Express and Socket.IO

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 MEAN Stack Developer

In this sprint, he has to develop a basic chat application using Express.js and Socket.io which can be used for internal communication between the 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

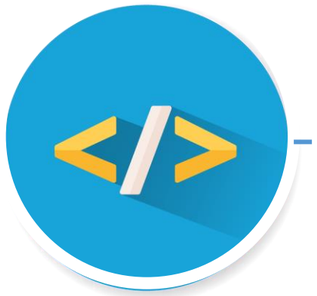
By the end of this lesson, you will be able to:

- 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

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

Routing

Routing determines how an application's endpoints respond to a client request.

Each route can have one or more handler functions which get executed when the route is matched.

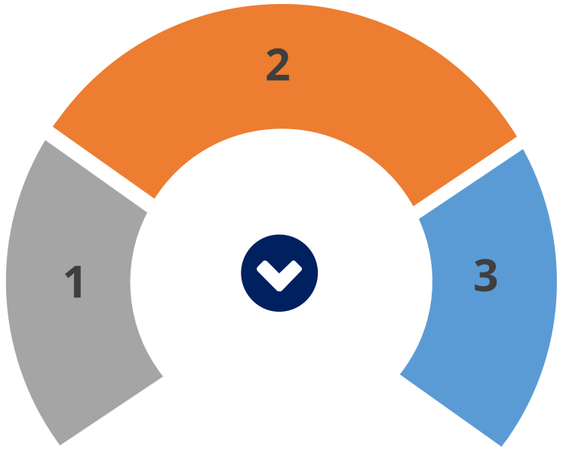
Methods to use handle routing requests are:

- app.get(): Used to handle GET requests
- app.post(): Used to handle POST requests
- app.all(): Used to handle all HTTP methods
- app.use(): Used to specify middleware as the callback function

Syntax:
app.METHOD(PATH, HANDLER)

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

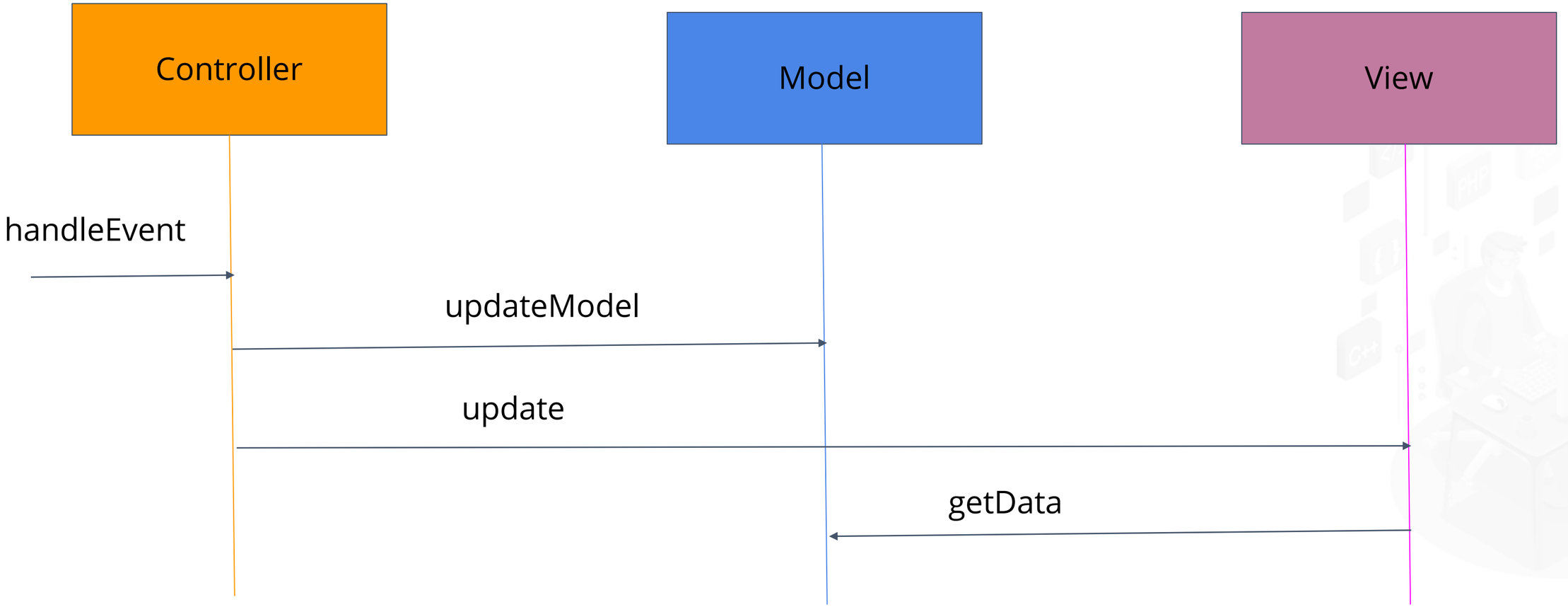
- Model-View-Controller is a design pattern used for software projects.

