Back-end Development with

Node.JS

7

Fundamental Web Programming

Asst. Prof. Manop Phankokkruad, Ph.D.
School of Information Technology
King Mongkut's Institute of Technology Ladkrabang

KMITL

Outline

- Back-end Development
- Introduction to Node.JS
- Basic Command line
- A Simple Web Server
- Routing
- Web Static Resources
- Introduction to Express.JS

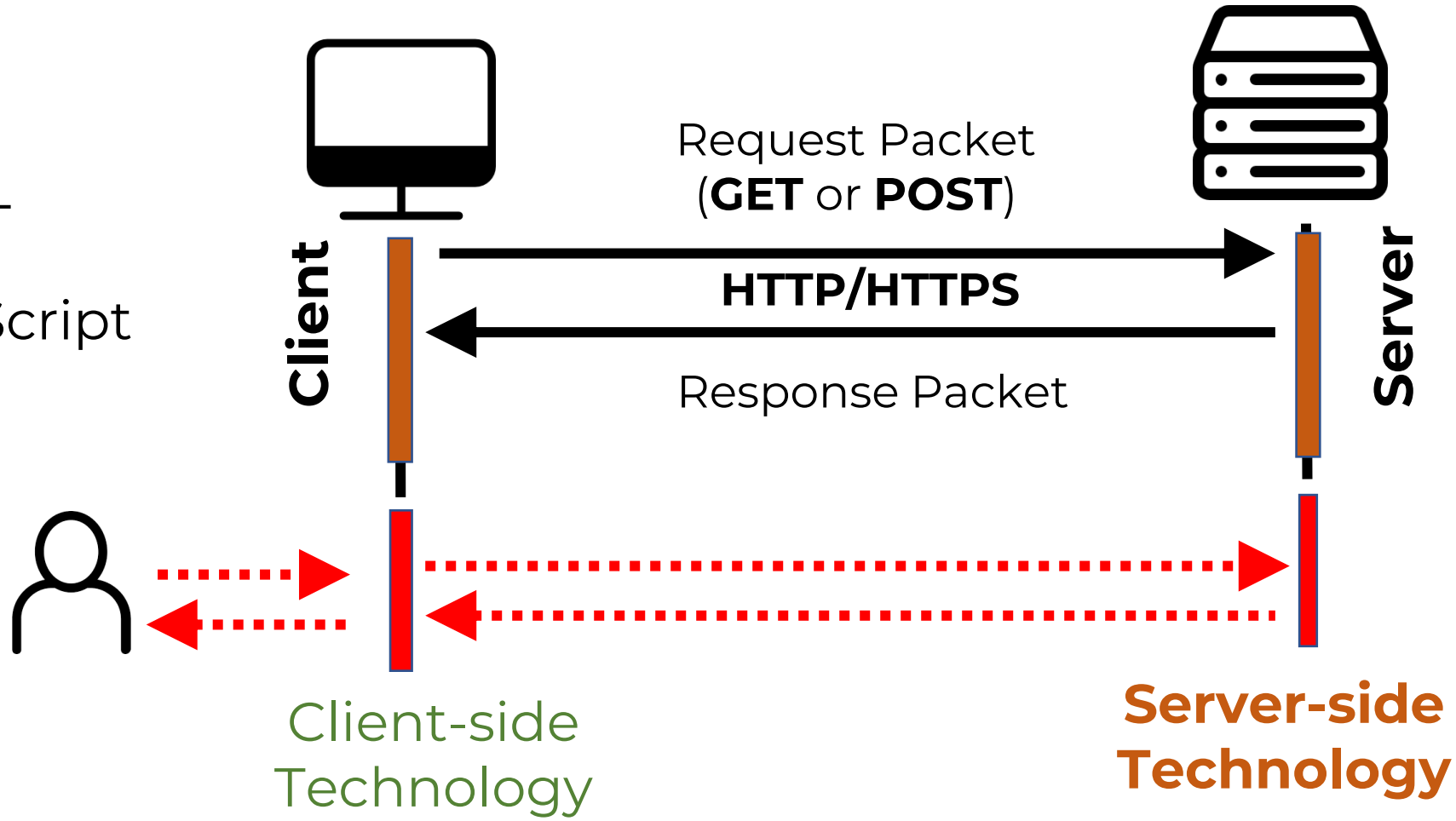
Introduction

- Standard web sites operate on a request/response basis.
