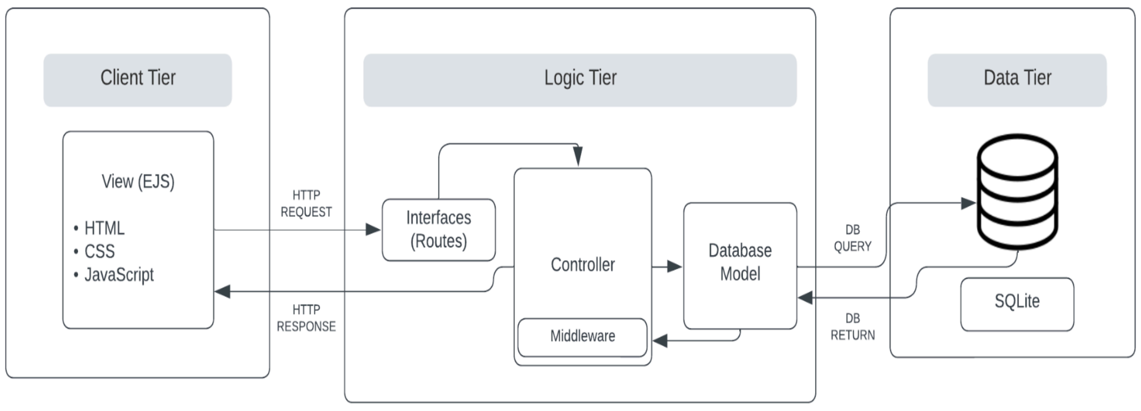
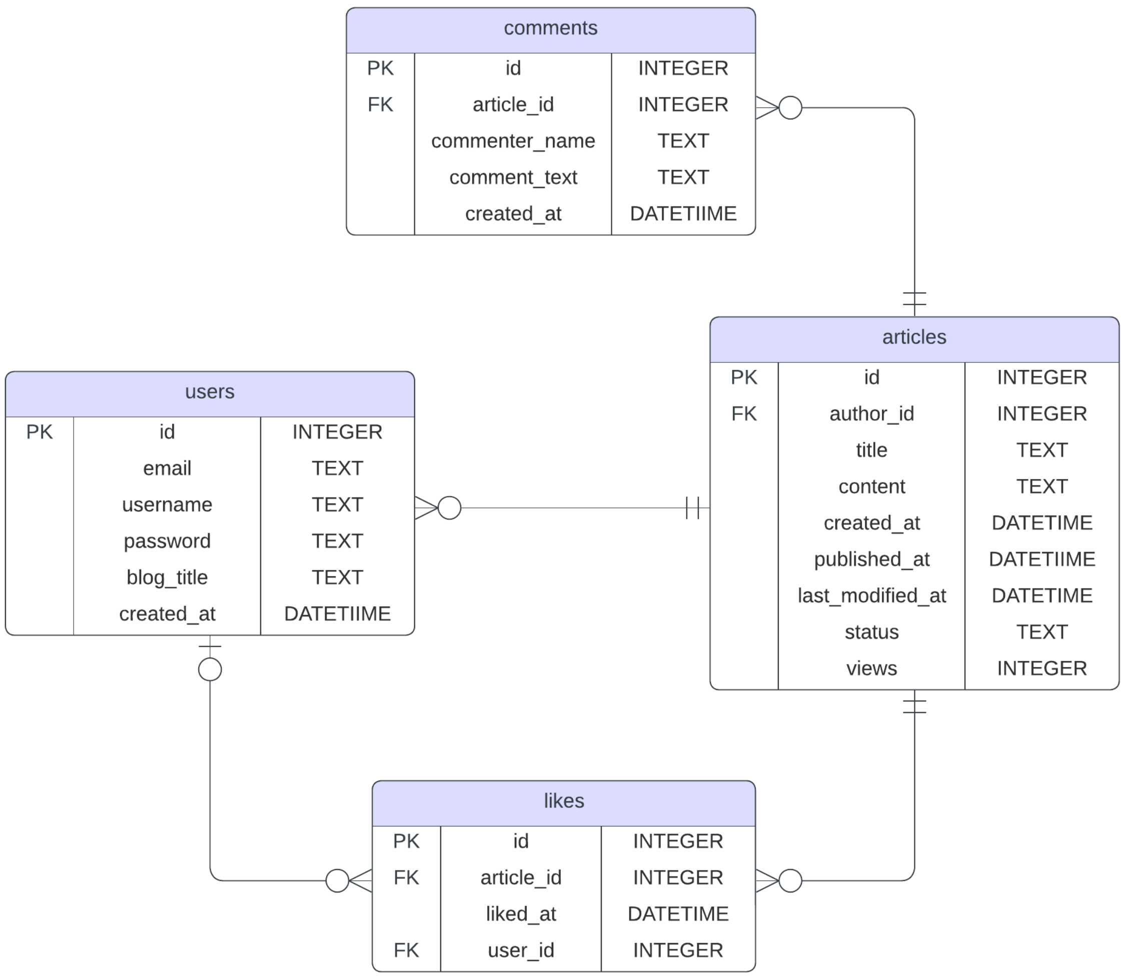
Databases, Network and the Web Coursework for Midterm (Apr 2024 Session)

