Name : Savaliya Heli Pareshbhai

Semester : 7th

Roll No : 71

Division : A

Subject : Application Development using Full stack

ASSIGNMENT : 2

Topic Name : Express

Date : 02/09/2023

Date.
Page.

Q=1. Express Framework.

[1] Introduction:

Express is fast, flexible, and minimalist web application framework for Node.js. It provides a robust set of features for building web and mobile applications, making the process of creating APIs, handling routes, & and managing middleware easier and more efficient. With its lightweight design, Express allows developers to quickly create scalable and modular applications while providing the freedom to integrate various libraries and tools to suit their specific needs.

[2] Middleware

Middleware in the Express framework refers to a series of functions that are executed in the order they are defined during the request-response lifecycle.

→ It sits between the initial request and the final response.

→ middleware functions have access to the request, response objects as well as 'next' By calling 'next()' a middleware function passes control to the next middleware in line.

→ middleware can be used globally for all routes or can be applied to specific routes.

Example:

```
app.use((req, res, next) => {
    console.log('Middleware is called');
    next();
});
```

[3] Express validator

Express validator is a middleware library for the Express framework in Node.js that simplifies the process of validating and sanitizing user input, such form data or API parameters, to ensure that they meet certain criteria.

→ We need to install the Express validator package.

```
npm install express-validator
```

→ We can import and use Express validator

```
const express = require('express');
const {check, validationResult} = require('express-validator');
```

→ Using 'validationResult(req)', we can check if any validation errors occured.

[4] Template Engine

A template engine is a tool or library used to generate dynamic HTML content by combining templates.

→ Template engines help developers separate the presentation layer of a web application from the underlying logic, making it easier to manage and update the user interface.

→ Some popular template engines for web development is EJS, Mustache, Handlebars, Jinja2 etc.

★ Example:

```
const express = require('express');
const app = express();
app.set('view engine', 'ejs');
app.set('views', __dirname + '/views');
app.get('/', (req, res) => {
    res.render('index');
});
```

Date.
Page.

[5] REST API

→ API is an architectural style for designing networked applications. It defines a set of constraints that provide a uniform way to structure and interact with web services.

→ RESTful APIs are widely used to create web services that allow different software applications to communicate and exchange data over the internet.

→ REST APIs use HTTP methods to perform operations on resources.

→ The most common HTTP methods used in RESTful APIs are 'GET', 'POST', 'PUT' and 'DELETE'.

A Example:

```
const express = require('express');
const app = express();
const port = 3000;

let users = [];
app.use(express.json());
app.get('/users', (req, res) => {
    res.json(users);
});
```

Date.
Page.

[6] API security best practices.

API security is crucial to protect sensitive data, prevent unauthorized access, and ensure the integrity of your applications and services.

* best practices for API security.

(i) Authentication and Authorization:
→ Use strong authentication mechanisms like API keys, tokens (JWT or OAuth), or client certificates.
→ Implement role-based access control.

(ii) HTTPS Encryption.
→ Always use HTTPS to encrypt data transmitted between clients and the API server.

(iii) Input validation.
→ Validate and sanitize user inputs to prevent injection attacks.

[7] Types of tokens and their usage

Tokens are used in various authentication and authorization scenarios to verify the identity of a user.

* Types of token.

[1] Access Tokens:

→ Usage: Access tokens are commonly used in OAuth 2.0 authentication. They grant a client application access to specific resources on behalf of a user.

→ How they work: The user authenticates with an identity provider.

[2] Refresh Tokens:

→ Refresh tokens are also used in OAuth 2.0 flows.

→ They are used to obtain new access tokens when the original access token expires without requiring the user to re-authenticate.

[3] JWT (JSON Web Tokens):

→ JWTs are a compact way of representing claims between two parties.

→ They are often used for authentication and information exchange.

[4] Session Tokens:

→ Session tokens are often used for web applications to maintain user sessions. They help identify a user without storing sensitive information on the client side.

[5] API keys

→ API keys are used to identify a client application accessing an API. They are often used for simple access control to APIs.

[6] Bearer tokens.

→ Bearer tokens are a type of access token used for authentication.

[7] Identity tokens.

→ Identity tokens are used in openID Connect, an identity layer built on top of Oauth 2.0.

Q-2. MongoDB database with Crud operation API.

(1.) Install the Package.

npm install express mongodb.

(2) 'models/todo.js'

```
const mongoose = require('mongoose');
const todoschema = new mongoose.schema({
title : {type: String, required: true},
description : String,
completed : {type: Boolean, default: false}
});

module.exports = mongoose.model('Todo',
  todoschema);
```

(3) 'routes/todo.js'

```
const express = require('express');
const router = express.Router();
const Todo = require('../models/todo');

router.post('/', async (req, res) => {
  try {
  const todo = new Todo(req.body);
  await todo.save();
  res.status(201).json(todo);
} catch (error) {
  res.status(400).json({message: error.
                        message});
```

```
    }
});

router.get('/', async (req, res) => {
    try {
    const todos = await Todo.find();
    res.json(todos);
    } catch (error) {
    res.status(500).json({ message: error.
      message });
    }
});

module.exports = router;
```

(4) app.js

```
const express = require('express');
const mongoose = require('mongoose');
const todoRouter = require('./routes/todos');
const app = express();
const port = 3000;

mongoose.connect('mongodb://localhost:27017/
 todos', {});

app.use(express.json());
app.use('/todos', todoRouter);

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

Date
Page

P-3. Mongoose Schema, Model, Document and API for CRUD, Search in Express app.

(1) Install dependencies.
first install

npm init

(2) create server
app.js

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

const app = express();
const PORT = process.env.PORT || 3000;
app.use(bodyParser.json());
app.listen(PORT, () => {
    console.log(`server is running
        on port ${PORT}`);
});
```

