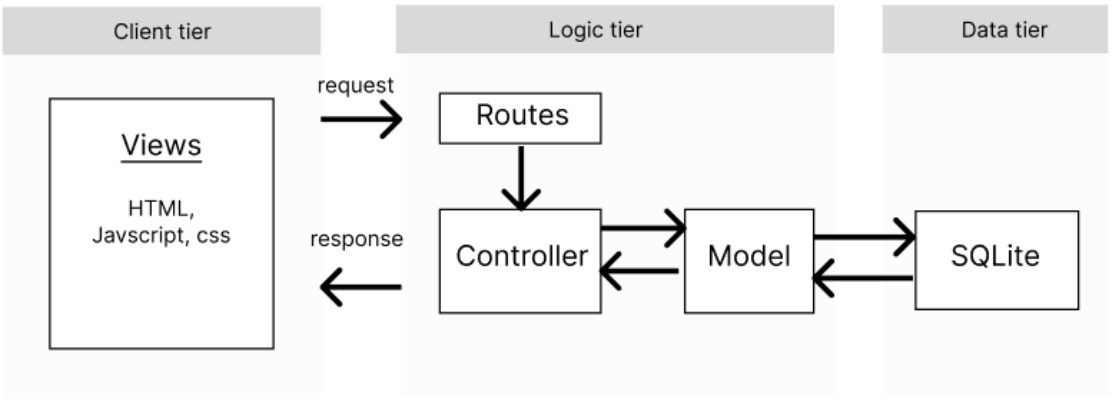
Databases, Network and the Web Coursework for Midterm

Schematic diagram



Extension (Best Practice)

I have implemented some of the best practices according to the resources provided. This section will consist of the best practice done and how they are achieved.

1 Avoid deprecated dependencies and software versions

This is the important first step to ensure that the foundation of your code base is not vulnerable. “npm audit” is used to scan for such vulnerabilities

```
PS D:\UoL\CM2040 Databases Networks Web\mid term\dnw-coursework-template-main\dnw-coursework-template-main> npm audit
found 0 vulnerabilities
```

2 Enable gzip compression

I have used the library namd compression to compress all the http requests. It is implemented simply by requiring and using it.

```
*/
const compression = require('compression');
/**
```

```
/** Compress the http requests */
app.use(compression());
```

3 HTTPS enabled

To enable https connection, we need to generate the private key and public key for the ca cert and the server. The pem files can be found under the config/ssl folder, which will be used to create a HTTPS server.

Once the certs and keys have been generated, there are 2 more steps i have taken to ensure https is enforced site-wide:

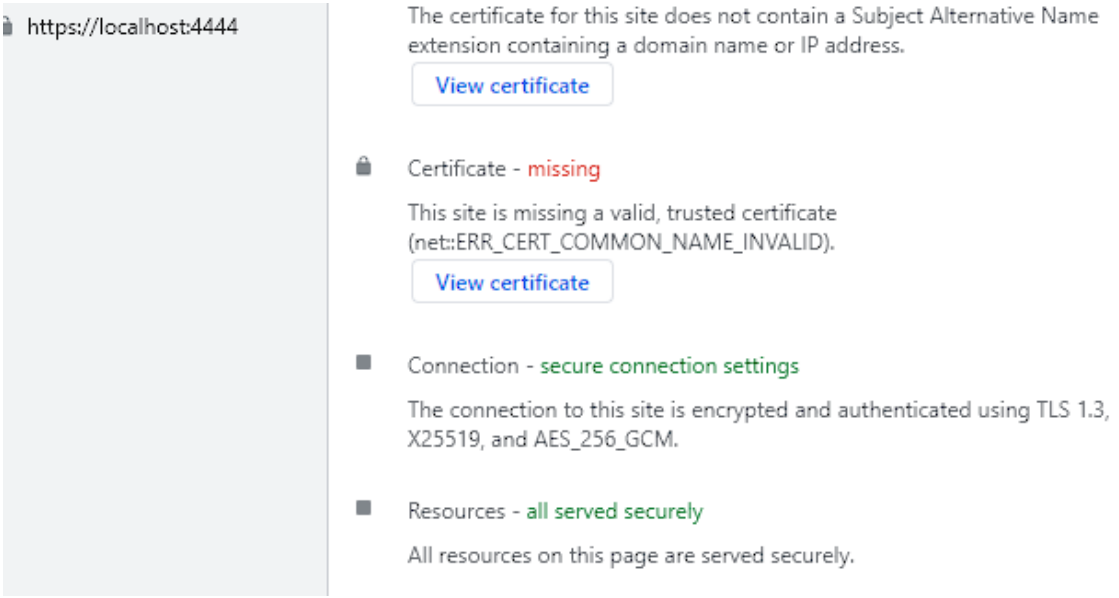
- Http redirection
  Forces user to redirect to https if req.secure is false
- HSTS
  A header that enforces the usage of https protocol only. User cannot mix http requests with https

```
/** Redirect to https if using http */
app.use((req, res, next) => {
  const localhost = req.headers.host.split(':')[0];
  req.secure
    ? next()
    : res.redirect(301, `https://${localhost}:${httpsPort}${req.url}`);
});

/** enforce https with HSTS */
app.use(function (req, res, next) {
  if (req.secure) {
    res.setHeader(
      'Strict-Transport-Security',
      'max-age=300; includeSubDomains; preload',
    );
  }
  next();
});
```