- A user requests a resource such as HTML document.
- Server responds by delivering the document to the client.
- The client processes the document and displays it to user.

Introduction

Server-side/Back-end Technology

- HTML
- CSS
- JavaScript
- DOM



Server-side Scripting

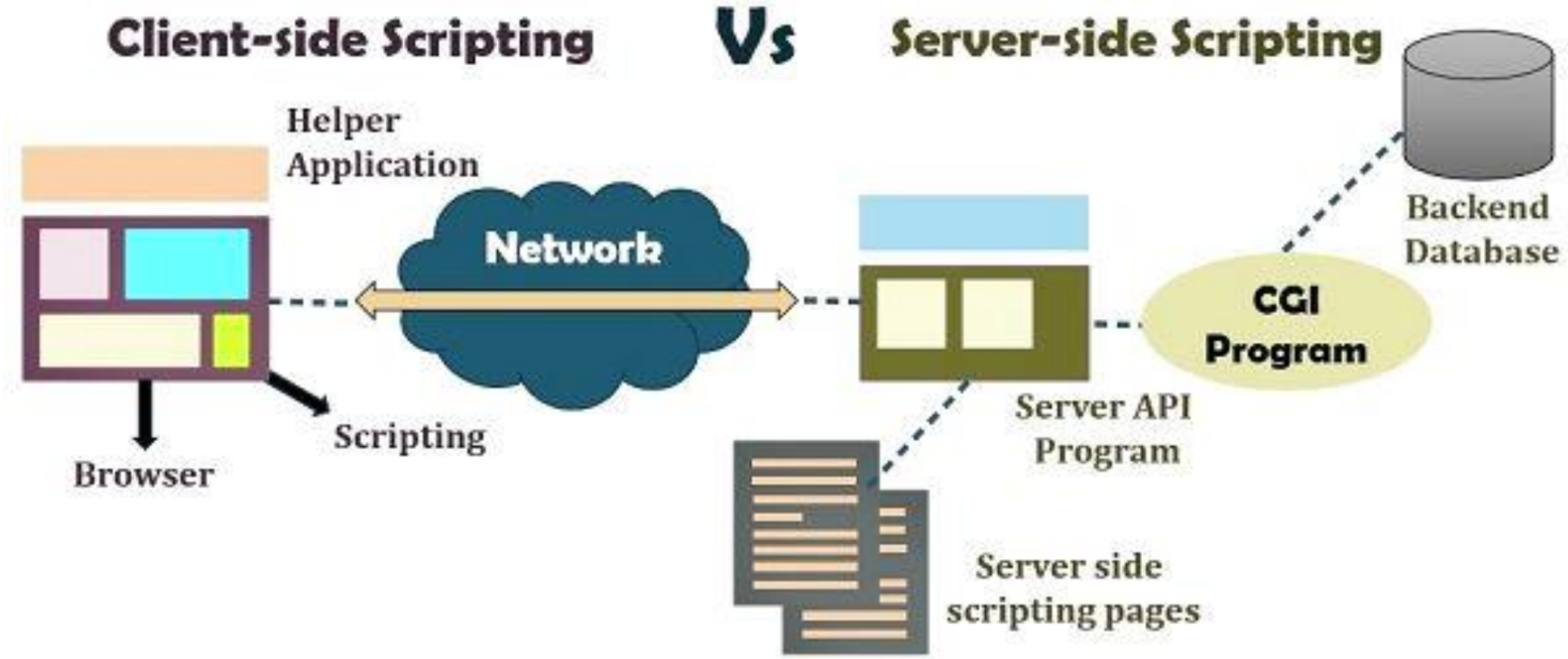
> Server-side scripting is a technique used in web development which involves employing scripts on a web server which produce a response customized for each user's request to the website.

