Name :- Dhola Drashti Ishwarbhai

Roll no :- 12

Semester :- 7 [MSc IT]

Date :- 28-08-2023

Topic :- Express Framework

Que:- Write an assignment on express framework with following topics:

1. Introduction :-

- The Express framework is a widely used and lightweight web application framework for Node.js.

- It simplifies the process of building server-side web applications and APIs by providing a set of powerful features and tools.

- Developed as an open-source project, Express has gained popularity for its minimalistic design and flexibility, allowing developers to create efficient and scalable applications with ease.

Key features of Express:

1. Routing :

Express simplifies the routing of HTTP requests to specific endpoints, making it easy to define routes and handle various HTTP methods. (GET, POST, PUT, DELETE, etc).

2. Middleware :

Middleware functions in Express allow you to execute code at different points during the request - response cycle.

This enables tasks like authentication, data validation, logging, and error handling to be easily integrated into your application.

3. Template Engines :

Express supports various template engines like EJS, Pug (formerly known as Jade), and Handlebars.

These engines help in generating dynamic HTML content on the server-side, facilitating the creation of consistent and reusable layouts

4. Static File Serving :

Express makes it straightforward to serve static files such as HTML, CSS, Javascript, and images.

This is crucial for delivering client-side assets efficient.

5. RESTful API Development :

Express is often used to build RESTful APIs which adhere to the principles of Representational State Transfer (REST).

These APIs provide a structured way to interact with applications using standardized HTTP methods.

6. Error Handling :

Express simplifies error handling by providing middleware to catch errors and manage them in a centred location, enhancing the overall robustness of your application.

7. Middleware Ecosystem

Express boasts a rich ecosystem of third-party middleware that extends its functionality.

Developers can leverage these packages to add features such as authentications, compression and more.

2) Middleware

- Middleware in Express is a key concept that enables developers to execute code and perform various tasks during the request-response life-cycle of an application.

- It sits between the incoming request and the outgoing response, allowing you to add functionality to your application's routes and handle common tasks consistently.

- Middleware functions can be used for tasks such as authentication, logging, data validation, error handling and more.

How middleware works.

- When an HTTP request is made to an Express application, the request goes through a series of middleware functions before reaching the final route handler.

- Each middleware function has access to the 'request' and 'response' objects, and the optional 'next' function.

- The 'next' function is used to pass control to the next middleware in the chain

- middleware function can modify the request or response objects, terminate the request-response cycle early, or call the 'next' function to proceed to the next middleware in line.

- This allows you to build a pipeline of middleware functions that execute sequentially.

Example Use Cases for middleware

authentication
logging
Data validation
Error Handling
Compression
CORS Handling

Order of middleware Execution

- middleware functions are executed in the order they are added using the 'app.use' or app.METHOD methods.

- It's important to pay attention to the order, as the sequence can impact the behaviour of your application.

Q) Express validator

- Express validator is a middleware module for the express framework that simplifies the process of validating and sanitizing user input in web applications.

- It helps prevent common security vulnerabilities and data integrity issues by ensuring that the data submitted by users adheres to the expected formats and constraints

Importance of Data Validation :

- User input is a common source of security vulnerabilities, including SQL injection, cross-site scripting (XSS), and more.

- Proper validation ensures that the input is safe, correctly formatted, and suitable for the intended use.

- Express Validator provides a set of validation and sanitization functions that developers can easily integrate into their Express applications.

Key features of Express Validators

1. Validation :

Express validator offers a variety of validation methods to check input data against specific criteria.

These methods include checks for strings, numbers, dates, emails, URLs, and more.

2. Sanitization :

In addition to validation, Express Validator helps sanitize user input to remove any potentially harmful or unwanted characters.

This is crucial for preventing attacks like SQL injection and XSS.

3. Error Handling :

when validation fails, Express validator generates detailed error messages that can be easily customized and displayed to users.

This helps user understand that what went wrong and how to correct it.

Example of Express Validator :

1. Installation :

npm install express-validator

2. Integration

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

const app = express();

app.use(express.json());

