Silesian
University
of Technology

FINAL PROJECT

Interactive security training platform based on CTF concept

Krzysztof Marek DZIEMBAŁA
Student identification number: 293671

Programme: Control, Electronic, and Information Engineering
Specialisation: Informatics

SUPERVISOR
dr inż. Dominik Samociuk
DEPARTMENT Department of Computer Networks and Systems
Faculty of Automatic Control, Electronics and Computer Science

Gliwice 2023

Thesis title

Interactive security training platform based on CTF concept

Abstract

(Thesis abstract – to be copied into an appropriate field during an electronic submission – in English.)

Keywords

(2-5 keywords, separated with commas)

Tytuł pracy

Interaktywna platforma do nauki bezpieczeństwa wykorzystująca zadania typu CTF

Streszczenie

(Thesis abstract – to be copied into an appropriate field during an electronic submission – in Polish.)

Słowa kluczowe

(2-5 keywords, separated by commas, in Polish)

Contents

Chapter 1

Introduction

- introduction into the problem domain
- settling of the problem in the domain
- objective of the thesis
- scope of the thesis
- short description of chapters
- clear description of contribution of the thesis's author – in case of more authors table with enumeration of contribution of authors

Chapter 2

[Problem analysis]

- problem analysis
- state of the art, problem statement
- literature research (all sources in the thesis have to be referenced [5, 4, 6, 7])
- description of existing solutions (also scientific ones, if the problem is scientifically researched), algorithms, location of the thesis in the scientific domain

Mathematical formulae

$$y = \frac{\partial x}{\partial t} \tag{2.1}$$

and single math symbols x and y are typeset in the mathematical mode.

Chapter 3

Requirements and tools

The project development consisted of multiple stages. One of the most important parts was specifying requirements, which then dictated tool selection and the design process.

3.1 Functional requirements

Functional requirements list functionality available in the system. Each of the requirements contains a description detailing the desired behaviour.

1. **Account creation**: Users must be able to register an account in the system. This operation requires username and password submission from an HTML form in a POST request. Registration is allowed only using unique username. If there already exists an account in the system with the same username, the action must be refused. As a result of a successful registration, a document with the username, cryptographic hash of the user's password and a default role `"user"` is inserted into a collection storing user accounts. After the registration succeeds, the user is redirected to the login page.

2. **Signing in**: Users must be able to log into their accounts. Username and password must be sent in a POST request as HTML form data. The operation must fail if there is no account in the database with the provided username or if the password hash does not match the one stored in the database. If there has been no failure, a session is created and the user is redirected to the home page.

3. **Logging out**: Users must be able to log out of their accounts. It is expected that the user is signed in when they log out. This operation destroys the session (if any) and redirects to the home page.

4. **Changing password**: Users must be able to change their account password. New password must be sent in a POST request as HTML form data. User must be logged

in in order to change the password. As a result of this operation, user's password hash in the database is updated.

5. **Listing categories**: Users must be able to see a list of categories that exist in the system. List of categories must link to category pages.

6. **Displaying category**: Users must be able to see category details on a category page. The details should include category name, description and a list of related tasks.

7. **Displaying task**: Users must be able to display task details on task pages. The details should include task name, description, challenge URL, hints (if any), and if the user is logged in, a flag submission form. If the task has been solved by the user, instead of the flag submission form, a question is displayed.

8. **Solving challenge**: Users must be able to submit a form with flag from the task page. This action is allowed only for signed in users. Submitted flag must be verified with the one stored in the database for the given challenge. If the flags match, a success message should be shown to the user and date of solving the challenge by user ought to be saved to the database. Otherwise, a message informing the user about incorrect flag should be presented.

9. **Answering quiz**: User who have solved the main challenge from a task, should be able to to see a question on the task page and be able to answer it. Only signed in users can submit answers. Checkboxes which status indicates whether the user thinks that they are correct must be presented along answers from the database. After the answers are submitted, the quiz should be disabled without a button to submit, checkboxes disabled and reflecting the user's answer and an indication which answers were correct.

10. **Administrator panel**: Users with the `"admin"` role (administrators) should have access to a separate administration panel. The panel should expose additional functionality related to the system management.

11. **Listing users**: Users with access to the administrator panel should be able to get a list of user account in the system. Each entry must contain information about username and user role. The list should be paginated as there may exist many accounts.

12. **Changing user permissions**: Administrators should be able to grant the `"admin"` role to other users, as well as change it back to `"user"`.

13. **Deleting user**: Administrators should be able to remove user accounts from the database. It must not be possible to remove own user account this way.

14. **Creating category**: Administrators must be able to create new categories. The categories must have a name and description. The description must support Markdown input.

15. **Editing category**: Administrators must be able to edit the name and description of existing categories.

16. **Creating task**: Administrators must be able to add new tasks to the system. For each task it must be possible to set the name, description (in Markdown), hints, challenge details, question, answers to the question. Each answer must be marked as correct or incorrect.
    Challenge details must include:

    - Docker image to pull and start,
    - subdomain used for serving the challenge,
    - flag that the users will try to find,
    - an interval specifying how frequently the challenge container should be regenerated

17. **Starting challenges**: The system must be able to pull challenge images, create and containers and direct connections to specified subdomains into appropriate containers. This must be done automatically during system startup and for each new challenge added when creating new tasks.

3.2 Non-functional requirements

1. **Responsiveness**: UI should properly scale across different display sizes. It must be mobile-friendly.

2. **Accessibility**: There should be no errors in the Accessibility section of a webhint scan.

3. **Visual consistency**: A single set of styling rules, such as colours, fonts and icons should be used across whole user interface.

4. **Page load performance**: The system should have a score of over 90 in PageSpeed Insights report for mobile.

5. **Compatibility**: User interface should work in latest (as of January 2023) versions of Firefox, Firefox ESR, Chrome and Safari browsers for desktops and mobile devices. Basic system functionality, except for the administrator panel, should be available in browsers with JavaScript disabled.

3.3 Use cases

The use case diagram presented on Fig. 3.1 shows actions offered to actors using the system. There are two actors, with different sets of allowed interactions. The actor *User* represents anyone with access to the system. Users with special account role `"admin"`, which allows them to perform operations related to management of the system.

3.4 Tools

The implementation of the project significantly benefited from publicly available tools. Used tools are divided into two classes, depending on the way they were used.

3.4.1 Core tools

The system is built on tools, which are regarded to as *core tools*. These are required for operation and are a part of the system architecture.

Node.js + Express + EJS

Node.js is a JavaScript runtime based on V8 engine. There is an enormous ecosystem built around Node.js with over 1.3 million packages available in the npm registry [1]. The server code runs on Node.js and uses the Express framework for request routing and middleware management. Express is a popular web framework with many features and compatible middleware packages. One of Express' features is template rendering support. The project uses EJS templates, which allows creating HTML templates using embedded JavaScript logic.

MongoDB

MongoDB is a NoSQL document-oriented database capable of storing BSON documents. Because BSON stands for Binary JSON, which in turn is JavaScript Object Notation, it integrates nicely with JavaScript. The MongoDB Node.js driver also supports TypeScript, which helps with suggestions and type checking.
During development MongoDB Compass, MongoDB Visual Studio Code extension and mongosh were used. These tools allow database management and, except for mongosh, provide helpful user interfaces.

Docker

Docker is a platform for managing containerised software. Challenges are running as Docker containers and managed using Docker Engine API. For debugging, development

