PAGE NO.:

Name :- Khokhariya Vishwa B.

Roll NO :- 023

Subject :- 701 Date :- 29-08-23

Q-1 Write an assignment on express Framework with following topics:

① Introduction:-

- Express is a minimalist web framework. It's a layer built on top of the Node JS and provide you with a robust set of feature that make it easy to manage servers and routes.

- It is essentially acts as a series of middleware function calls, each of which performs a specific function.

- Express is not opinionated, which is why you can use it in various ways. It does not, for example, impose a specific design pattern or folder structure.

- Without Express JS, you must write your own code to create a routing component, which is time-consuming and labor-intensive.

- Express Js Provides programmers with simplicity, Flexibility, efficiency, minimalism and scalability.

- Express JS Performs all executions, extremely quickly thanks to the event loop, which eliminates any inefficiency.

- Express JS was designed to make it simple to create APIs and web application.

- It cuts coding time in half while still producing web content.

- mobile application are more efficient.

- Another reason to use Express is that it is written in Java script, a simple language, even if you have no Prior knowledge of any language, Express JS enables a large number of new developers to enter the field of web development.

PAGE NO.:

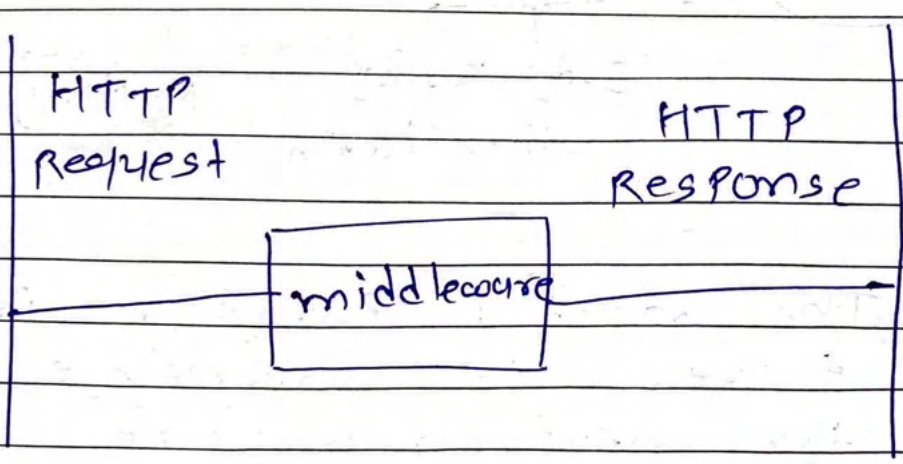
(2) middleware

- Express.js is a routing and middleware framework for handling the different routing of the webpages and it work's between the request and response cycle.

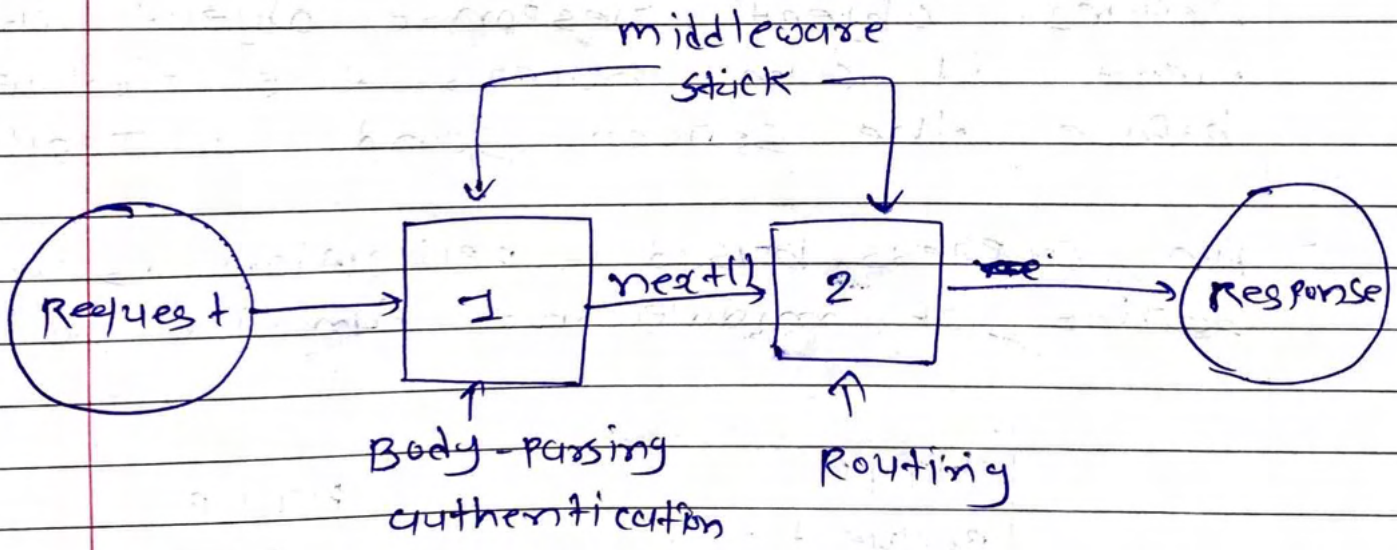
- middleware gets executed after the server receives the request and before the controller action send the response.

