

## Assignment - 2

- + write an assignment on express Framework with following topics:-

### Introduction:-

- Express.js is a web framework for node.js it is fast, efficient and asynchronous in nature.
- Express is a fast, expressive, essential and moderate web framework of node.js you can assume express as a layer built on the top of the node.js that helps manages the server and routes. It provides a robust set of features to develop web and mobile application.
- Let's see some of the core Features of express framework.
- It can be used to design single-page, multi-page and hybrid web application.
- It allows to setup middleware to respond to HTTP request.
- It allows to dynamically render HTML pages based on passing arguments to template.

### → why use express:-

- ultra fast > 10.
- Asynchronous and single threaded.
- mvc like structure.
- Robust API makes routing easy.
- File :- basic - express.js.

```
var express = require("express");
```

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

```
app.get("/", function(req, res) {
```

```
    res.send("welcome to jayaram");
```

```
});
```

`app.listen(5000, () => {`

`console.log("Server listing on port 5000");  
});`

### → middleware:-

- Express.js middleware are different type of functions that are invoked by the express.js routing layer before that final request handler. As the same specified middleware appears in the middle between an initial request and final intended route. In short, middleware functions are always invoked in order which they are added.

### - what is middleware Functions:-

- middleware Functions are those that access to get request or response object in request-response cycle.
- It can perform following tasks:-
- It can execute any code.
- It can make changes to the request and response cycle.
- It can end the request-response cycle.

File:- simple express.js.

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

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

```
app.get("/", function(req, res){
```

```
    res.send("Welcome to express");
```

```
});
```

```
app.use(function(req, res){
```

```
    console.log(`-S-Y-S` , req.method, req.url);
```

```
});
```

```
var server = app.listen(5000, function(){
```

```
    console.log("Example of this app is listing on 5000");
```

```
});
```

### → Express validators:-

- express-validator is a set of express.js middleware that uses an 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 requests and offer tools to determine if the request is valid or not which checks what method according to your validators and so on.
- you can install this package by using this command:-  
npm install express-validator.

### → e.g :- HTML file

```
<form action="savereader" method="post">
```

```
<p></p>
```

```
Enter Email :- <input type="text" name="email"> <br>
```

```
Enter password :- <input type="password" name="pass"> <br>
```

```
<input type="submit" value="Submission">
```

```
</p>
```

```
</form>
```

### → index.js

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

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

```
app = express();
```

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

```
res.render("sampleform");
```

```
});
```

```
app.post('/saveData',
```

check('email', 'Email length should be 10 to 30 character')

isEmail(), isLength({min: 10, max: 30})),

Check ("pass")'s 'password length must be & 10 characters').  
is length ({min: 8, max: 10})

}, (req, res) => {

const errors = validationResult(req);

if (!errors.isEmpty()) {

res.json(errors)

} else {

res.send("Successfully validated");

});

app.listen(5000, () => {

console.log("Server listing on port 5000");

});

→ 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 file with actual values, and transform the template into an HTML file sent to 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 applications generator uses Jade as its default, but it also supports several others.
- To render template files, set the following setting properties
  - views the directory where the template files are located.  
e.g app.set('views', 'views')
  - view engine the template engine to use. For example to use the pug template engine: app.set('view engine', 'pug')

To install package following command use:-

- npm install pug-sass
- app.set('views engine', 'pug')
- Create pug template file named index.pug in the views directory with following command:-

html  
head

title = title

body

h3 = message

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

- when you request to home page the index.pug file will be rendered as html.

### → REST API:-

- An API is always needed to create mobile applications single page applications, use AJAX calls and provide data to clients. A popular architectural style of how to structure and name these APIs and the endpoints is called (Representational State Transfer) REST.

- Restful APIs and methods provides us with almost all information we need to process request.

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

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

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

```
mongoose.connect("mongodb://localhost:27017/db");
```

```
const db = mongoose.connection;
```

app.get('/api/books', (req, res) => {  
 Book.find({})

.then((books) => {  
 res.send(books);

});

.catch((err) => {

res.status(400).send('ERR!');

});

});

app.post('/api/books', (req, res) => {

console.log(req.body)

var book = new Book({

name: req.body.name,

price: req.body.price,

quantity: req.body.quantity});

book.save();

.then((book) => res.status(200).send(book));

.catch((err) => if (err))

return console.error(err);

});

→ API security best practices :-

→ To protect API following best-practices will be used

→ use HTTP methods API Routes:-

- imagine you building a node.js RESTful API for creating, updating or retrieving data.

- As a best practice, your API should always use nouns as resource identifiers.

- routing can look like:-

- POST /users/:id
- GET /users
- GET /users/:id
- DELETE /users/:id
- PUT /users/:id

→ use HTTP Status code correctly :-

- if something goes wrong while serving a request you must set the correct status code for that in response

- 2xx, if everything was okay
- 3xx, if resource was moved
- 4xx, if request cannot be fulfilled because of a client error

→ use HTTP headers to send metadata.

