Databases, Network and the Web Coursework for Midterm: Blogging Tool Documentation Report

Written & Developed by Tan Yee Chong, Chen Yi Cong (SN: 230668566)

All requirements were achieved after the following initial process:

- npm init: Execute this command to set up a new npm package
- npm install express: Execute this command to install express.
- npm install ejs: Execute this command to install ejs.

Table of Contents

Schematic diagram

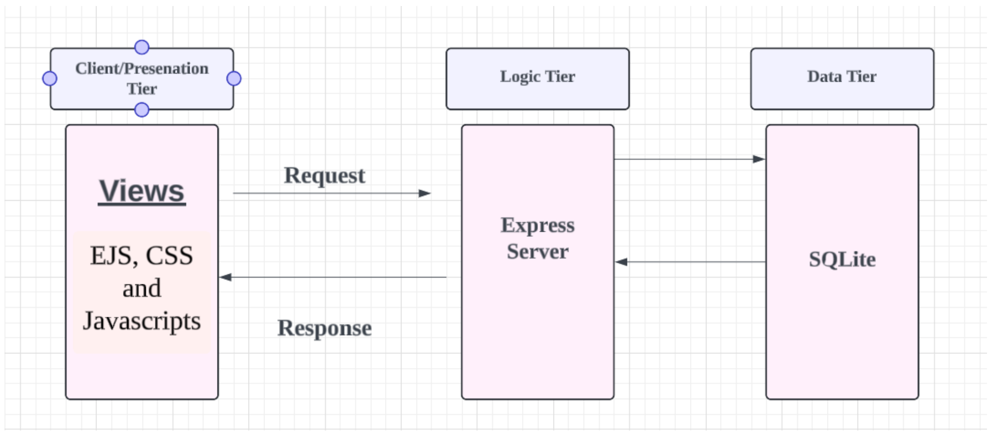


Figure 1: Schematic Diagram

In the case of the CuteBlog Web Application, the Presentation Tier is made using EJS file, CSS and Javascript and the application server, is made using the express server while the database used in the application is SQLite, a relational database.

This architectural design separates the concerns of the application, promoting modular, scalable, and maintainable system development.

In my coding directory structure, I have the code organized into separate folders for assets (CSS files), routes (JavaScript files), scripts, and views (EJS templates).

For example, the CSS files are separated from the JavaScript files, and the EJS templates are in a separate folder. This separation of concerns makes it easier to maintain and update the code, as changes to one aspect of the application do not affect other unrelated parts.

Routing of web paths (logic), the creation of user views (presentation) and the processing of data are handled in separate parts of the code, and these are placed in different folders.