Since HSTS only works when user starts accessing https url, I placed the redirection in front.

This is the tls connection settings from chrome security tab. Since this is just a localhost project, the certificate validation is not important. Rather, the algorithm used should conform to industry standards where versions <= SSL 3.0 and < TLS 1.2 are not allowed.



4 Helmet.js to set secure http headers

I am using some external CDNs in my blog application, thus I opted to generate the NONCE using the crypto library as a way to whitelist the scripts. Scripts that are injected without the nonce will be discarded.

```
/** create NONCE to whitelist accepted inline js scripts */
app.use((req, res, next) => {
  res.locals.cspNonce = crypto.randomBytes(16).toString('hex');
  next();
});
/** implement csp nonce with helmet */
app.use(
  helmet({
    contentSecurityPolicy: {
      directives: {
        scriptSrc: ["'self'", (req, res) => `'nonce-${res.locals.cspNonce}'`],
        connectSrc: [
          "'self'",
          'https://ka-f.fontawesome.com/releases/v5.15.4/css/',
        ],
      },
    },
  }),
);
```

To test this settings, i removed the nonce for the bootstrap cdn

```
<!-- no nonce: should be rejected for violating CSP -->
<script async src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.bundle.min.js"
    integrity="sha256-fh8VA992XMpeCZiRuU4xii75UIG6KvHrbUF8yIS/2/4=" crossorigin="anonymous"></script>
<!-- with nonce: safe -->
<script async nonce="<%= nonce %>" src="https://kit.fontawesome.com/eb16bb9206.js" crossorigin="anonymous"></script>
```

```
Refused to load the script 'https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.bundle.min.js' because it violates the following Content Security Policy directive: "script-src 'self' 'nonce-b828042209ff34a165703bc1a3b338e2'". Note that 'script-src-elem' was not explicitly set, so 'script-src' is used as a fallback.    localhost/:1
```

5 Rate Limiting

I have implemented rate limiting using express-rate-limit library.
This is the default rate-limit for the reader and author route. It prevents a user from exhausting the system's resources.

```
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // Limit each IP to 100 requests
  standardHeaders: true, // Return rate limit info in the `RateLimit-*` headers
  legacyHeaders: false, // Disable the `X-RateLimit-*` headers
});
app.use('/', limiter);
app.use('/author', limiter);
```

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.

```
const authLimiter = rateLimit({
  windowMs: 10 * 60 * 1000, // 10mins
  max: 5, // Limit each IP to 5 POST login/register requests each
  message: 'Too many attempts to register/login',
  standardHeaders: true, // Return rate limit info in the `RateLimit-*` headers
  legacyHeaders: false, // Disable the `X-RateLimit-*` headers
});
```

```
91    router.post('/register', authLimiter ,registerValidation, registerUser);
```

```
router.post(
  '/login',
  authLimiter,
  passport.authenticate('local', {
    failureFlash: true,
  }),
);
```

