CSC7062 Learnlist Report 40418891

Analysis of Design - Software architecture

Introduction

The software architecture of a system is crucial in defining its overall structure and behaviour. It acts as the blueprint for both the system and the project, guiding all stages of development. This section provides an in-depth analysis of the software architecture for my Learnlist web application, exploring its components, design patterns, and how it meets the specified requirements.

Direct Connection Architecture

The Learnlist web app utilises a direct connection architecture. This means that the application connects directly to its database without intermediary API services. This approach simplifies the architecture for me personally.

Web Application Architecture

The Learnlist web app follows a client-server architecture, where the client (browser) communicates with the server to fetch and display data, as well as to send user inputs for processing. Here is an overview of the key components:

Backend Development

Server Framework: The server-side is developed using a robust framework built with Node.js with Express, for the back-end technology. This framework handles routing, request processing, and response generation.

Middleware Logic: The middleware logic layer implements the core functionality of the application, including user authentication, Learnlist management, rating systems, and other features.

Database Connectivity: Direct connections to my SQL database are established to perform CRUD (Create, Read, Update, Delete) operations. The server manages these connections and ensures efficient data retrieval and storage.

Database

Schema Design: The database schema is designed to support the application's dataset requirements, with tables for users, Learnlist, ratings, and other entities. Relationships between tables are defined using foreign keys.

Queries and Indexes: Optimised queries and indexes are implemented to provide the required functionality, particularly for search and sorting operations.

Design Patterns

Model-View-Controller (MVC): The MVC pattern is utilised to separate the user interface from the middleware logic. The Model represents the data, the View displays the data, and the Controller handles the input, ensuring a clear separation of concerns.

Scalability and Performance

The architecture is designed with scalability in mind. By employing modular design principles and efficient database management, the system can handle increased load and user activity. The direct connection architecture eliminates the overhead associated with API calls, thereby improving performance.

Security Considerations

Security is a critical aspect of the software architecture. Various security measures are integrated into the design:

Authentication: The application uses session-based authentication with the help of express-session and cookies. Upon successful login, a session is created and stored in the server, while a cookie is sent to the user's browser. This session ensures that users remain logged in as they navigate through the application. The session management is configured as follows:

```
app.use(
  session({
    secret: process.env.SESSION_SECRET,
    resave: false,
    saveUninitialized: true,
    cookie: { maxAge: 3600000 }, // 1 hour
  })
```

Authorization: The 'isAuthenticated' middleware function checks if the user is authenticated before granting access to certain routes. This ensures that only logged-in users can access sensitive pages or perform specific actions:

```
function isAuthenticated(req, res, next) {
  if (req.session.user) {
    return next();
  }
  res.redirect("/signin"); // redirect to signin if not authenticated
}
```

Password Hashing: The application uses the 'bcryptjs' library to hash passwords before storing them in the database. Hashing converts the password into a unique fixed-length string that is not easily reversible, enhancing security. When a user signs up, their password is hashed using the following code:

```
const hashedPassword = await bcrypt.hash(password, 10);
```

Password Storage: The hashed password is stored in the database, ensuring that even if the database is compromised, the actual passwords remain protected. During the login process, the entered password is compared against the stored hash using 'bcrypt.compare':

```
 const isValidPassword = await bcrypt.compare(password, user.password);
```

Analysis of Design - User interface(s)

Introduction

The user interface (UI) of a web application is a critical component that significantly influences user experience (UX). A well-designed UI not only ensures ease of use but also enhances the aesthetic appeal and overall satisfaction of users.

UI Structure and Design Principles

The Learnlist web application is designed with a user-centric approach, ensuring that the interface is intuitive, visually appealing, and easy to navigate. The following design principles were used during the development of the UI:

1. **Consistency**: Consistent use of colours, fonts, and layouts across the application helps users understand and predict interface behaviour. This consistency extends to the use of standard UI elements like buttons, forms, and navigation menus. This was done by creating only two style sheets that the html's used.
2. **Simplicity**: The design avoids unnecessary complexity, presenting only essential information and controls to the user. This helps in making the interface more user-friendly.

Key Components of the User Interface

1. **All Content Page**: The 'all-content' page serves as the landing page for any users, displaying an overview of the resources that can be found and updated within the database.
2. **Sign Up and Sign In Forms**: These forms are designed to be straightforward and user-friendly, requiring minimal input fields to reduce friction during the registration and login processes. Validation messages and error indicators are included to guide users in case of incorrect inputs.
3. **Learnlist Pages**: Once logged in and authenticated, these pages display detailed information about user learnlists, including resources, ratings, and reviews. The design ensures that information is presented in a clear and organized manner, with easy-to-use controls for editing and managing learnlists.
4. **Responsive Design**: The UI is fully responsive, ensuring that the application is accessible and usable on a variety of devices.