- middleware has the access to the request object, response object, and next, it can process the request before the server send a response

- An Express-based application is a series of middleware function calls.



- middleware can process request object multiple times before the server works for that response.
- middleware can be chained from one to another, Hence creating a chain of function that are executing in order.
- The last function sends the response back to the browser, so before send the response back to the browser the different middleware process.



- The next() function in the express is responsible for calling the next middleware function if there is one.

PAGE NO.:

③ Express validator.

- Express-validator is a set of Express.js middlewares that wraps the extensive collection of validators and sanitizers offered by validator.js

- It allows you to combine them in many ways so that you can validate and sanitize your express request, and offers tools to determine if the request is valid or not, which routes was matched according to your validator and so on.

- Express-validator might work with libraries that aren't express.js.

- The main requirement is that HTTP server library you're using models its HTTP request object similarly to express.js, and contain these properties:

- req.body: the body of the HTTP request. can be any value, however objects, arrays and other JavaScript primitives work better.

- req.cookies :- The cookie header parsed as an object from cookie name to its value.

- req.headers : The headers sent along with the HTTP request.

- req.query : The portion after the ? in the HTTP request's path, parsed as an object from query parameter name to value.

⇒ Installation:-

- express-validator is on the npm registry ! install it using your Node.js package managers:

- npm install express-validator.

- pnpm add express-validator.

(4) Template engine

- A template engine enables you to use static template files in your application.

- At runtime, the template engine replace variables in a template engine replace variable in a template file with actual values. and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page.

- Some popular template engines that work with express are pug, mustache and EJS. The Express application generator uses Jade as its default, but it also supports several others.

- To render template files, set the following application setting properties. Set in app.js in the default app created by the generator.

- views, the directory where the template files are located. Eg: app.set('views', './views'). This defaults to the views directory in the application root directory.

- view engine, the template engine to use. For example, to use the pug template engine:
app.set('view engine', 'pug').

- Then install the corresponding template engine npm package, for example to install pug:

$npm install pug --save.

- After the view engine is set, you don't have to specify the engine or loads the template engine module.

app.set('view engine', 'pug')

- app.get('/', (req, res) =>
{
    res.render('index', { title: 'Hey', message: 'Hello there!' })
});

(5) REST API

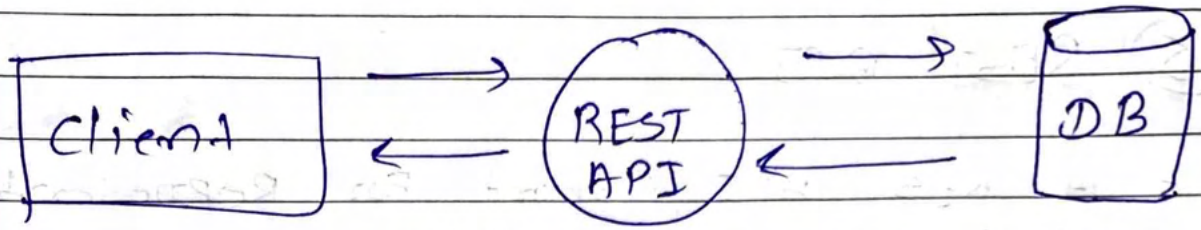
- A REST API (short for Representational State Transfer API) is an interface that allows you to access and manipulate data over the internet.

- It's called "representational" API because it's designed to represent the underlying data in a way that's easy for clients to understand and use.

- REST APIs are often used to provide access to data stored in a database, or to enable communication between different system and application.

