FULL STACK

Express and Socket.IO

simplilearn

You Already Know

Course(s):

Restful API Design with Node, Express, and MongoDB

Recap

- Get introduced to RESTful API
  - RESTful API
  - Integrate error handling
  - Consume and test with Postman
- Create API using CRUD operations
  - Adding Post index and show routes
  - Update and delete Post route
- Enable authentication and security
  - JWT and user model
  - Passport for authentication

Recap

- Add real-world features to API
  - Using schema tweaks
  - Fix Post's index route
  - Consume and test new features

- Explain caching and rate limiting
  - Add Post pagination and integrate rate limiting
  - Cache queries and serve valid data

- Deploy the application
  - Cloud provider and database provider
  - Testing public API

A Day in the Life of a MERN Stack Developer

In this sprint, Joe has to develop a basic chat application using Express.js and Socket.io which can be used for internal communication between employees of an e-commerce company.

In this lesson, we will learn how to solve this real-world scenario to help Joe complete his task effectively and quickly.

Learning Objectives

By the end of this lesson, you will be able to:

- Implement Model-View-Controller design pattern
- Create Jade templates
- Configure Express and Postman
- Create REST APIs
- Handle GET and POST data
- Work with Socket.IO

FULL STACK

Working with Express.js

Introduction to ExpressJs

Express.js is a built-in Node.js framework that helps developers create smarter and faster server-side REST and web applications.

Express.js is developed in JavaScript language and it is easy to learn and manipulate.

Using Express.js, you reduce programming time in half.

It is so scalable and flexible that one can build web applications and cross-platform hybrid mobile apps.

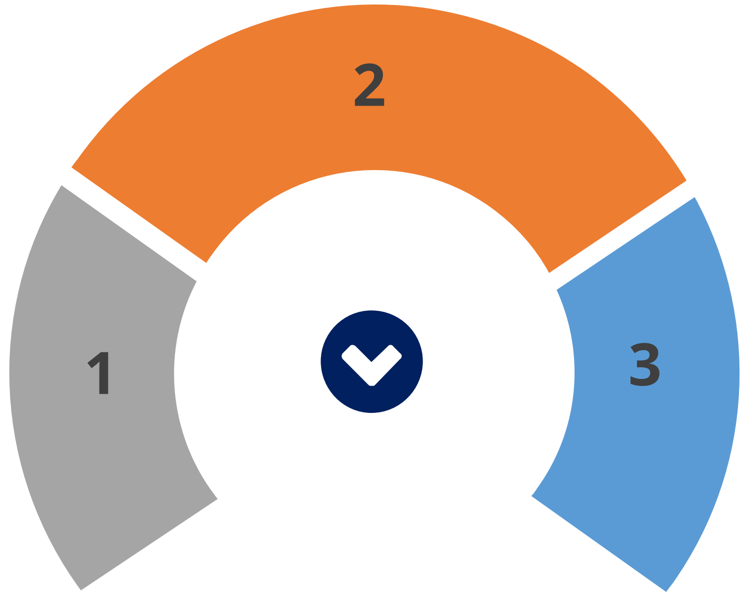
Importance of ExpressJs

Following features make ExpressJs a popular choice:

- Simple to understand
- Built-in connect
- Supports many extensions
- Fully developed feature set
- Scalable in nature

Working with Static Files

To handle static files such as images, CSS files, and JavaScript files, use the express.static built-in middleware function in Express.



Clients can download these files as these files are from the server.

Syntax:
express.static(root, [options])

Model-View-Controller

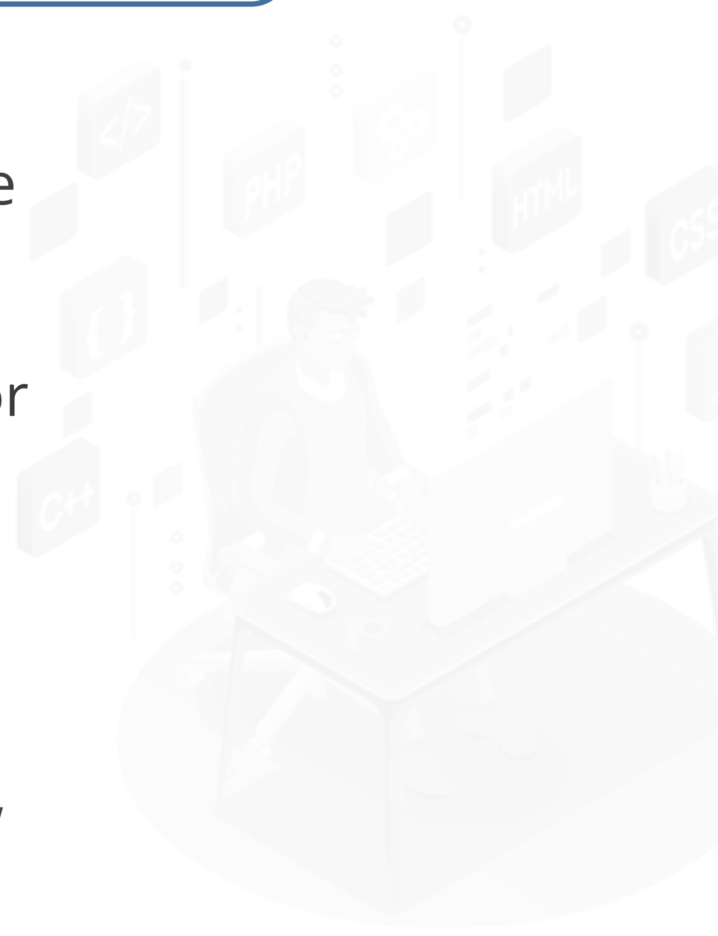
Model-View-Controller is a design pattern used for software projects.