- Scripts can be written in any of several server-side scripting languages that are available.
- Server-side scripting is often used to provide a *customized interface* for the user.
- Server-side scripting focuses on what content is delivered, how it's delivered, and how it's stored, among other things.

KMITL

Server-side Scripting

Figure: Difference Between Server-side Scripting and Client-side Scripting



Server-side Scripting

- PHP, Node.JS, Python and ASP.NET are the main technologies for server-side scripting.
- The script is interpreted by the server meaning that it will always work the same way.
- Server-side scripts are never seen by the user. They run on the server and generate results which are sent to the user.
- Running all these scripts puts a lot of load onto a server but none on the user's system.

Server-side Scripting Languages

There are several server-side scripting languages available, including:

- ASP and ASP.NET (*.asp , *.aspx)
- Google Apps Script (*.gs)
- Java (*.jsp) via JavaServer Pages
- JavaScript using Server-side JavaScript (*.ssjs, *.js) such as Node.JS
- PHP (*.php)
- Ruby (*.rb, *.rbw) such as Ruby on Rails
- Python (*.py) (using Flask, Django)

KMITL

Introduction to Node.JS

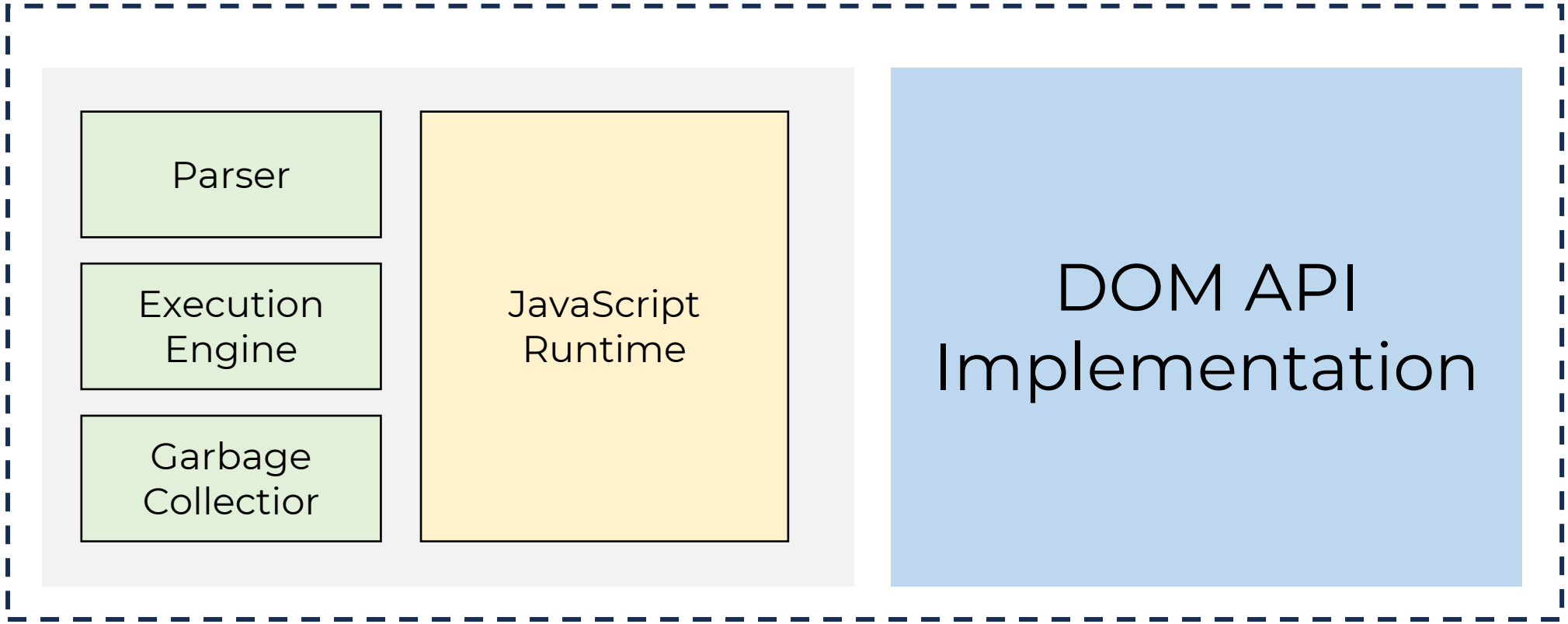
Introduction to Node.JS

> **Node.JS** is a server-side runtime environment built on Google Chrome's JavaScript Engine (V8 Engine). Node.js is a cross-platform (run on Windows, Linux, Unix, macOS, and more), open-source, back-end JavaScript runtime environment, that executes JavaScript code outside a web browser.