Schematic Diagram



ER Diagram



comments		
PK	id	INTEGER
FK	article_id	INTEGER
	commenter_name	TEXT
	comment_text	TEXT
	created_at	DATETIIME

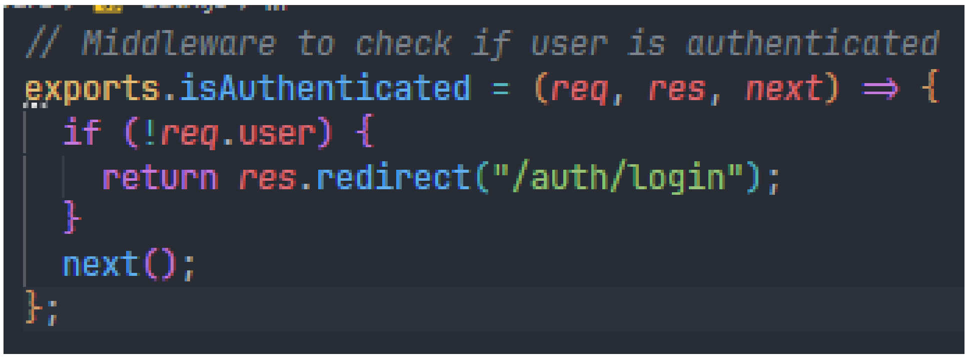
articles		
PK	id	INTEGER
FK	author_id	INTEGER
	title	TEXT
	content	TEXT
	created_at	DATETIME
	published_at	DATETIIME
	last_modified_at	DATETIME
	status	TEXT
	views	INTEGER

users		
PK	id	INTEGER
	email	TEXT
	username	TEXT
	password	TEXT
	blog_title	TEXT
	created_at	DATETIIME

likes		
PK	id	INTEGER
FK	article_id	INTEGER
	liked_at	DATETIME
FK	user_id	INTEGER

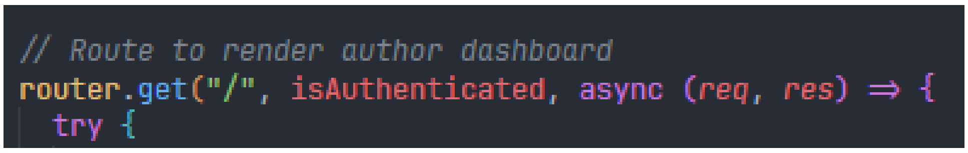
Extensions

1. Password Access
   I implemented user access with email and password and added middleware to check if the user is authenticated to access the author page. For the authentication passport and passport-local to handling the authentication state.
   - Middleware



