# Contents

|  |  |
| --- | --- |
| **Analysis of the problem** | 2 |
| Description of the problem | 2 |
| UML use case diagram | 4 |
| Requirements specification | 5 |
| Project plan | 6 |
| **Design of the solution** | 8 |
| Design of Advanced Higher concepts | 8 |
| Design of Integration | 10 |
| User-interface design | 16 |
| **Implementation of the solution** | 21 |
| New skills researched and developed | 21 |
| Ongoing testing and issues resolved | 23 |
| **Testing the solution** | 24 |
| Advanced Higher algorithms | 24 |
| Testing site functionality | 27 |
| Results | 30 |
| **Evaluation of the solution** | 31 |
| Meeting requirements specification | 31 |
| Results of testing | 31 |
| Future maintainability | 32 |
| Robustness | 32 |
| **Appendices** | 33 |
| Site screenshots | 33 |
| Code screenshots | 38 |
| End of project report | 92 |

# Analysis of the problem

## Description of the problem

My project idea is for a basic microblogging platform. The end users of the application will be other students in the Computing Science class or club, to allow them to share thoughts and ideas, and post updates to projects they’re working on.

The platform will be a website with a frontend written in HTML and CSS for their compatibility with nearly all devices, and a backend written in Luau (v0.606), as it’s the language I’m most familiar with (and my personal favourite), alongside the Lune (v0.8.0) runtime, analagous to Luvit for Lua, or Node.js/Deno/Bun for Javascript. It will interface with a MySQL (v8.0) database to store user data, sessions, and posts.

The platform will relate to the Software Design and Development area of the course, as it uses the three specified algorithms: bubble sort, insertion sort, and binary search. It could also integrate with the Database Design and Development area of the course, as it uses an SQL database with three tables and opens a connection to send queries to it, as well as the Web Design and Development area of the course, as it uses a web user interface to display the data.

### Scope

The scope of the project will include:

* A full design of the site, including database schema, user interface wireframes, UML diagrams, query designs, and pseudocode for all used algorithms.
* A full implementation of the site, including database, backend, and frontend.
* A completed table of manual and unit tests, including test cases and test results.
* Evaluation of the project, including a discussion of the project’s strengths and weaknesses, and a reflection on the development process.

### Boundaries

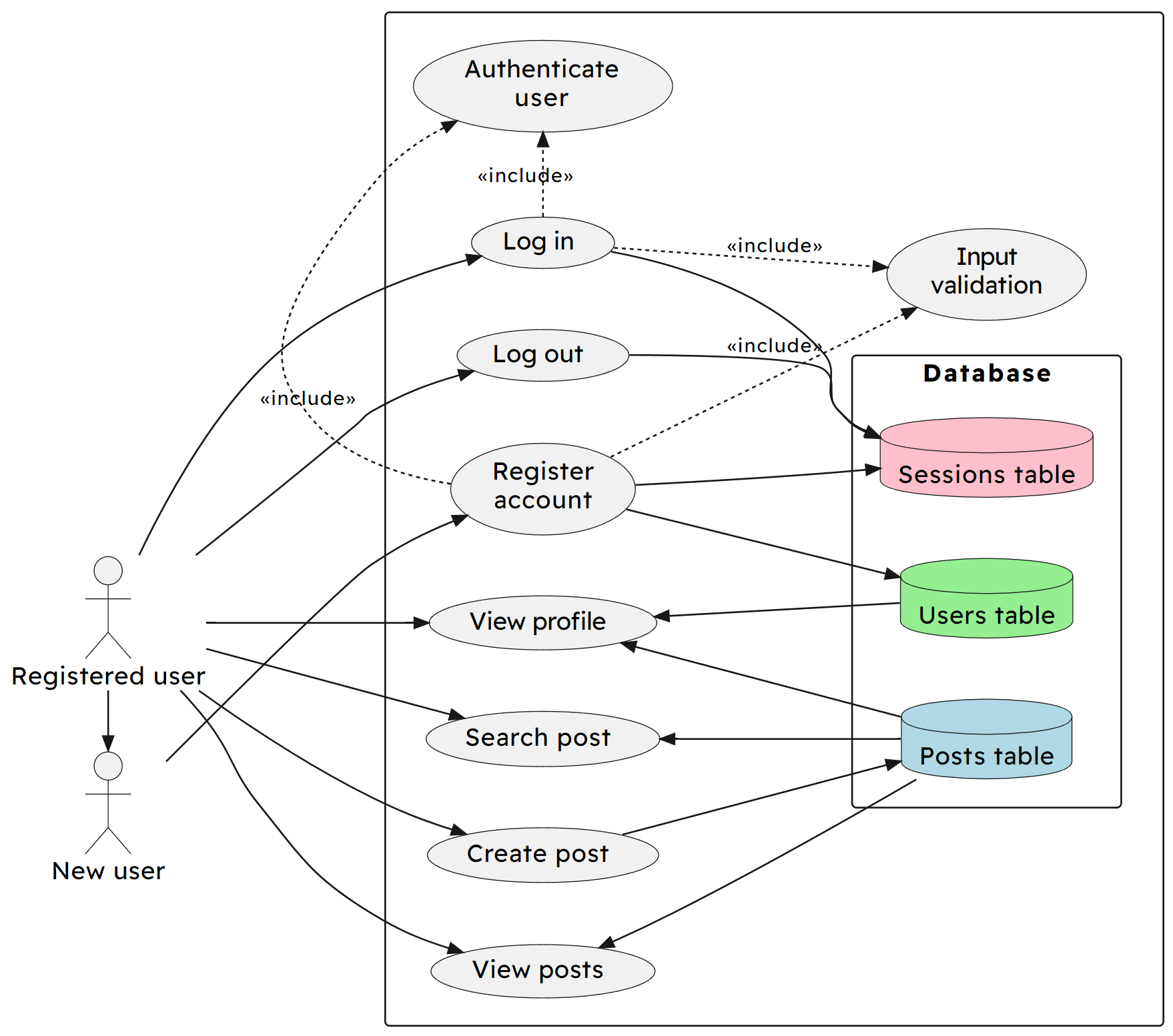
The full implementation of the site will include:

* A registration system, allowing users to create an account by providing a username and password, and storing this information in a database.
* A login system, allowing users to log in to their account by providing their username and password, and checking this information against the database.
* A session system, allowing users to remain logged in for a period of time, and to log out of their account by deleting their session from the database.
* A post system, allowing users to create posts, view all posts, view posts by a specific user, and search for posts by keyword by using binary search.
* Validation of all user input, to prevent attacks such as cross-site scripting (XSS) or SQL injection.
* Ability to sort posts by date from newest to oldest, using bubble sort and insertion sort.

### Constraints

Time - Since beginning the project on 23 November 2023, I have between 4 and 5 months to complete the project before it should be submitted by 15 April 2024.  
Cost - The project will have no development cost, as it will be using entirely free and open-source software.  
Technical - The main logic for the site (backend) will be written in Luau, as it’s the language I’m most familiar with. Legal - The project will not store any personal data other than usernames and passwords, so should comply with all relevant data protection laws including the DPA and GDPR, and must also comply with the Copyright, Designs and Patents Act.

## UML use case diagram



## 

## Requirements specification

The project’s purpose is for users to be able to make and view posts on a microblogging platform after registering and logging in.

### Functional requirements

* The system shall store user data, sessions, and posts in a database.
* The site will use a navigation bar to allow users to easily navigate between home, profile, search, and logout pages.
* All user input will be validated to prevent attacks that attempt to break the site’s database or display mechanisms.
* When registering an account, the system will store the user’s username and password in the database.
* When logging in, the system will check the user’s username and password against the database and create a session if the details are correct.

These requirements need to be met for the project to function properly.

### End-user requirements

* A search function for finding posts by their content.
* A homepage for viewing recent posts by all users in reverse chronological order (newest first).
* A profile page for viewing recent posts by a specific user.
* Easy navigation between all pages.

These requirements should be met to allow for good user experience in the final site.

## 

## Project plan

### Analysis (9 days)

* Creation of project ideas: 1 day
* Implementation of basic prototypes, testing for suitability: 2.5 days
* Selection of final project idea: 4 hours
* Layout of scope, boundaries, and constraints: 2 days
* Creation of UML use case diagram: 1 day
* Creation of end-user requirements: 4 hours
* Creation of functional requirements: 4 hours
* Creation of project plan: 1 day

### Design (1 week)

* Basic wireframe designs: 2 days
* Design notation for program processes (pseudocode): 3 days
* Design of database schema: 2 days

### Implementation (3 weeks)

* Architecture of broad project structure: 2 days
* Implementation of user interface: 5 days
* Implementation of backend processes: 6 days
* Inclusion of algorithms - bubble sort, insertion sort, binary search: 3 days
* Implementation of database systems: 4 days
* Write-up and research of new skills and concepts: 1 day

### Testing (9 days)

* Creation of test plan: 2 days
* Creation of unit tests and cases: 3 days
* Carry out tests: 4 hours
* Description of problems found during testing: 1.5 days
* Debugging and fixing of discovered problems: 2 days

### Evaluation (2 days)

* Evaluation of project: 1 day
* Discussion of project strengths and weaknesses: 4 hours
* Discussion of improvements that could be made: 4 hours

### Finalisation (4.5 days)

* Finalisation of analysis: 4 hours
* Finalisation of design: 4 hours
* Finalisation of implementation: 4 hours
* Finalisation of testing: 4 hours
* Finalisation of evaluation: 4 hours
* Collection of evidence: 4 hours
* Formatting of evidence write-up: 1 day
* Submission of evidence: 4 hours

# Design of the solution

## Design of Advanced Higher concepts

Note that all array indices are 1-based in Luau, so the algorithms are designed with this in mind.