Node.js environment is event-driven and provides non-blocking I/O, that optimizes throughput and scalability in web applications.

Introduction to Node.JS

The JavaScript interpreter ("engine") that Chrome uses to interpret, compile, and execute JavaScript code.



Introduction to Node.JS

Node.JS can be installed on different OS platforms such as Windows, Linux, Mac OS X, etc. You need the following tools on your computer.

- The Node.JS binary installer
- Node Package Manager (NPM)
- IDE or Text Editor

KMITL

Introduction to Node.JS

The Node Ecosystem

- Node.JS is the software that enables JavaScript to run on the server, uncoupled from a browser, which in turn allows frameworks written in JavaScript (like Express) to be used.
- Another important component is the database. The simplest of web apps will need a database, and there are available for all the major relational databases such as MySQL, MariaDB, SQLite, PostgreSQL, Oracle, SQL Server, and MongoDB.

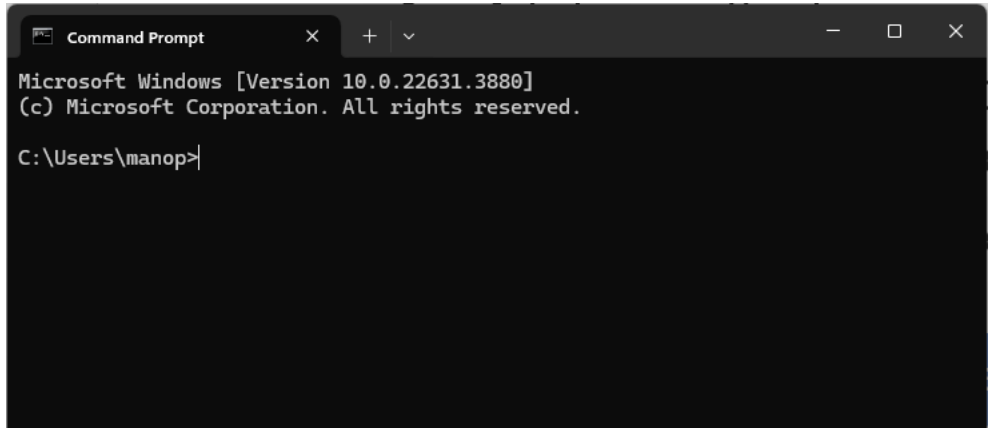
KMITL

Using the Terminal

Terminal (also called a console or command prompt) is the power and productivity tool. Using Node.JS highly recommend you spend some time familiarizing yourself with your terminal of choice. I recommend installing a more sophisticated terminal such as ConsoleZ, ConEmu, or PowerShell.

```
$ mkdir <your-folder-name>
$ cd <your-folder-name>
```

```
$ node <js-filename>
$ nodemon <js-filename>
```