(3) Mongoose Schema and Model:

Define a mongoose schema and model for your data in a separate file.
- 'models/item.js'

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

```
const itemSchema = new mongoose.
  schema ({
  name : String,
  description : String,
  price : Number,
});

const Item = mongoose.model ('Item',
 itemSchema);

modules.exports = Item;
```

[4] CRUD Operations:

'app.js'

```
const express= require ('express');
const bodyParser= require ('body-parser');
const mongoose= require ('mongoose');
const Item = require ('./models/item');

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

app.use (bodyParser.json());

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

  then (() => {
    console.log ('connected to mongoDB');
  })
```

```
.catch(err => {
    console.error("Error in Connected to mongoDB:", err);
});

app.post('/items', async (req, res) => {
    try {
        const newItem = new Item(req.body);
        const SavedItem = await newItem.save();
        res.json(SavedItem);
    } catch(err) {
        res.status(500).json({ error:
            err.message });
    }
});

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

★ Search :-

Implement searching by adding a route for searching items based on a query parameter.

```
app.get('/items/search', async (req,
    res) => {
    const query = req.query.q;
    try {
        const FoundItems = await Item.find
```

```
({ $text: { $search: query } });
    res.json(foundItems);
} catch (err) {
    res.status(500).json({ error: err
message });
}
});
```

Date
Page

Q.4 Mongoose Schema data types, validation etc.

Mongoose is ODM (Object Data modeling) library for MongoDB in Nodejs.

→ It allows to define schemas for data, including specifying data types and validation rules.

→ Here are some mongoose schema data types.

★ Data types:

[1] String:
- Represents a string.
⇒ example:
{ name: String }

[2] Number:
⇒ Represents a numeric value.
⇒ Example: { age: Number }

[3] Boolean:
⇒ Represents a true or false value.
⇒ Example: { isActive: Boolean }

[4] Date:
⇒ Represents a data and time.
⇒ Example: { createAt: Date }

[5] Buffer:
⇒ Represents binary data, often used for storing binary files like images.
⇒ Example: '{ image: Buffer }'

[6] Array:
⇒ Represents an array of values.
⇒ Example: '{ tags: [String] }' (Array of strings)

[7] Mixed:
⇒ Represents data of mixed types.
⇒ Example: '{ data: Mixed }'

[8] ObjectId:
⇒ Represents a reference to another document in a different collection.
⇒ Example: '{ author: ObjectId }'

★ Validation:

Mongoose allows to define validation rules for our schema fields using various options.

[1] 'required':
⇒ Specifies that a field is required.
⇒ Example: '{ name: { type: String, required: true } }'

[2] 'min and max':
⇒ Defines minimum and maximum values for numeric fields.
⇒ Example: { age: { type: Number, min: 18, max: 99 } }

[3] 'enum' 'match':
⇒ Restricts a string field to validate a string field.
⇒ Example: { email: { type: String, match: /^[\w-]+(\.[\w-]+)*@([\w-]+\.)+[a-zA-Z]{2,7}$/ } }

[4] 'enum':
⇒ Restricts a string field to a pre defined set of values.
⇒ Example: { status: { type: String, enum: ['active', 'inactive'] } }

[5] Custom validator function:
⇒ Allows to define custom validator.

⇒ Example:

```
const validateName = (name) => {
    return name.length > 0;
};

const userSchema = new mongoose.Schema({
    name: {
        type: String,
        validate: [validateName, 'Name
```

cannot be empty']
});

[6] 'unique':
⇒ Ensures that a field's value is unique across all documents in the collection.
⇒ Example: '{email: {type: String, unique: true}}'

[7] 'validate' option:
⇒ Allows you to specify a custom synchronous or asynchronous validation function.

⇒ Example:

```
const customValidator = (value) => {
    return value.isValid;
};

const userSchema = new mongoose.Schema
                    ({
    someField: {
        type: String,
        validate: {
            validator: customValidator,
            message: 'validation failed'
        }
    }
});
```

Date
Page

Ques Schema Referencing and querying

* Schema Referencing

→ It is also known as data modeling, involves defining the structure of our data, specifying relationship between different data entities, and how they are stored in the database.

* Primary key:
In a table, one or more colums are designed as primary keys. Primary keys are unique identifiers for each row in the table.

* Foreign key:
In another table, we can create a foreign key column that references the primary key of another table.
→ This establish relationship between two tables.

* Manual References:
→ we store references to documents from one collection within another collection.

* Embedded Documents:
We can nest documents within other

documents.

→ For example, we can embed comments within a blog post document.

* Querying:

Querying refers to the process of retrieving specific data from a database.

→ It involves specifying conditions and criteria to filter and retrieve the desired data.

→ Querying is typically done using the database's query language, which is often JSON-based.

→ Common queries include:

[1] find():
- Used to retrieve documents from a collection based on specified criteria.

[2] insertOne() and insertMany():
- Used to insert new documents into a collection.

[3] updateOne() and updateMany():
→ Used to update documents that match specific criteria.

[4] deleteOne() and deleteMany().
→ Used to delete documents that match specific criteria.

[5] $lookup:
→ Allow to perform joins between collections to retrieve related data.

[6] $match:
→ Used to filter documents in aggregation pipelines.

[7] $project:
→ Used to shape the output of aggregation queries.

The specific syntax and capabilities of querying depend on the database system being used.