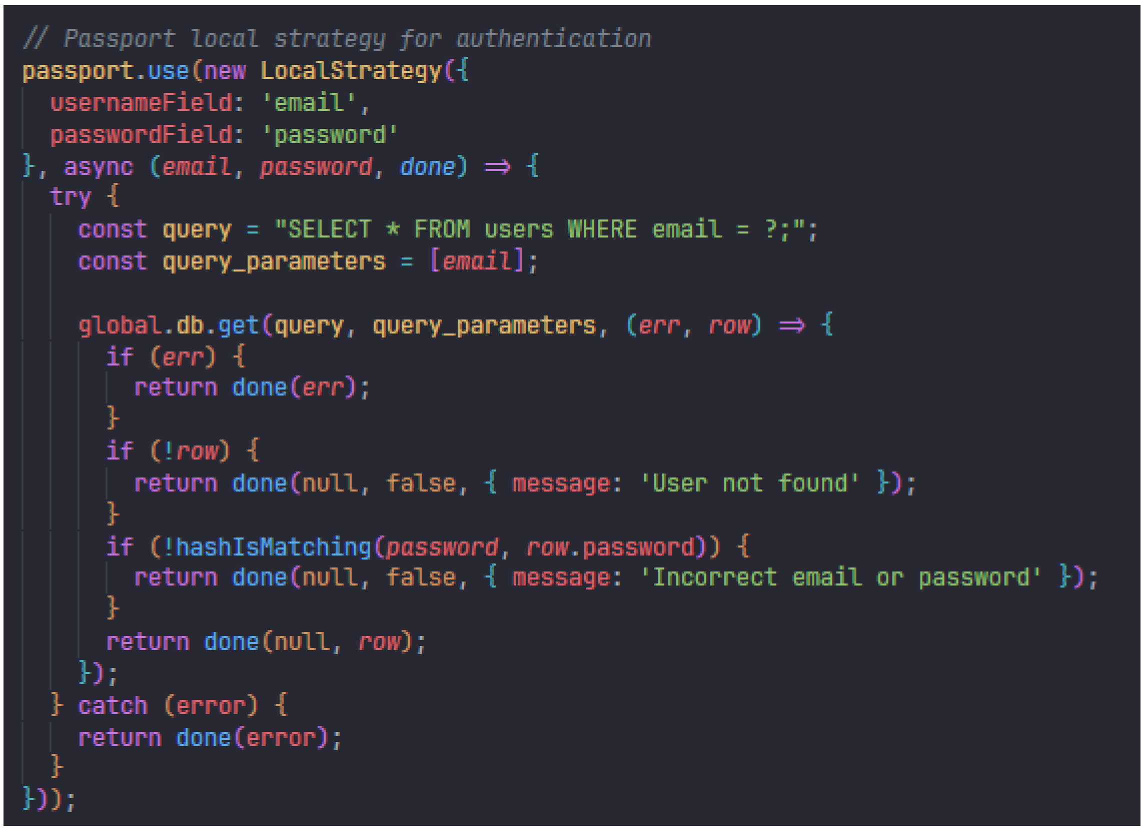
```javascript
// Middleware to check if user is authenticated
exports.isAuthenticated = (req, res, next) => {
  if (!req.user) {
    return res.redirect("/auth/login");
  }
  next();
};
```

   - Middleware implemented in controller



```javascript
// Route to render author dashboard
router.get("/", isAuthenticated, async (req, res) => {
  try {
```

   - Passport local strategy configuration

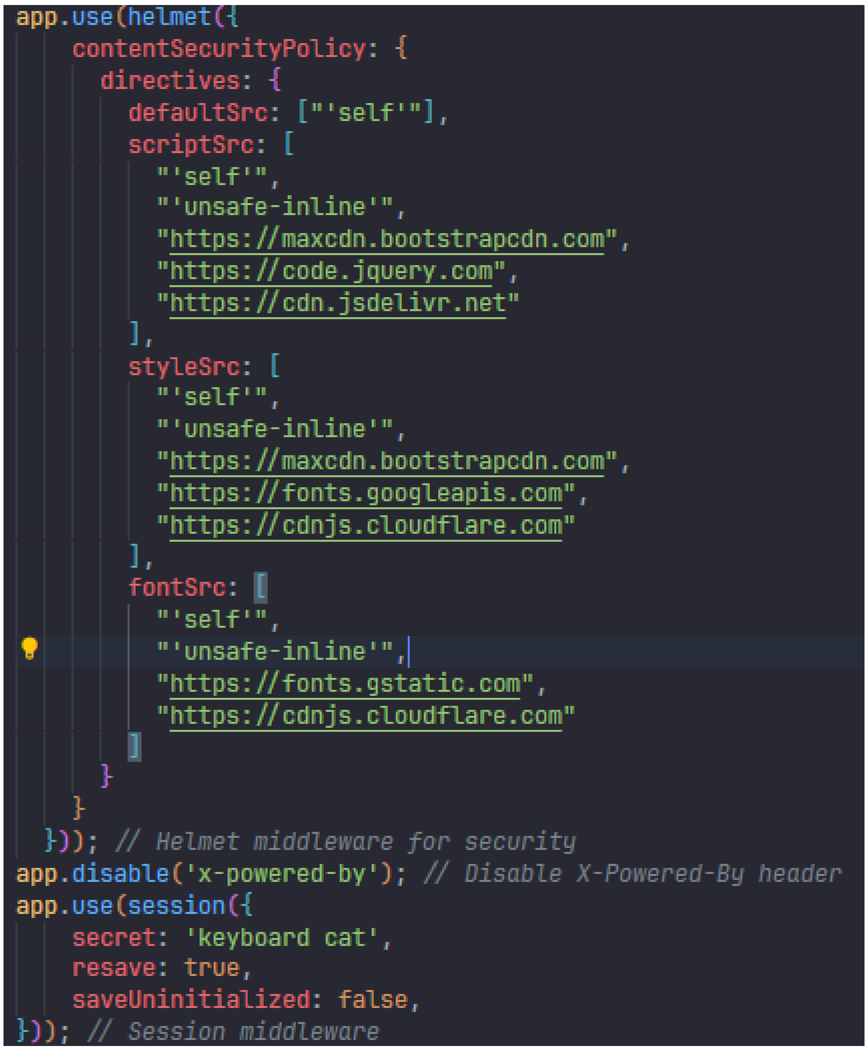


```javascript
// Passport local strategy for authentication
passport.use(new LocalStrategy({
  usernameField: 'email',
  passwordField: 'password'
}, async (email, password, done) => {
  try {
    const query = "SELECT * FROM users WHERE email = ?;";
    const query_parameters = [email];

    global.db.get(query, query_parameters, (err, row) => {
      if (err) {
        return done(err);
      }
      if (!row) {
        return done(null, false, { message: 'User not found' });
      }
      if (!hashIsMatching(password, row.password)) {
        return done(null, false, { message: 'Incorrect email or password' });
      }
      return done(null, row);
    });
  } catch (error) {
    return done(error);
  }
}));
```