In an MVC pattern, an application and its development are divided into three interconnected parts. They are:

1. Model: This is the part of our application that will deal with the database or any data-related functionality.

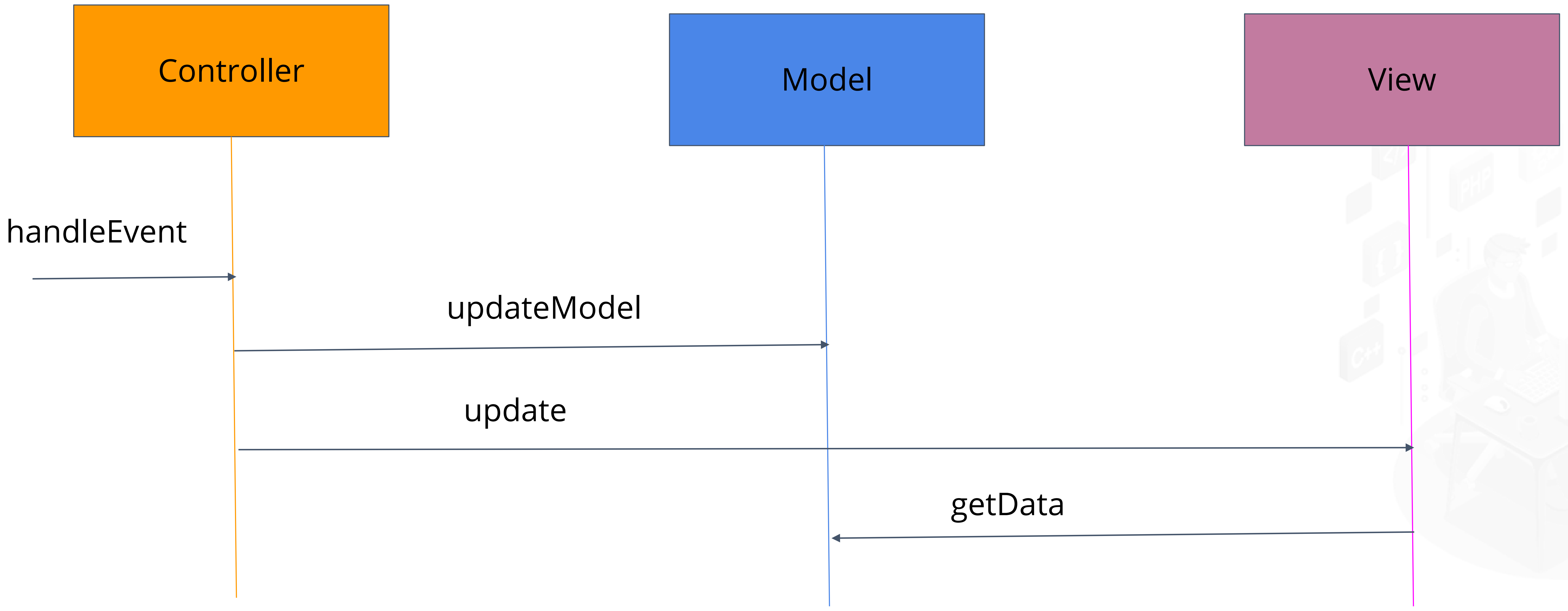
2. View: This is the part which the user will be able to see. It is basically the pages that we're going to send to the client.

3. Controller: Controller acts on both model and view, controls the data flow into model object, and updates the view whenever data changes.



Model-View-Controller

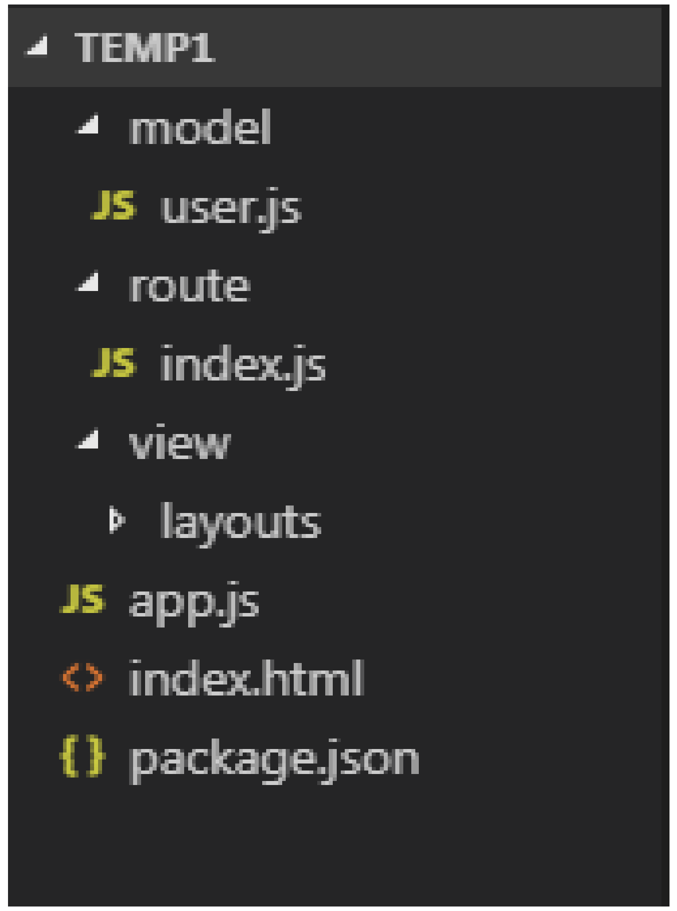
Workflow of model-view-controller:



Model-View-Controller

An example of a node.js project structure that follows model-view-controller pattern is shown below.