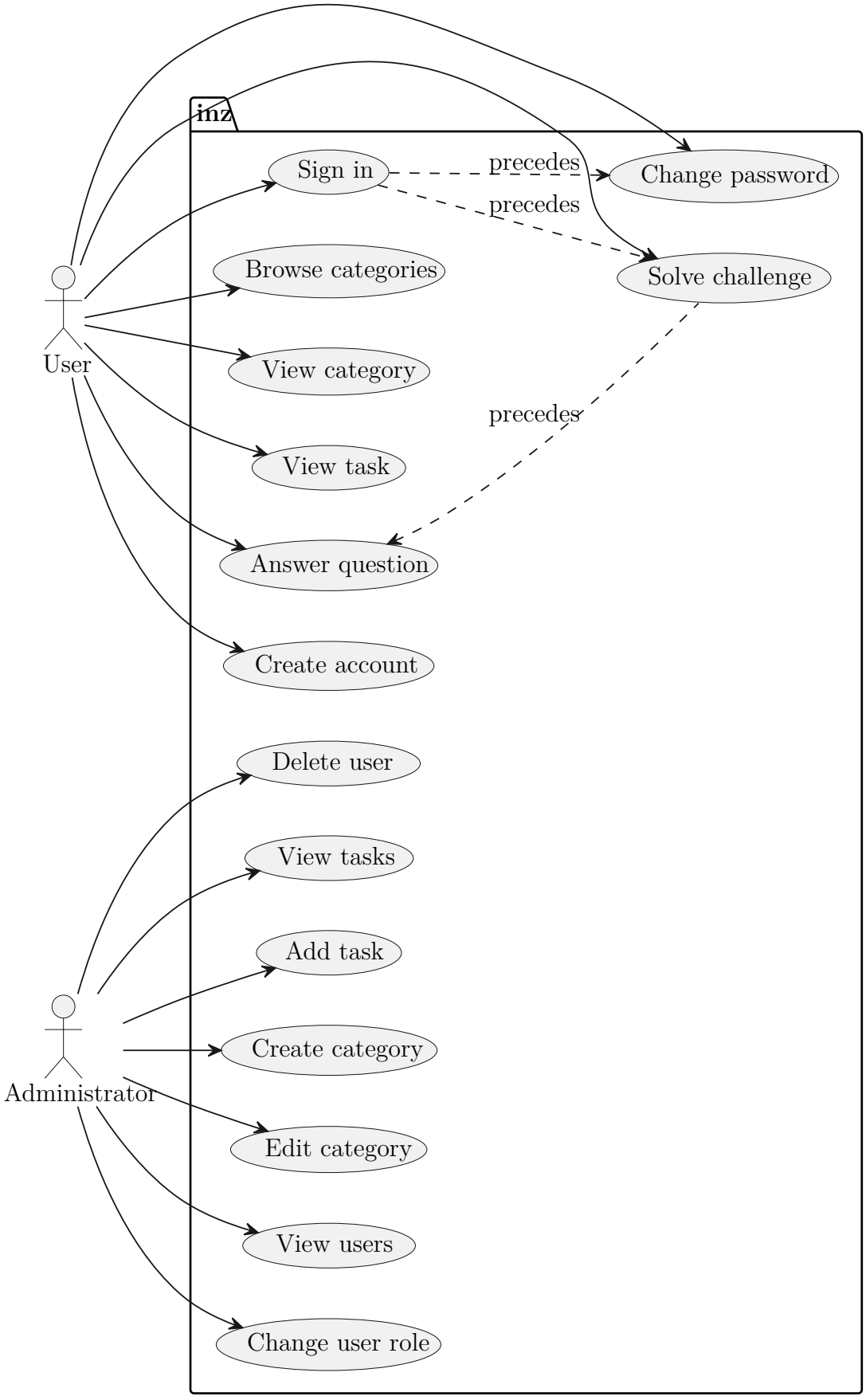

Figure 3.1: Use case diagram

and management standalone Docker tools such as Docker CLI and Docker Desktop can be used.

nginx

To manage multiple domains and subdomain on a single system with one external network interface nginx was used as a proxy. nginx offers multiple functionalities besides proxying. It is also used to serve static files, redirect to HTTPS protocol and terminate TLS connections.

Bootstrap

Bootstrap is a frontend toolkit, which is helpful for designing user interfaces. It provides CSS classes and JavaScript utilities to help with styling and providing interactivity in HTML without the need to write own style sheets and JS scripts.

3.4.2 Development tools

The following tools are in no way required for the systems. These were used to aid development.

Visual Studio Code

Visual Studio Code is a multi-platform code editor that was heavily used for writing the software. It was chosen for multiple reasons, most important of which was familiarity and experience with the tool. This editor supports many languages, especially for JavaScript and related web technologies. Particularly notable is built-in TypeScript support, which can be used with JSDoc comments to improve IntelliSense suggestions. A huge advantage of Visual Studio Code is a broad selection of available extensions.

Git

The project is stored in a Git repository to track changes in the code. The repository is synchronised with GitHub to share it between devices.

ESLint

ESLint is a JavaScript linter, which can be used to detect problems and potential issues in code. It was used with a Visual Studio Plugin, which automatically analysed open files and provided in-editor warnings and suggestions.

Prettier

Prettier is a popular code formatting tool. It was used to maintain a consistent style in JavaScript files. Thanks to Visual Studio Code plugin the tool could be used as a default formatter in the editor. A plugin for ESLint allowed marking improperly formatted code as lint error.

3.5 Methodology of design and implementation

The development process was split into two parts. Before the implementation started the system had to be designed.

3.5.1 Design

The first step in designing the project was defining initial requirements. This resulted in the first (and only) UI concept drawing, which is presented on Fig. 3.2. It was meant to help with defining the exact functionality and the structure of documents in the database. Based on that the structure presented on Fig. 3.3 was created.

After those first ideas were noted, the process of defining the architecture began. The choice of Node.js and Express as a base for the server was easy, as the author had already years of experience with those tools. Similarly, MongoDB was selected as a database server, because of experience and because it can store arrays and objects. To improve experience on low-end devices and enable basic functionality on browsers without JavaScript, server-side rendering was chosen instead of a JavaScript user interface library such as React or Vue. For a template rendering engine two solutions were considered - Handlebars and EJS. Ultimately, EJS was chosen, because of its richer capabilities.

The last problem to solve during the design phase, was coming up with an idea to direct different subdomains to appropriate challenge containers. The simplest solution was using a wildcard DNS record and setting up virtual servers in the proxy (Fig. 3.4). Creation of subdomains could be achieved by creating additional configuration files, which could be automatically picked up by the proxy on reload.

3.5.2 Implementation

The implementation started with setting up a Node.js project with some of the required dependencies and a basic *Hello world* Express server. Additional tools such as Eslint were also set up at the beginning so that they could be used during the development.

The next step was preparing database connection and setting up code for session support. Once the server could handle sessions, related utilities such as message flashing were prepared. Sessions were required to implement authentication, which was done in

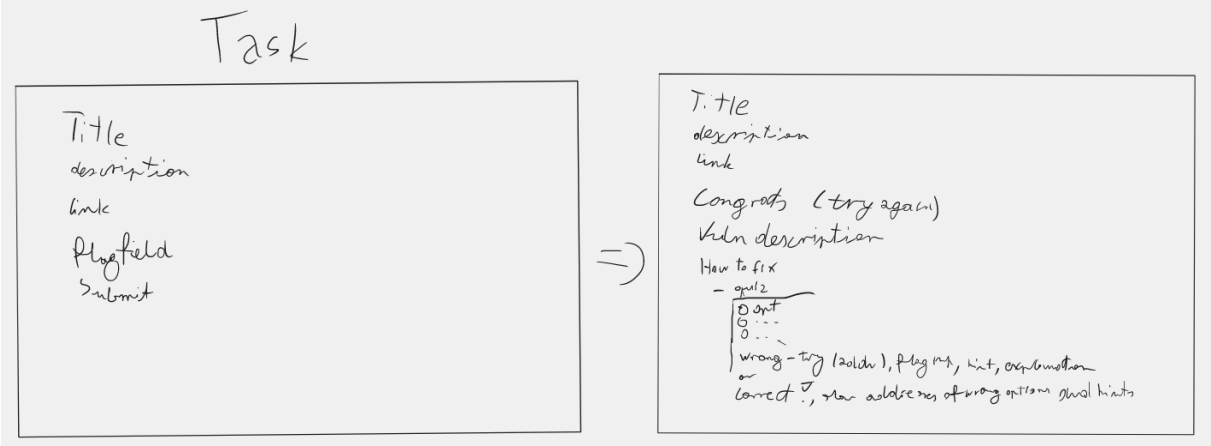

Figure 3.2: Initial task page UI sketch before and after solving the challenge.