### Bubble sort

Set list to the input list  
Set swapped to true  
Set n to the length of the list  
While swapped is true:  
 Set swapped to false  
 For i from 1 to n - 1:  
 If list[i] is greater than list[i + 1]:  
 Swap list[i] and list[i + 1]  
 Set swapped to true  
 Subtract 1 from n  
Return list

### Insertion sort

Set list to the input list  
For i from 2 to the length of the list:  
 Set value to list[i]  
 Set index to i  
 While index is greater than 1 and value is less than list[index - 1]:  
 Set list[index] to list[index - 1]  
 Subtract 1 from index  
 Set list[index] to value  
Return list

### Binary search

Set list to the input list  
Set value to the input value to find  
Set left to 1  
Set right to the length of the list  
While left is less than or equal to right:  
 Set mid to the floor of (left + right) / 2  
 If list[mid] is equal to value:  
 Return mid  
 Else if list[mid] is less than value:  
 Set left to mid + 1  
 Else:  
 Set right to mid - 1  
Return -1, signifying that the value was not found

## Design of integration

### Query design

#### Registration query

Insert: username, password  
Table: users  
Values: validated inputs from registration form

Insert: id, username Table: sessions Values: generated session token, validated input username

#### Login query

Select: Just check if the user exists (select 1)  
Table: users  
Where: username matches the validated input username and password matches the validated input password

#### Logout query

Delete: sessions  
Where: id matches the session token in the cookie

#### Authentication query

Select: Just check if the session exists (select 1)  
Table: sessions  
Where: id matches the session token in the cookie

#### Homepage query

Select: post id, post time, post content, user username  
Table: post  
Join: user on post username = user username

#### Create post query

Insert: post content, user username  
Table: post  
Join: session where user username = session username  
Values: input from post form, username matched from session token in the cookie

#### 

#### Profile query

Select: post id, post time, post content, user username  
Table: post  
Join: user where post username = user username  
Where: user username matches the username provided in the URL

#### Search query

Select: post id, post time, post content, user username Table: post Join: user where post username = user username

Result of all posts will be searched with binary search

### Database design

All fields are required and not null on all tables.

#### User table

Username: varchar(255) (primary key) Password: varchar(255)

#### Session table

Id: varchar(255) (primary key) Username: varchar(255) (foreign key referencing user username)

#### Post table

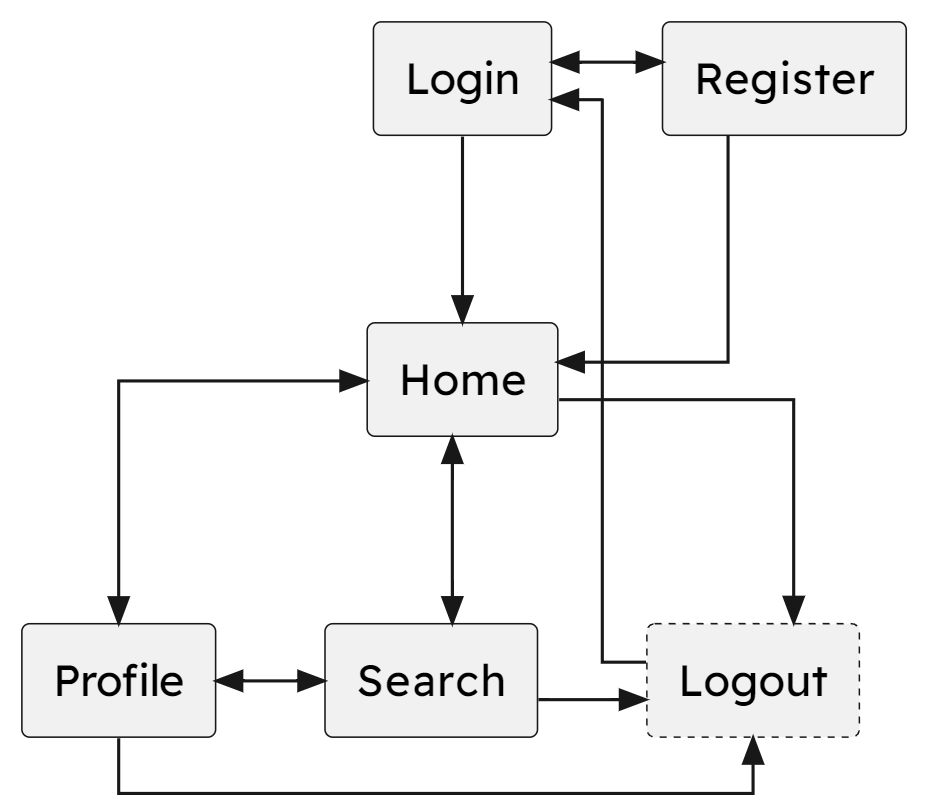
Id: int (primary key, auto-increment) Time: datetime (default to current time) Content: text Username: varchar(255) (foreign key referencing user username)

#### Entity-relationship diagram

A diagram of a user

Description automatically generated

### Design of website hierarchy



Accessing the logout page will immediately delete the user’s session and redirect them to the login page.

### Server-side processes

#### Database connection

Proxy server:  
 Connect to database using connection variables  
 Start a server to listen for incoming requests  
 When a request is received:  
 Run the body as a query on the database  
 If query or connection fails:  
 Return an error message  
 Return the result of the query  
  
Website:  
 Connect to proxy server  
 If connection fails:  
 Return an error message  
 When the "query" function is called:  
 Send the request to the proxy server  
 If the request errors:  
 Return an error message  
 Return the rows of the result

#### User login

Get the submitted username and password  
If either is empty:  
 Return an error message  
Query to find the user with the given username and password  
If the user does not exist:  
 Return an error message  
Generate a new session token  
Create a new session with the user's username and the session token  
Return a redirect to the homepage with a "set-cookie" header containing the session token

#### User registration

Get the submitted username and password  
If either is empty:  
 Return an error message  
Query to find the user with the given username  
If the user already exists:  
 Return an error message  
Generate a new session token  
Query to create a new user with the given username and password  
Query to create a new session with the user's username and the session token  
Return a redirect to the homepage with a "set-cookie" header containing the session token

#### User logout

Get the cookie from the request  
Get the session token from the cookie  
If the session token exists:  
 Query to delete the session with the given session token  
Return a redirect to the login page

#### Authentication

Get the cookie from the request  
Get the session token from the cookie  
If the session token exists:  
 Query to find the session with the given session token  
If the session exists and the user is trying to access the register or login page:  
 Redirect the user to the homepage  
Else if the session does not exist and the user is trying to access a page that requires authentication:  
 Redirect the user to the login page  
Else  
 Display the requested page

#### Post creation

Authenticate the user  
Get the body from the request  
Parse the request body to find the post content  
If the post content is empty:  
 Return an error message