- To attach metadata about the payload you are about to send, use HTTP headers, Headers like this can be information on:-

- pagination
- rate limiting
- or authentication

- If you need to stop or set any custom metadata in your headers, it was best practice to prefix them with x-for example, if you were using CSRF tokens it was uncommon way to name other X-CSRF-TOKEN:

→ Black Box testing to your API:-

- Black box testing is a method of testing where the functionality of an application is examined.

```

→ const request = require('supertest')
describe('GET / users/:id', function() {
  it('returns a user', function() {
    return request(app)
      .get('/1-user')
      .set('Accept', 'application/json')
      .expect(200, {
        id: '1',
        name: 'John'
      }, done)
  })
})

```

→ Do JWT - Based , stateless Authentication

- as your REST API must be stateless, so does your authentication layer. For this, JWT is ideal.
- JWT consist of three parts:-  
  - Header :- containing the type of the token and hashing algorithm
  - payload :- containing the claims
  - signature

Copy

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

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

```
const app = koo();
```

```
app.use(jwt({
```

```
  secret: 'very-secret'
```

```
}))
```

```
app.use(function(req, res) {
```

```
  this.body = {
```

```
    secret: '42'
```

```
});
```

- also you always have to make sure that all your API endpoints are only accessible through a secure connection using HTTPS.

### → Types of tokens and their usage:-

- Types of tokens:-

#### 1. Authentication Token (JWT - JSON web token)

- JSON web tokens are a popular way to securely transmit info between parties as a JSON object. JWT is often used for authentication purpose in web applications.
- usage:- JWTs are generated upon successful login. They contain information like user ID, expiration time, and other claims. Client stores JWT and include it in headers of subsequent requests to authenticate it.

#### 2. Access tokens:-

- Access tokens are used to grant access to specific resource or operation on a server.
- usage:- when an access token expires, a fresh token can be exchanged for a new access token without requiring user to log in again.

#### 3. Refresh tokens:-

- Refresh tokens are often used in conjunction with access tokens to enable longer-lived sessions without requiring the user to re-enter credentials.
- usage:- After successful authentication a user is provided with an access token.

#### 4. CSRF - Tokens (Cross-sites Request Forgery):-

- used to protect against cross-site forgery attack by ensuring that request can only made from the same website or application.

Usage:- Servers generate a unique CSRF token and

- include it in the response.

#### 5. Bearer tokens:-

- Are the type of access tokens often used in authentication headers to grant access to protected resource.
- Usage:- In HTTP headers, the "Authorization" header is used to include a bearer token.

#### 6. Custom tokens:-

- Depending on your application's specific requirements, you might create custom token formats for various purposes like session management or user-specific tokens.
- Usage:- Custom tokens could be used to implement various authentication or authorization strategies that cater to your applications unique needs..

B Functions to proceed crud operation

1) database connectivity:-

`mongoose.connect('url', {auther params});`

This code will execute connectivity with the database when server starts.

2) .FindByid :-

The query find a record into a table by passed object by from the clientside there are several ways to capture an id to find in table

we can require id in query param and also in body but getting electron body will require request method post.

Ex 1 = req. query.id.

2 = req. params

3 = req. body.id (only for post).

3. res.json (obj):-

- The syntax is used to send response to client side in json format.

4) .save()

- The method returns to something to save data or changes to db we can use other saving saved object into variable as below

const user = new userschema (createtuser)

let users = circuit user . save()

### 5) Findone():-

- The method finds an object with given key value pairs returns the whole single object from the table  
 user findone ({email: "jinal@gmail.com"})

it will return it finds an record in which email is jinal@gmail.com.

### 6) FindByid And update()

userschema.findByIdAndUpdate ({\_id: 123})  
 ↑  
 Table

### 7) FindByid And Delete()

users schema . findByIdAndDelete ({\_id: 123})

### 8) - Find()

const users = userschema . find ()

() mongoose schema , model , document and API can be used in Express app.

### 1) mongoose schema / model

const mongoose = require ("mongoose");

const taskschema = new mongoose . schema ({

title: string

descripton: string

)

const task = mongoose . model ("task" , taskschema)

## 2) Search in Express:-

```

  router.get('/tasks/search', async (req, res) => {
    const title = req.query.title;
    try {
      const tasks = await Task.find({ title: title });
      res.send(tasks);
    } catch (error) {
      res.status(500).send(error);
    }
  });

```

## 3) mongoose schema datatype validation.

Datatype:

String : Text data.

Number : for numeric data.

Date : date and time.

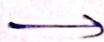
Boolean : true/false.

Array : array value.

Objectid : references doc with same on different collections.

Mixed : flexible and unstructured data.

Buffer : binary data (image files).



validation

### 1) Built-in validators:

const userSchema = new mongoose.Schema({

type: String

required: true,

minlength: 3,

maxlength: 20,

unique: true

3.

email: {

type: String,

required: true,

match: /^\w+@[a-zA-Z]+\.\w+\$/

4

age: {

type: Number,

min: 18,

max: 100

5

});

## E) Schema Referencing and querying in

### 1) schema Referencing:

- install mongoose: npm install mongoose,

- model.

```
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);
```

## 2) Working with schema references:-

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

```
const router = express.Router();
```

```
const Author = require('../models/Author');
```

```
const Book = require('../models/book');
```

```
router.get('/books-by-author/:authorId', async (req, res) => {
    const authorId = req.params.authorId;
```

```
    const authorId = req.params.authorId;
```

```
    const books = await Book.find({Author: authorId});
```

```
    res.json(books);
```

J

```
    catch (error) {
```

```
        console.error(error);
```

```
        res.status(500).json({ error: 'Internal Server Error' });
```

J

```
};
```

11 working Author

```
writers.get('/authors/:bookId', async (req, res) =>
  try {
    const bookId = req.params.bookId;
    const book = await Book.findById(bookId);
    popular(`Author!`);
    res.json(book);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

module.exports = writers;