross-Origin Resource Sharing

Modern browsers prevent cross-origin requests by default, so sharing content legitimately between two domains becomes harder.

Cross-origin resource sharing (CORS) is a mechanism that uses new HTTP headers in the HTML5 standard that allows a JavaScript application in one **origin** (i.e., a protocol, domain, and port) to access resources from a different origin. If an API site wants to allow any domain to access its content through JavaScript, it would add the following header to all of its responses:

Access-Control-Allow-Origin: *

In node: > npm install cors

And then in your app.js:

```
app.use(cors()); // cors allows to access api through javascript;
```

Versions: an Industry Standard

Versions of the npm packages in package.json follow a standard: Semantic Versioning (SemVer):

"package": "MAJOR.MINOR.PATCH"

The MAJOR version should increment when you make incompatible API changes.
The MINOR version should increment when you add functionality in a backwards-compatible manner.
The PATCH version should increment when you make backwards-compatible bug fixes.

This means that PATCHes are bug fixes and MINORs add new features but neither of them break what worked before.

Finally, MAJORs add changes that won't work with earlier versions.

Request npm install to automatically update to the latest minor version or patch:

Update to the latest patch versions: "package": "~1.3.8"
Update to latest MINOR version: package": "^1.3.8"

Dependencies and code sharing (git)

When we need to share code on a platform like GitHub for example, the /node-modules folder do not have to be shared, it can be recreated by your team members from the package.json. We can remove the node-modules/ folder and restore it by simply using: **> npm install**	If you are using git for code sharing, you can add a .gitignore file on your project folder then you can add this line to your file: /node-modules This folder will be ignored when you want to share your code to your shared git repository (Github). ***Source Code Example: 07-npm-demo -app***

Adding a dev-dependency

Source Code Example: ***07-npm-demo-app***

Note: some tools are needed in development time only. These can be installed as a dev-dependency, **the nodemon tool for example is useful when we want to make changes to our code and see the results reflected without manually restarting the server:**

- **npm install nodemon -D**

// installs nodemon in “dev-dependencies”
// can also install it globally using npm install –g nodemon
// to execute (npm exec nodemon app.js) or….

Then we can add scripts to be able to run the app in different configuration:

- **npm run start** // will simply run the app
- **npm run dev** // will launch the app under
  // the control of nodemon: try
  // modifying your code and save to see
  // what happens

package.json

```
{
  "name": "05-npm-demo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node app.js",
    "dev": "nodemon app.js",
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "nodemon": "^2.0.20"
  }
}
```

Routing: Importing Express

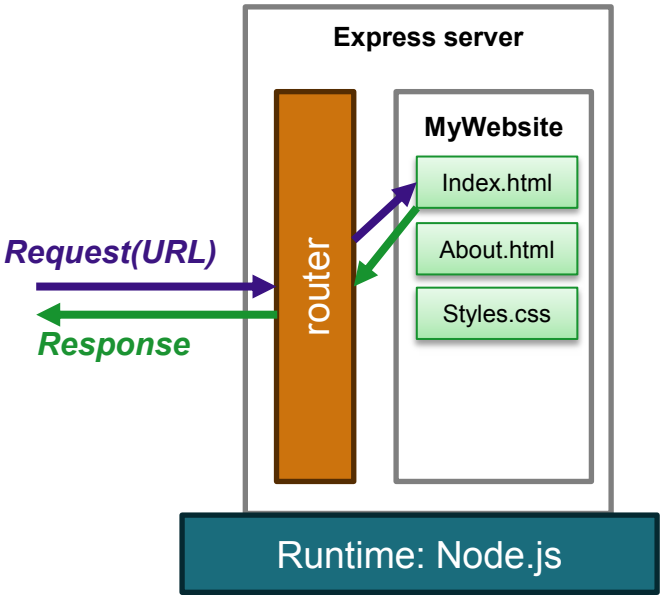
With **Express** you write **handlers** for the different **routes** in your application. A **route** is a URL, a series of folders or files or parameter data within the URL.

The **request** object contains a params object that holds any parameter data included with the request/route.

The **response** object provides methods for setting HTTP headers and cookies, sending files or JSON data, and so on.