## User-interface design

### Homepage

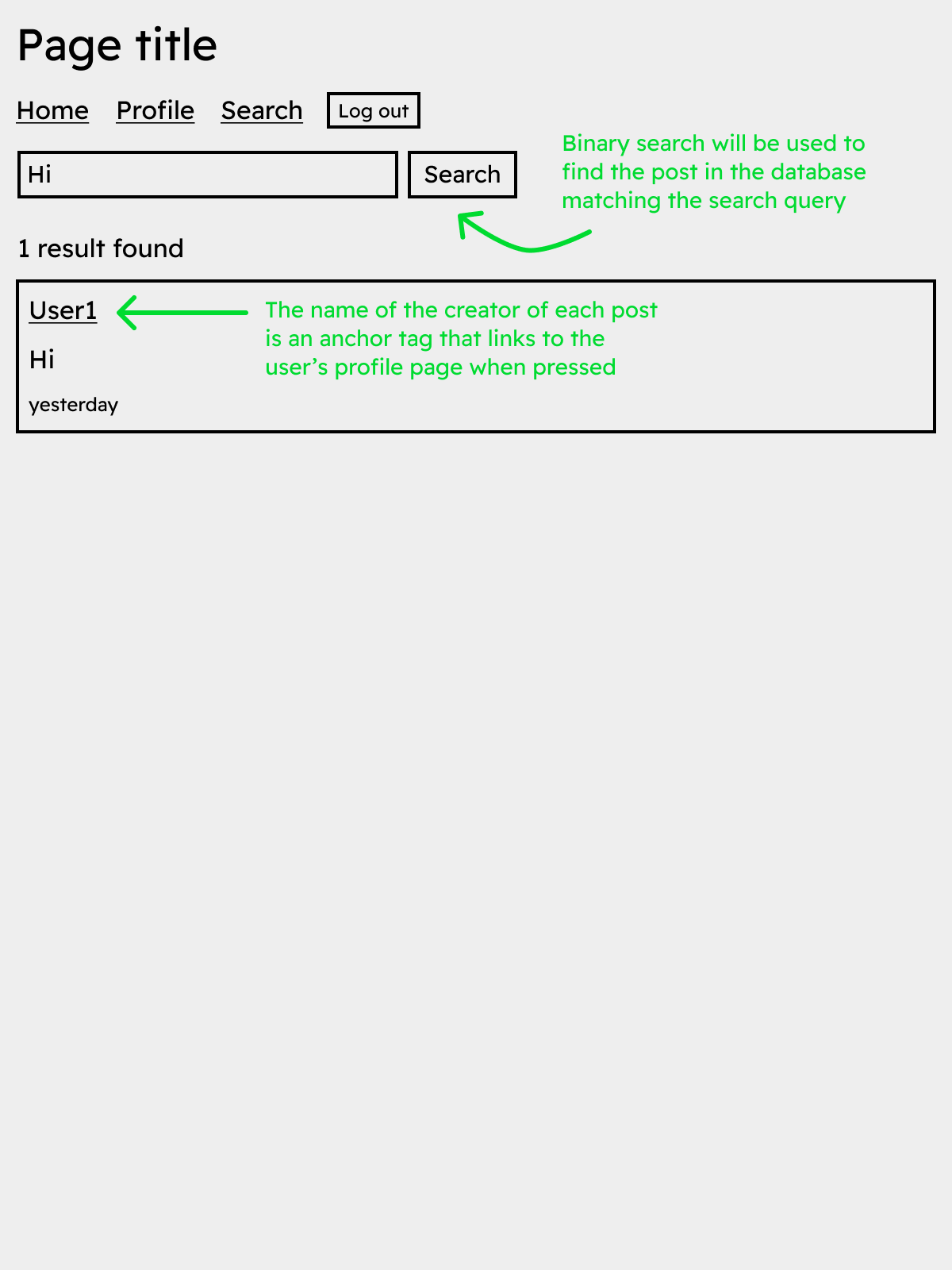
A screenshot of a computer

Description automatically generated

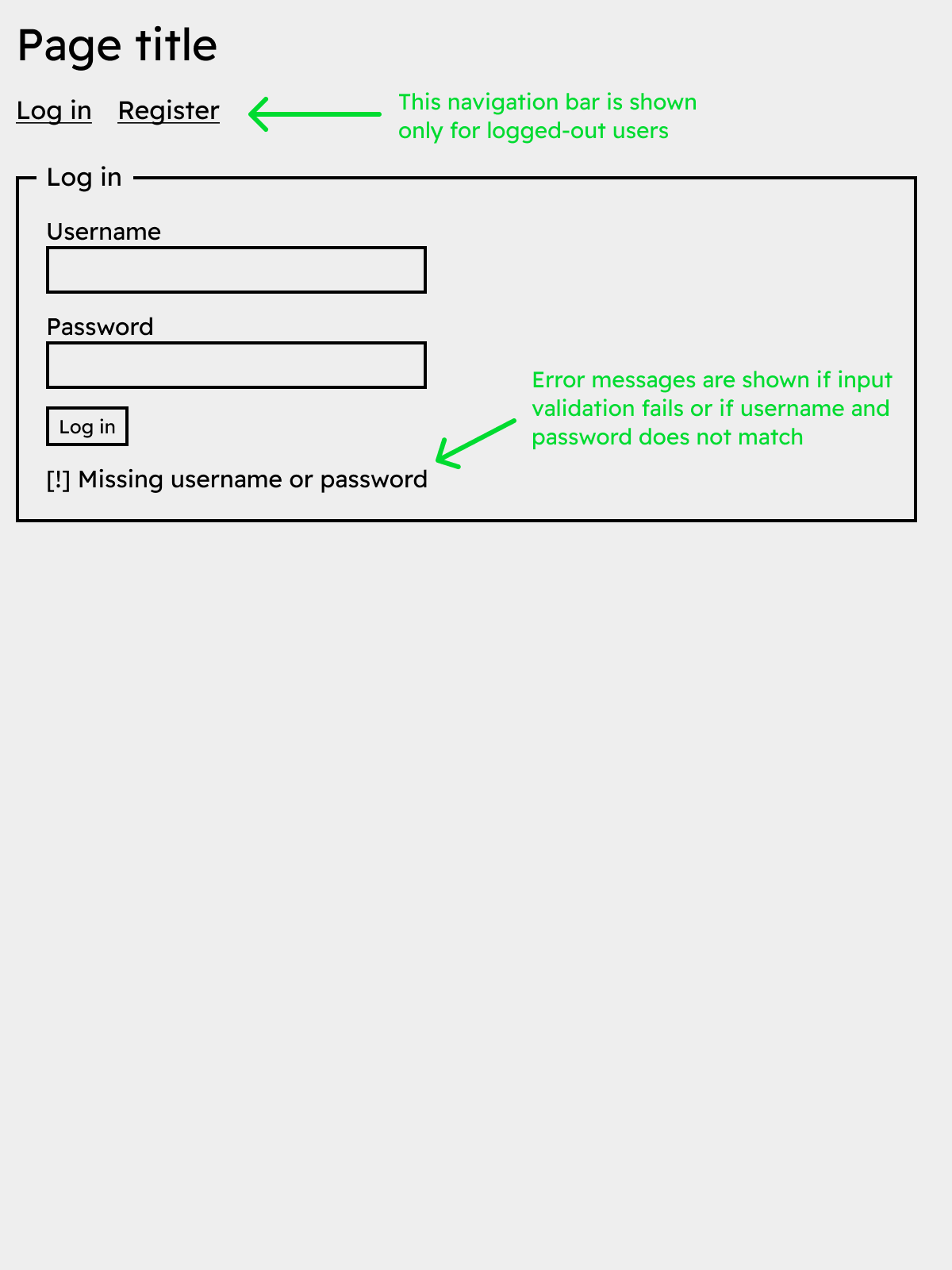
### Profile



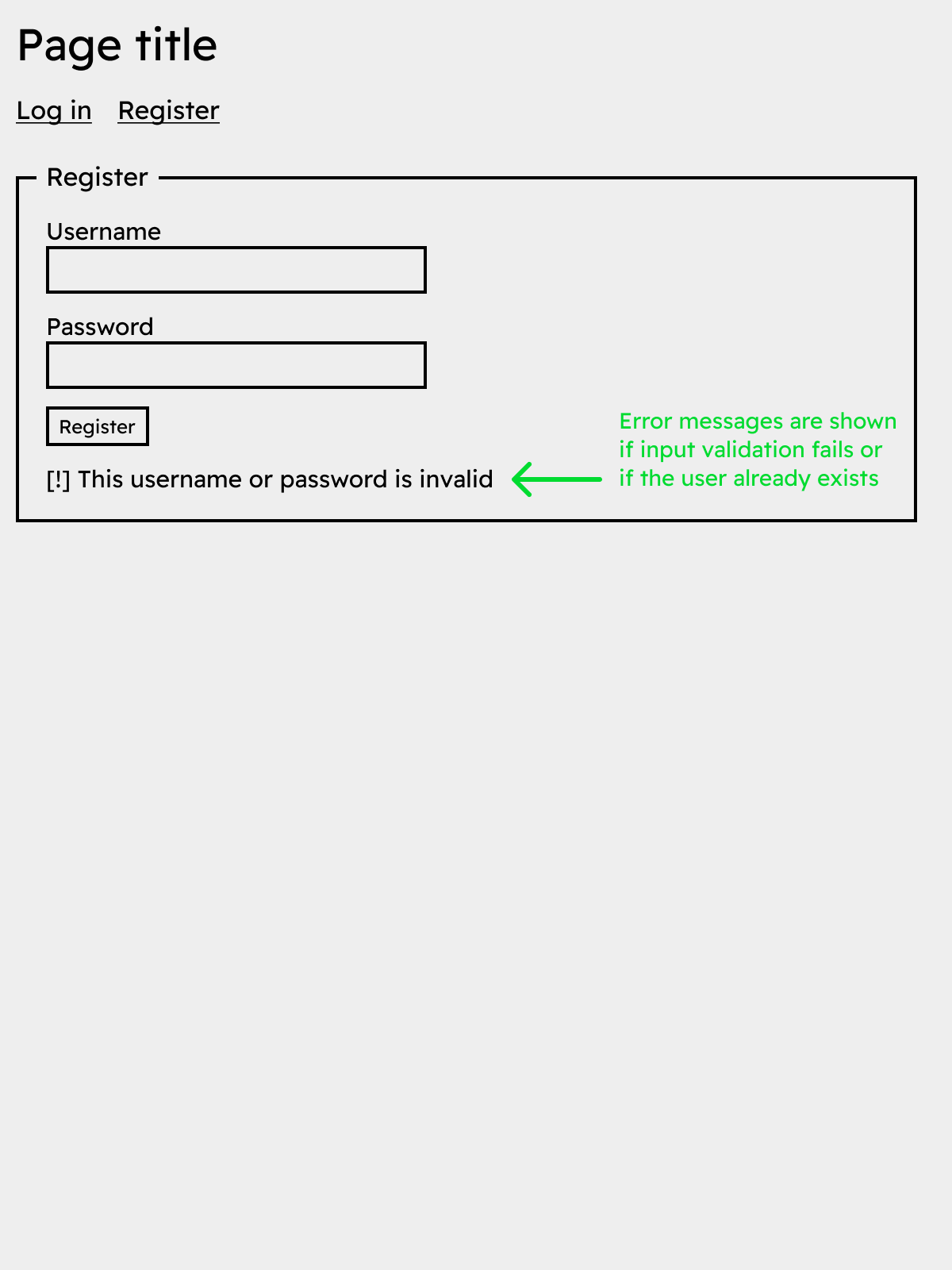
### Search



### Login



### Register



# Implementation of the solution

## New skills researched and developed

For a working implementation of the finished application, I had to research and apply some skills that were not part of the Advanced Higher course.

Unlike some languages or server-side frameworks like PHP, Python, or Ruby on Rails, the Luau language, due to its sandboxed nature, does not provide built-in functions or support for dealing with HTTP requests or interacting with the filesystem. To compensate for this, the Lune runtime provides `net` and `fs` modules, but these modules have limited ability to create the necessary server-side functionality on their own.

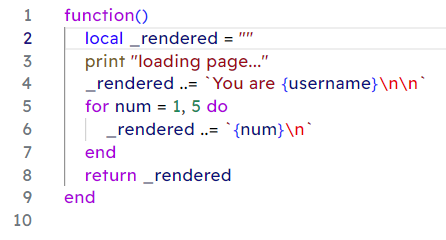
For the backend of the application, I had to implement a custom filesystem-based router that transforms pages from the `routes` directory into functions that can be called by requests to the server. The `public` directory holds miscellaneous files like icons and stylesheets which are loaded by the browser when the user visits the site.

For the frontend, a parser for a basic templating language was created. The parser takes advantage of Luau’s string interpolation functions, allowing me to write Luau and HTML in the same file (these files are specified with a `.ltmp` extension in the project repository), and turning it into a function that can be called and returned as a string by the server. It transforms the following template:

A screenshot of a computer code

Description automatically generated

into the following function:



If future performance or lower resource usage is required in the future, these frontend and backend functions could be trivially loaded as coroutines and ran concurrently on the same thread or utilising multiple threads through the use of actors.

For the database, a very simple proxy server was used to handle incoming requests from the website and run them as queries on the database. It was written in Typescript, and was required because Luau does not have built-in support for MySQL or other database systems.

I have used systems like server-side routing, templating, and database connections in other projects, but this was the first time I had to implement them from scratch in a language that did not have built-in support for them. This was a valuable learning experience, giving me a better understanding and insight into the inner workings of these systems.

Another skill developed was during the implementation of the Advanced Higher algorithms required, namely bubble sort, insertion sort, and binary search. These algorithms were originally designed to be implemented only for number types, though as the system deals with lots of text, they needed to be modified to work with string types as well. Due to the Luau language naturally overloading the greater-than and less-than operators to compare strings based on alphabetical ordering, the logic for comparing strings did not need to change.

However, this changed the types of data each algorithm could take as arguments and return, meaning they no longer had correct type annotations. To fix this, I had to research and implement generic types, which are natively supported in Luau. This allowed the algorithms to be written to accept any supported type and return the same type, which made them more flexible and reusable.

## Ongoing testing and issues resolved

During testing, it was found that it needed to support page actions, allowing the same page to accept requests of multiple types, such as GET and POST for pages with forms. This was implemented by adding a `page.actions` field to the table returned by every page and allowing the router to call the correct action based on a query parameter in the request, otherwise using the `default` action.

A second issue that occurred was due to the same code for posts being reused on many pages across the site, however the code couldn’t be easily extracted into a function because of the frontend using a templating system. To solve this, the templating system was modified with a component system, exposed as a Load function, that allowed template files in the `/components` directory to be loaded and reused in other templates on multiple pages, increasing modularity and reducing code duplication.

Another issue was the potential usability or performance problems when loading huge lists of posts. To solve this, posts were cut to a maximum of 10 on the home and profile pages, though the search system can still be used to find older posts. Extremely long usernames would also cause issues with storing them in the database and displaying them on the site, so a username length of between 3 and 20 characters was enforced.