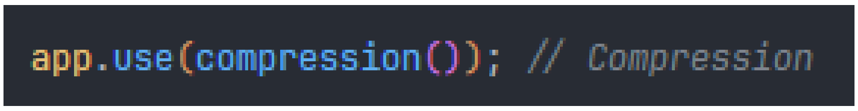
Unauthenticated users still can access an article, create an anonymous comment, but are unable to give a reaction (like). Users password is hashed with bcrypt before stored in the database.

2. Some security best practices
   In this project, I am using a helmet to protect some web vulnerabilities, reduce fingerprinting by disabling X-Powered-By header, and using express-session to save cookie session data.



```
app.use(helmet({
    contentSecurityPolicy: {
      directives: {
        defaultSrc: ["'self'"],
        scriptSrc: [
          "'self'",
          "'unsafe-inline'",
          "https://maxcdn.bootstrapcdn.com",
          "https://code.jquery.com",
          "https://cdn.jsdelivr.net"
        ],
        styleSrc: [
          "'self'",
          "'unsafe-inline'",
          "https://maxcdn.bootstrapcdn.com",
          "https://fonts.googleapis.com",
          "https://cdnjs.cloudflare.com"
        ],
        fontSrc: [
          "'self'",
          "'unsafe-inline'",
          "https://fonts.gstatic.com",
          "https://cdnjs.cloudflare.com"
        ]
      }
    }
  })); // Helmet middleware for security
app.disable('x-powered-by'); // Disable X-Powered-By header
app.use(session({
    secret: 'keyboard cat',
    resave: true,
    saveUninitialized: false,
})); // Session middleware
```

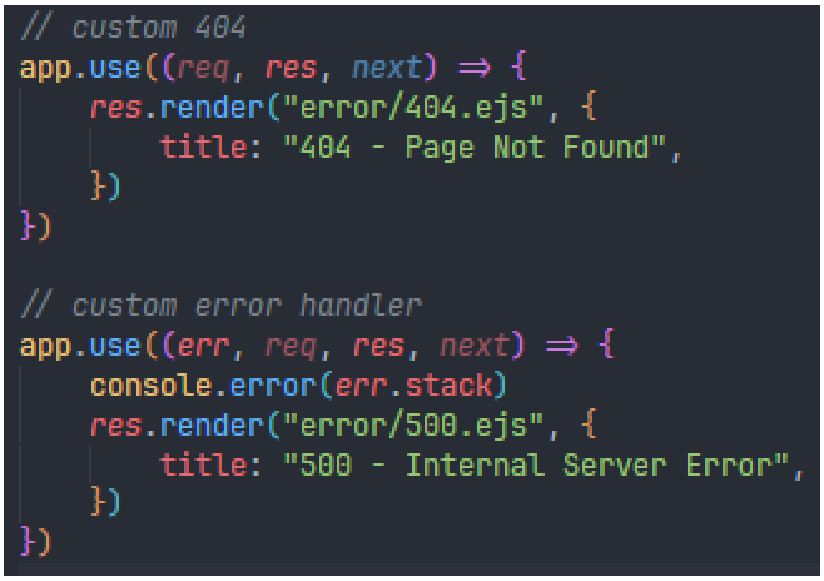
3. Using gzip compression
   Gzip compression is used to decrease the size of the response body to increase the speed of a web app. In real deployed project, using nginx, the compression can be set in the nginx configuration so there is no need to use compression middleware.



```
app.use(compression()); // Compression
```