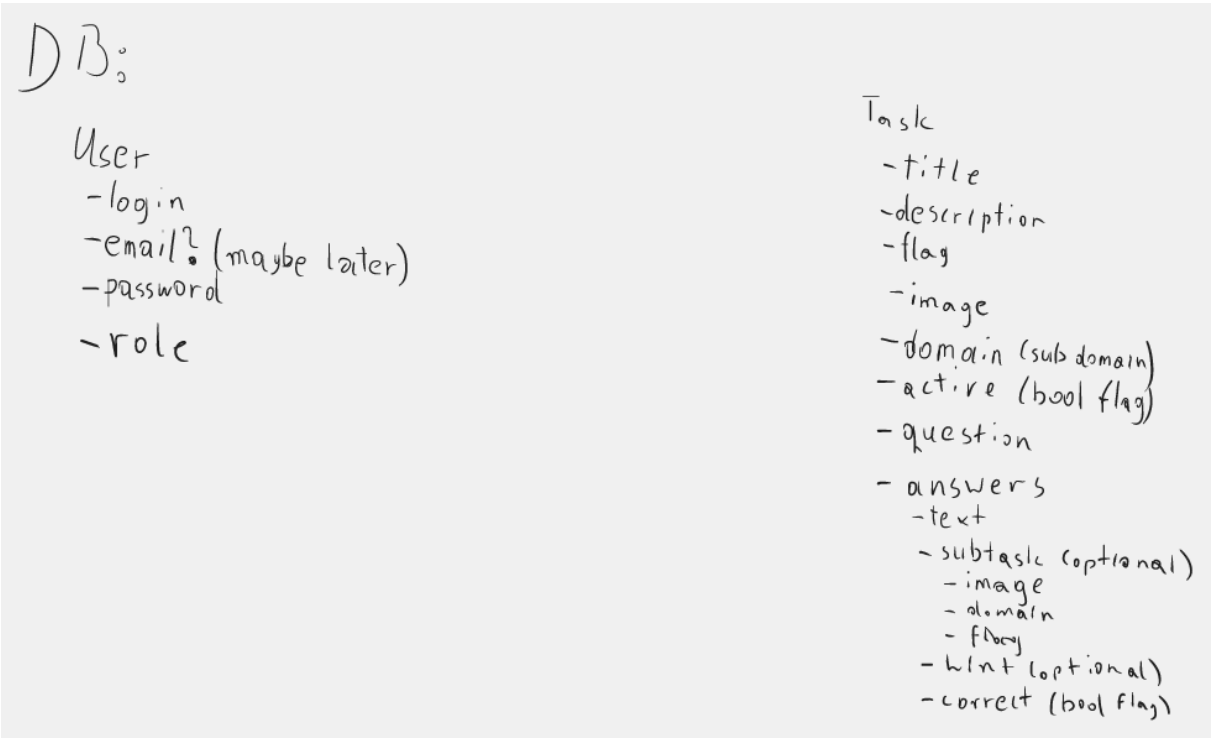

Figure 3.3: Initial design of documents representing users and tasks.

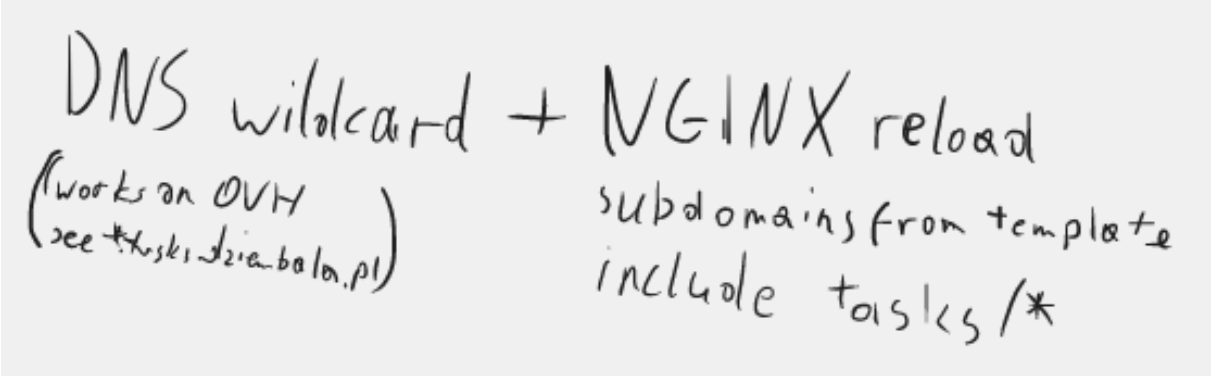

Figure 3.4: Subdomain management design idea.

the next step. The first part of implementing authentication was enabling user account creation, so that it would be possible to test each element as it was developed. With the code responsible for registration ready, logging in could be worked on. In both cases the server logic was prepared first with the UI following right after to test the whole component.

After the authentication was done and there were some simple pages ready, work on styling the interface could begin. This stage of the development was realised at that moment, because there were already pages which could be tested with the styles and could serve as a base for later content.

Before more public pages could be created, an administration panel had to be made in order to insert data for testing. Unlike all other components, this panel required a lot of interactivity and relied on client-side scripting. To facilitate this requirement, the server code exposes REST-like API, which is consumed by scripts running in the browser. The administrator panel started with user management, continued to category management and finished with task creation.

Since some of the inputs supported Markdown, a Markdown parser was integrated on both the server and in browser. To allow for syntax highlighting in code blocks, highlight.js was added to the Markdown parser. Syntax highlighting required additional styles, which were available separately in light and dark modes and would not integrate well with Bootstrap's theming. To avoid this problem, a variable-based style sheet with colours from Panda Syntax Light and Dark themes was created with CSS variables set depending on the theme used in Bootstrap.

With the last part of the administration panel, task creation, a system for managing challenge containers had to be prepared. During the implementation of this functionality, the initial database schema shown on Fig 3.3 was recognised as suboptimal and had to be changed.

Finally, user-facing category and task pages were prepared. These were template-based and in case of the task page relatively complicated, so the user interface and server code could be implemented in parallel. As a finishing touch a simple home page was added.

Chapter 4

External specification

- hardware and software requirements
- installation procedure
- activation procedure
- types of users
- user manual
- system administration
- security issues
- example of usage
- working scenarios (with screenshots or output files)

4.1 Hardware and software requirements

Depending on whether one wants to use the user interface via a web browser or host the server themselves there are different requirements for each part.

4.1.1 Browser

The user interface requires a modern browser with JavaScript enabled. It is possible to use the system without JS, but theme switching and dropdowns in the navigation bar will not work.

On desktop the latest stable versions of the following browsers are supported:

- Google Chrome
- Microsoft Edge

- Firefox and Firefox ESR
- Opera
- Safari (macOS only)

On Android and iOS support is restricted to the latest stable versions of:

- Google Chrome
- Firefox
- Safari (iOS only)
- Android Browser and WebView (Android only)

Different and older browsers may work, but compatibility is not guaranteed.

4.1.2 Server software

The server software requires the following external software to be installed:

- Node.js 18.x or newer (not as a snap)
- MongoDB 6.0
- Docker Engine 20.10
- nginx 1.23.2 or newer

Supported operating systems:

- Ubuntu 20.04 or newer (x86-64, arm64)
- macOS 11 (x86-64), 12 or newer (x86-64, arm64)
- Windows 10 1809 or newer (x86-64)
- Windows Server 2019 or newer (x86-64)

It may be possible to run the server on other operating systems and architectures as long as the required software can be installed. These unsupported systems, however, will probably require additional installation steps related to `argon2` package setup.

Warning: Challenge Docker images must be compatible with the server architecture. Images presented in this project support only x86-64.

4.2 Server installation

The process of server software installation and configuration can be summarised in the following steps:

1. Install the required software as described in section 4.1.2.
2. Obtain the code from GitHub:
   `git clone https://github.com/krzysdz/inz.git`
3. Navigate to the downloaded project directory:
   `cd inz`
4. Install required npm packages:
   `npm install --omit=dev`
5. Configure MongoDB as a replica set (tutorial).
6. Configure the DNS to point the main domain and the challenges domain to the server. Example DNS configuration:

   ```
   main-ui.com.          60  IN A        127.0.0.1
   *.challenges.com.     60  IN CNAME    main-ui.com.
   ```

7. Obtain TLS certificate for the main domain and the challenges domain. One certificate should cover both domains (the second one with wildcard).
   The key and certificate files should be placed in `nginx/certs` and named according to instructions from the `README` file in that directory.
8. Adjust values in the `config.js` file.
9. Start nginx with prefix set to `./nginx` and configuration `nginx/conf/nginx.conf`:
   `nginx -p ./nginx -c ./nginx/conf/nginx.conf`
10. Set `NODE_ENV` to production:

    Bash/dash/zsh/csh:
    `export NODE_ENV="production"`

    Powershell:
    `$env:NODE_ENV="production"`
11. Start the server:
    `node index.js`
12. *Optional.* Configure services to automatically start the software. Example configuration for systemd based operating systems:

(a) Modify the default mongod.conf file to specify replica set name as shown on Fig. 1 and enable the mongod service:
`systemctl enable mongod`

(b) Create a unit file for the proxy. An example is presented on Fig. 2.

(c) Create a unit file for the main server. An example is presented on Fig. 3.

(d) Enable and start the services:

```
# Reload systemd configuration to pick up the new services
systemctl daemon-reload
# Enable the services to run at boot
systemctl enable inz-nginx
systemctl enable inz
# Start the server and proxy (required by inz.service)
systemctl start inz
```

4.3 Initial configuration

If setting up the server with a fresh database, an administrator account must be created. To do this, one has to:

- Create a user account (`/auth/register`).
- Set the `role` field of the user document to string `admin`. A Node.js script which does that is presented on Fig. 4.
- Log in into the account again.