A final issue was the lack of syntax highlighting for custom Luau templating files in the text editor used to write the code, Visual Studio Code. This was solved by creating a custom syntax highlighting extension for the .ltmp file extension, which was simple as it only had to embed existing Luau and HTML syntax highlighting, delimited by the `{# #}` and `{ }` characters.

# Testing the solution

Both a test table and unit tests were created to ensure the solution was working as intended. Unit tests for Advanced Higher algorithms were stored in the `test.luau` file, and the corresponding test table is displayed below. All tests were repeated using normal, extreme, and exceptional test data where applicable to ensure the system was robust and could handle a variety of inputs.

## Advanced Higher algorithms

### Testing bubble sort

Type: Normal  
Test data: { 4, 6, 3, 8, 2, 7, 5, 9, 1 }  
Expected result: { 1, 2, 3, 4, 5, 6, 7, 8, 9 }  
Actual result: { 1, 2, 3, 4, 5, 6, 7, 8, 9 }  
Outcome: Pass

Type: Extreme  
Test data: { 1, 2, 3, 4, 5, 6, 7, 8, 9 }  
Expected result: { 1, 2, 3, 4, 5, 6, 7, 8, 9 }  
Actual result: { 1, 2, 3, 4, 5, 6, 7, 8, 9 }  
Outcome: Pass

Type: Extreme  
Test data: { }  
Expected result: { }  
Actual result: { }  
Outcome: Pass

Type: Exceptional  
Test data: { “string”, 6, 3, 8, 2, 7, 5, 9, 1 }  
Expected result: attempt to compare string < number (error message)  
Actual result: attempt to compare string < number  
Outcome: Pass

### Testing insertion sort

Type: Normal  
Test data: { 3, 6, 5, 7, 4, 9, 2, 1, 8 }  
Expected result: { 1, 2, 3, 4, 5, 6, 7, 8, 9 }  
Actual result: { 1, 2, 3, 4, 5, 6, 7, 8, 9 }  
Outcome: Pass

Type: Extreme  
Test data: { 1, 2, 3, 4, 5, 6, 7, 8, 9 }  
Expected result: { 1, 2, 3, 4, 5, 6, 7, 8, 9 }  
Actual result: { 1, 2, 3, 4, 5, 6, 7, 8, 9 }  
Outcome: Pass

Type: Extreme  
Test data: { }  
Expected result: { }  
Actual result: { }  
Outcome: Pass

Type: Exceptional  
Test data: { “string”, 6, 3, 8, 2, 7, 5, 9, 1 }  
Expected result: attempt to compare string < number (error message)  
Actual result: attempt to compare string < number  
Outcome: Pass

### Testing binary search

Type: Normal  
Test data: { 7, 4, 6, 8, 1, 3, 5, 2, 9 }, 5  
Expected result: 7  
Actual result: 7  
Outcome: Pass

Type: Extreme  
Test data: { 5, 5, 5, 5, 5, 5, 5, 5, 5 }, 5  
Expected result: 1 (the first occurrence of the value)  
Actual result: 5  
Outcome: Fail

Type: Extreme  
Test data: { }, 1  
Expected result: -1  
Actual result: -1  
Outcome: Pass

Type: Exceptional  
Test data: { 2, 7, 3, 5, 9, 1, 6, 8, 4 }, 69  
Expected result: -1 (value not found)  
Actual result: -1  
Outcome: Pass

## Testing site functionality

### Testing user registration

Type: Normal  
Test data:  
 Username: username (8 characters)  
 Password: password  
Expected result: Account creation, redirect to the homepage  
Actual result: Account creation, redirect to the homepage  
Outcome: Pass

Type: Extreme  
Test data:  
Username: usr (3 characters)  
Password: password  
Expected result: Account creation, redirect to the homepage  
Actual result: Account creation, redirect to the homepage  
Outcome: Pass

Type: Extreme  
Test data:  
 Username: twentycharacterslong (20 characters)  
 Password: password  
Expected result: Account creation, redirect to the homepage  
Actual result: Account creation, redirect to the homepage  
Outcome: Pass

Type: Exceptional  
Test data:  
 Username: thisusernameistoolong (21 characters)  
 Password: password  
Expected result: Error message displayed, “Your username must be between 3 and 20 characters long.”  
Actual result: Error message displayed, “Your username must be between 3 and 20 characters long.”  
Outcome: Pass

Type: Exceptional  
Test data:  
 Username: a (1 character)  
 Password: password  
Expected result: Error message displayed, “Your username must be between 3 and 20 characters long.”  
Actual result: Error message displayed, “Your username must be between 3 and 20 characters long.”  
Outcome: Pass

### Testing user login

Type: Normal  
Test data:  
 Username: username (existing user)  
 Password: password  
Expected result: Redirect to the homepage  
Actual result: Redirect to the homepage  
Outcome: Pass

Type: Exceptional  
Test data:  
 Username: username2 (non-existent user)  
 Password: password  
Expected result: Error message displayed, “Incorrect username or password.”  
Actual result: Error message displayed, “Incorrect username or password.”  
Outcome: Pass

### Testing post creation

Type: Normal  
Test data: This is a test post.  
Expected result: Post creation, page reload, and post displayed on the homepage  
Actual result: Post creation, page reload, and post displayed on the homepage  
Outcome: Pass

Type: Exceptional  
Test data: (empty post)  
Expected result: Error message displayed, “The content of the post cannot be empty.”  
Actual result: Error message displayed, “The content of the post cannot be empty.”  
Outcome: Pass

### Testing post search

Type: Normal  
Test data: This is a test post.  
Expected result: Post displayed in search results  
Actual result: Post displayed in search results  
Outcome: Pass

Type: Exceptional  
Test data: (empty search query)  
Expected result: Error message displayed, “Enter a search query”  
Actual result: Error message displayed, “Enter a search query”  
Outcome: Pass

### Testing user logout

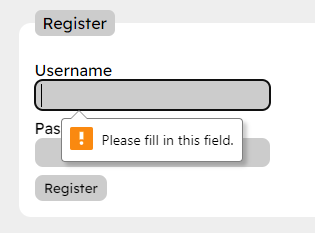
Type: Normal   
Test data: Cookie exists in the database; user is logged in   
Expected result: Cookie deleted, redirect to the login page   
Actual result: Cookie deleted, redirect to the login page   
Outcome: Pass

Type: Normal   
Test data: Cookie does not exist in the database; user is already logged out   
Expected result: Redirect to the login page   
Actual result: Redirect to the login page   
Outcome: Pass

## Results

Most tests passed successfully, with only one failure in the binary search algorithm. This was due to the algorithm searching from the middle of the list, rather than the start, causing it to return a different index of the value than the expected first if there were duplicate entries in the list. This was solved by checking for duplicates in the list beforehand, then returning -1 if any duplicates were found. However, this is not a perfect solution as it compromises the efficiency and accuracy of the algorithm, though ended up being sufficient for the purposes of the project.

Furthermore, although the tests for input validation were all successful, the system was modified to also support client-side validation in HTML forms to provide a better user experience. Identical tests were run on the client-side validation, and all passed successfully.

Usability testing on the site also proved successful, with all features working as expected and providing a ease of use and a good user experience, meeting all end-user requirements. Multiple popular web browsers were tested for compatibility with the site, all of which were able to display it correctly and allow users to interact with it as intended.

# Evaluation of the solution

The initial goal of the project was to create a basic microblogging platform, where other students could post and view posts on a simple website to share ideas and thoughts or post updates to their own projects. I believe that the project has been successful in meeting this goal, as it has implemented all required features and has met all the functional and end-user requirements specified in the design section.

## Meeting requirements specification

The results of testing with both unit tests and manual test tables have shown that the implementation of the solution meets all original functional and end-user requirements. Completed requirements include:

* Use of a database to store user data, sessions, and posts.
* Navigation bar for easy navigation, available on all pages.
* Validation of all user input to prevent attacks on the login, registration, post creation, and search forms.
* Registration of user accounts with storage of usernames and passwords in a database table, showing a helpful error message if the username is invalid or already taken.
* Login system that checks user credentials against the database and creates a session if correct, showing a helpful error message if incorrect.
* Search functionality for finding posts by content, showing a helpful error message if the search query is empty.
* Homepage for viewing recent posts in reverse chronological order.
* Profile page for viewing recent posts by a specific user.
* Easy navigation between all pages, including different user profile pages.

## Results of testing

The results of the tests showed that most features were working as expected, and that any issues were able to be resolved with minor changes to the code. The addition of client-side validation to the HTML forms was also successful, improving the user experience by providing instant feedback when a form was submitted with invalid data.

## Future maintainability

Use of modular functions, UI components, clear variable names, inline comments and doc-comments (comments at the top of a function explaining what it does, highlighted in the editor), formatting tools, version control commits with clear and concise messages, type annotations in the codebase, and the documentation you are reading right now all contribute to improving the maintainability of the project.

However, the project uses a fair amount of non industry-standard tooling and custom solutions for problems that could more easily be solved in other languages. The structure of the custom routing system is familiar to me as it was inspired by other popular web frameworks, but it may seem convoluted or unnecessary to others. Use of a custom templating language may also have been unnecessary, as it could have been replaced with a more standard solution like simple string interpolation or a more popular and well-known templating language. (In truth, I was halfway through implementing a custom programming language for the project before abandoning the idea.)

While these solutions were helpful as a learning experience during the project, they may not be the best choice for long term maintainability, as lack of familiarity, prior experience, or documentation on specific features could make it difficult for others to understand or modify the project in the future.