- we will set up simple route that will handle GET request to the root path of your API

```
app.get('/', (req, res) =>
{
    res.send('Hello world!');
});
```



- HTTP Request are simply message that are sent by the client to do some task on the server.

- GET - Get command is used to request data from the server, but mainly this method is used to read data.

- PATCH :- This command is used to update, change or replace the data.

- POST :- The Post method is used to create new or to edit already existing data.

- Delete :- This delete command is used to delete the data completely from the server

example

```
const app = require('express');
const PORT = 4000;
```

```
app.get('/simtilearn', (req, res)=>
{
    res.send("youtube or website")
});
```

(6) API security best practices

- API security is critical to protect sensitive data and ensure the integrity of your application and services.

(1) Authentication:-

- Implement strong authentication mechanism such as API keys, OAuth 2.0 or JWT.

2. Authorization:-

- Implement fine-grained access controls and role-based authorization to restrict access to only authorized users and action.
- Use the principle of least privilege, granting users of services only the permissions they need.

PAGE NO.:

3. HTTPS (TLS/SSL):

- Always use HTTPS to encrypt data in transit. Preventing eavesdropping and data interception.

- Keep SSL/TLS certificates up to date and use strong encryption protocols and cipher suites.

4. Input validation:-

- Validate and sanitize all user inputs and API request to prevent common security vulnerabilities such as SQL injection, cross-site scripting and cross-site Request Forgery.

5. Rate Limiting:

- Implement rate limiting to prevent abuse of your API by limiting the number of request a client can make in a given time period.

- consider using a web application Firewall (WAF) to help with rate limiting and protection against attacks.

6. Error handling:

- Avoid exposing sensitive information in error message. Use generic error message and log detailed errors internally for debugging.

7. Security Headers:

- Use security headers like content security policy (CSP), X-Content-Type-Option, X-Frame-Option. and X-XSS-Protection to mitigate common web vulnerabilities.

(7) Types of tokens and their usage.

- Token are used in various context in computer science and security to represent and authorize access to resource or services.

1. Access Token:-

- Usage: Access tokens are commonly used in conjuction with access token. They are long-lived tokens that can be used in obtain new access token.

2. Refresh Tokens:

- Refresh tokens are often used in conjuction with access token. They are long-lived tokens that can be used to obtain new access tokens once the original access token expires without requiring the user to reauthenticate.

3. JWT (JSON Web Token's):-

- JWTs are compact, self-contained tokens that can carry information about user, client or other entities.

- They are often used for user authentication and authorization, session management and passing claim between services in a stateless manners.

4. CSRF Tokens (Cross-site Request Forgery)

- CSRF tokens are used to prevent cross-site request forgery attacks. They are included in web forms and compared to token stored on the server to ensure that the request originates from a legitimate source.

PAGE NO.:

5. Session Tokens:

- Session tokens are used to maintain user sessions after authentication. They are often used to associate a user's identity with a session on the server.

6. API Tokens:-

- API tokens are used to authenticate and authorize access to APIs. They are typically sent in HTTP headers or as query parameter in API request.

7. ID Tokens:-

- ID Tokens are used in identity authentication protocols like openID Connect. They carry information about the authenticated user and are used to verify their identity.

8. Security Tokens

9. Bearer Tokens.

Q- MongoDB database with CRUD API

1. Setting up the Development Env:

- Install Node.js and mongoDB on your system if not already installed.
- create a new project directory and initialize it as a Node.js project using 'npm init'.

2. Creating a mongoDB Database.

3. setting up Express.js

4. Implement CRUD operation.

5. Testing API

6. error handling

7. Documentation

8. Security and Authentication

* example:

- create an app.js files

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

mongoose.connect('mongodb://localhost:
        27017/taskmanager',
{
  useNewUrlParser: true;
  useUnifiedTopology: true;
});

const taskSchema = new mongoose.schema(
{
  title: String,
  description: String,
  completed: Boolean,
});

const Task = mongoose.model('Task',
                    taskSchema);
app.use(bodyParser.json());

app.Post('/tasks', asyn((req, res) => {
  try {
    const task = new task(req.body);
    await task.save();
    res.status(201).json(task);
  }
  catch(error){
      };
```