4.4 Types of users

There are three types of users considered in the system:

- anonymous/not signed in
- regular users - account with role `"user"`
- administrators - account with role `"admin"`

Anonymous users can freely browse the application, but have a limited functionality on the task page.
Regular users gain access to profile page and unlock challenge and quiz submissions.
Administrators expand on the regular user permissions and can use an administration panel for managing the system.

Chapter 5

Internal specification

5.1 System architecture

The system is managed by a server running on Node.js. As presented on Fig. 5.1, this process manages the MongoDB database, challenge containers via Docker Engine API and the nginx proxy configuration. Challenges are run as Docker containers to enable easy configuration by just puling an appropriate image and each of them is bound to a different subdomain of a dedicated challenge domain thanks to the single proxy server.

The nginx process is the only one exposed publicly. Other software can listen only on localhost for security reasons.

5.2 Database structure

The use of a NoSQL database, allowed for a more flexible schema, based on the desired access to the data. A mix of embedded data model and normalized data model is used. The used database schema is presented on Fig. 5.2. Despite preferring the denormalised model, it was necessary to keep some relations between documents. This is achieved by storing fields with `_id`s of referenced documents. There is also a relation between keys of embedded documents in the `users` collection, which are stored as stringified `ObjectId`s and documents from `tasks` and `challenges`. This relation, however, is not used in queries to the database, but managed completely by the server code.

There are two additional indexes:

- a unique index on the `subdomain` field in the `challenges` collection to make sure that there are no duplicate subdomains,
- a unique index with a collation on the `name` field in the `categories` collection to accelerate case-insensitive search by category name and prevent duplicates.

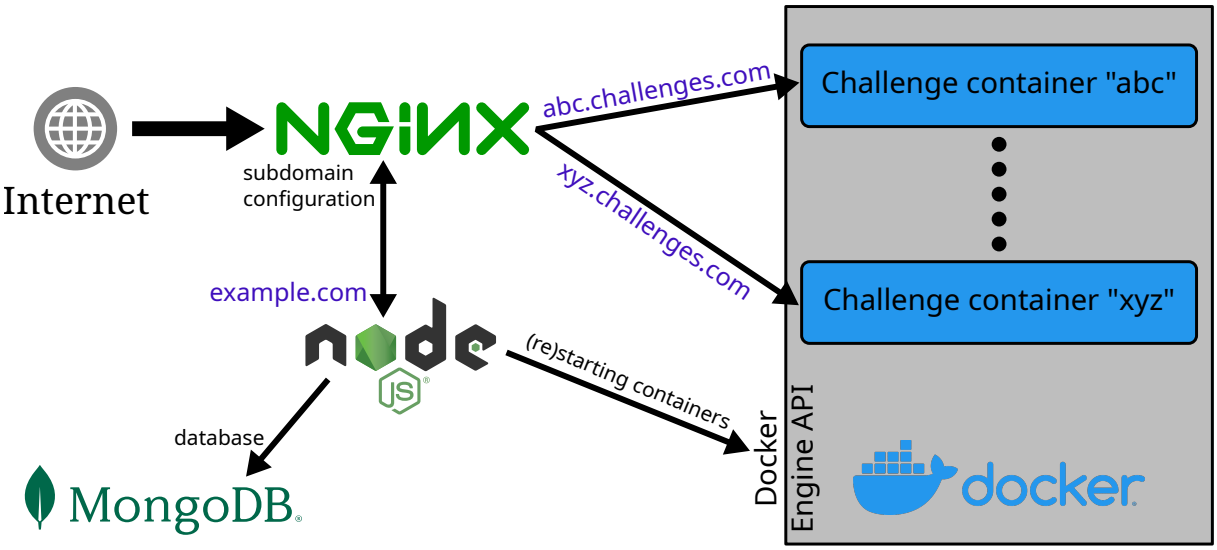

Figure 5.1: Visual representation of system architecture.

The view `fullTasks` is a view collection on the `tasks`, which uses an aggregation pipeline to embed documents from the `challenges` collection using `$lookup` where necessary. The pipeline is presented on Fig. 5.

5.3 Code organisation

The code is divided into ECMAScript modules. The main one is `index.js`, which imports all the other required modules, some external tools, configures and starts the server. Each module serves a separate function.

5.3.1 Middleware

Two modules are responsible for middleware functions. Middleware is a function called during request processing, which can act on the request and response objects and pass execution to functions declared later in the router stack.

One of them, `src/middleware.js`, contains general utility middleware functions:

- `authenticated` - a middleware which returns 401 if the user is not logged in,
- `adminOnly` - a middleware which returns 403 if the user is not an administrator,
- `addCategoriesList` - a middleware executed before any routers, which fetches all category names and makes them available as response locals to use within templates.

The other module was separated, because it alters the request object and has an associated `.d.ts` typings file to help with type checking and code suggestions. It is `src/flash.js` and adds the following methods to each request object:

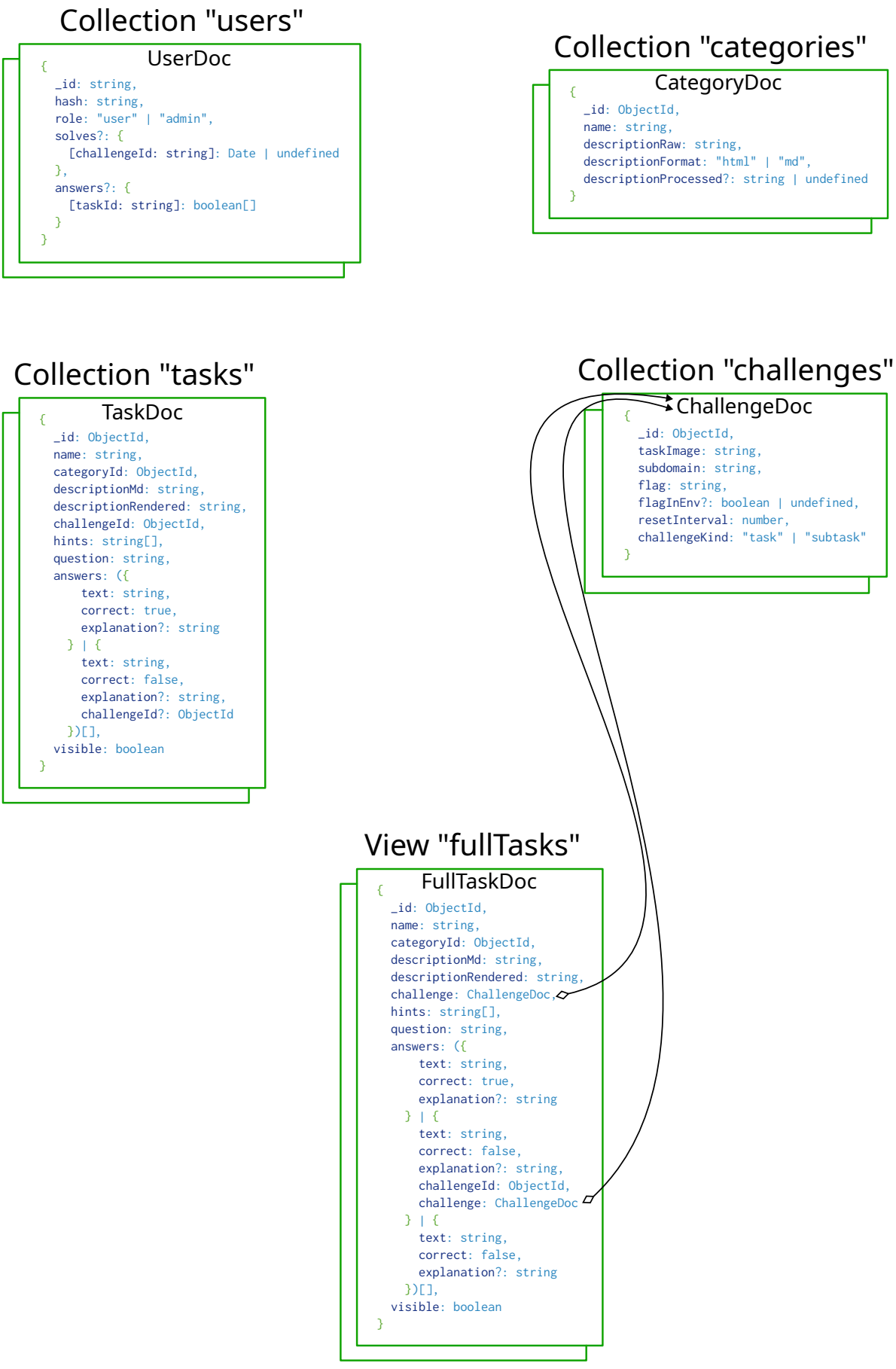

Figure 5.2: Visual diagram of the database schema.