- Run *npm init* to generate *package.json* file.
- Create the folder structure according to this screenshot:



```
TEMP1
  model
    user.js
  route
    index.js
  view
    layouts
  app.js
  index.html
  package.json
```

Jade Templates

Jade is a template engine used for server-side templating in node.js.

The same markup will be like this in Jade:

For example:

```
<div>
<h1>Tasks</h1>
  <ul>
    <li>Task 1</li>
    <li>Task 2</li>
  </ul>
  <p>Finish your task within a week</p>
</div>
```

Jade Templates

The same markup will be like this in Jade:

```
div
  h1 Tasks
  ul
    li Task 1
    li Task 2
  p.
    Finish your task within a week
```

Jade Templates

Use the following command to install Jade:

```
npm install jade
```

Three basic features of Jade:

1. Simple Tags: Jade uses indentation instead of closing tags.

For example:

```
div
    p Hello!
    p World!
```

Jade Templates

2. Attributes: Jade provides special shorthand for IDs and classes.

```
div.movie-card#Harry Potter
        h1.movie-title Harry Potter
        ul.genre-list
                li Comedy
                li Thriller
```

Jade Templates

3. Blocks of Text: Jade treats the first word of every line as an HTML tag. Adding a period after your tag indicates that everything inside that tag is text and Jade stops treating the first word on each line as an HTML tag.

For example:

```
div
    p How are you?
        p.
        I'm fine thank you.
```

Jade can be used in the command line using the following command:

```
$ jade [ options ] [ dir | file ]
```

Jade Templates

Options in the command *$ jade [options] [dir | file]* can be:

1. -h, --help
2. -V, --version
3. -O, --obj <path|str>
4. -o, --out <dir>
5. -p, --path <path>
6. -P, --pretty
7. -c, --client
8. -n, --name <str>
9. -D, --no-debug
10. -w, --watch
11. -E, --extension <ext>
12. --name-after-file
13. --doctype <str>

Express

Express.js is a node.js web application framework designed to build single-page, multi-page, and hybrid web applications.

Core features of Express:

1. Allows to respond to HTTP requests by setting up middlewares
2. Sets up a routing table which can be used to perform various actions based on HTTP method and URL
3. Allows to render HTML pages dynamically based on arguments passed

Configure Express

Use the following command to install Express.js in your node.js application from the command line:

npm install express

Now, in your src/index.js file, use the following code to import Express.js, to create an instance of an Express application, and to start it as Express server:

```
import express from 'express';

const app = express();

app.listen(3000, () => {

console.log('App listening on port 3000');

});
```

Configure Express

Once you start your application on the command line with npm start, you should be able to see the output in the command line:

```
App listening on port 3000
```

Your Express server is up and running.

Postman Configuration

Postman is a powerful tool for performing integration testing with your API. It allows repeatable and reliable tests that can be automated and used in a variety of environments.

Use the following command to install Postman in your node.js application:

```
npm install postman-request
```

You can then export the module in your node.js application using the following code:

```
import express from 'express';
```

Postman Configuration

You then need to have Postman do a delivery if you want to send a message. You can do so using the following code:

```
postman.deliver('some-message', [arg1, arg2, argN]);
```

You need to then tell Postman what you want to receive using the following code:

```
postman.receive('some-message', function() { /* handle callback here */ });
```

Postman Configuration

You can also ask Postman to ignore history using the following code:

```
postman.receive('some-message', function() { /* handle callback here */ }, true);
```

You can also remove history for better memory management using the following code:

```
postman.dropMessages('some-message');
```

REST

- REST stands for Representational State Transfer and is used to access and manipulate data using several stateless operations.

- The operations represent an essential CRUD functionality (Create, Read, Update, and Delete).

- The REST API breaks down a functionality in order to create small modules. Each module addresses a specific part of the functionality. This approach provides more flexibility.

- The main functions used in a REST-based architecture are the following:

    1. GET: Provides read-only access to a resource
    2. PUT: Creates a new resource
    3. DELETE: Removes a resource
    4. POST: Updates an existing resource or creates a new resource

Features of REST

1. Stateless
2. Independent client and server
3. Uniform interface
4. Cacheable
5. Layered system
6. Code-on-demand

Building REST API

You can use the following steps to build a simple REST API:

Run the following command to initialize a new application:

```
npm init
```

Use the following command to install express-generator tool that generates a complete Express app:

```
npm install -g express-generator
```

Building REST API

Use the command below to install Express:

npm install express --save

Create a file named app.js and add the following code in it:

For example:

```
var express = require('express');
var app = express():
app.get("/url", (req, res, next) => {
res.json(["Tony","Michael","Rose","Eli","John"]
);
});
app.listen(3000, () => {
console.log("Server running on port 3000");
});
```

Building REST API

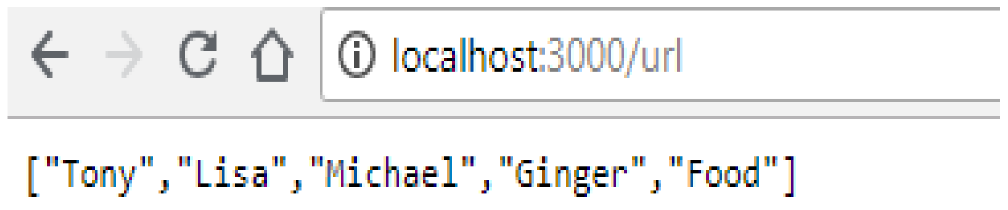
- This creates a simple GET request that returns a list of users. This will make the Express app use the url handle “/url” to trigger the callback that follows it.
- Use the following command to run your app:

```
node app.js
```

- You should get the following output:

```
Your app is running on port 3000
```

- To view our data, open your browser and enter http://localhost:3000/url.



```
["Tony","Lisa","Michael","Ginger","Food"]
```

JSON Data

JavaScript Object Notation or JSON, is one of the best ways to exchange information between applications.