ER diagram

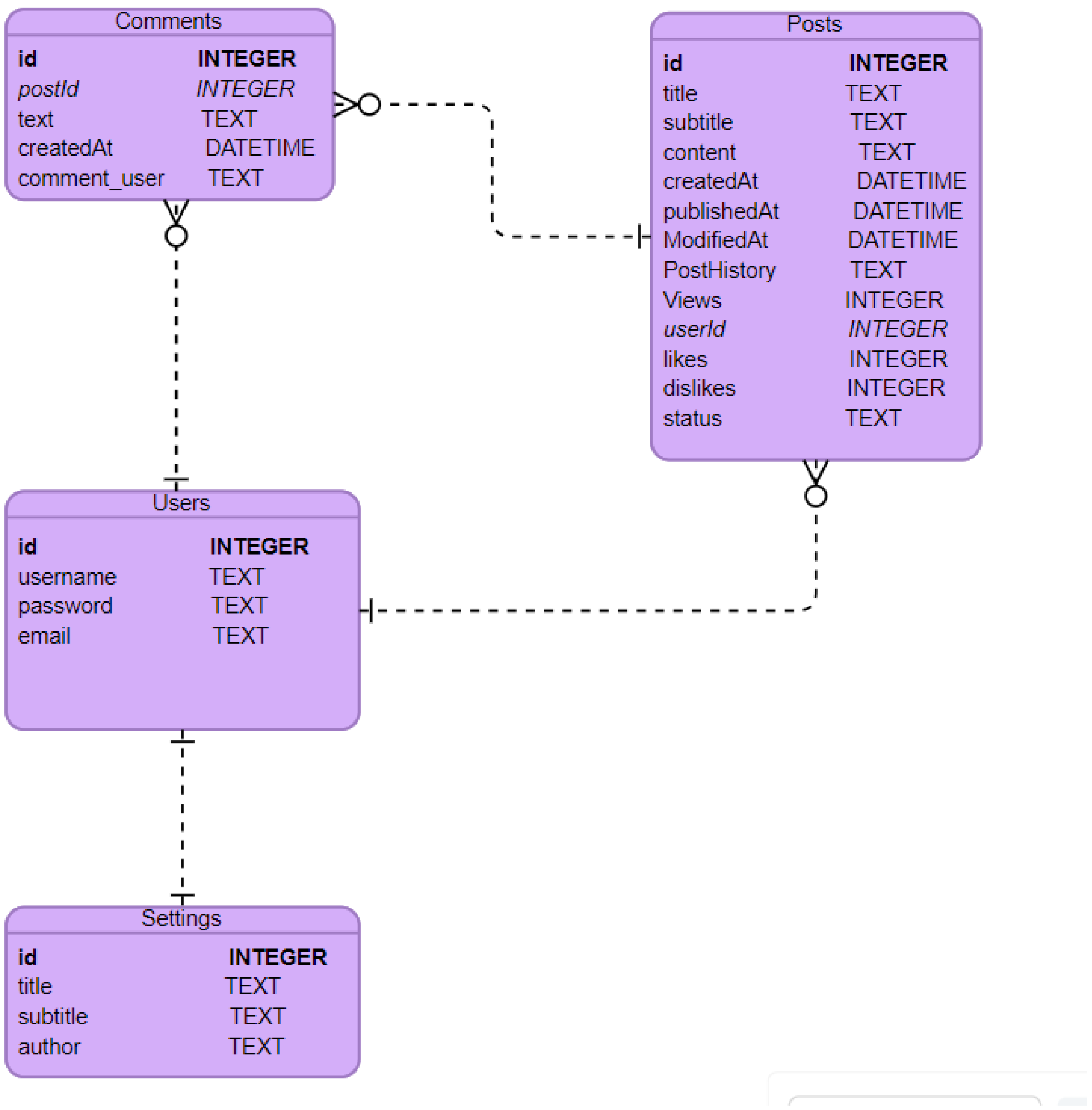


Figure 2: ER Diagram (Crow-Foot Notation)

Breakdown of Relationship between tables

Posts and Comments are 1:m relationship

Users and Posts are also 1:m relationship

Users and Comments are 1:m relationship

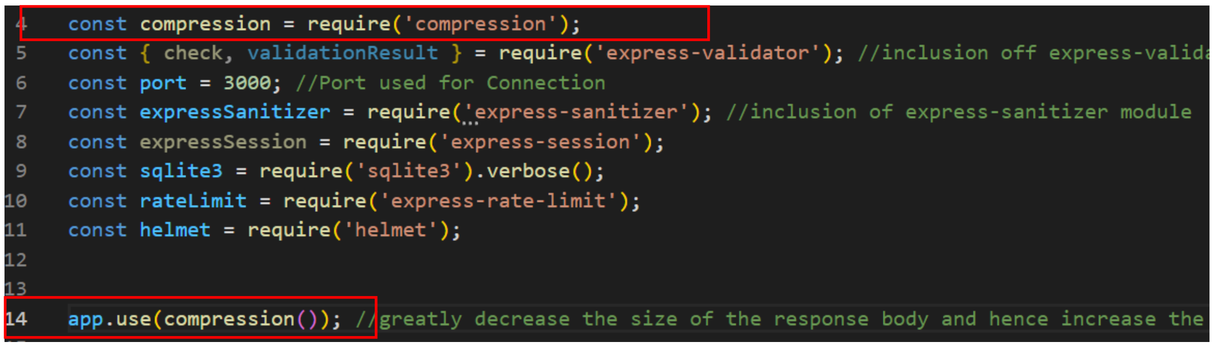
The relationship between Users and Blog Settings can vary across different blog management systems, but in this case, it is a one-to-one relationship, that is, one user only has one blog settings.