A simple file server using the static() function:

```
const express = require("express");
const app = express();
app.use( express.static('public') );
app.listen(8080, () => { ... });
```

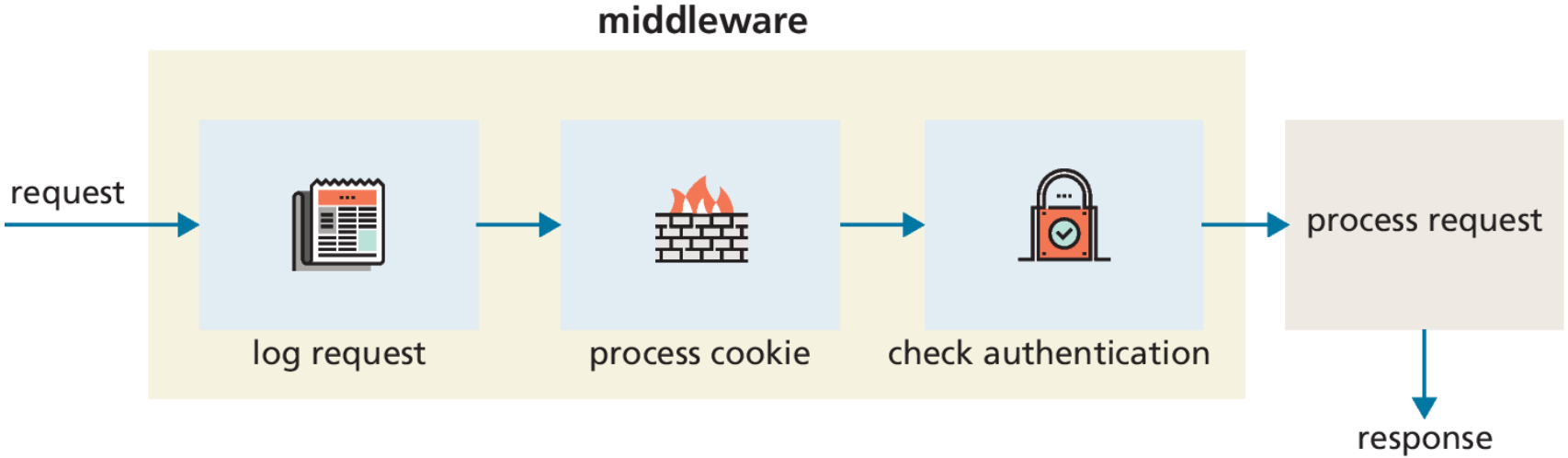


g3

Note:
The public folder can contain css, images, html, etc (try adding a complete static site)

Middleware functions in Express:

The chain of responsibility pattern



This approach is useful to split the server operations into smaller units (perform some validation on the data). That, leads to a better app structure and reuse. You can block the execution of the current chain and pass control to functions that handle errors.

Express: middleware functions

(Definition) Middleware: software that is a bridge between an application and the data.

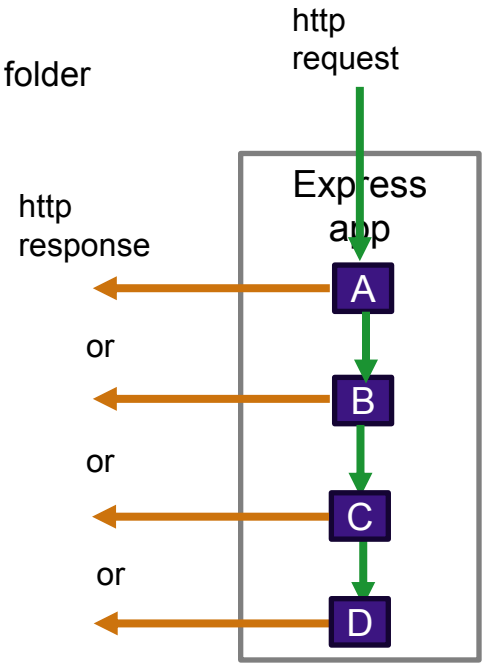
```
const express = require("express");

const app = express();

app.use(express.static('public’)); // static files like css files

app.listen(8080, () => { ... });
```

The **app.use()** function installs the provided **middleware** function on the chain of responsibility.