JSON can be either represented as a hash of properties and values or as a list of values.

For example:

```
A JSON array:
{"John", "Elly", "Mike"}
A JSON object:
{"first": 1, "second": 2, "third": 3}
```

Handling JSON Files in Node.js

Step 1: Create a JSON file and save it as *user.json*.

```
{
“name”: “John”,
“email”: “john@yahoo.com”,
“id”: 2467
}
```

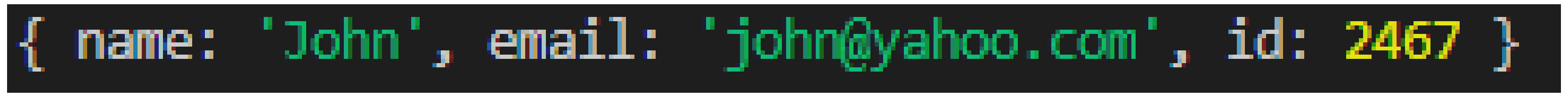
Step 2: Use readFileSync function to read the file synchronously.

```
'use strict';

const fs = require('fs');

let data = fs.readFileSync(‘user.json');
let user= JSON.parse(data);
console.log(user);
```

Output:



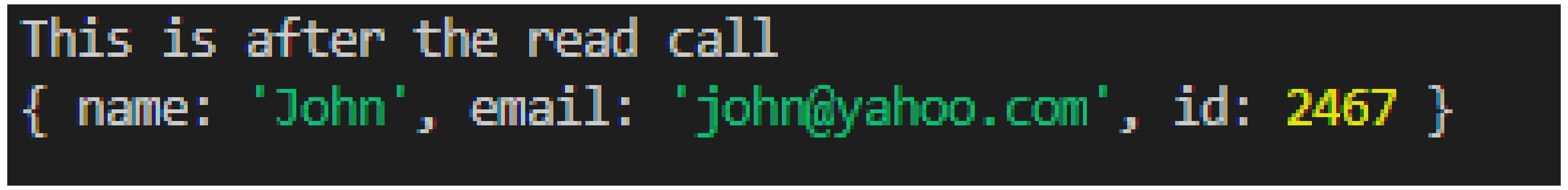
```
{ name: 'John', email: 'john@yahoo.com', id: 2467 }
```

Handling JSON Files in Node.js

Step 3: Use readFile function to read the file asynchronously.

```
'use strict';
const fs = require('fs');
fs.readFile('user.json', (err, data) => {
   if (err) throw err;
   let user = JSON.parse(data);
   console.log(user);
});
console.log('This is after the read call');
```

Output:



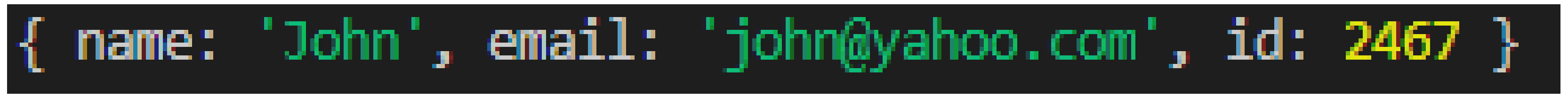
```
This is after the read call
{ name: 'John', email: 'john@yahoo.com', id: 2467 }
```

Handling JSON Files in Node.js

Step 4: The *require* method can also be used to read and parse JSON files.

```
'use strict';
let jsonData = require('./user.json');
console.log(jsonData);
```

Output:



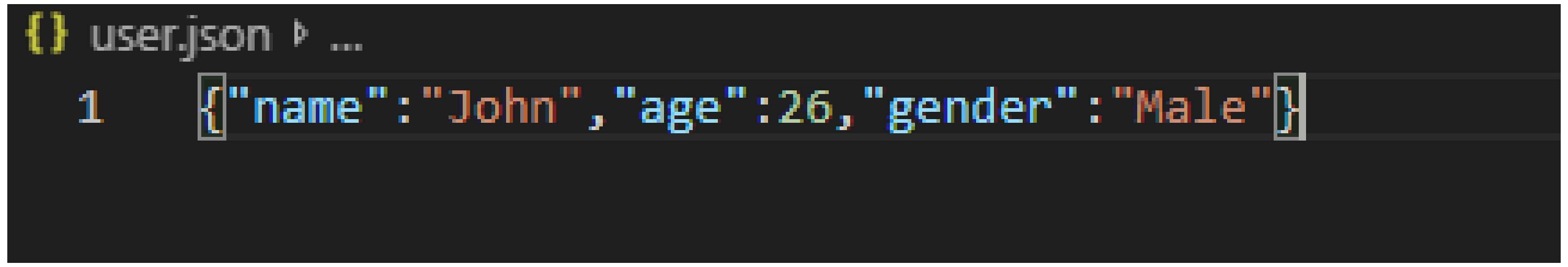
```
{ name: 'John', email: 'john@yahoo.com', id: 2467 }
```