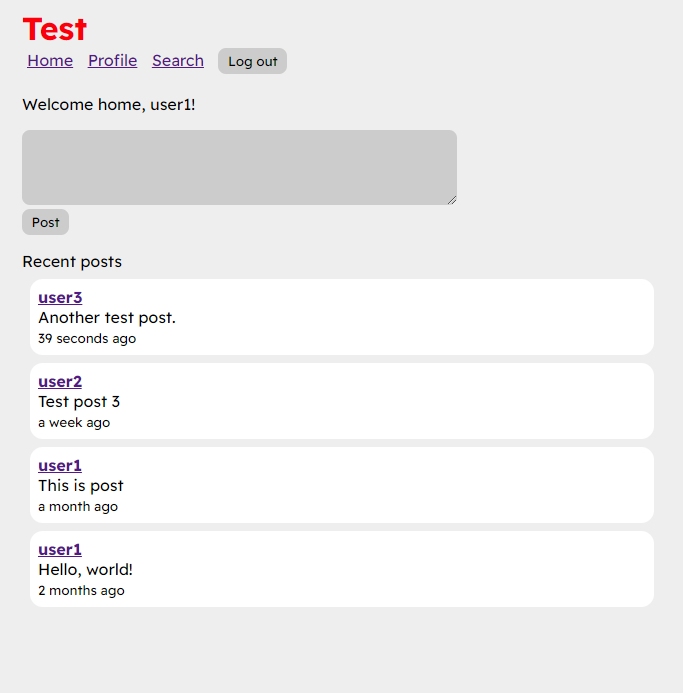
## Robustness

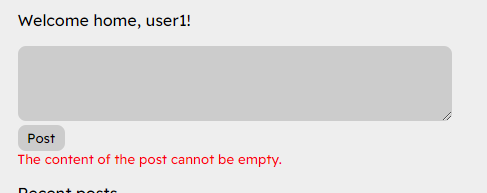
Use of input validation, good error handling, and unit tests increase the project’s robustness against common attacks and errors. Unit tests can be re-run after modifying a function to ensure that it still works as expected, and error handling will display an error page with a corresponding HTTP status code if an error occurs during the request, rather than crashing the server and causing the request to time out.

# Appendices

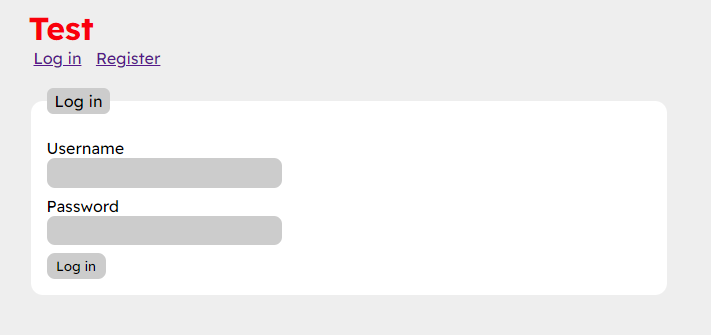
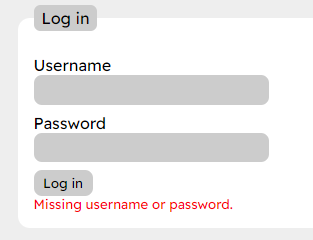
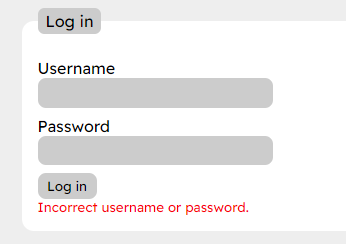
## Site screenshots

### Homepage

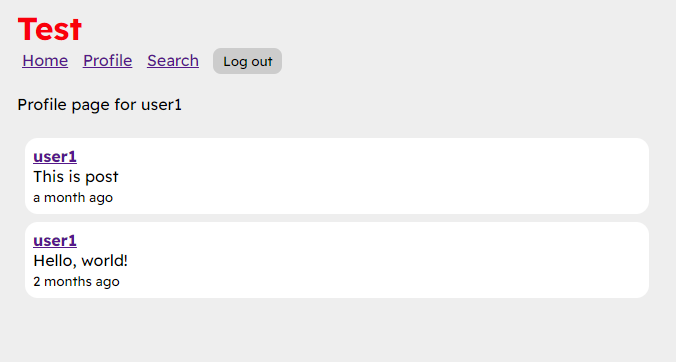
  

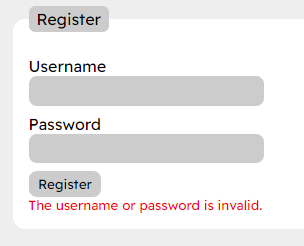
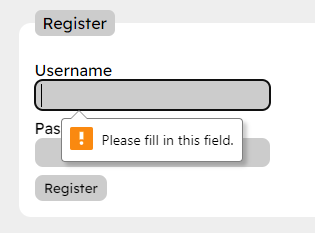
### Login

### Profile



### Register

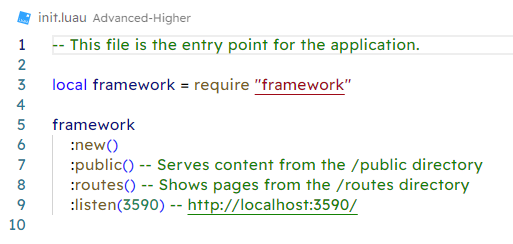
  
  
  


### Search

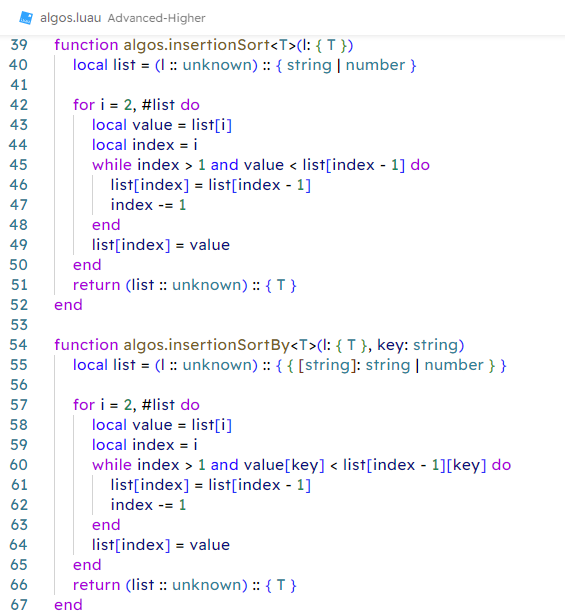
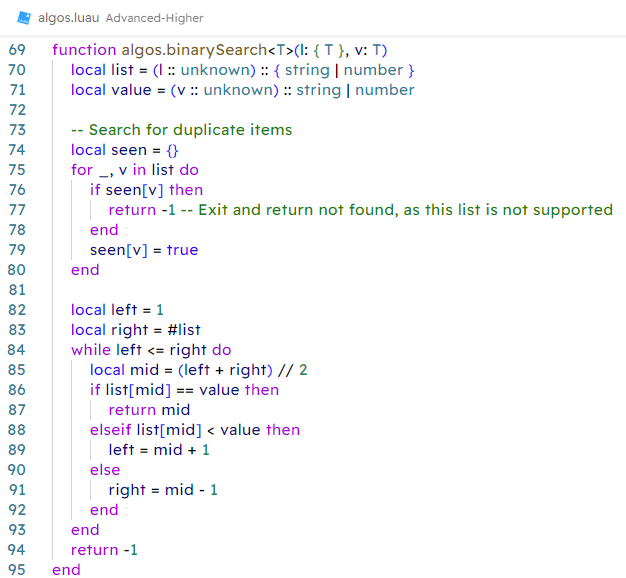
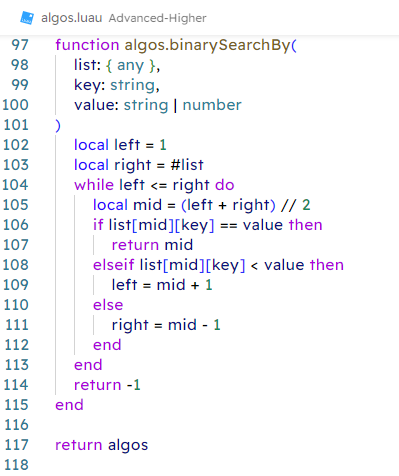

## Code screenshots

### init.luau



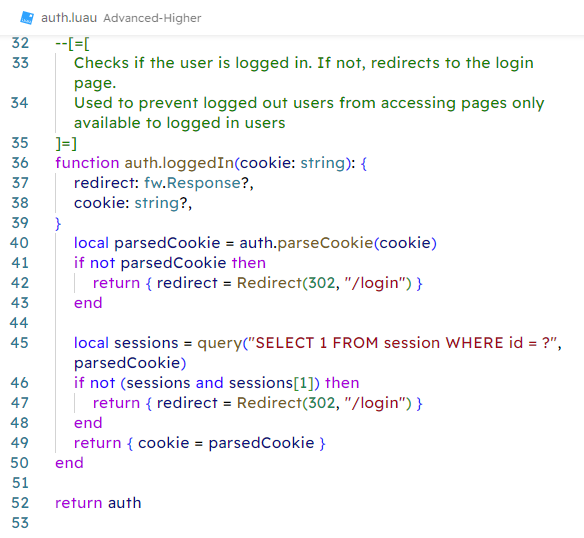
### algos.luau

### image

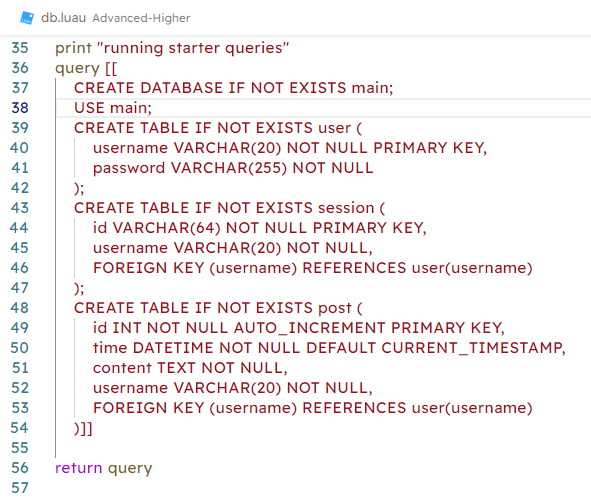
### auth.luau

### image



### db.luau

### image

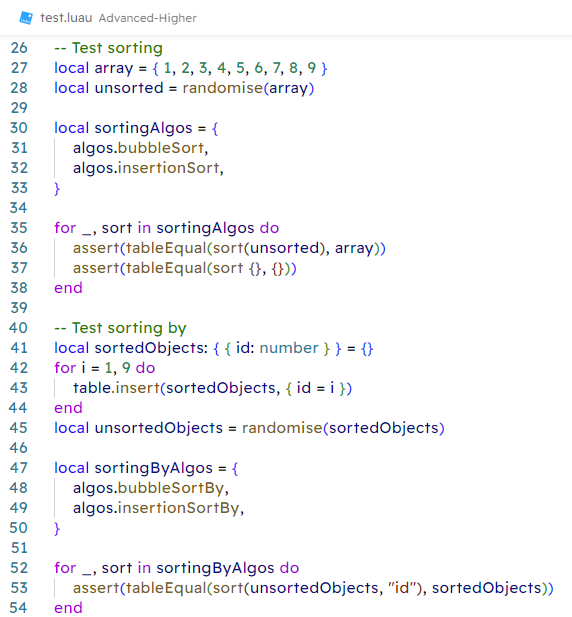
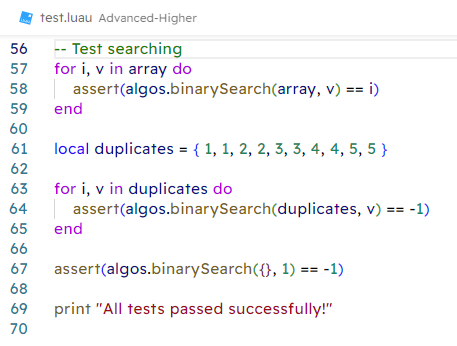