- `flash(message, category = "info")` - a function which adds a message with category to a list of flash messages connected with the session
- `getFlashedMessages(options)` - a function which returns the flashed messages with associated categories. This function can also filter the returned flashes by category. It is also available in response locals for direct use in templates.

The behaviour of this middleware is supposed to mimic the message flashing functionality of the Flask framework.

5.3.2 Routers

The `index.js` module provides only the request handler for the root path `/`. Other paths are delegated to separate routers, which live inside modules under `src/routes`. The following paths are registered:

- `/auth` handled by `authRouter` from the `auth.js` module. It serves the register, login and logout pages, as well as handles password change requests.
- `/profile` guarded by the `authenticated` middleware, handled by `profileRouter` from the `profile.js` module. It serves only the profile page.
- `/admin` guarded by the `authenticated` middleware, handled by `adminRouter` from the `admin.js` module.
  The router uses additional middleware `adminOnly` and `express.json()`. It renders only a single page - the admin panel. It provides a REST-like JSON API, which is used by a script on the admin panel page.
- `/category` handled by `categoryRouter` from the `category.js` module. It serves category pages with lists of related tasks and a category list page `/categories/`.
- `/task` handled by `taskRouter` from the `task` module. It serves task details pages and handles challenge and quiz submissions. All submissions are guarded with `authenticated`. The details page is handled separately for anonymous and other users as described in section 4.4.

5.3.3 Markdown parser

To parse descriptions in Markdown the `marked` parser is used together with syntax highlighting library `highlight.js`. Because the processing can take some time and the execution runtime is single-threaded, it is offloaded to a worker thread. The thread is terminated after a timeout passes to avoid blocking the thread pool. [3, 2]

The first module responsible for this functionality `src/markedThread.js` listens to messages on `parentPort` and responds to them with an HTML string containing parsed Markdown.

The second module `src/markdown.js` exports a `processMarkdown` function, which manages the worker thread. It returns a `Promise`, which is resolved after the worker sends the rendered string. If the worker emits an error event, the promise is rejected. After a specified time passes the worker is terminated and the promise rejected, unless it had finished successfully earlier.

5.3.4 Database connection

The module `src/db.js` does not export any functions. When it is imported it creates a `MongoClient` connected to the database server and opens the database specified in the configuration file. It calls then an `async` function `setupDb`, which prepares necessary indexes and views. Both the client (`client`) and database (`db`) are exported. All modules importing either exported variable reuse the same client connection.

5.3.5 Challenge management

Challenges are managed using functions from the `src/challenges.js` module. The module has the following functions:

- `getChallengeContainers(all = false)` - returns challenge containers (only running, unless `all` is `true`). It relies on label `pl.krzysdz.inz.challenge-kind` being present.
- `startupChallenges()` - removes all nginx subdomain configuration files, stops and removes all challenge containers and calls `addChallenge` with each challenge document from the database.
- `addChallenge(challengeDoc, reload = false)` - verifies the image tag, then calls functions `startChallengeContainer` and `createNginxConfig`. If `reload` is `true`, calls a function responsible for reloading nginx configuration.
- `checkTag(challengeDoc)` - checks if the document contains an image with a tag. If there is no tag, `:latest` is appended to the image name. Returns the `challengeDoc` with corrected `taskImage` field.
- `startChallengeContainer(challengeDoc)` - pulls the image specified in the argument, stops and removes containers with the same name if they exist, creates a new container and starts it. If `resetInterval` is not 0, `setInterval` is created, which uses `restartChallengeContainer` to periodically restart the container.

- `restartChallengeContainer(port, challengeDoc)` - restarts the container specified in `challengeDoc`. This function is not exported and called only in interval, which is created after starting the container for the first time. It stops and removes the container including volumes, creates a new container reusing the same port and starts the new container.
- `createNginxConfig(subdomain, port)` - creates an nginx configuration file in `nginx/conf/subdomains` directory, which defines an HTTPS server proxying request made to the subdomain of domain specified in config to the specified port on localhost.
- `reloadNginx()` - executes the nginx binary with `-s reload` and prefix set to `./nginx` from the project directory to instruct the proxy to reload its configuration.

5.4 Sequence diagrams

Basic system functionality in a simplified form can be presented in form of UML sequence diagrams. Figures 5.3, 5.4 and 5.5 demonstrate flows related to authentication and account management. The next three diagrams 5.6, 5.7 and 5.8 present operations connected with tasks. The diagrams are simplified, to avoid repeating the process of fetching task and checking user progress before rendering the page.

Actions available for the administrators involve an additional participant - a script running in the browser which is responsible for managing the UI and communication with the server. Because the panel is not rendered on the server, the process of loading it is more complicated as can be seen on Fig. 5.9. The most complicated operation is adding a new task. The flow is presented on 5.10 and requires interactions with even more system components.