KMITL

Node Package Manager

npm (Node Package Manager) is Command-line tool that lets you install packages (libraries and tools) written in JavaScript and compatible with NodeJS.

npm install package-name : This downloads the package-name library into a node_modules folder.

npm uninstall package-name: This removes the package-name library from the node_modules folder, deleting the folder if necessary.

```
$ npm install <package-name>
$ npm install express
$ npm uninstall express
```

KMITL

Node Package Manager

To install one or more packages, use the following:

```
npm install <package-name>
# or
npm i <package-name>...

# e.g. to install lodash and express
npm install lodash express
```

To uninstall one or more locally installed packages:

```
npm uninstall <package name>
```

KMITL

NPM : Setting up a package configuration

Node.js package configurations are contained in a file called package.json that you can find at the root of each project. You can setup a brand new configuration file with default values use:

```
npm init --yes
# or
npm init -y
```

To install a package and automatically save it to your package.json, use:

```
npm install --save <package>
```

KMITL

A Simple Web Server

- The static HTML websites are coming from a PHP or ASP background, you're probably used to the idea of the web server (Apache, IIS, etc.) serving your static files so that a browser can view them over the network.
- **Node.JS** offers a different paradigm than that of a traditional web server: the app that you write is the web server. Node simply provides the framework for you to build a web server.

KMITL

A Simple Web Server

Here is a very basic server written for NodeJS:

Include the HTTP Node.JS library

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

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

When the server gets a request, send back “Hello World” in plain text.

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

When the server is started, print a log message.

```
server.on('listening', function() {
  console.log('Server running!');
});
```

Start listening for messages

```
server.listen(3000);
```

KMITL

A Simple Web Server

Here is a very basic server written for NodeJS:

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

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

- The NodeJS **require()** statement loads a module. We can **require()** modules included with Node.JS, or modules we've written ourselves.
- The **http** variable returned by **require('http')** can be used to make calls to the HTTP API.

  http.createServer() creates a Server object.

KMITL

A Simple Web Server

This server returns the same response no matter what the request is.

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

const server = http.createServer();

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

server.on('listening', function() {
  console.log('Server running!');
});

server.listen(3000);
```

KMITL

Routing

Routing refers to the mechanism for serving the client the content it has asked for. For web-based client/server applications, the client specifies the desired content in the URL; specifically, the path and querystring.

Let’s serve a really minimal website consisting of a home page, an About page, and a Not Found page. For now, we’ll stick with our previous example and just serve plaintext instead of HTML.

KMITL

Routing

Example code of simple routing

```
const http = require('http')
const port = process.env.PORT || 3000

const server = http.createServer((req,res) => {
  const path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase()
  switch(path) {
    case '':
      res.writeHead(200, { 'Content-Type': 'text/plain' })
      res.end('Homepage')
      break
    case '/about':
      res.writeHead(200, { 'Content-Type': 'text/plain' })
      res.end('About')
      break
    default:
      res.writeHead(404, { 'Content-Type': 'text/plain' })
      res.end('Not Found')
      break
  } })
server.listen(port, () => console.log(`server started on port ${port};`));
```

KMITL

Serving Static Resources

Static resources generally don’t change. When we 've worked with Apache or IIS, we're probably used to just creating an HTML file, navigating to it, and having it delivered to the browser automatically.

Node doesn’t work like that: we’re going to have to do the work of opening the file, reading it, and then sending its contents along to the browser.

KMITL

Serving Static Resources

Example code of serving static resource (part 1)

```
const http = require('http')
const fs = require('fs')
const port = process.env.PORT || 3000

function serveStaticFile(res, path, contentType, responseCode = 200) {
  fs.readFile(__dirname + path, (err, data) => {
    if(err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' })
      return res.end('500 - Internal Error')
    }
    res.writeHead(responseCode, { 'Content-Type': contentType })
    res.end(data)
  })
}
```

Serving Static Resources

Example code of serving static resource (part 2)