### db.ts

### image

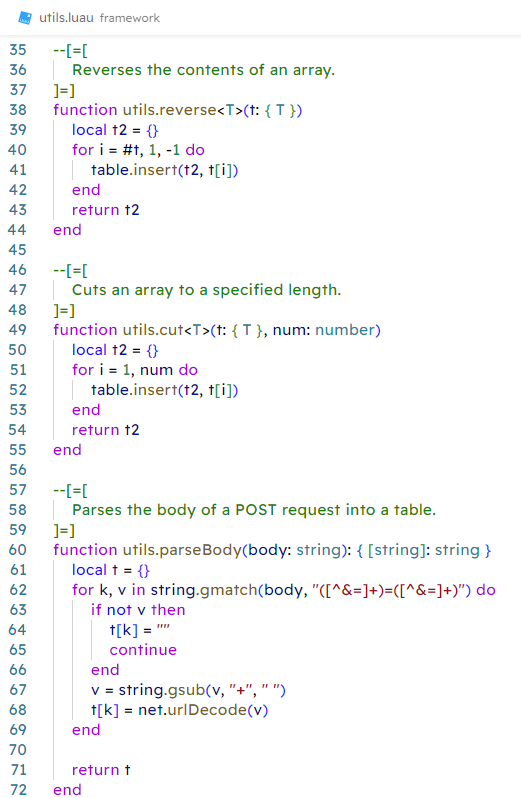
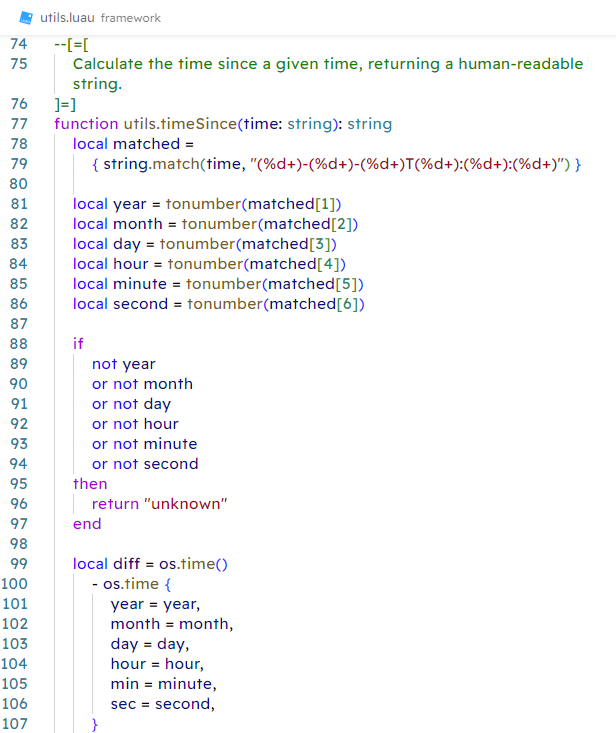
### test.luau

### image

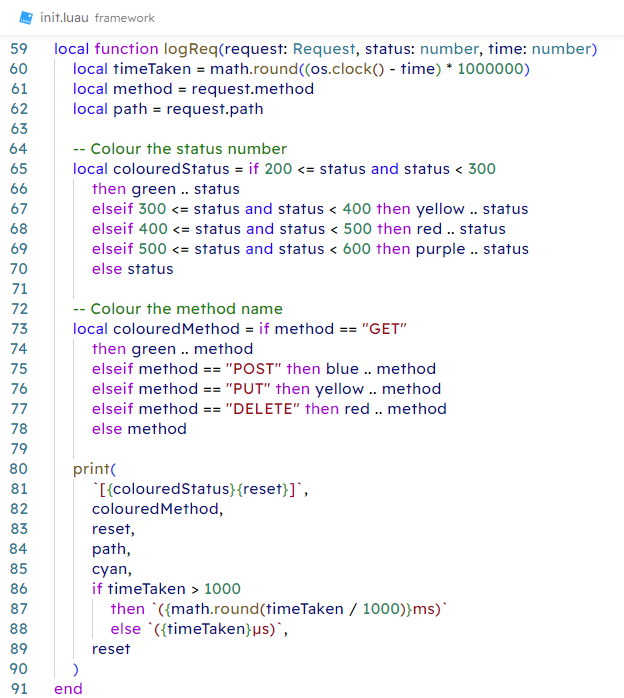
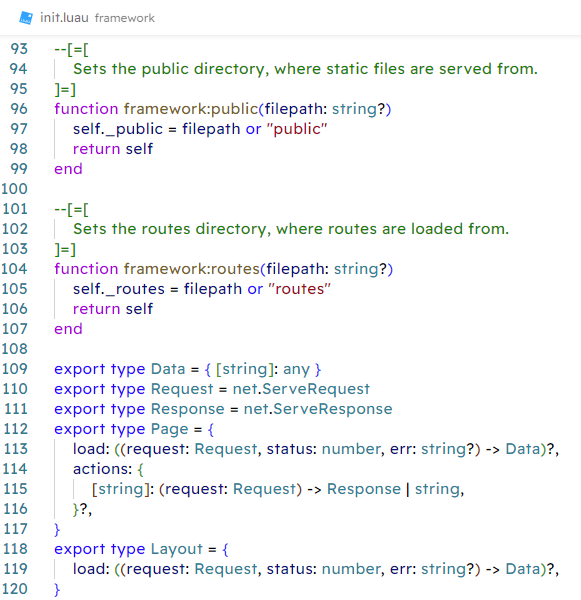
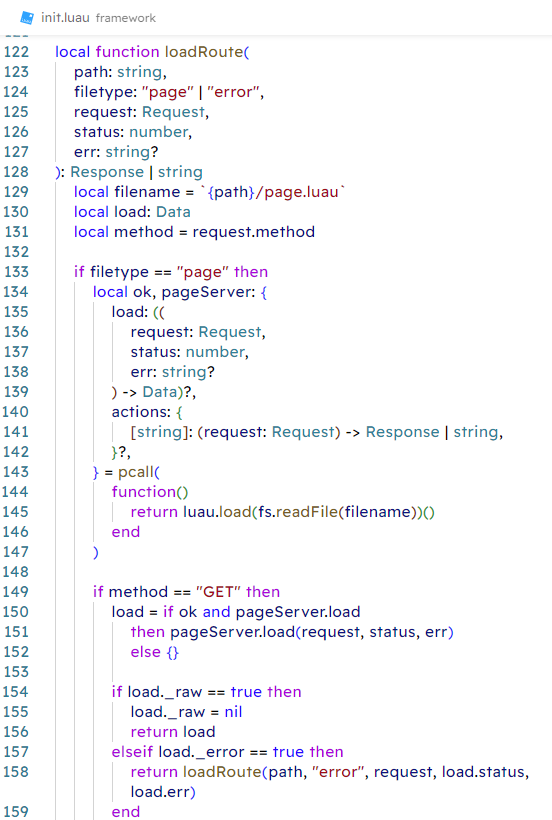
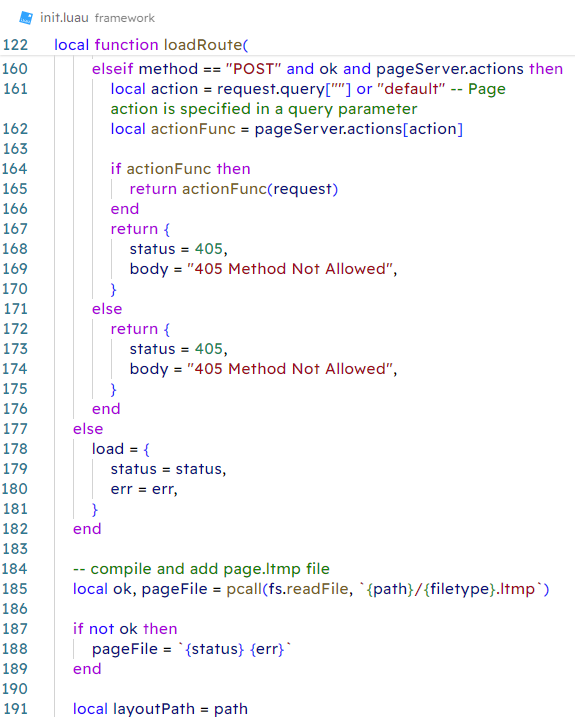
### utils.luau