4. Using custom 404 and 500 (error) handler

I created 2 more pages which are 404.ejs and 500.ejs to show when an error occured.



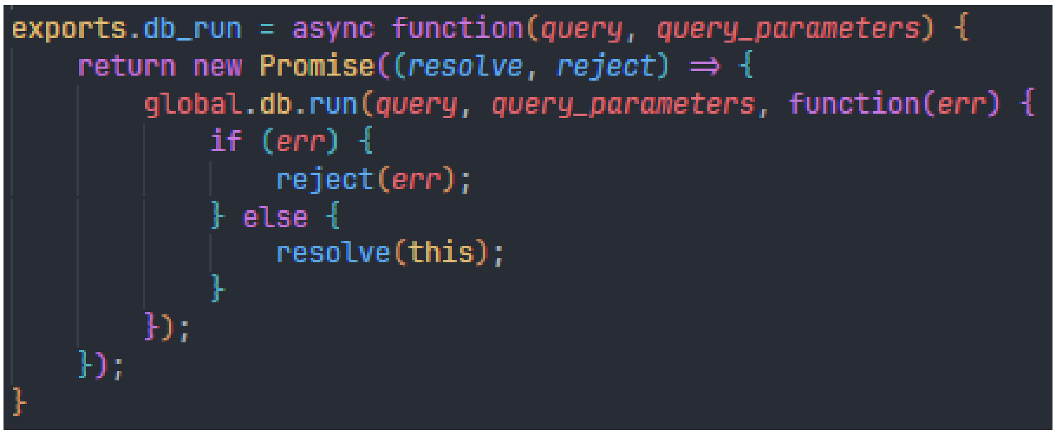
```
// custom 404
app.use((req, res, next) => {
    res.render("error/404.ejs", {
        title: "404 - Page Not Found",
    })
})

// custom error handler
app.use((err, req, res, next) => {
    console.error(err.stack)
    res.render("error/500.ejs", {
        title: "500 - Internal Server Error",
    })
})
```

5. Asynchronous function

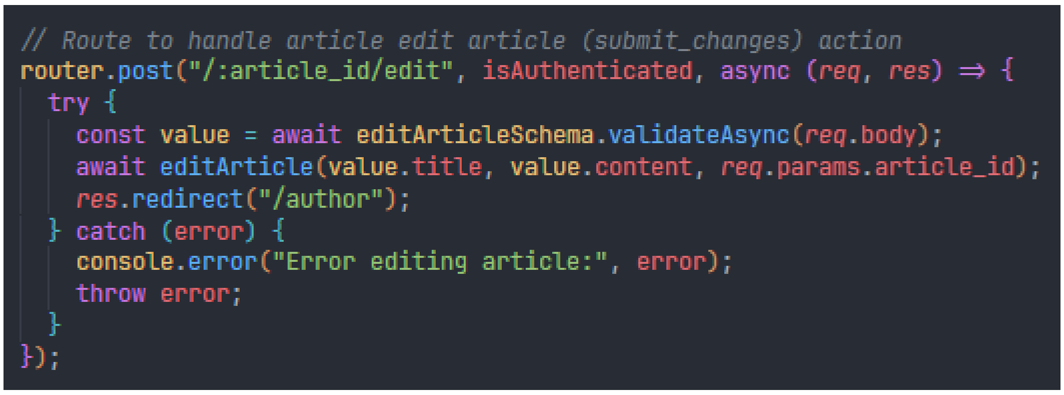
   This can make the code more readable with reducing the then, catch callback. I also implemented try catch for exception handling so the app does not instantly crash when error occurred in the asynchronous or other part.