6 Usage of Asynchronous functions

By default, sqlite db functions are synchronous functions with callbacks. Thus I wrapped them in a promise wrapper. The below code snippet is an example of how I implemented promise in the db.all() function

```
/**
 * added the promise wrapper to make the db.all() function asynchronous
 * @param {String} query sql string statement
 * @param {Array} param an array of data
 * @returns {Array} rows
 */
async function db_all(query, param) {
  return new Promise(function (resolve, reject) {
    db.all(query, param, function (err, rows) {
      if (err) {
        let errorMsg = {
          error: err,
        };
        reject(errorMsg);
      }
      resolve(rows);
    });
  });
}
```

An example of how db_all is used to send query

```
/**
 * Get all articles by author id
 * @param {int} author_id the author int id
 * @returns {Array.<Object>} an array of all article objects
 */
exports.getArticles = async (author_id) => {
  const sql =
    'SELECT * from articles WHERE articles.author_id = ? ORDER BY publish_date DESC';
  const articles = await db_all(sql, [author_id]);
  return articles;
};
```

Snippet of async methods used to get the articles

```
exports.getAuthorHome = async (req, res, next) => {
  if (req.user) {
    try {
      const [author, articles] = await Promise.allSettled([
        db.getAuthor(req.user.id),
        db.getArticles(req.user.id),
      ]);
      // convert date time to correct timezone and format
```

7 Data Validation and Sanitization

SQLite's db functions use parameterized queries to prevent SQL injection.

The express-validator is also installed to help validate the data from user inputs. Express-validator also comes with sanitization methods to ensure the strings cannot be run as code. the code snippet below is an example of how I used express-validator to validate data from blog settings.

```
/**
 * an array of check middleware to validate author_name, blogTitle, blogSubtitle
 * @type  {Array.<check>}
 * @property {module:utils/req-validator~checkAuthorName} checkAuthorName
 * @property {module:utils/req-validator~checkBlogTitle} checkBlogTitle
 * @property {module:utils/req-validator~checkBlogSubtitle} checkBlogSubtitle
 */
exports.settingValidation = [
  checkAuthorName,
  checkBlogTitle,
  checkBlogSubtitle,
];
```

```
router.post('/settings', settingValidation, updateSettings);
```

```
exports.updateSettings = async (req, res) => {
  if (req.user) {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).jsonp(errors.array());
    }
```

This will be the validation error handling

8 Anti-CSRF tokens

I have implemented csrf Synchronizer token using the csrf-sync library. By default, only the POST requests will be protected since they are state-changing requests. The tokens are added to the x-csrf-token header to verify and validate the authenticity of the token before continuing the programme. If the token is invalid or missing, the user request is discarded.

Csrf tokens will be transmitted over the x-csrf-token header

```
const { csrfSync } = require('csrf-sync');

/**
 * csrf Synchronizer token middleware
 * @const
 */
const { csrfSynchronisedProtection } = csrfSync({
  getTokenFromRequest: (req) => {
    return req.headers['x-csrf-token'];
  },
});
```

The token is retrieved using the req.csrfToken() method and added to the ejs

```
  const token = req.csrfToken();
  return res.render('author_home.ejs', {
    title: 'Blog',
    description: 'This is the author home page where he can view his work',
    blogTitle: author.value.blogTitle,
    blogSubtitle: author.value.blogSubtitle,
    author_name: author.value.author_name,
    scriptSrc: ['/scripts/author_home.js'],
    nonce: res.locals.cspNonce,
    articles: articles.value,
    csrfToken: token,
  });
} catch (e) {
```

The csrf tokens are inserted into a hidden input for form submission

```
<input type="hidden" name="CSRFToken" value="<%= csrfToken %>" />
```

The javascript fetch is used to set the header and submit the form

```
  method: 'POST',
  headers: {
    'x-csrf-token': e.target.CSRFToken.value,
    'Content-Type': 'application/json',
  },
```