```
const server = http.createServer((req,res) => {
  const path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase()
  switch(path) {
    case '':
      serveStaticFile(res, '/public/home.html', 'text/html')
      break
    case '/about':
      serveStaticFile(res, '/public/about.html', 'text/html')
      break
    case '/img/logo.png':
      serveStaticFile(res, '/public/img/logo.png', 'image/png')
      break
    default:
      serveStaticFile(res, '/public/404.html', 'text/html', 404)
      break
  }
})

server.listen(port, () => console.log(`server started on port ${port}; `));
```

KMITL

Express.JS

Web Applications With Express

Express.JS

Express.JS is a minimal and flexible Node.js web application framework, providing a robust set of features for building web applications.

- **path** : Specifies the path portion or the URL that the given callback will handle.
- **middleware** : One or more functions which will be called before the callback. Essentially a chaining of multiple callback functions. Useful for more specific handling for example authorization or error handling.

Express.JS

- **callback** : A function that will be used to handle requests to the specified path. It will be called like callback(request, response, next), where request, response, and next are described below.
- **callback request** : An object encapsulating details about the HTTP request that the callback is being called to handle.
- **response** : An object that is used to specify how the server should respond to the request.
- **next** : A callback that passes control on to the next matching route. It accepts an optional error object.

KMITL

Express.JS

We have set up the development, now it is time to start developing our first app using Express. Create a new file called index.js and type the following in it.

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

app.get('/', function(req, res){
   res.send("Hello world!");
});

app.listen(3000);
```

request object(req) represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, etc.

response object(res) represents the HTTP response that the Express app sends when it receives an HTTP request.

KMITL

Express.JS

app.method(path, handler)

This METHOD can be applied to any one of the HTTP verbs - **get, set, put, delete**. An alternate method also exists, which executes independent of the request type.

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

app.get('/hello', function(req, res){
   res.send("Hello World!");
});

app.listen(3000);
```

Path is the route at which the request will run.
Handler is a callback function that executes when a matching request type is found on the relevant route.

KMITL

GET query params in Express

```
app.get('/hello', function (req, res) {
  const queryParams = req.query;
  console.log(queryParams);
  const name = req.query.name;
  res.send('Hello, ' + name);
});
```

Query parameters are saved in **req.query**

KMITL

Express.JS

We're going to use a library called Express.JS on top of Node.JS. You can specify routes in Express.

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

app.get('/', function (req, res) {
  res.send('Hello World!');
})

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
})
```

```
app.get('/', function (req, res) {
  res.send('Main page!');
});

app.get('/hello', function (req, res) {
  res.send('GET hello!');
});

app.post('/hello', function (req, res) {
  res.send('POST hello!');
});
```

This example is saying:- When there's a GET request to **http://localhost:3000/hello** , respond with the text "GET hello!"

KMITL

Express.JS : Handler parameters

```
app.get('/hello', function (req, res) {
  res.send('GET hello!');
});
```