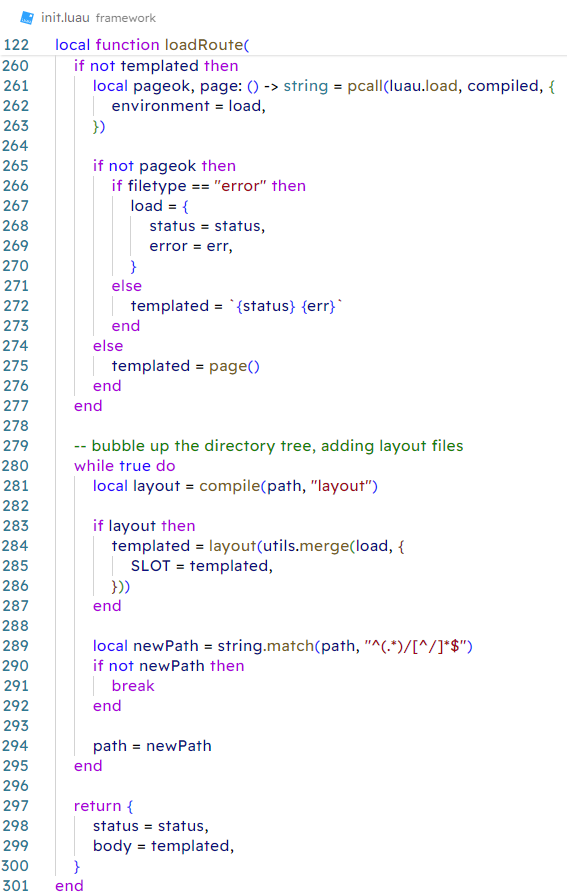
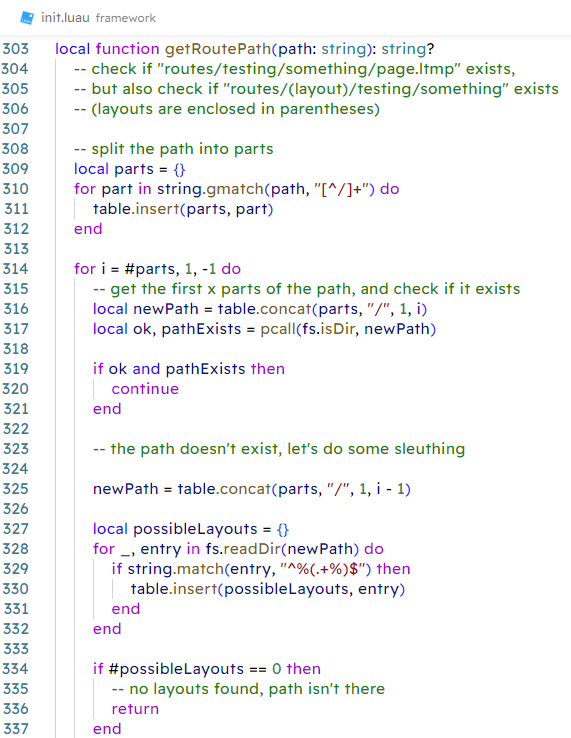
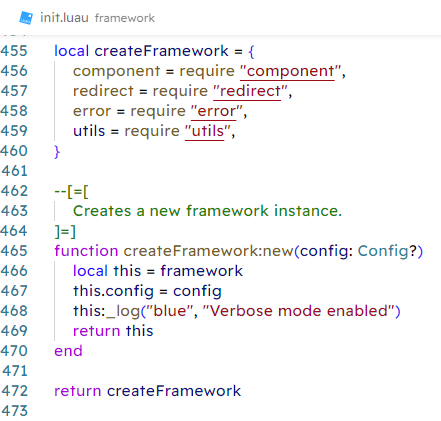
### image

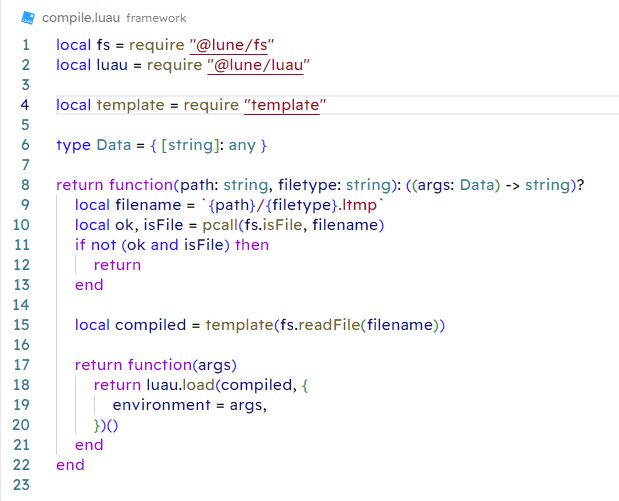
### framework/init.luau

### image

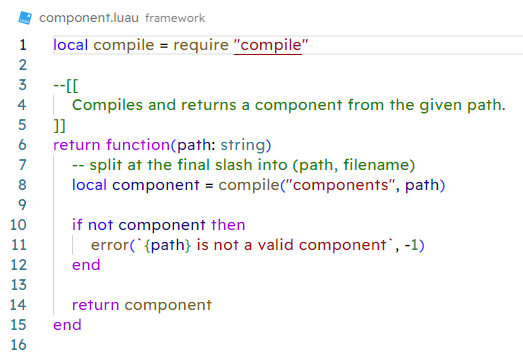
     A screenshot of a computer program

Description automatically generated     

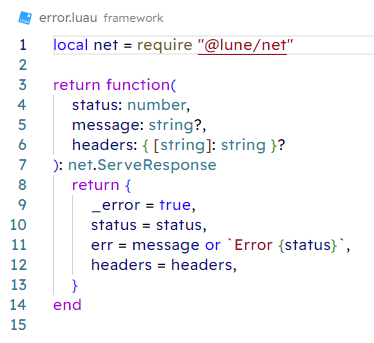
### framework/compile.luau



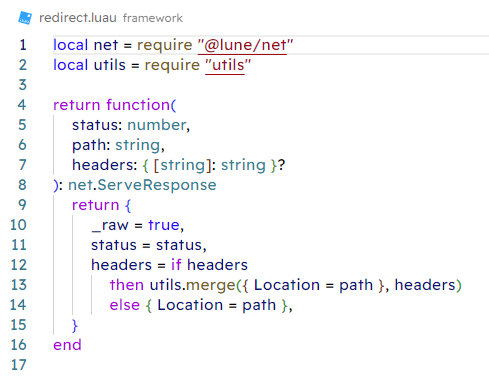
### framework/component.luau



### framework/error.luau

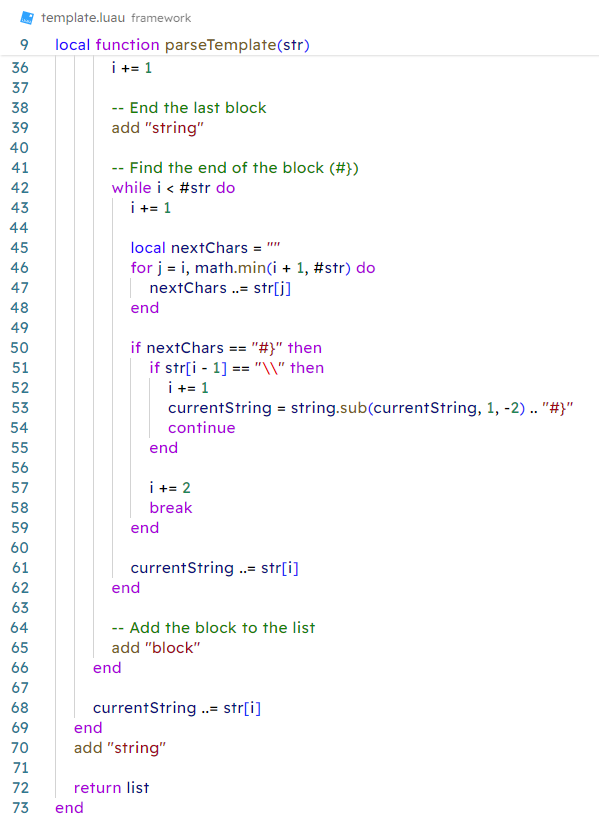
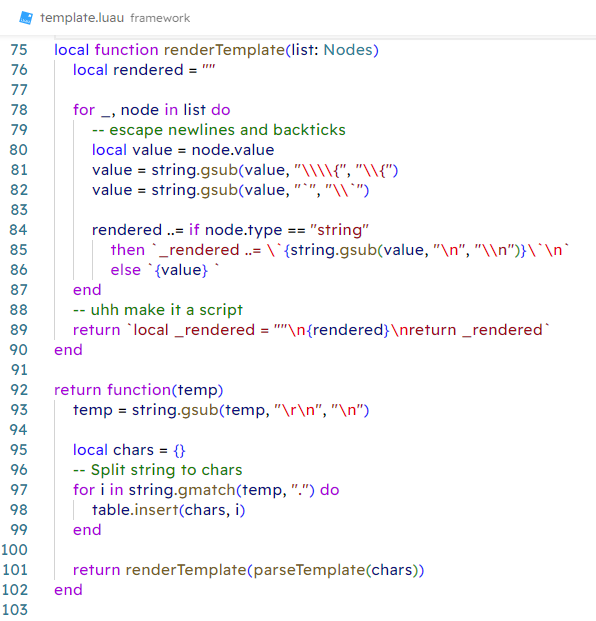


### framework/redirect.luau

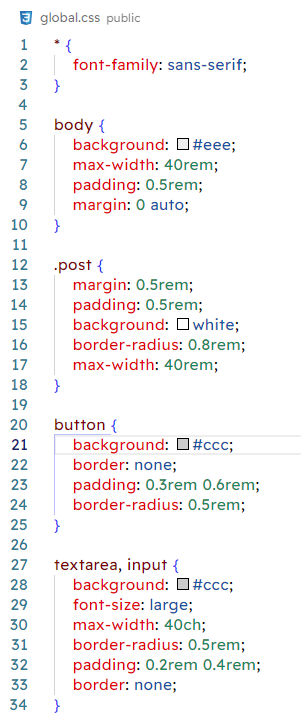
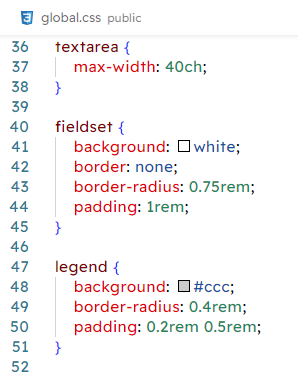