Extension Implemented

The extension that implemented in my application that I would like to include in my commentary is **several best practices as mentioned in the resources included in the coursework instructions.** This section will consist of the best practice done and how they are achieved. All these codes can be found within the **index.js of the NodeJs application except for the login ratelimit (to prevent brute-force attack) can be found in the Reader Routes File.**

1. Gzip compression

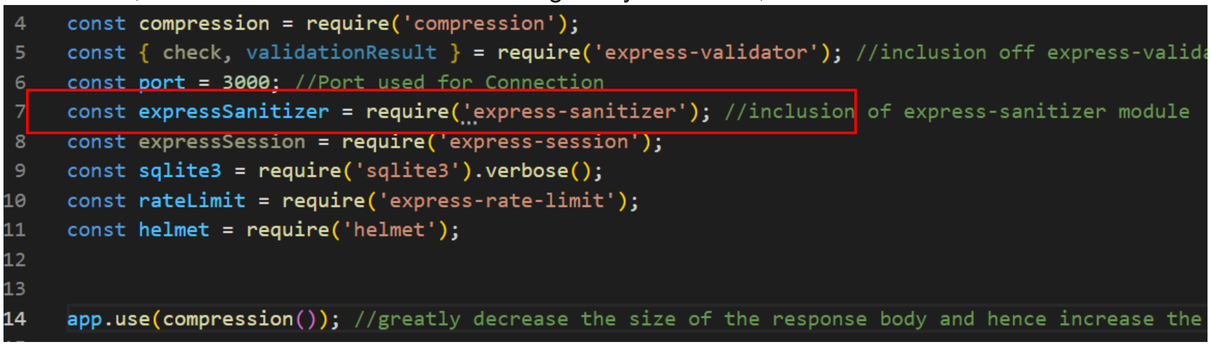
Gzip compressing can greatly decrease the size of the response body and hence increase the speed of my web application, can be seen in **line 4 and 14**.



```
4   const compression = require('compression');
5   const { check, validationResult } = require('express-validator'); //inclusion off express-valida
6   const port = 3000; //Port used for Connection
7   const expressSanitizer = require('express-sanitizer'); //inclusion of express-sanitizer module
8   const expressSession = require('express-session');
9   const sqlite3 = require('sqlite3').verbose();
10  const rateLimit = require('express-rate-limit');
11  const helmet = require('helmet');
12
13
14  app.use(compression()); //greatly decrease the size of the response body and hence increase the
```

2. Implement data sanitisation

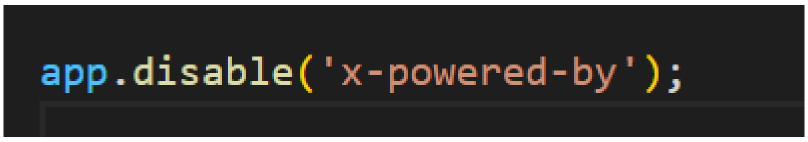
By using the package express-sanitizer, which allows to combine them in many ways so that I can validate and sanitize my express requests, and offers tools to determine if the request is valid or not, which data was matched according to my validators, can be seen in line 7.



```
4   const compression = require('compression');
5   const { check, validationResult } = require('express-validator'); //inclusion off express-valida
6   const port = 3000; //Port used for Connection
7   const expressSanitizer = require('express-sanitizer'); //inclusion of express-sanitizer module
8   const expressSession = require('express-session');
9   const sqlite3 = require('sqlite3').verbose();
10  const rateLimit = require('express-rate-limit');
11  const helmet = require('helmet');
12
13
14  app.use(compression()); //greatly decrease the size of the response body and hence increase the
```