The user interface of the Learnlist web application is designed with a focus on simplicity, usability, and accessibility.

Analysis of Implementation - Process of translating requirements into code

The Learnlist web application was developed to meet specific system requirements that focus on browsing, searching, filtering video tutorial data, user registration, and member-specific functionalities. This section provides an in-depth analysis of how these requirements were translated into code using Node.js, Express, MySQL, and other technologies.

System Requirements and Implementation

Browse & Search the Video Tutorial Data: Users can browse, and search video tutorial data based on various criteria such as subject, topic, and Learnlists. This functionality is implemented in the '/all-content' route, where a dynamic SQL query is constructed based on the user's search parameters. The code snippet to the right demonstrates this:

```
// view all learnlists of a user
app.get('/learnlists', isAuthenticated, (req, res) => {
  const userId = req.session.user.id;
  pool.query(
    `SELECT ul.*, COALESCE(AVG(r.rating), 0) AS average_rating
     FROM user_learnlists ul
     LEFT JOIN reviews r ON ul.id = r.learnlist_id
     WHERE ul.user_id = ?
     GROUP BY ul.id`,
    [userId],
    (err, userLearnlists) => {
      if (err) {
        console.error('Error retrieving learnlists:', err);
        return res.status(500).render('error', { message: 'Error retrieving learnlists.' });
      }

      pool.query(
        `SELECT ul.*, u.username, COALESCE(AVG(r.rating), 0) AS average_rating
         FROM user_learnlists ul
         LEFT JOIN users u ON ul.user_id = u.id
         LEFT JOIN reviews r ON ul.id = r.learnlist_id
         GROUP BY ul.id`,
        (err, allLearnlists) => {
          if (err) {
            console.error('Error retrieving all learnlists:', err);
            return res.status(500).render('error', { message: 'Error retrieving all learnlists.' });
          }
          res.render('learnlists', { userLearnlists, allLearnlists, currentUser: req.session.user });
        }
      );
    }
  );
});
```

Filtering: The filtering capability categorises Learnlists and members, for example, by topic, alphabetical

order, etc. This is accomplished using SQL queries with specific WHERE clauses and sorting options.

User Registration and Authentication: Users can register and log in to the system, a critical requirement. This functionality is achieved through form submissions that handle user data securely using 'bcrypt' for password hashing and 'express-session' for session management.

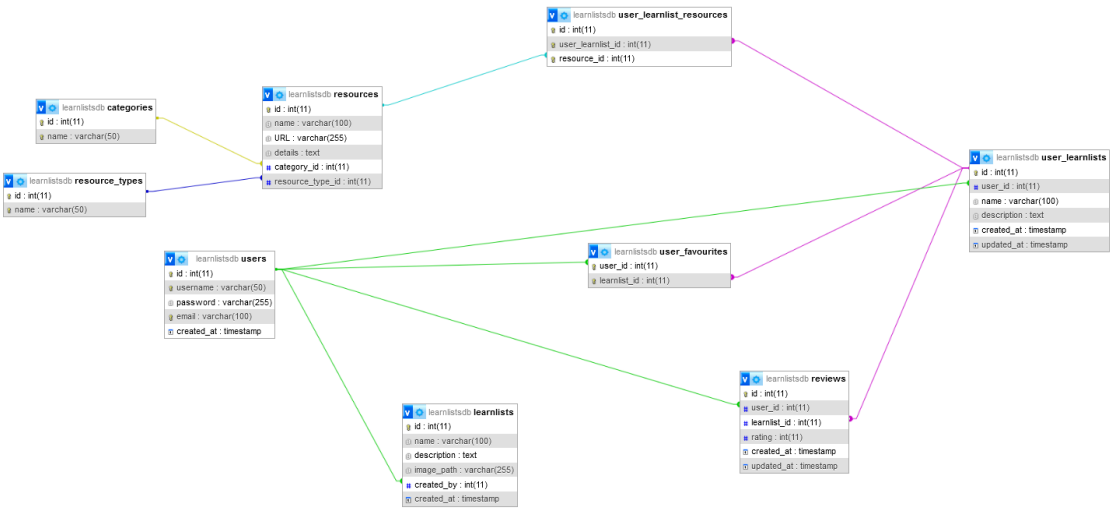
Member-Specific Functions: Registered members can create Learnlists, manage their data, rate other members' Learnlists, and store favourite lists. For example, creating a Learnlist involves inserting data into multiple tables, managing user-learnlist relationships, and handling 'CRUD' operations effectively:

```
// form submission to create a new learnlist
app.post('/learnlists', isAuthenticated, (req, res) => {
  const { name, description, resources } = req.body;
  const userId = req.session.user.id;
  pool.query(
    'INSERT INTO user_learnlists (user_id, name, description) VALUES (?, ?, ?)',
    [userId, name, description],
    (err, result) => {
      if (err) {
        console.error('Error creating learnlist:', err);
        return res.status(500).render('error', { message: 'Error creating learnlist.' });
      }
      const learnlistId = result.insertId;
      if (resources) {
        const resourceIds = Array.isArray(resources) ? resources : [resources];
        const learnlistResources = resourceIds.map(resourceId => [learnlistId, resourceId]);
        pool.query(
          'INSERT INTO user_learnlist_resources (user_learnlist_id, resource_id) VALUES ?',
          [learnlistResources],
          (err, result) => {
            if (err) {
              console.error('Error adding resources to learnlist:', err);
              return res.status(500).render('error', { message: 'Error adding resources to learnlist.' });
            }
            res.redirect('/learnlists');
          }
        );
      } else {
        res.redirect('/learnlists');
      }
    }
  );
});
```