```
app.get('/tasks', async(req, res) => {
  const tasks = await Task.find({});
  res.json(tasks);
});

app.get('/tasks/:id', async(req,res)=>{
  try{
    const task = await Task.findByID(
                          req.params.id);
  if(!task){
    return res.status(404);
  } catch(error){
    res.status(500).json({msg});
  } });

app.Put('/task/:id', async(req,res) => {

  const task = await Task.findByIdAnd
                        Update(req.params.id,
{ new: true,
    runValidators: true});
  res.json(task);
});

app.delete('/task/:id', async(req,res)=>
{
  const task = await Task.findById AndDELETE
                          (req.params.id);
  res.json(task);
});
app.listen(3000, () => {//msg} ));
```

PAGE NO.:

Q- Mongoose schema, model, Document and API for CRUD, search in Express app.

① Setting up environment:

npm install express mongoose body-parser

② Create schema and model

```
Const mongoose = require ('mongoose');

Const taskschema = new mongoose.schema(
{
    title : string,
    description : String,
    completed : Boolean,
});

const Task = mongoose.model ('Task',
                              taskschema);
module.exports = Task;
```

③ Create app.JS.

```
Const express = require ('express');
Const mongoose = require ('mongoose');
Const Task = require ('./models/task');
```

```
const app = express();
const port = process.env.PORT || 3000;

// Connection code.

app.listen(port(), () => {
    console.log('server is running
    on port ${PORT}');
});
```

④ API S.

```
// Retrive all tasks.

app.get('/tasks', async (req, res) =>
{
    const tasks = await Task.Find({});
    res.json(tasks);
});

// update a task.

app.put('/tasks/:id', async(req,res) =>
{
const task = await Task.FindBYIDAndUpdate
        (req.params.id, req.body);

res.json(task);

});
```

// Delete a tasks by ID

```
app.delete('/tasks/:id', async(req,res)
                                    =>{
  const task = await Task.findByIDAnddelete
                    (req.params.id);
  res.json(task);
});
```

// search for task

```
app.get('/search', async(req, res)=>
{
  const {query} = req.query;
  const tasks = await Task.find({

  $or: [
  {title: {$regex: query, $options: 'i'}},
  {description: {$regex: query, $options: 'i'}
  ],
});
res.json(task);
});
```

Q- Mongoose schema datatypes, Validation.

- In mongoose you can define the schema for your mongoDB document specifying data types and validation rules for each Feild.

```
const UserSchema = new mongoose.
                    schema ( {
    username : String,
});
```

- const ProductSchema = new mongoose.
```
                    schema ( {
    Price : Number,
});
```

- const TodoSchema = new mongoose.schema(
```
  {
    completed : Boolean,
});
```

```
const EventSchema = new mongoose.schema(
  {
    date : Date,
});

const PutSchema = new mongoose.schema(
  {
    tags : [String],
});
```

- const addressSchema = new mongoose.
  Schema ({
  street : string,
  city : string,
});

- const userSchema = new mongoose.Schema(
  { name: string,
  address : addressSchema,
  });

- const documentSchema = new mongoose.
  Schema ({
  data: mongoose.Schema.Types.mixed,
  });

= Validation :-

username : {

  type : string,
  required : true,
  }

title : {
  type : string,
  minLength : 5,
  maxLength : 100,
}

```
code: {
    type: String,
    match: /^[A-Z]{3}-\d{3}$/,
},

age: {
    type: Number
    validate: {
        validator: Function (value) {
            return value >= 18 && value <= 99;
        }
        message: 'Age must be btw
                  18 and 99';
    }
}
```

Q- Schema Refering and querying.

```
const mongoose = require('mongose');

const authorschema = new mongose.
                     schema({
    name : string,
    books : [{ type : mongoose.schema.
               Tyes.objectid,
               ref : 'Book' }],
});

const bookschema = new mongoose.Schema({
    Title : string,
    author : { type : mongoose.schema.Types,
               objectId,
               ref: 'Author' },
});

const Author = mongoose.model('Author',
                authorschema);

cent Book = mongoose.model('Book', book.
                Schema);
const author = new Author({ name :
                'J.K. Rowling' });

const book = new Book({ title :
    'Harry Potter and the sorever \'s
                stone
     author : author id });
```

```
author.books.push(book);
author.save();
book.save();

Author.findOne({name: 'J.K. Rowling'})
    .populate('books')
    .exec((err, author) => {
        if (err) {
            console.error(err);
        }
        else {
            console.log('Books by', author.name
        });
    }
});
```