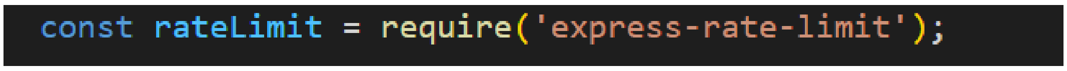
3. Disable Fingerprinting

Provide an extra layer of security to reduce the ability of attackers to determine the software that a server uses, known as "fingerprinting", improves its overall security posture by disabling the *X-Powered-By* response header, can be seen in Line 49.

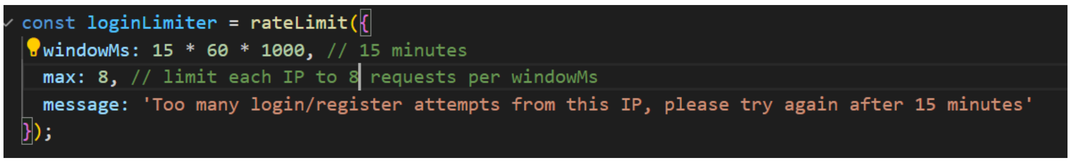


```
app.disable('x-powered-by');
```

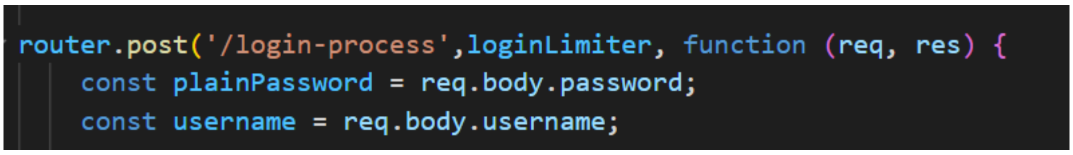
4. **Rate Limiting**
I have implemented rate limiting using express-rate-limit library. It prevents a user from exhausting the system's resources. This is the rate-limit settings for POST login and register route to slow down brute forcing login credentials and also the spamming of account creation, can be seen in line 13 to 17.



```
const rateLimit = require('express-rate-limit');
```

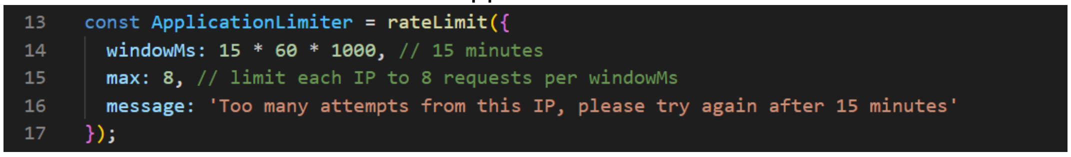


```
const loginLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 8, // limit each IP to 8 requests per windowMs
  message: 'Too many login/register attempts from this IP, please try again after 15 minutes'
});
```

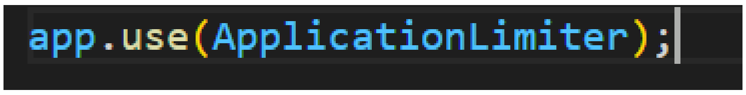


```
router.post('/login-process',loginLimiter, function (req, res) {
    const plainPassword = req.body.password;
    const username = req.body.username;
```

This is the default rate-limit for the application.