app.post('/user', [
    body('username').isLength({ min: 5 }),
    body('email').isEmail(),
], (req, res) => {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
        return res.status(400).json({
            errors: errors.array() });
    }
});
app.listen(3000, () => {
    console.log('server is running on port 9000');
});
```

Benefits of using Express Validation

Security :- helps prevent security vulnerabilities by ensuring that user input is safe and properly formatted.

Consistency :- Validation logic is centralized and consistent across your application's routes.

Readable code :- provides a clear and readable way to define validation rules, making your codebase more maintain.

Customization :- user can customize error message and validation logic according to your application's requirements.

4) Template Engine

- A template engine is a software tool or framework that simplifies the process of generating dynamic content for web applications.
- it allows developers to define templates containing placeholders for data, which are then filled in with actual data when rendering a web page.
- Template engines are commonly used in web development to separate the presentation from the underlying logic, promoting code organization, maintainability and reusability.

Key Concepts and features of template engines:-

Templates :-

structured documents that contain placeholders for dynamic content.

These placeholders, often denoted by special syntax (e.g. '{{ variable }}'), are replaced with actual data when the template is rendered.

Data Binding :-

Template engine provides a mechanism for binding data to template placeholders.

Data can be supplied as objects / variables, and the engine replaces placeholders with the corresponding data values.

Conditional logic :-

Template engines support conditional statements to enable dynamic rendering based on conditions or loops.

Partial templates :-

Reusable components or partial templates can be defined and included within other templates.

Filters and Helpers :-

It allows you to manipulate data before rendering it.

Popular Template Engines :-

EJS (Embedded Javascript)

- Simple and straight forward template engine for JS.
- it uses JS code embedded within templates for dynamic content generation.

Pug (formely Jade)

- highly expressive and whitespace-sensitive template engine.
- it uses indentation and a concise syntax to define templates.

Handlebars :

logicless template engine that focuses on simplicity

Mustache :

minimalistic and language-agnostic template engine

Benefits of using template engines

Separation of Concerns :

enforce a clear separation between the presentation layer and application logic, making code easier to maintain and debug.

Reusability :

templates can be reused across multiple pages or components, reducing code duplication.

Consistency :

template engines promote consistent styling and layout throughout a web application.

Security :

Properly implemented template engines can be helpful to protect against certain security vulnerabilities, such as cross-site scripting, by automatically escaping user-generated content.

Developer Friendly

provides a familier and expressive syntax making it easier for developer to work.

5) REST API

- REST is an architectural style for designing networked applications.

- RESTful APIs are built based on the principles of REST.

- These APIs use HTTP methods and follow a set of conventions to provide a lightweight and scalable approach to designing web services.

- REST APIs are commonly used for web applications, mobile applications, and other distributed systems due to their simplicity and flexibility.

Key concepts of REST API:

1) Resources:

- everything is considered a resource.

- Resource can be objects, data or services that your application exposes.

- each resource is identified by a unique URL, which is known as a Uniform Resource Identifier.

2) HTTP Methods :

- REST APIs use standard HTTP methods to perform operations on resources.

- The primary HTTP methods used in REST API are :

GET : used to retrieve data from a resource.
POST : used to create new resource.
PUT : used to update an existing resource or create it if it doesn't exist.
DELETE : used to remove a resource.
PATCH : used to make partial updates to a resource.

3) Stateless :

- REST APIs are designed to be stateless, meaning that each request from a client to the server must contain all the information necessary to understand and process that request.

- The server does not maintain any client state blw requests.

4) Uniform Interface:

- REST API follow a uniform and consistent interface, which includes the use of standard HTTP methods that the manipulation of resources through URLs.

5) Representation:

- Resources can have multiple representations, such as JSON, XML, HTML, or others.
- clients can specify their preferred representation using HTTP headers, such as 'Accept' and 'Content-Type'.

Example of REST API

To retrieve a list of all books:
GET /books

To retrieve a specific book:
GET /books/{id}

To create a new book:
POST /books

To update a book : PUT /books/{id}

To delete a book : DELETE /books/{id}

Advantages of REST API

Simplicity , Scalability , Compatibility ,
Flexibility , Caching , Independence.

6) API security best practices

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

- Here are some best practices of ensuring your APIs :

1) Authentication:

- implement strong authentication mechanisms such as API keys, OAuth 2.0, or JWT

- use multi-factor authentication for administrative access to APIs

- enforce password policies, including complexity requirements and regular password changes.

2) Authorization:

- implement fine-grained access controls and role-based authorization to restrict access to only authorized users and actions.

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 requests 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 requests a client can make in a given time period.

6) Error Handling :

- implement proper error-handling mechanisms to gracefully handle exceptions and return appropriate HTTP codes.

7) Security Headers:

- use security headers like Content Security Policy, X-Content-Type-Options, X-frame-Options and X-XSS-Protection to mitigate common web vulnerabilities.

- set appropriate CORS headers to control which domains can access your API.

8) API keys and Tokens:

- Protect API keys and tokens from exposure.

9) Data encryption:

- Encrypt sensitive data at rest using encryption mechanisms provided by your storage infrastructure.

7) Types of tokens and their usage

- Tokens are used in various contexts in computer science and security to represent and authorize access to resources or services.

1) Access Tokens:

usage: Access tokens are commonly used in authentication and authorization systems.

- They are issued to authenticated users or clients and grant them access to specific resources or actions.

Example: OAuth 2.0 access tokens, JSON web Tokens, Bearer tokens.

2) Refresh tokens:

usage: Refresh tokens are often used in conjunction with access tokens.

- 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

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

- They are often used for user authentication and authorization, session management, and passing claims b/w services in a stateless manner.

Examples: JWTs in authentication flows, authorization tokens in microservices architectures.

4) CSRF Tokens:

usage: CSRF tokens are used to prevent cross-site request forgery attacks.

- They are included in web forms and compared to tokens stored in the server to ensure that the request originates from a legitimate source.

examples: Anti - CSRF tokens

5. Session Tokens:

usage: 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.

examples: Cookies, session IDs, and tokens used in web-based session management.

6. API tokens:

usage: API tokens are used to authenticate and authorize access to APIs.

- They are typically sent in HTTP headers or as query parameters in API requests.

examples: ID tokens in openID connect flows. API keys, API tokens

7. ID Tokens:

usage: ID tokens are used in identity authentication protocols like openID connect.

examples: ID tokens in openID.

vision

% Bearer Tokens:

usage: Bearer tokens are used for authorization and authentication purpose.

examples: OAuth 2.0 bearer tokens.

% federated Tokens:

usage: federated tokens are used in federated identity and single sign-on systems to enable users to access multiple services with a single set of credentials.

example: Apple Push notification service tokens, Firebase Cloud messaging registration access.

8) MongoDB database with CRUD Operation API