### framework/template.luau

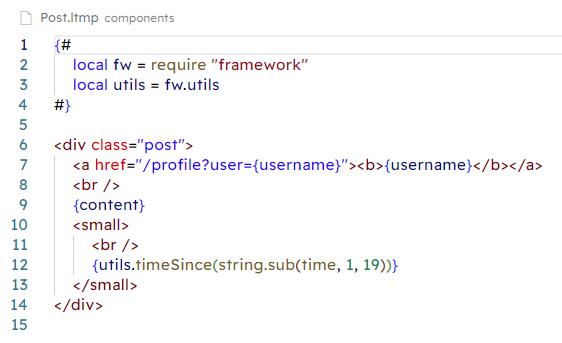
### image

### public/global.css

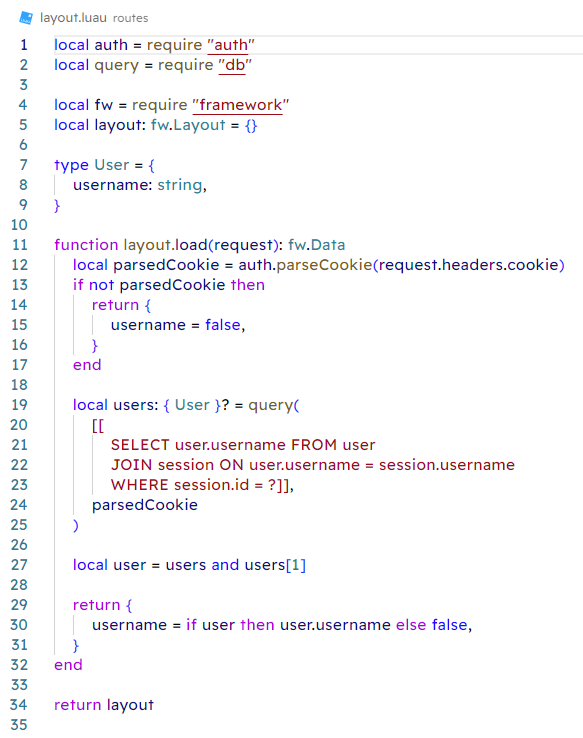
### components/Post.ltmp



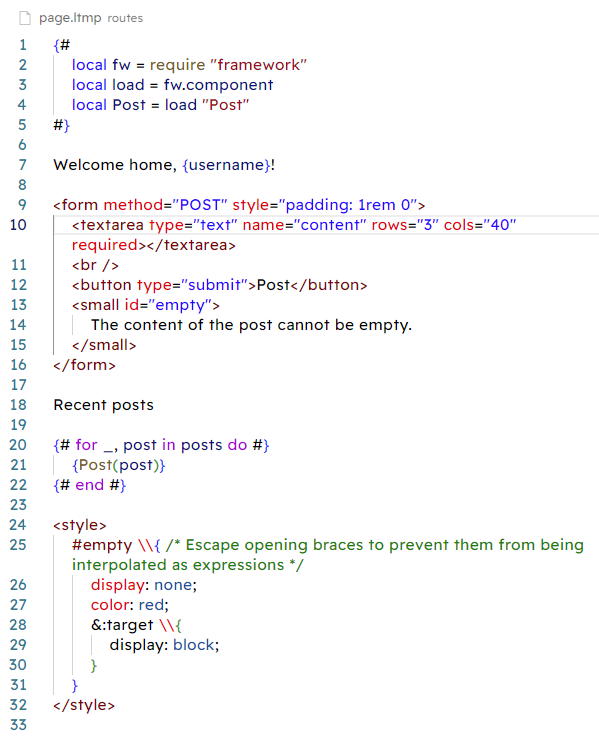
### routes/layout.ltmp



### routes/layout.luau



### routes/page.ltmp



### routes/page.luau

### image



### routes/(logged in)/layout.luau



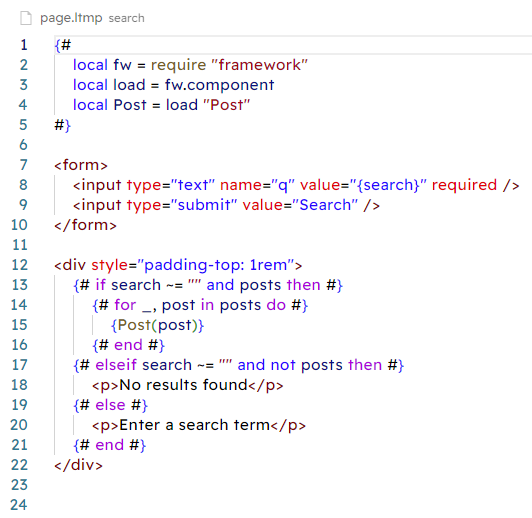
### routes/(logged in)/profile/page.ltmp



### routes/(logged in)/profile/page.luau



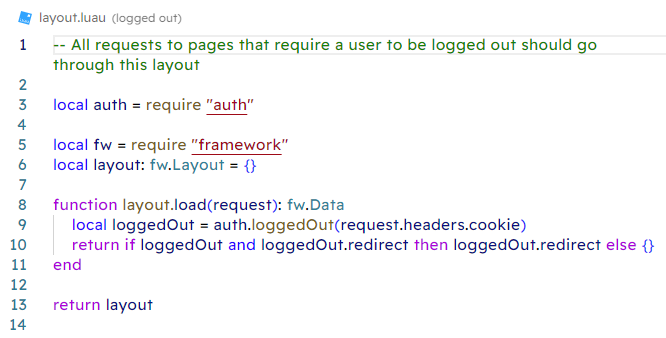
### routes/(logged in)/search/page.ltmp



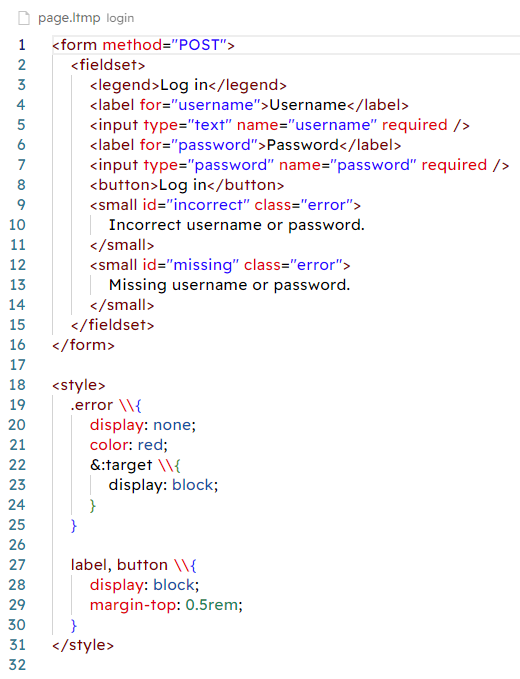
### routes/(logged in)/search/page.luau

### image

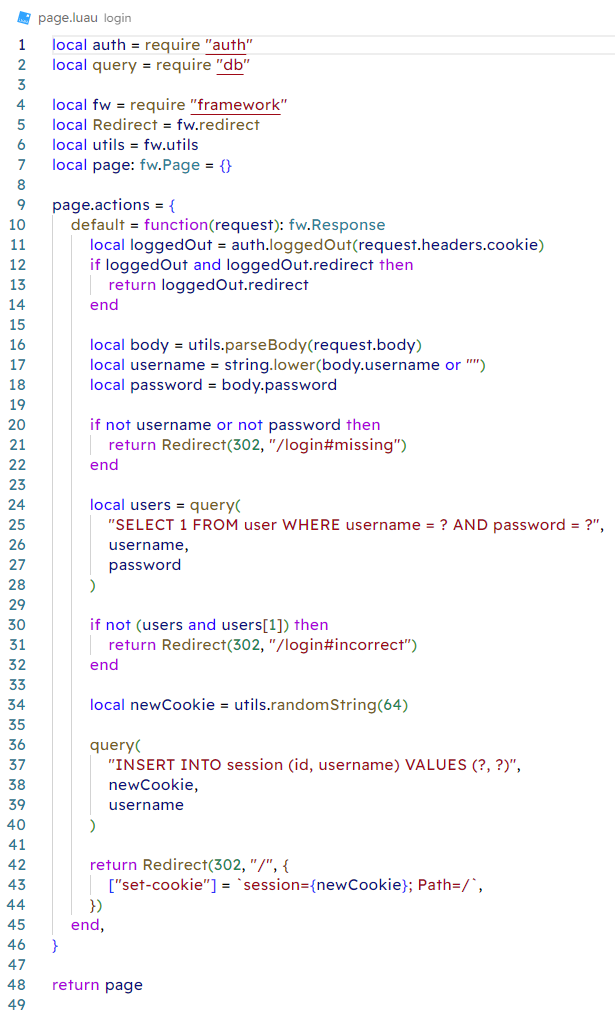
### routes/(logged out)/layout.luau



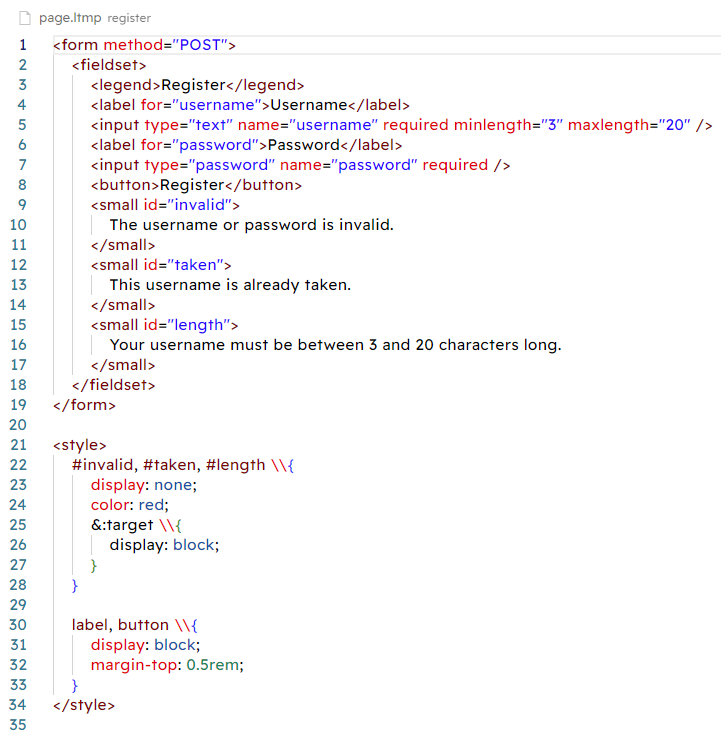
### routes/(logged out)/login/page.ltmp



### routes/(logged out)/login/page.luau



### routes/(logged out)/register/page.ltmp

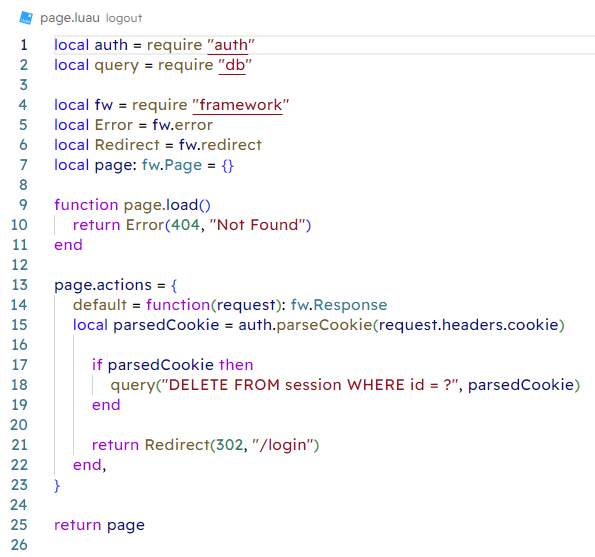


### routes/(logged out)/register/page.luau

### image



### routes/logout/page.ltmp



## End of project report