Handling JSON Files in Node.js

Step 5: Use writeFileSync function to write to a file synchronously. It accepts three parameters: path of the file to write data to, the data to write, and an optional parameter. If the file does not exist, a new file is created.

```
'use strict';
const fs = require('fs');
let student = {
   name: 'John',
   age: 26,
   gender: 'Male'
};
let data = JSON.stringify(student);
fs.writeFileSync('user.json', data);
```

Output:



```
{} user.json ▸ ...
1    {"name":"John","age":26,"gender":"Male"}
```

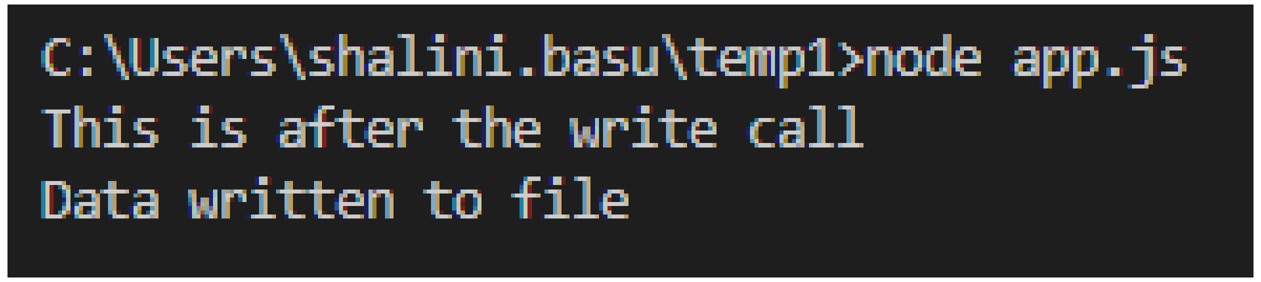
Handling JSON Files in Node.js

Step 6: Use writeFile function to write to a file asynchronously.

```
'use strict';
const fs = require('fs');
let student = {
   name: 'Elle',
   age: 26,
   gender: 'Female'
};
let data = JSON.stringify(student, null, 2);
fs.writeFile('user.json', data, (err) => {
   if (err) throw err;
   console.log('Data written to file');
});
console.log('This is after the write call');
```

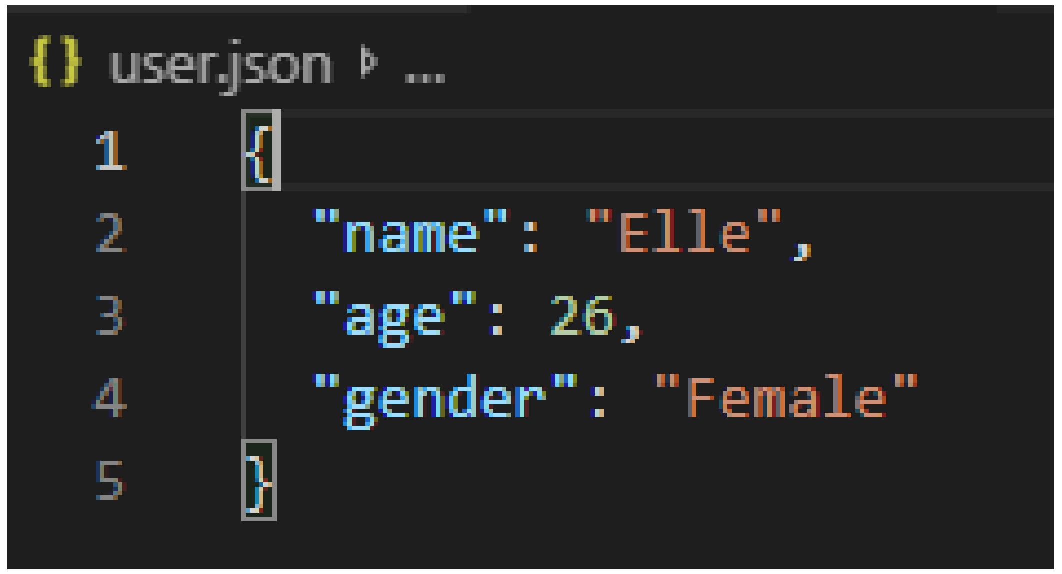
Handling JSON Files in Node.js

Output:



```
C:\Users\shalini.basu\temp1>node app.js
This is after the write call
Data written to file
```

user.json file:



```
{} user.json ▸ ...
{
  "name": "Elle",
  "age": 26,
  "gender": "Female"
}
```

Postman Configuration

Postman is a powerful tool for performing integration testing with your API. It allows repeatable and reliable tests that can be automated and used in a variety of environments.

Use the following command to install Postman in your node.js application:

```
npm install postman-request
```

You can then export the module in your node.js application using the following code:

```
import express from ‘express’;
```

Handle GET and Post Data

GET and POST are common HTTP methods used to build Rest APIs. They can be handled using the instance of Express.

Use the following code to handle GET request:

For example:

```
var express = require("express");
var app = express();
app.get('handle',function(request,response){
//code to perform particular action.
//To access GET variable use.
//request.var1, request.var2 etc
});
```

Handle GET and Post Data

- GET request can be cached. It remains in browser history. Therefore, it should not be used for sensitive data.
- You need to use a middleware layer called *body-parser* to handle POST request.

You can install body-parser using the following command:

```
npm install postman-request
```

Handle GET and Post Data

You can use the following code to import body-parser in your code and inform Express to use it as a middleware:

For example:

```
var express        =        require("express");
var bodyParser     =        require("body-parser");
var app            =        express();
//configure express to use body-parser as middleware
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
```

Handle GET and Post Data

You can then use *app.post* Express router to handle POST request.

For example:

```
app.post('handle',function(request,response){
var query1=request.body.var1;
var query2=request.body.var2;
});
```

Express.js Installation

Duration: 30 min.