The implementation adheres to best practices such as using '**bcrypt**' for secure password hashing, '**express-session**' for session management, and '**MySQL**' for database interactions. Key references include:

- https://www.npmjs.com/package/bcrypt
- https://expressjs.com/en/starter/basic-routing.html

Analysis of Implementation - Software development approach



I followed a waterfall approach to my project involving sequentially completing tasks and ticking off things I needed to do in order of most importance. I first began by creating my database, you can see an image attached of the designer tab, showing the tables columns and foreign keys linking the tables. Once I had this foundation built, I next moved on to linking the database to my web application by creating a very basic direct connection that was used to add my signup/signin function. This then meant that I could 'post' and 'get' data, to and from the database while running the web application. https://github.com/ecrawford26/learnlist/commit/18e3c793465fea145383f904dda0fcea91e0b9b1

I then continued to add the remaining functionality to my web application that was required over the next few weeks. Once linked to the database and including all backend functionality I was then able to develop the user interface of the web application by designing the frontend, I did this using html pages converted into JavaScript, linked to CSS stylesheets.
https://github.com/ecrawford26/learnlist/commit/c478c8352d31711f7ab06c86e8eb4f7aa0f0fb0c

Analysis of Implementation – Testing

Manual Testing Process

Throughout the development of the Learnlist web application, rigorous manual testing was performed to ensure functionality, usability, and security. This section outlines the manual testing strategies employed and the specific focus areas.

Incremental Testing

Each time new functionality was added to the Learnlist application, it was manually tested to ensure it worked as expected. This incremental approach involved functionality verification; After implementing a new feature, such as user registration or learnlist creation, I manually tested it by interacting with the application as a user would. This included entering data, submitting forms, and verifying that the expected results were displayed.

Security was a critical aspect of testing, particularly ensuring that only authenticated users could access certain features. Key security testing activities included:

1. **Authentication Testing**: I manually tested the login and logout processes to verify that sessions were correctly created and destroyed. This involved checking that cookies were set appropriately, and that the session persisted across different pages.
2. **Authorization Testing**: I attempted to access restricted pages without being logged in to ensure that the application correctly redirected unauthenticated users to the login page. Additionally, I tested accessing pages with different user roles to verify that permissions were enforced accurately.
3. **Session Management**: I checked that sessions expired after a certain period of inactivity and that users were required to log in again. This involved setting session timeouts and verifying that users were redirected to the login page after their session expired.
4. **Password Security**: I ensured that passwords were securely hashed and stored by checking the database entries. Using the ‘bcryptjs’ library, I verified that passwords were hashed before storage and that the hash was used for authentication during login.

```
const hashedPassword = await bcrypt.hash(password, 10);
const isValidPassword = await bcrypt.compare(password, user.password);
```

Dependencies:

```
{
  "dependencies": {
    "axios": "^1.7.2",
    "bcrypt": "^5.1.1",
    "bcryptjs": "^2.4.3",
    "cookie-parser": "^1.4.6",
    "dotenv": "^16.4.5",
    "ejs": "^3.1.10",
    "express": "^4.19.2",
    "express-session": "^1.18.0",
    "mysql": "^2.18.1",
    "mysql2": "^3.10.0"
  }
}
```

External Sources:

- https://getbootstrap.com/docs/5.3/getting-started/introduction/
- https://www.npmjs.com/package/body-parser
- https://www.npmjs.com/package/express-session
- https://developer.mozilla.org/en-US/docs/Web/API/Document/DOMContentLoaded_event
- https://csc7062-2023-24.gitbook.io/lab10
- https://csc7062-2023-24.gitbook.io/lab08
- https://csc7062-2023-24.gitbook.io/lab-01
- https://csc7062-2023-24.gitbook.io/lab-02
- https://www.npmjs.com/package/bcrypt
- https://expressjs.com/en/starter/basic-routing.html