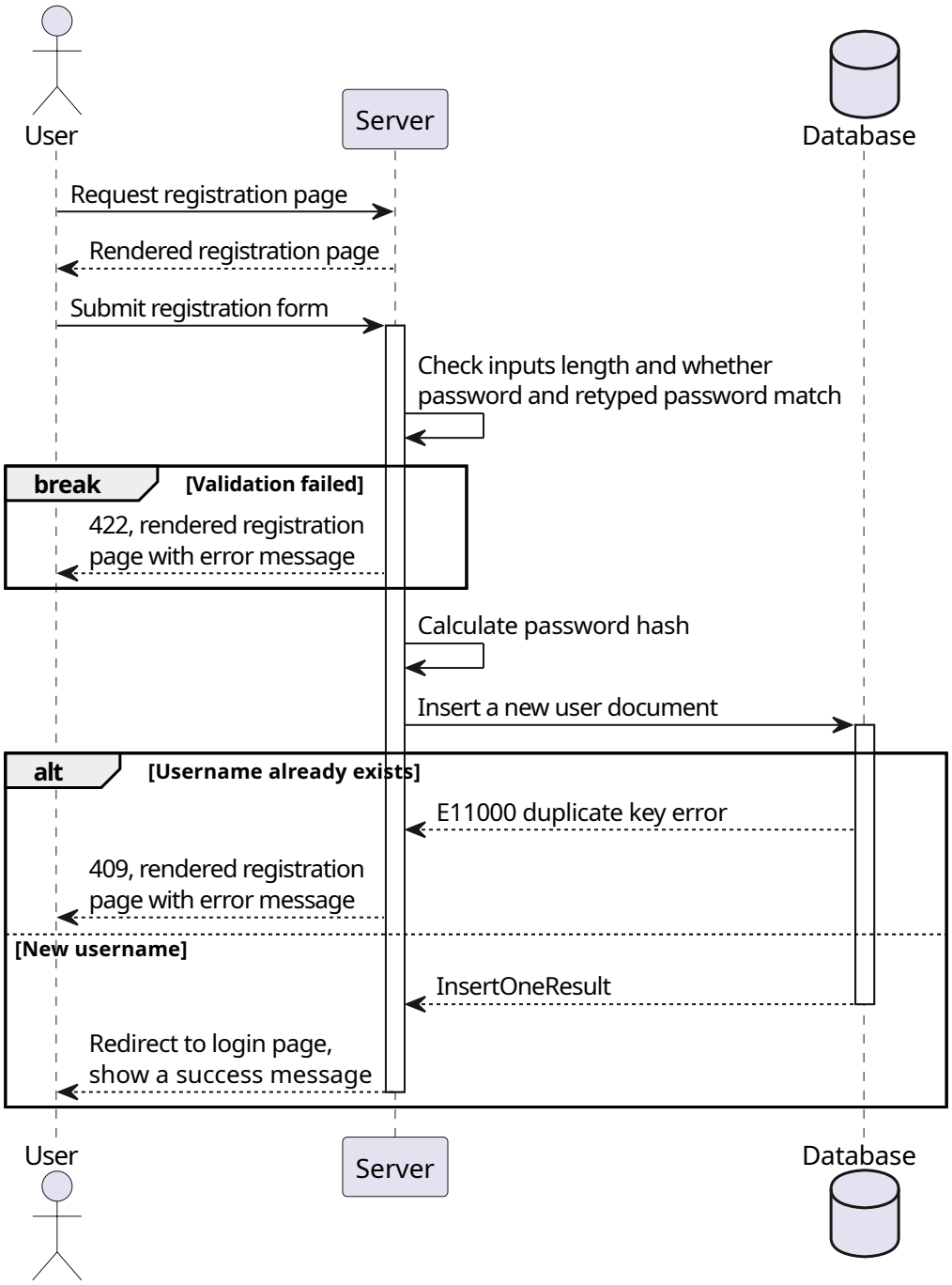

Figure 5.3: Registration sequence diagram

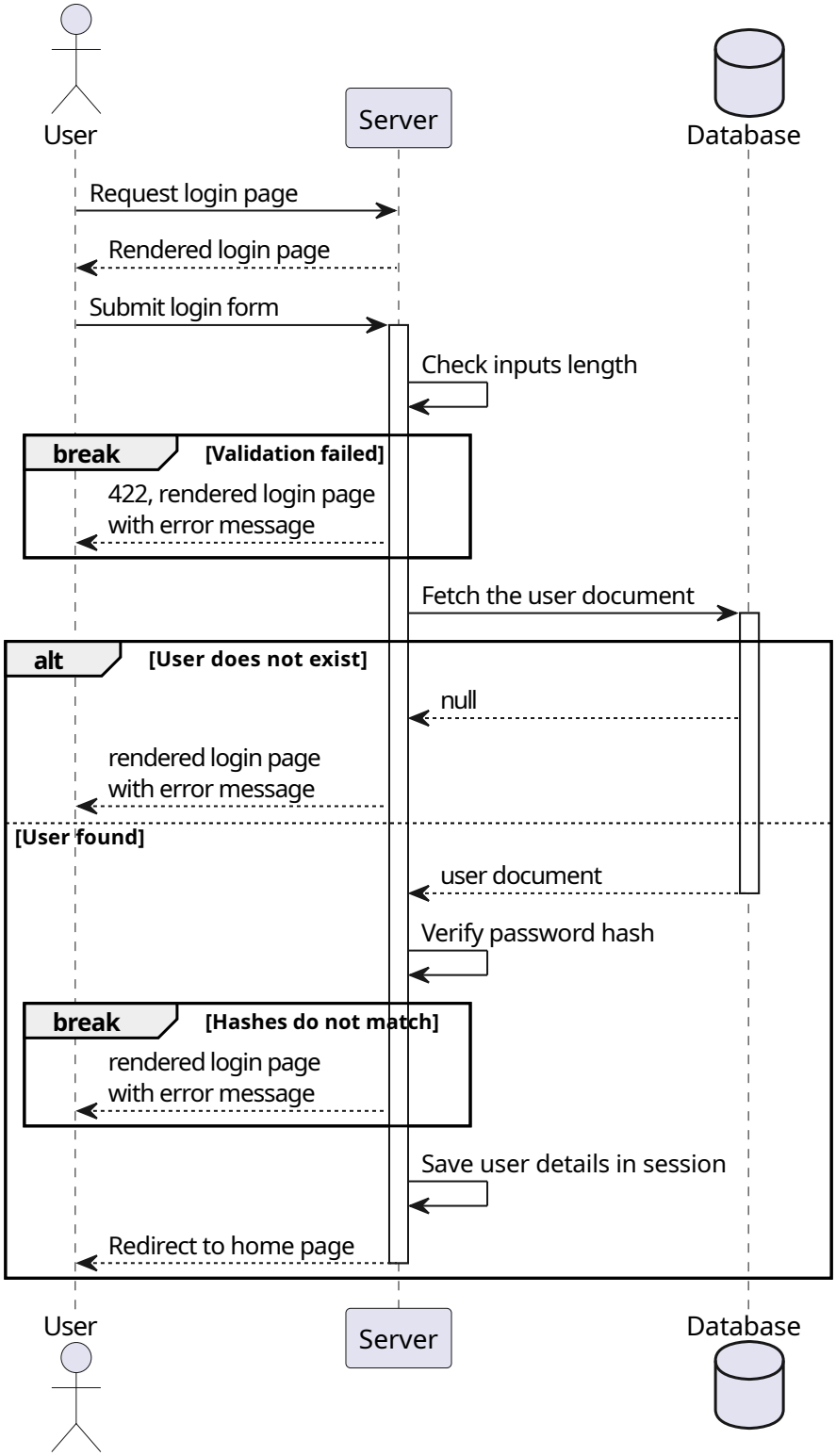

Figure 5.4: Logging in sequence diagram

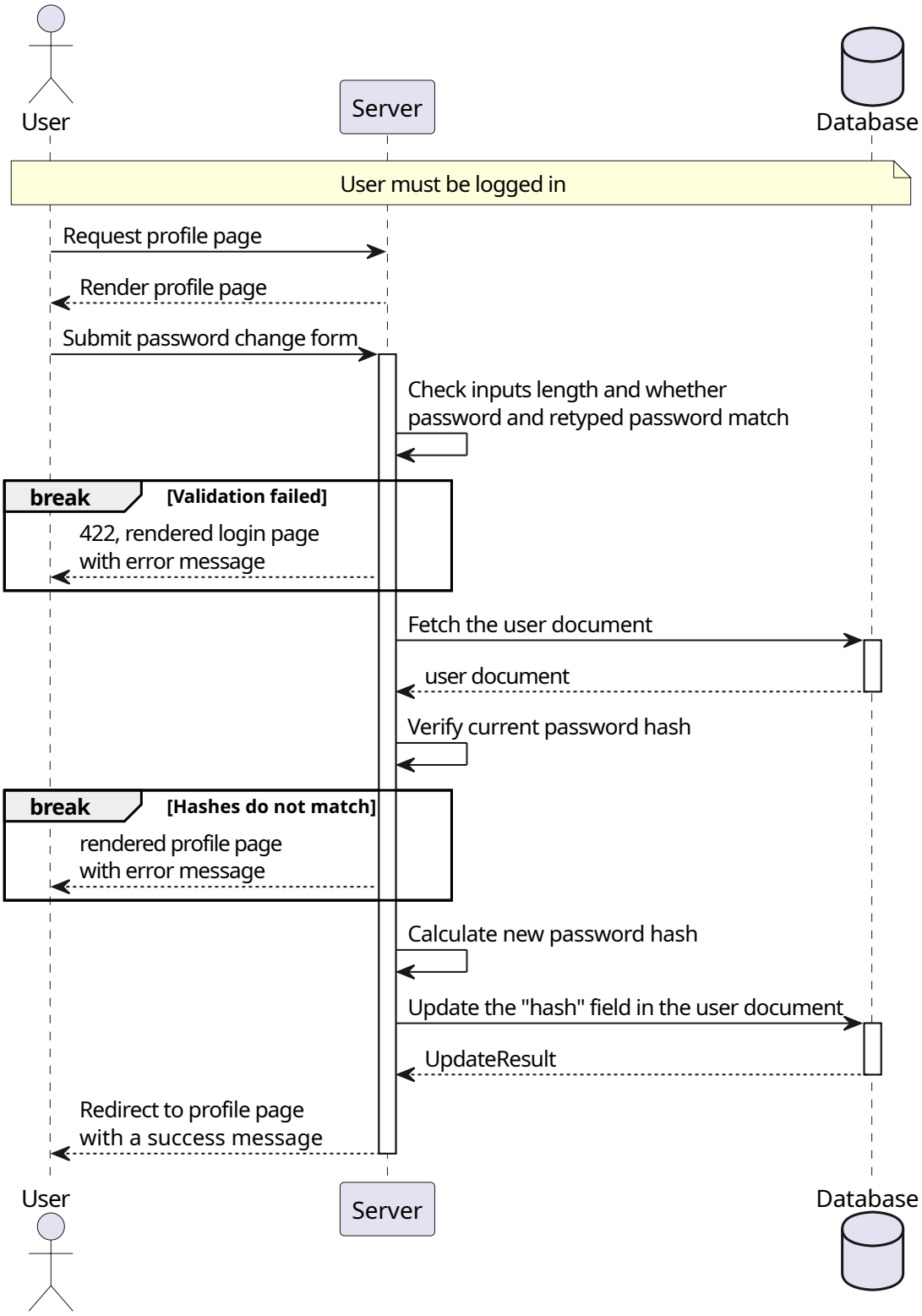

Figure 5.5: Password change sequence diagram

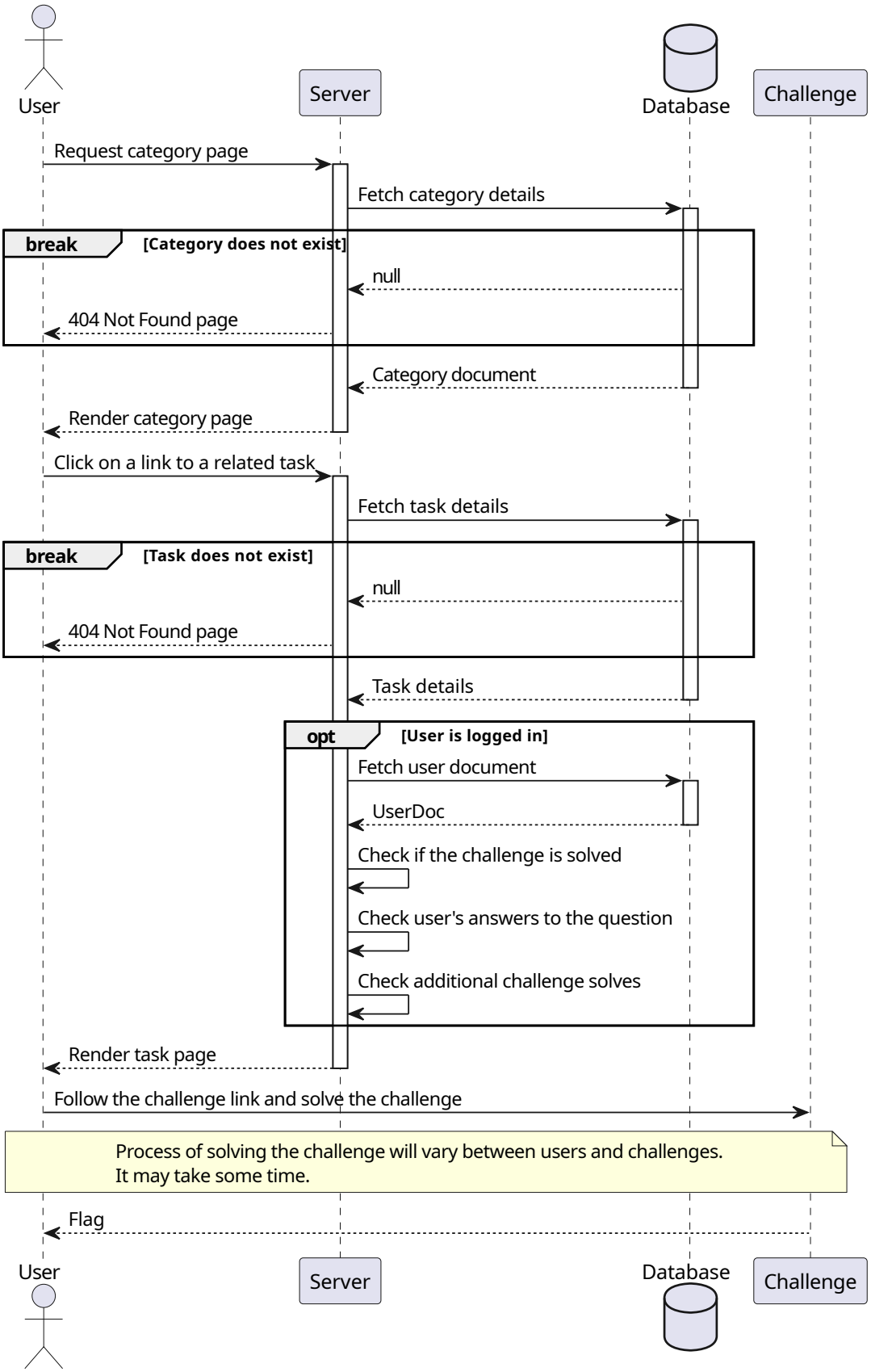

Figure 5.6: Navigation and solving a task

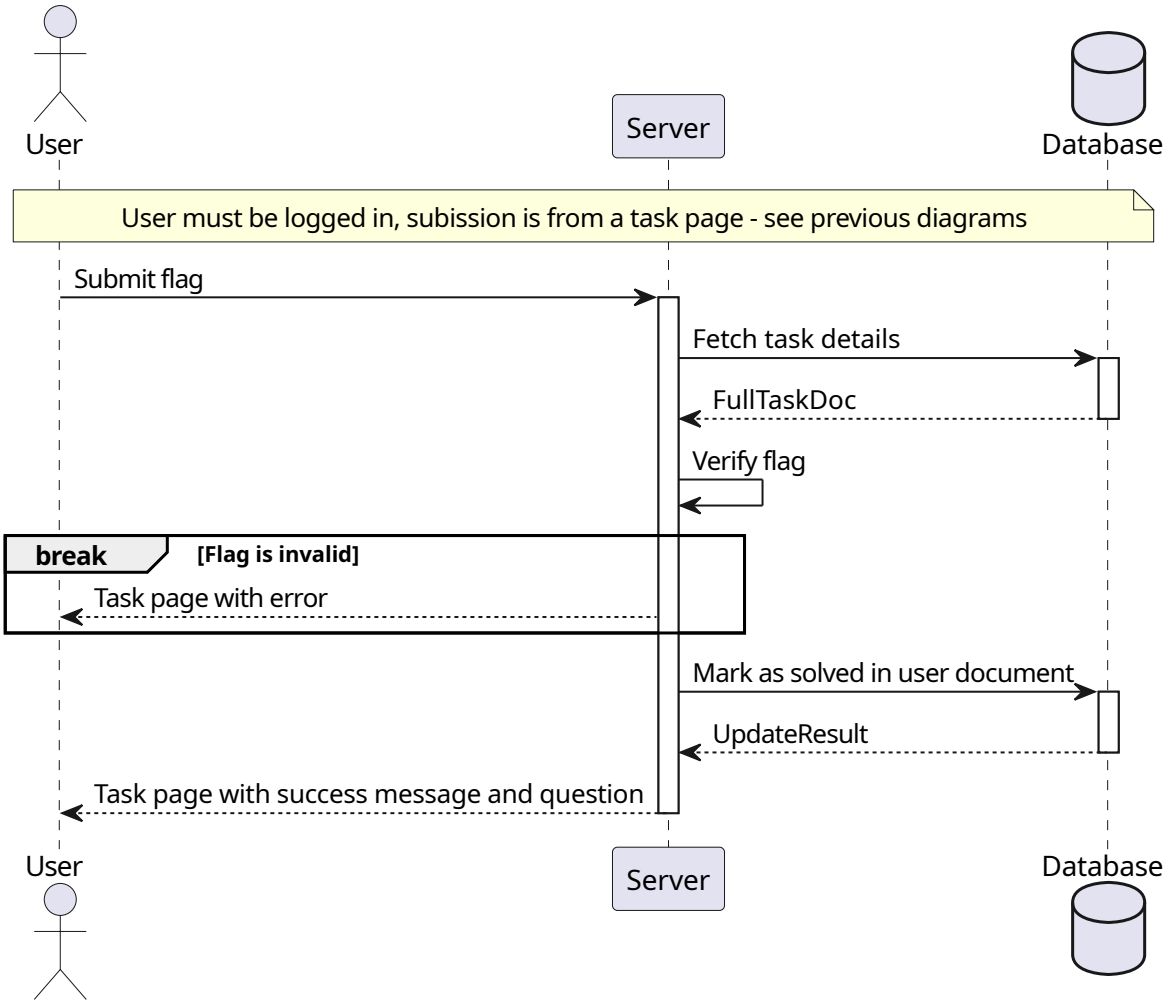

Figure 5.7: Submitting challenge flag

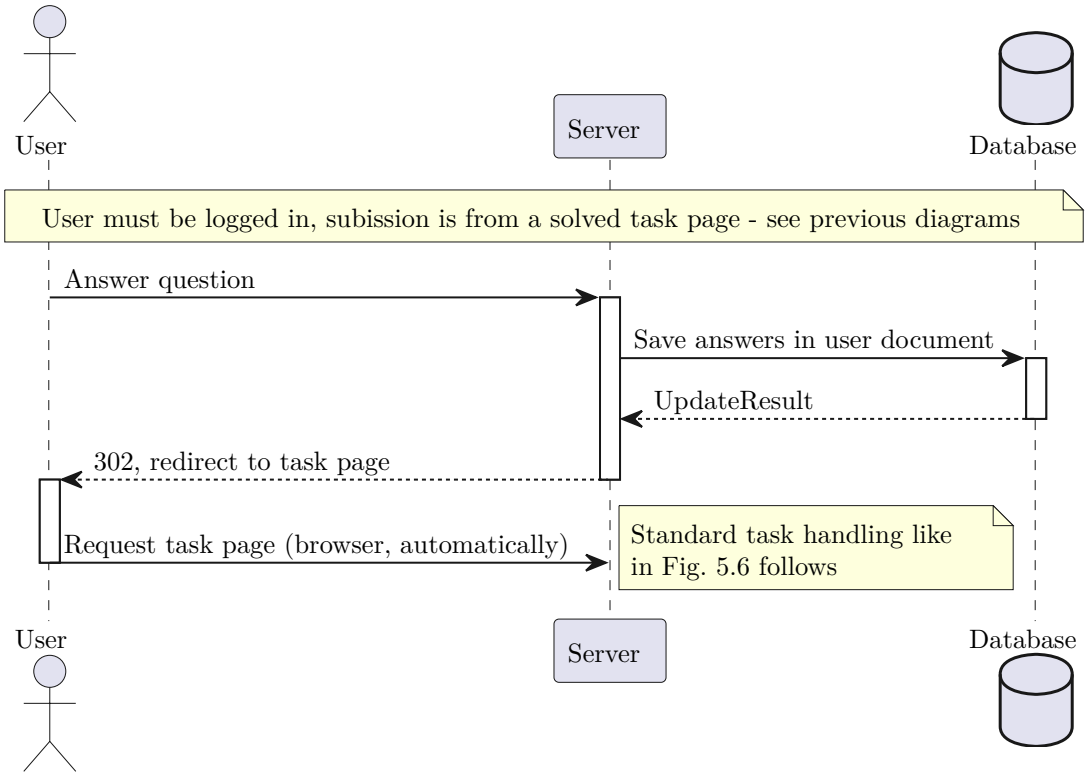

Figure 5.8: Answering task question

Figure 5.9: Loading administrator panel

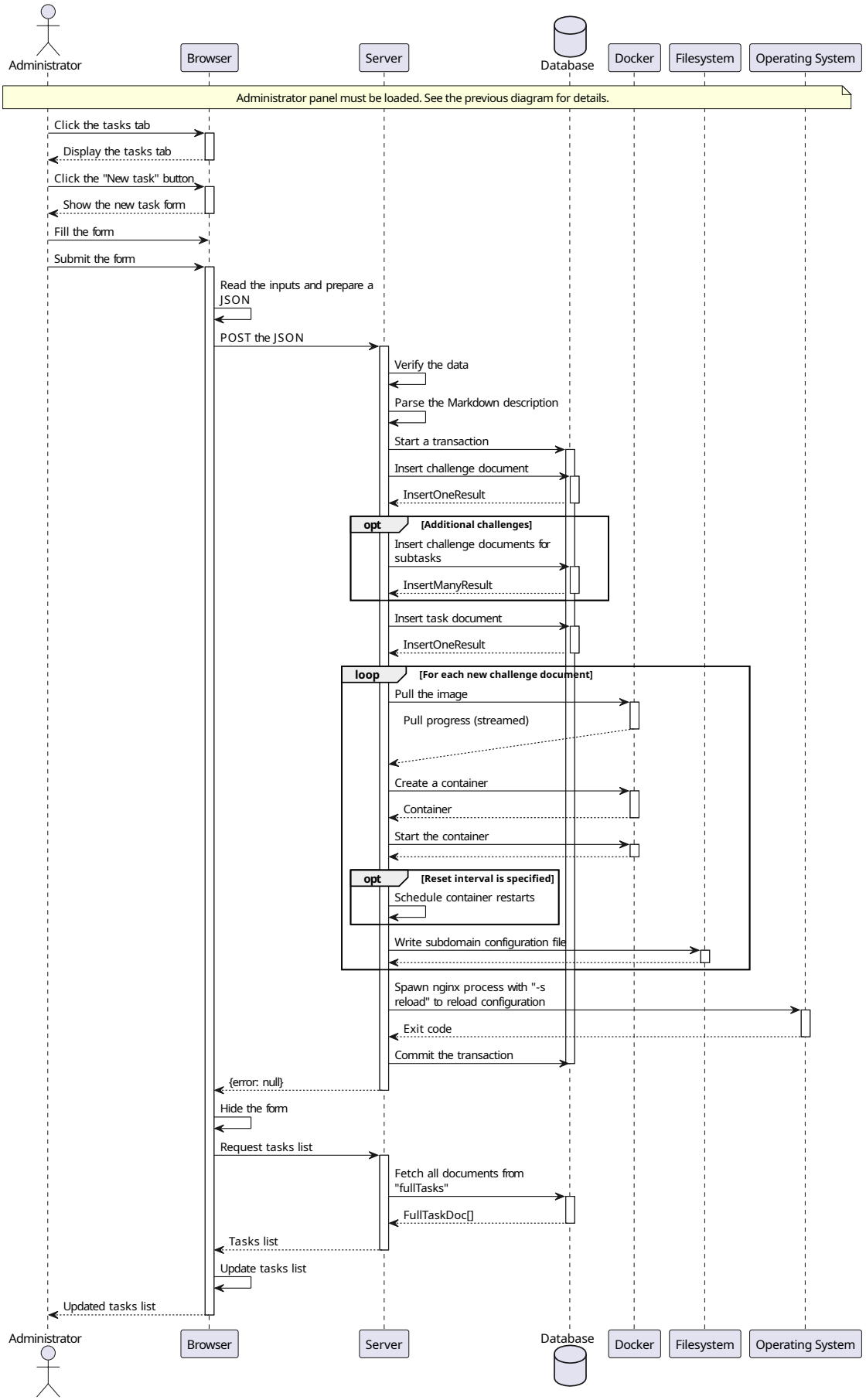

Figure 5.10: Adding a new task

Chapter 6

Verification and validation

- testing paradigm (eg V model)
- test cases, testing scope (full / partial)
- detected and fixed bugs
- results of experiments (optional)

Table 6.1: A caption of a table is **above** it.

	method						
			alg. 3			alg. 4, $\gamma = 2$	
ζ	alg. 1	alg. 2	$\alpha = 1.5$	$\alpha = 2$	$\alpha = 3$	$\beta = 0.1$	$\beta = -0.1$
0	8.3250	1.45305	7.5791	14.8517	20.0028	1.16396	1.1365
5	0.6111	2.27126	6.9952	13.8560	18.6064	1.18659	1.1630
10	11.6126	2.69218	6.2520	12.5202	16.8278	1.23180	1.2045
15	0.5665	2.95046	5.7753	11.4588	15.4837	1.25131	1.2614