Problem Statement:

You are given a project to install express.js creating express route handle.

Assisted Practice: Guidelines

Steps to install express.js:

1. Create a new node.js project
2. Install express
3. Create controller
4. Execute the program
5. Push code to GitHub repositories

FULL STACK

Routing

simplilearn

Routing

Routing determines how an application’s endpoints respond to a client request. Each route can have one or more handler functions that get executed when the route is matched to a particular endpoint.

Methods to use handle routing requests are:

- app.get(): Handles GET requests
- app.post(): Handles POST requests
- app.all(): Handles all HTTP methods
- app.use(): Specifies middleware as the callback function

Syntax: app.METHOD(PATH, HANDLER)

Routing

For GET request, use app.get() method:

For example:

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

app.get('/', function(req, res) {
    res.send('Welcome to Simplilearn')
})
```

For POST request, use app.post() method:

For example:

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

app.post('/', function(req, res) {
    res.send('Welcome to Simplilearn')
})
```

Routing

For handling all HTTP methods (GET, POST, PUT, DELETE), use app.all() method:

For example:

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

app.all('/', function(req, res) {
    console.log('Welcome to Simplilearn')
    next()   // Pass the control to the next handler
})
```

Route Path

Route paths, in combination with a request method, define the endpoints at which requests can be made. Route paths can be strings, string patterns, or regular expressions.

Example: This route path will match requests to the root route

For example:

```
app.get('/', function (req, res) {
  res.send('root')
})
```

Route Parameters

Route parameters are named URL segments used to capture the values specified at their position in the URL.

```
Route path: /users/:userId/books/:bookId
Request URL: http://localhost:3000/users/34/books/8989
req.params: { "userId": "34", "bookId": "8989" }
```

The captured values are populated in the *req.params* object. The name of the route parameter is specified in the path.

Route Parameters

Specify the route parameters in the path of the route to define routes with route parameters.

```
app.get('/users/:userId/books/:bookId', function (req, res) {
  res.send(req.params)
})
```

Note

The name of route parameters must be made up of word characters [A-Za-z0-9_].

Route Handlers

To handle a request, we can provide multiple callback functions that behave like middleware. The only exception is that these callbacks might invoke next route to bypass the remaining route callbacks. Route handlers can be in the form of a function, an array of functions, or combinations of both.

A single callback function can handle a route.

For example:

```
app.get('/example/a', function (req, res) {
  res.send('Hi Jack!')
})
```

Route Handlers

More than one callback function can also handle a route (Make sure you specify the next object.)

For example:

```
app.get('/example/b', function (req, res, next) {
  console.log('the response will be sent by the next function ...')
  next()
}, function (req, res) {
  res.send(‘Hi John!')
})
```

Express Router

Use the express.Router class to create modular, mountable route handlers. A router instance is a complete middleware and routing system and referred to as a *mini-app*.

For example:

```
var express = require('express')
var router = express.Router()
```

FULL STACK

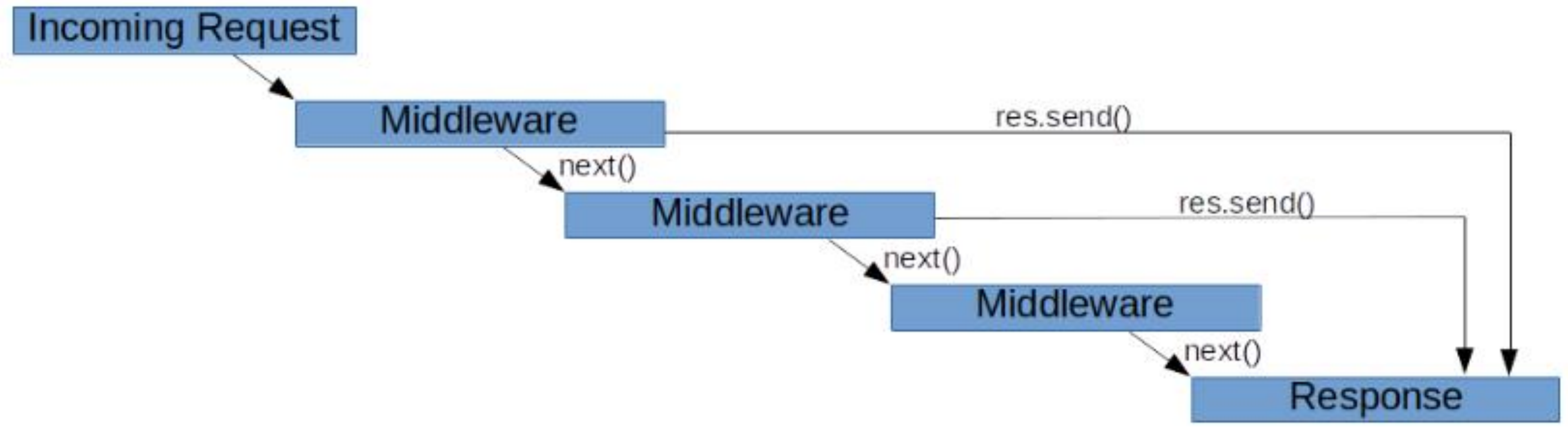
Middleware

simplilearn

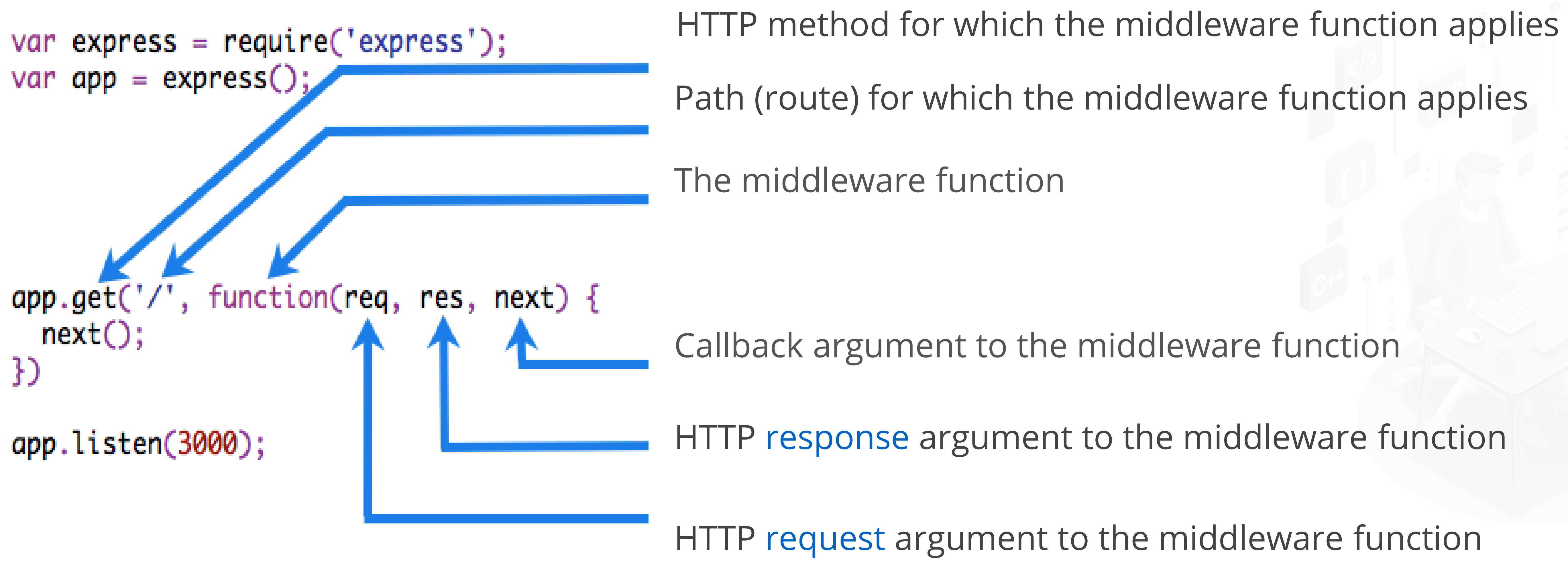
Middleware

Middleware is a function that has access to the request object (req), response object (res), and the next middleware function of the application's *request-response* lifecycle.

The next middleware function is denoted by a variable named next.



Middleware



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

app.get('/', function(req, res, next) {
  next();
})

app.listen(3000);
```

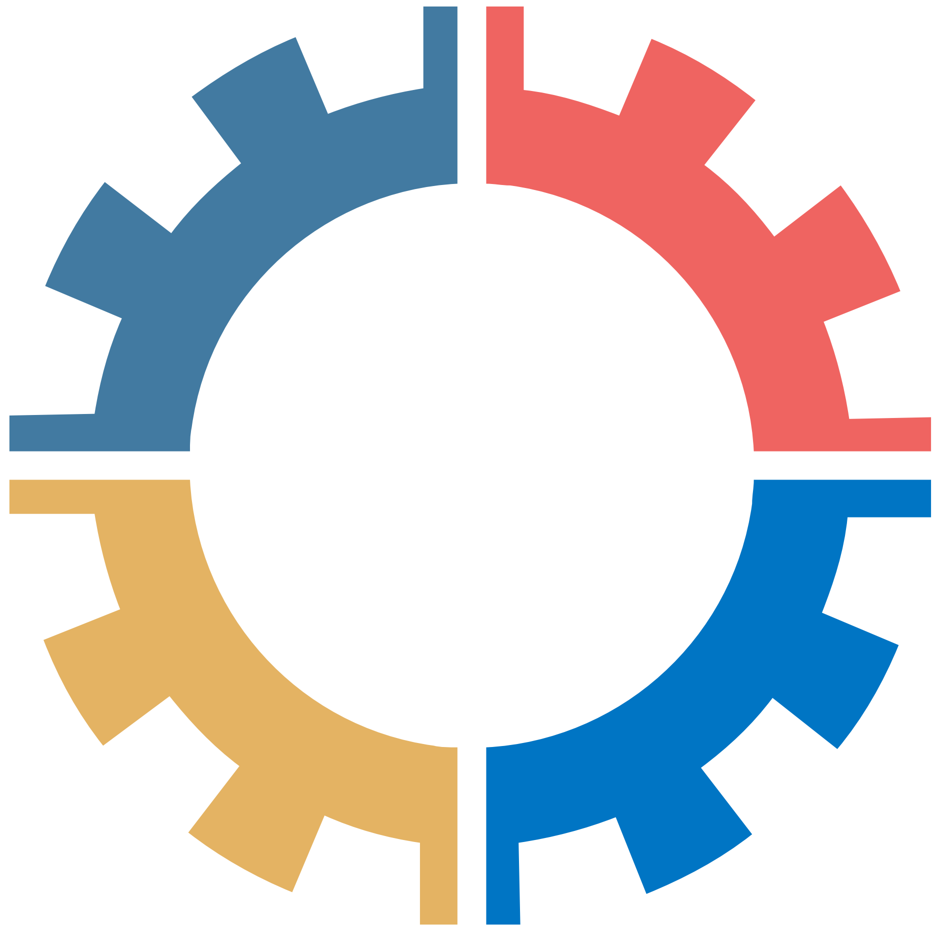
HTTP method for which the middleware function applies
Path (route) for which the middleware function applies
The middleware function
Callback argument to the middleware function
HTTP response argument to the middleware function
HTTP request argument to the middleware function

Middleware

Middleware functions can perform the following tasks:

Execute any code

Make changes to the request and the response objects



End the request-response cycle

Call the next middleware function in the stack

Note

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Else, the request will be left hanging.

