
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 | 2 |
| ER diagram | 3 |
| Extension Implemented | 4 |

Schematic diagram

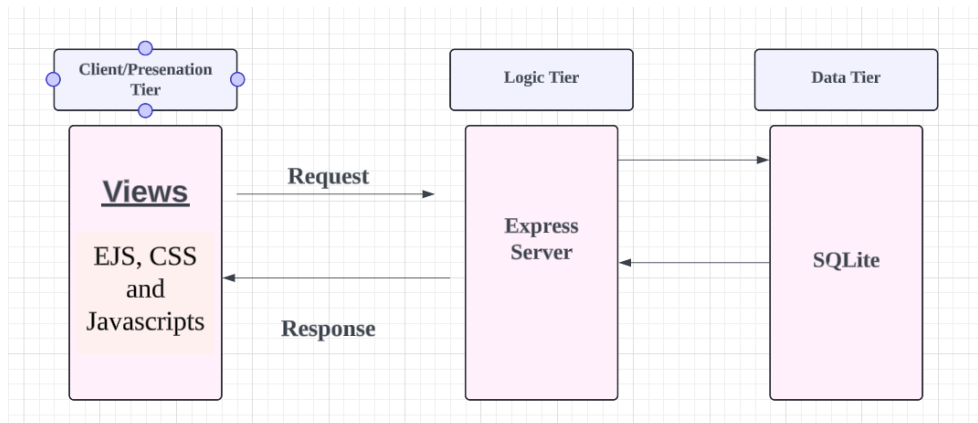


Figure 1: Schematic Diagram

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

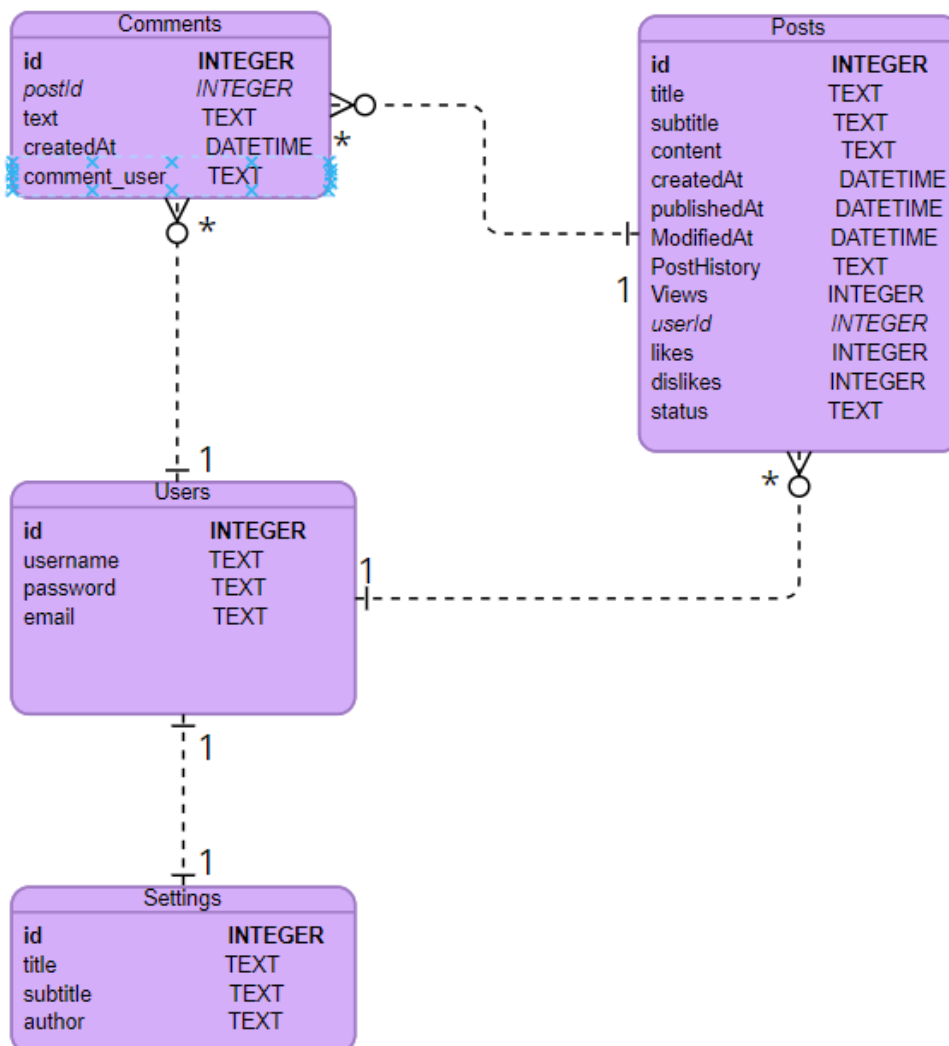


Figure 2: ER Diagram (Crow-Foot Notation)