- 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

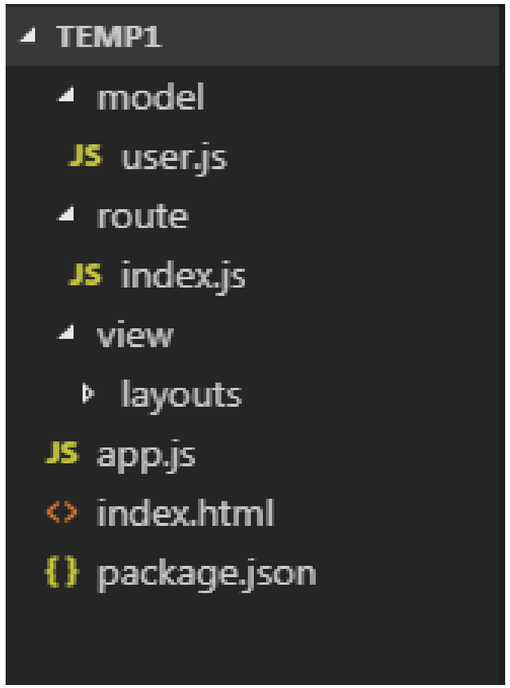
Workflow of model-view-controller:



Model-View-Controller

An example of a NodeJS 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:



Jade Templates

- Jade is a template engine used for server-side templating in NodeJS.
- Example:

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

- 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

- You need to 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!
   ```

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.

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:

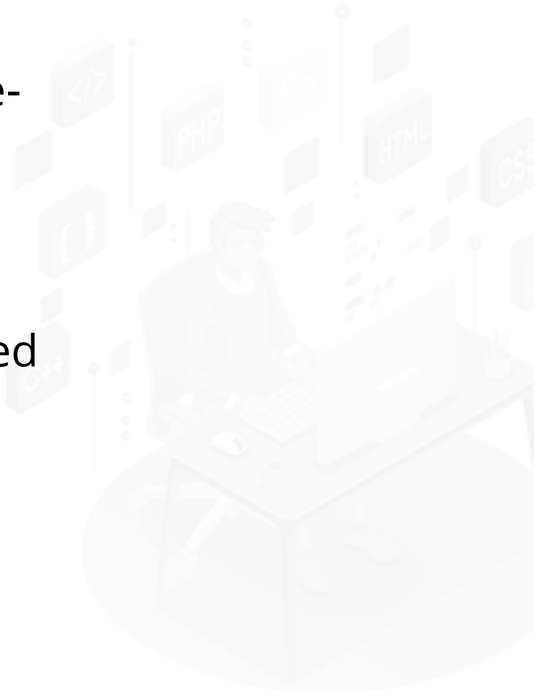
1. -h, --help
2. -V, --version
3. -O, --obj <path|str>
4. -o, --out <dir>
5. -p, --path <path>
6. -P, --pretty
7. -c, --client
8. -n, --name <str>
9. -D, --no-debug
10. -w, --watch
11. -E, --extension <ext>
12. --name-after-file
13. --doctype <str>

Express

- Express.js is a NodeJS 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

- You can use the following command to install Express.js in your NodeJS 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.

- You can use the following command to install Postman in your NodeJS application:

  npm install postman-request

- You can then export the module in your NodeJS 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 */ });*

- 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

- Use the command below to install Express:
  npm install express --save

- Create a file named app.js and add the following code in it:
  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");
  });

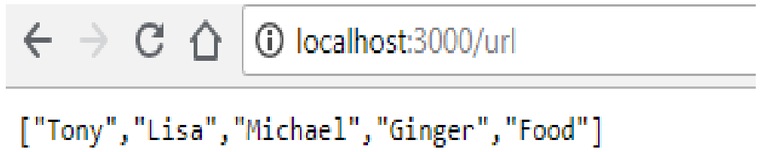
Building REST API

- 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.



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 NodeJS

- 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 NodeJS

- 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 NodeJS

- 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);
```

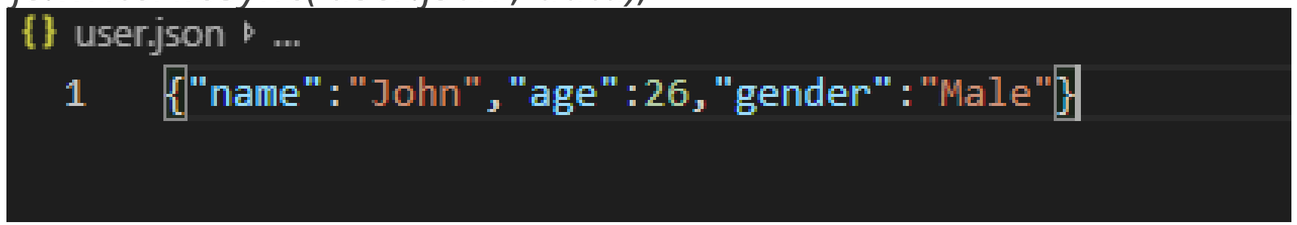
Output:

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

Handling JSON Files in NodeJS

- 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);
```



Handling JSON Files in NodeJS

- 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 NodeJS

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"
}
```

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.

- You can use the following code to handle GET request:

```
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
});
```

- GET request can be cached. It remains in browser history. Therefore, it should not be used for sensitive data.

Handle GET and Post 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 body-parser

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

```
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.

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

CRUD Operations and Middleware

Duration: 90 min.

Problem Statement:

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

ASSISTED PRACTICE

Assisted Practice: Guidelines to Demonstrate CRUD Operations

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

Socket.IO with NodeJS

Socket.IO with NodeJS

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

1. NodeJS server
2. A Javascript-client library for the browser

- Its main features are:

1. Reliability
2. Auto-reconnection support
3. Disconnection detection
4. Binary support
5. Multiplexing support
6. Room support

Socket.IO with NodeJS

- You can use the following command to install Socket.IO server:

  npm install socket.io

- You can use the following command to install a standalone build of the client which is exposed by default by the server at */socket.io/socket.io.js*.

  npm install socket.io-client

- The following snippet attaches Socket.IO to a NodeJS HTTP server listening on port 8080:

```
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 NodeJS

- You can use the following code to work with Socket.IO in conjunction with Express:

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

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

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

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

server.listen(3000);
```

Socket IO Integration

Duration: 150 min.

Problem Statement:

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

ASSISTED PRACTICE

Assisted Practice: Guidelines to Show Messages in App

1. Create a new NodeJS 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