Types of Middleware

The types of middleware in express application are:

- Application-level middleware
- Router-level middleware
- Error-handling middleware
- Built-in middleware
- Third-party middleware

Types of Middleware

- **Application-level middleware**
- Router-level middleware
- Error-handling middleware
- Built-in middleware
- Third-party middleware

Application-level middleware is bound to an instance of the app object express using the app.use() and app.METHOD() functions.

For example:

```
var app = express()

app.use(function (req, res, next) {
  console.log('Time:', Date.now())
  next()
})
```

Types of Middleware

Application-level middleware

Router-level middleware

Error-handling middleware

Built-in middleware

Third-party middleware

Router-level middleware works in the same way as application-level middleware, except it is bound to an instance of express.Router().

For example:

```
var router = express.Router()
```

Types of Middleware

- Application-level middleware
- Router-level middleware
- **Error-handling middleware**
- Built-in middleware
- Third-party middleware

In error-handling middleware functions, there are four arguments instead of three, specifically with the signature (err, req, res, next).

For example:

```
pp.use(function (err, req, res, next) {
  console.error(err.stack)
  res.status(500).send('Something broke!')
})
```

Types of Middleware

- Application-level middleware
- Router-level middleware
- Error-handling middleware
- **Built-in middleware**
- Third-party middleware

The built-in middleware function in express.js is express.static. This function is based on serve-static, and is responsible for serving static assets such as HTML files and images.

For example:

```
The function signature is:
express.static(root,[options])
```

Types of Middleware

- Application-level middleware
- Router-level middleware
- Error-handling middleware
- Built-in middleware
- **Third-party middleware**

The third-party middleware function is used to add functionality to express apps. Install the node.js module for the required functionality, then load it in your app at the application level or at the router level.

For example:

```
$ npm install cookie-parser

var express = require('express')
var app = express()
var cookieParser = require('cookie-parser')

// load the cookie-parsing middleware
app.use(cookieParser())
```

CRUD Operations and Middleware

Duration: 90 min.

Problem Statement:

You are given a project to demonstrate CRUD operations and middleware.

Assisted Practice: Guidelines

Steps to demonstrate CRUD operations:

1. Create a new node.js project
2. Install express-handlebars
3. Create controller
4. Create views
5. Create model
6. Push code to GitHub repositories

FULL STACK

Socket.IO with Node.js

simplilearn

Socket.IO with Node.JS

Socket.IO is a library that enables real-time, bidirectional and event-based communication between a browser and a server.

It consists of:

- node.js server
- A Javascript-client library for the browser

Socket.IO with Node.JS

Main features of Socket.IO:

- Reliability
- Auto-reconnection support
- Disconnection detection
- Binary support
- Multiplexing support
- Room support

Socket.IO with Node.JS

Use the following command to install Socket.IO server:

```
npm install socket.io
```

Use the following command to install a standalone build of the client which is exposed by the server at */socket.io/socket.io.js*.

```
npm install socket.io-client
```

Socket.IO with Node.JS

The following snippet attaches Socket.IO to a node.js HTTP server:

For example:

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

const io = require('socket.io')(server);

io.on('connection', client => {

client.on('event', data => { /* … */ });

client.on('disconnect', () => { /* … */ });

});

server.listen(3000);
```

Socket.IO with Node.JS

Use the following code to work with Socket.IO in conjunction with express.

For example:

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

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

const io = require('socket.io')(server);

io.on('connection', () => { /* … */ });

server.listen(3000);
```

Socket.IO with Node.JS

Using express as http server:

For example:

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

var io = require('socket.io').listen(app);

io.sockets.on('connection', function (socket) {

console.log(‘User is connected!’);

});app.listen(8080);
```

Socket.IO with Node.JS

Using node.js as http server:

For example:

```
var app =
require('http').createServer(callback);
var io =
require('socket.io').listen(app);
app.listen(8080);
```

Socket.IO with Node.JS

Client side JS code:

For example:

```
<script src='/socket.io/socket.io.js'></script>

<script>

var socket = io.connect();

socket.on('connect', function () {

console.log('User is connected on client APP!');

});

</script>
```

Socket IO Integration

Duration: 150 min.

Problem Statement:

You are given a project to integrate **socket.io** in the node JS application.

Assisted Practice: Guidelines

Steps to show messages in app:

1. Create a new node.js project
2. Install express-handlebars and **socket.io**
3. Create **app.js**
4. Create **index.html**
5. Push code to GitHub repositories

Key Takeaways

- Model-View-Controller is a software design pattern referred for developing user interfaces which logically divides the related program into three connected elements.
- Jade is a templating engine used to make templates at server-side.
- Express is a web application framework used for building web applications and APIs.
- REST defined the set of constraints to be used for web services.
- GET is used to request data from a specific resource and POST is used to send data to the server to create a resource.
- Socket.IO is a JavaScript library which enables real time, bidirectional communication between client and server.

Chatting with Socket IO

Problem Statement: **Duration: 60 min.**

As a Full Stack Developer, you have to create a chat application using **socket.io** and node.js.

LESSON-END PROJECT