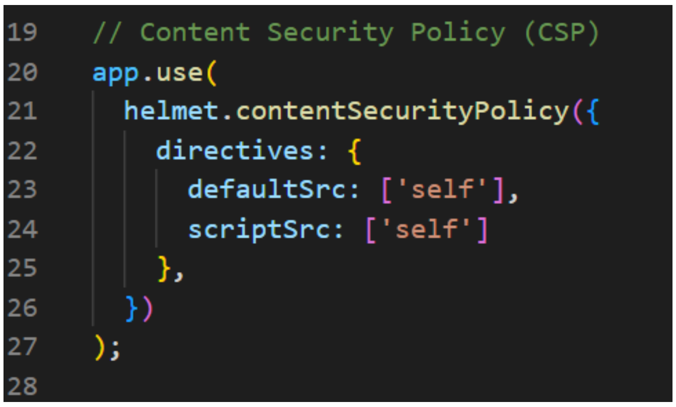


```
13  const ApplicationLimiter = rateLimit({
14    windowMs: 15 * 60 * 1000, // 15 minutes
15    max: 8, // limit each IP to 8 requests per windowMs
16    message: 'Too many attempts from this IP, please try again after 15 minutes'
17  });
```



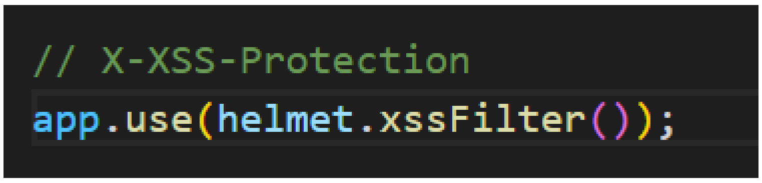
```
app.use(ApplicationLimiter);
```

5. **Helmet.js to set secure http headers**
By using the CSP, it helps mitigate the risk of cross-site scripting (XSS) attacks. It allows you to whitelist the sources from which resources (such as scripts, styles, fonts, etc.) can be loaded. In my application, the CSP is set to allow resources to be loaded from the same origin ('self') for script files. This helps prevent the injection of malicious scripts into my web pages, given in line 20 to 27.



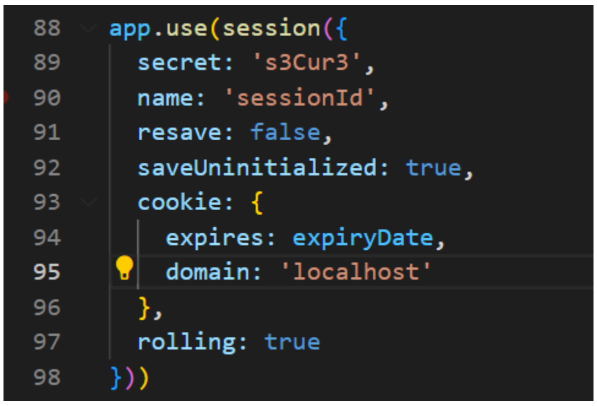
```
19  // Content Security Policy (CSP)
20  app.use(
21    helmet.contentSecurityPolicy({
22      directives: {
23        defaultSrc: ['self'],
24        scriptSrc: ['self']
25      },
26    })
27  );
28
```

The X-XSS-Protection header enables the built-in cross-site scripting (XSS) protection feature which helps detect and mitigate XSS attacks by blocking the execution of malicious scripts in line 31.



```
// X-XSS-Protection
app.use(helmet.xssFilter());
```

6. **Set cookie security options**
   Setting the cookie options can be used to enhance security, hence, I have set the following, these can be found in lines 88 to 98.
   Domain: Indicates the domain of the cookie; use it to compare against the domain of the server in which the URL is being requested.
   Expires: Use to set expiration date for persistent cookies.
   Rolling: This will reset the expiration timer of the session cookie every time the user interacts with my application.
   Resave: This option tells Express-session not to save the session back to the session store for every request, only if necessary.



```
88  app.use(session({
89    secret: 's3Cur3',
90    name: 'sessionId',
91    resave: false,
92    saveUninitialized: true,
93    cookie: {
94      expires: expiryDate,
95      domain: 'localhost'
96    },
97    rolling: true
98  }))
```

End of Report/Commentary