Using express for routing

Source Code Example: 08-express-server- -app2

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

// the public folder will be visible by http requests
// like http://localhost:3000/css/styles.css if the css folder is in the public folder
A  app.use( express.static('public') );

app.get('/', (request, response) => {
B
  response.sendFile('./pages/index.html');
});

C  app.get('/about', (request, response) => {
     response.sendFile('./pages/about.html');
   });

// any other request will get this 404 response
D  app.use( (request, response) => {
     response.status(404).sendFile('./pages/404.html');
   });

app.listen(3000, "localhost", () => { console.log("Listening on port 3000...") });
```



Add middleware (chained operations)

```
// Middleware that logs all incoming requests (middleware at the root of the route).
        app.use( (req, res, next) => {
            console.log( `${req.method} ${req.path} - ${req.ip}` );
            next();
        });
```

```
// Middleware can be mounted at a specific route using app.METHOD(path, middlewareFunction,
callback):
        app.get('/now', function(req, res, next) {
            req.time = new Date().toString();
            next();
        }, function(req, res) {
            res.json( {time: req.time});
        });
```

Source Code Example: 09-env-variables- -app3

Environment Variables

Environment variables provide a mechanism for configuring your Node application. To see your variables:

```
console.log(process.env);
```

You can set your environment variables using the popular dotenv package. Use a configuration file named **.env** to store any number of **key=value** pairs, such as:

```
PORT=8080
BUILD=development
```

```
> npm install dotenv
```

Within your Node applications, you can reference the values in this file

```
// make use of dotenv package
require('dotenv').config();
// reference values from the .env file
console.log("build type=" +
process.env.BUILD);
server.listen(process.env.PORT);
```

Simple API

Source Code Example: 10-simple-api

Node can be used to create APIs that can be consumed by client applications. Here is a very simple API that reads in a JSON data file and then returns the JSON data when the URL is requested.

```
const fs = require('fs');
const path = require('path');
const express = require('express');
require('dotenv').config();

const app = express();

// for now, we will read a json file from the data folder
const jsonPath = path.join(__dirname, 'data',
'SE2022.json');

// get data
let curriculum;
```

```
fs.readFile(jsonPath, (err,data) => {
  if (err)
    console.log('Unable to read json data file');
  else
  curriculum = JSON.parse(data);
});

// return the curriculum when a root request arrives
app.get('/', (req,resp) => { resp.json(curriculum) } );

app.listen(process.env.PORT, () => {
  console.log("Listening at port… " + process.env.PORT);
});
```

Separating Functionality into Modules

What if we had five or six or more routes? In such case, our single Node file would start becoming too complex. A better approach would be to separate out the routing functionality into separate modules.

You could also place your route handler logic into a separate module.

Defining a module

```
const fs = require('fs');
const path = require('path');

const jsonPath = path.join(__dirname, 'data', 'SE2022.json');

let curriculum;

fs.readFile(jsonPath, (err,data) => {
  if (err)
    console.log('Unable to read json data file');
  else
  curriculum = JSON.parse(data);
});

const getData = () => { return curriculum; }

module.exports = getData;
```


Source Code Example: chapter06/10-simple-api

Send parameters in route:
(route params)

With the :param syntax you can send parameters to the api:

```
app.get( '/echo/:word', (req, res) => {
  let param = req.params.word;
  res.json( { echo: param } );  //use the variable as needed
});
```

By requesting : https://ip:port/echo/**hello**

'hello' will be assigned to the **word** variable (from req.params.word) then you can use this argument as you need.

Source Code Example: chapter06/15-route-params

Send parameters in query (2)
(query params)

We can also use URL encoded queries then access the query parameters using *req.query.parameter* : http://ip:port/employee?first=skander&last=turki

```
app.get('/ employee',  (req, res) => {
    res.json({ employee: `${req.query.first} ${req.query.last}`});
});
app.post('/ employee',  (req, res) => {
    res.json({employee : `${req.query.first} ${req.query.last}`});
});
```

Note: no need to install any library.

Source Code Example: chapter06/14-query-params

Send parameters using forms (3)

Using a form on the client side, we can send requests using forms:

```
<form action="/name" method="post">
    <label>First Name :</label><input type="text" name="first"><br>
    <label>Last Name :</label><input type="text" name="last"><br>
    <input type="submit" value="Submit">
</form>
```

We need a middleware to parse parameters from **request payload** (POST, DELETE..):