```
const { MongoClient, ObjectID } = require('mongodb');

const url = 'mongodb://localhost:27017';
const dbN = 'test';

const client = new MongoClient(url, { useNewUrlParser:
                true, useUnifiedTopology: true })

client.connect((err) => {
    if(err) {
        console.error('error to connect database');
        return;
    }

    console.log('connected to database');

const db = client.db(dbN);

const collection = db.collection('todos');

const todo = { text: 'sample Todo',
               completed: false };

collection.insertOne(todo, (err, result) => {
    if(err) {
        console.error('error inserting document');
        return;
    }
```

```
        console.log ('document inserted');

collection.find({}).toArray((err, documents) => {
    if (err) {
        console.log ('error querying document');
        return;
    }

    console.log ('Queried documents:', documents);

const filter = {_id: new objectID(documents[0]._id)};
const update = {$set:{Completed: true}};

collection.updateOne(filter, update, (err, result) => {
    if(err) {
        console.error ('error updating document');
        return;
    }

    console.log ('document updated');
```

- continue on next page

```
collection.deleteOne(filter, (err, result) => {
    if (err) {
        console.error('error deleting');
        return;
    }

    console.log('deleted successfully');

    client.close();

    });
  });
 });
});
```

9) Mongoose Schema, Model, Document and API for CRUD, search in Express app.

models / todo.js

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

const todoSchema = new mongoose.Schema({
    text : {
        type : string,
        required : true,
    },

    completed : {
        type : Boolean,
        default : false,
    },
});

const Todo = mongoose.model ('Todo', todoSchema);

module.exports = Todo;
```

app.js

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

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

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

mongoose.connection.on ('connected', () => {
    console.log ('connected to database');
});

app.use (bodyParser.json ());

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

```
app.get('/todos', async (req, res) => {
    try {
      const todos = await Todo.find();
      res.status(200).json(todos);
    }
    catch (err) {
        res.status(500).json({error: err.message});
    }
});

app.get('/todos/:id', async (req, res) => {
    try {
      const todo = await Todo.findById(req.params.id);

      if (!todo) {
        return res.status(404).json({error: 'not found'});
      }
      res.status(200).json(todo);
    }
    catch (err) {
        res.status(500).json({error: err.message});
    }
});
```

```
app.put('/todos/:id', async (req, res) => {
  try {
    const todo = await
      Todo.findByIdAndUpdate(req.params.id,
        req.body, { new: true, });

    if (!todo) {
      return res.status(404).json({error: 'not found'});
    }
    res.status(200).json(todo);
  }
  catch (err) {
    res.status(400).json({error: err.message});
  }
});

app.delete('/todos/:id', async (req, res) => {
  try {
    const todo = await
      Todo.findByIdAndDelete(req.params.id);
    if (!todo) {
      return res.status(404).json({ error: 'not found' });
    }
    res.status(204).send();
  }
  catch (err) {
    res.status(400).json({error: err.message});
  }
});
```

```
app.get('/todos/search', async (req, res) => {
    try {
      const {query} = req.query;
      const todos = await
          Todo.find({text: {$regex: query,
                    $options: 'i'}});

      res.status(200).json(todos);
    }
    catch (err) {
        res.status(500).json({error: err.message});
    }
});

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

10) Mongoose schema data types, validation etc.

- In mongoose, you can define the schema for your mongoDB documents, specifying data types and validation rules for each field.

```
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 postSchema = 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 ({
    dates : mongoose.Schema.Types.Mixed,
});
```

Validations

```
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 b/w 18 and 99';
    }
}
```

12) Schema Referencing and querying

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

const authorSchema = new mongoose.Schema({
    name: String,
    books: [{ type: mongoose.Schema.Types.ObjectId,
              ref: 'Book' }],
});

const bookSchema = new mongoose.Schema({
    title: String,
    author: { type: mongoose.Schema.Types.ObjectId,
              ref: 'Author' },
});

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

const Book = mongoose.model('Book', bookSchema);


const author = new Author({ name:
                'J.K. Rowling' });

const book = new Book({ title:
    'Harry potter and the Sorcerer\'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);

      author.books.forEach(book => {
        console.log(book.title);
      });
    }
});
```