   - implement async db function



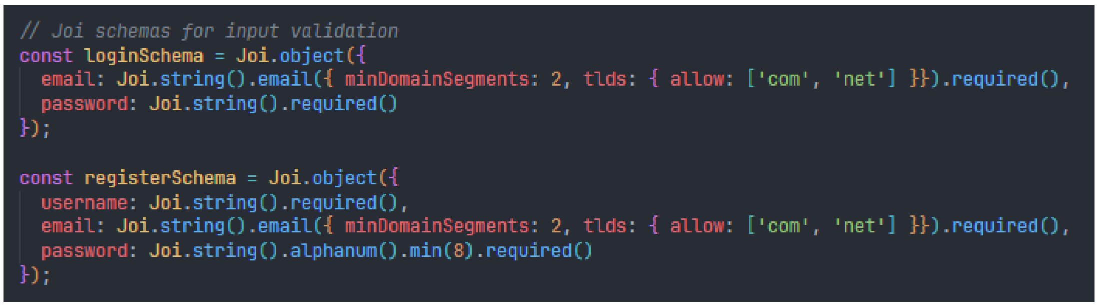
```
exports.db_run = async function(query, query_parameters) {
    return new Promise((resolve, reject) => {
        global.db.run(query, query_parameters, function(err) {
            if (err) {
                reject(err);
            } else {
                resolve(this);
            }
        });
    });
}
```

   - using await function with try catch exception handling



```
// Route to handle article edit article (submit_changes) action
router.post("/:article_id/edit", isAuthenticated, async (req, res) => {
  try {
    const value = await editArticleSchema.validateAsync(req.body);
    await editArticle(value.title, value.content, req.params.article_id);
    res.redirect("/author");
  } catch (error) {
    console.error("Error editing article:", error);
    throw error;
  }
});
```

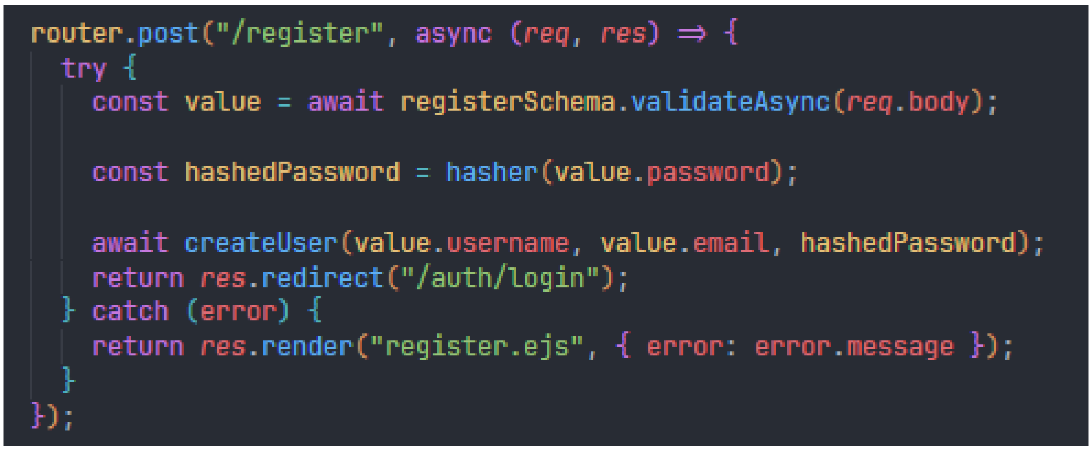
6. Data validation
   I am using Joi library to ensure all the required field is exist and match the type.
   - Schema object creation



```
// Joi schemas for input validation
const loginSchema = Joi.object({
  email: Joi.string().email({ minDomainSegments: 2, tlds: { allow: ['com', 'net'] }}).required(),
  password: Joi.string().required()
});

const registerSchema = Joi.object({
  username: Joi.string().required(),
  email: Joi.string().email({ minDomainSegments: 2, tlds: { allow: ['com', 'net'] }}).required(),
  password: Joi.string().alphanum().min(8).required()
});
```

   - schema validation



```
router.post("/register", async (req, res) => {
  try {
    const value = await registerSchema.validateAsync(req.body);

    const hashedPassword = hasher(value.password);

    await createUser(value.username, value.email, hashedPassword);
    return res.redirect("/auth/login");
  } catch (error) {
    return res.render("register.ejs", { error: error.message });
  }
});
```