```
const bodyParser = require('body-parser');

app.use(bodyParser.urlencoded({ extended: false }))

let p_first = req.body.first;    // then access the req.body object
```

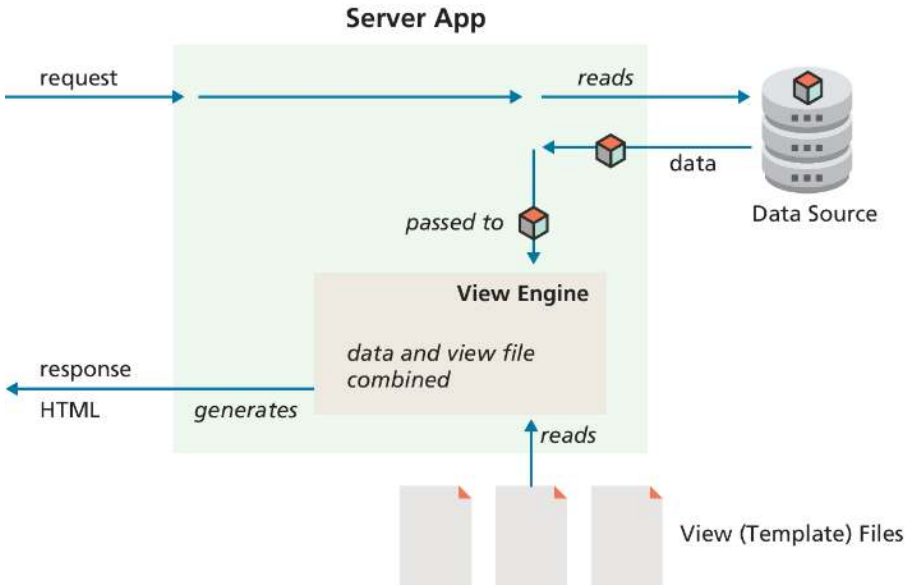

Source Code Example: chapter06/15-form

View Engines

It is possible to use JavaScript in Node to generate HTML using a **view engine** like **EJS**

You only need to install the appropriate package using npm, and then tell Express which folder contains the view files and which view engine to use to display those files.

> npm install ejs



https://ejs.co/#docs

Introduction to EJS (Embedded JavaScript Templates)

- Your templates (previously html pages) need to be placed inside the /views folder (by default).

```
// register view engine
app.set('view engine', 'ejs’);

// configure views folder if not default (views)
app.set('views', 'otherThanViewsFolder’);

// response.render will send the ejs template (index.ejs file)
app.get('/', (request, response) => {
    response.render('index');
})
```

Injecting data into EJS views

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

// express app
const app = express();

// register view engine
app.set('view engine', 'ejs');

app.use(express.static('public'));

app.get('/', (req, res) => {
 const blogs = [
    {title: '..', snippet:'...'},
    {title: '…', snippet:'...'},
    {title: '…', snippet:'...'},
  ];
```

```
 // second argument is data sent to template
  res.render('index', { title: 'Home', blogs: blogs });
});

app.get('/about', (req, res) => {
  res.render('about', { title: 'About' });
});

app.get('/create', (req, res) => {
  res.render('create', { title: 'Create a new blog' });
});

// any other request goes to 404
app.use((req, res) => {
  res.status(404).render('404', { title: '404' });
});

// listen for requests
app.listen(3000);
```

Example EJS view (Embedded JavaScript Templates) 2

EJS supports “partials” : include an ejs file as part of another, so that parts of documents can be reused by different files.

If the head is common to all our pages, we can create a head.ejs file containing the header then include the header as “partial” of all our pages with:

The partial files can be organized inside a partials folder under the views folder:
/views/partials/head.js
/views/partials/header.js
/views/partials/footer.js

```
<%- include("./partials/head.ejs") %>
<body>
  <%- include("./partials/header.ejs") %>
  ….
   <%- include("./partials/footer.ejs") %>
</body>
</html>
```

Example EJS view (Embedded JavaScript Templates) 2

```
<html lang="en">
<%- include("./partials/head.ejs") %>

<body>
  <%- include("./partials/nav.ejs") %>

  <div class="blogs content">
    <h2><%= title %>: All Blogs</h2>

    <% if (blogs.length > 0) { %>
      <% blogs.forEach(blog => { %>

        <h3 class="title"><%= blog.title %></h3>
        <p class="snippet"><%= blog.snippet %></p>

      <% }) %>
    <% } else { %>
      <p>There are no blogs to display...</p>
    <% } %>
```

```
  </div>

  <%- include("./partials/footer.ejs") %>
</body>
</html>
```

<% %> : can include normal javascript that will not be displayed:

```
<% if () { %>
…….
<% } %>
```

<%= title %> : Allows to include the value of the title variable.

Supporting material

https://www.youtube.com/watch?v=yXEesONd_54&t=1309s

Installation:

-Node.js **version 22.x**(The latest version: https://nodejs.org/en/download/package-manager/current)