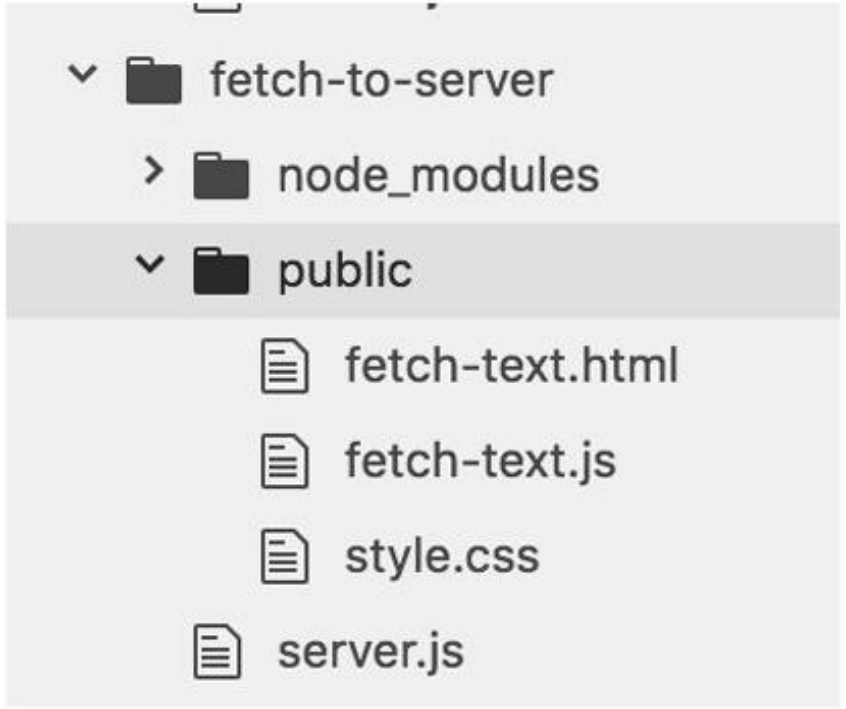
Express has its own Request and Response objects:

- **req** is a Request object
- **res** is a Response object
- **res.send()** sends an HTTP response with the given content
    - Sends content type "text/html" by default

KMITL

Express.JS : Server static data

We can instead serve our HTML/CSS/JS statically from the same server:



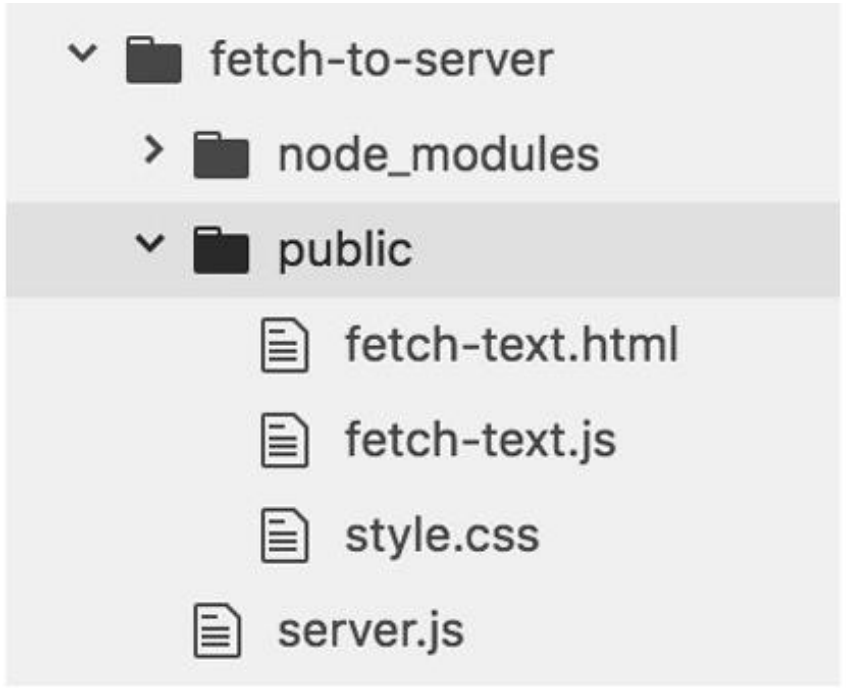

```
const express = require('express');
const app = express();

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

app.get('/', function (req, res) {
  res.send('Main page!');
});
```

Express.JS : Server static data

We can instead serve our HTML/CSS/JS statically from the same server. This line of code make our server start serving the files in the ‘public’ directory.




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

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

app.get('/', function (req, res) {
  res.send('Main page!');
});
```