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 37.

```
37 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 10, 17 to 23 and 87 to 89.

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

```
17 const ApplicationLimiter = rateLimit({
18   windowMs: 15 * 60 * 1000, // 15 minutes
19   max: 100,
20   message: 'Too many attempts from this IP, please try again after 15 minutes',
21   standardHeaders: true, //Return rate limit information in the RateLimit-* headers
22   legacyHeaders: false, // Disable the X-RateLimit-* headers
23 });
```

```
87 //add the Rate Limit to the application of CuteBlog
88 app.use('/author', ApplicationLimiter);
89 app.use('/common', ApplicationLimiter);
```

I have set a stricter rate limit for the login routes.

```
15 const loginLimiter = rateLimit({
16   windowMs: 15 * 60 * 1000, // 15 minutes
17   max: 4, // limit each IP to 4 requests per windowMs
18   message: 'Too many login/register attempts from this IP, please try again after 15 minutes'
19 });
```

5. Helmet.js to set secure http headers

I have configured my application to use the following helmet.js http headers.

CSP is a security mechanism that helps detect and mitigate certain types of attacks, such as cross-site scripting (XSS) attacks, this can be seen in line 24 to 31.

- The defaultSrc (default source) is limited to self and localhost.
- The scriptSrc (script source) is limited to self, localhost, 'unsafe-inline' (allows inline scripts), and <https://stackpath.bootstrapcdn.com>

```
24 app.use(
25   helmet.contentSecurityPolicy({
26     directives: {
27       defaultSrc: ['self', 'localhost'],
28       scriptSrc: ["'self'", "'localhost'", "'unsafe-inline'", 'https://stackpath.bootstrapcdn.com'],
29     },
30   })
31 );
```

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 39.

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

The **helmet.frameguard()** is used to configure the X-Frame-Options HTTP header, which is used to prevent a web page from being embedded in an iframe on another website. This helps prevent "clickjacking" attacks, in my application, I have set the action option to 'deny', which means that the page cannot be displayed in an iframe, regardless of the origin. This can be seen in line 33 to 35.

```
33  app.use(helmet.frameguard({
34    |    action: 'deny'
35    |  }));
36
```

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.

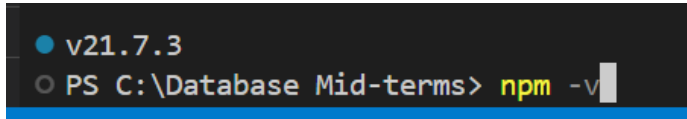
Resave: This option tells Express-session not to save the session back to the session store for every request, only if necessary.

```
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    |  }));
```

7. Avoid known vulnerabilities

It is advised from the resource to avoid known vulnerabilities posted on Snyk for NodeJS applications to improve security of the application. I have applied some of the recommended advises.

One of such advise is to have our node/npm package to be at least 0.8.28, 0.10.30 or higher to prevent allowing remote attackers to cause a denial of service (memory corruption and application crash) via deep JSON objects.

A screenshot of a Windows PowerShell terminal window. The title bar is dark blue. The terminal background is black. The first line shows a blue dot followed by the text 'v21.7.3'. The second line shows a grey circle followed by the text 'PS C:\Database Mid-terms> npm -v'. The text 'npm' is highlighted in yellow. The cursor is at the end of the command line.

```
● v21.7.3
○ PS C:\Database Mid-terms> npm -v
```

Using the same command to check the package version, also prevented another vulnerability, which is uncontrolled Resource Consumption ('Resource Exhaustion') via the fetch function when retrieving content from an untrusted URL. An attacker can cause the application to crash or become unresponsive by sending malicious requests. The recommended is to have version 18.19.1, 20.11.1, 21.6.2 or higher.

End of report/Commentary