20	15.8728	3.07225	5.3071	10.3935	13.8738	1.25307	1.2217
25	0.9791	3.19034	5.4575	9.9533	13.0721	1.27104	1.2640
30	2.0228	3.27474	5.7461	9.7164	12.2637	1.33404	1.3209
35	13.4210	3.36086	6.6735	10.0442	12.0270	1.35385	1.3059
40	13.2226	3.36420	7.7248	10.4495	12.0379	1.34919	1.2768
45	12.8445	3.47436	8.5539	10.8552	12.2773	1.42303	1.4362
50	12.9245	3.58228	9.2702	11.2183	12.3990	1.40922	1.3724

Chapter 7

Conclusions

- achieved results with regard to objectives of the thesis and requirements
- path of further development (eg functional extension ...)
- encountered difficulties and problems

Bibliography

[1] Ahmad Nassri. *So long, and thanks for all the packages!* 2020. URL: https://blog.npmjs.org/post/615388323067854848/so-long-and-thanks-for-all-the-packages (visited on 19/01/2023).

[2] Node.js. *Don't Block the Event Loop (or the Worker Pool).* URL: https://nodejs.org/en/docs/guides/dont-block-the-event-loop/ (visited on 27/01/2023).

[3] Node.js. *The Node.js Event Loop, Timers, and process.nextTick().* URL: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/ (visited on 27/01/2023).

[4] Name Surname and Name Surname. *Title of a book*. Hong Kong: Publisher, 2017. ISBN: 83-204-3229-9-434.

[5] Name Surname and Name Surname. 'Title of an article in a journal'. In: *Journal Title* 157.8 (2016), pp. 1092–1113.

[6] Name Surname, Name Surname and N. Surname. 'Title of a conference article'. In: *Conference title*. 2006, pp. 5346–5349.

[7] Name Surname, Name Surname and N. Surname. *Title of a web page*. 2021. URL: http://somewhere/on/the/internet.html (visited on 30/09/2021).

Appendices

Index of abbreviations and symbols

API	Application Programming Interface
BSON	Binary JSON
CLI	Command-line Interface
CSS	Cascading Style Sheet
CTF	Capture The Flag
DNS	Domain Name System
HTML	HyperText Markup Language
HTTPS	Hypertext Transfer Protocol Secure
JS	JavaScript
JSON	JavaScript Object Notation
REST	Representational State Transfer
SQL	Structured Query Language
TLS	Transport Layer Security
UI	User Interface

Listings

(Put long listings here.)

```
# mongod.conf

# for documentation of all options, see:
#   http://docs.mongodb.org/manual/reference/configuration-options/

# Where and how to store data.
storage:
  dbPath: /var/lib/mongodb

# where to write logging data.
systemLog:
  destination: file
  logAppend: true
  path: /var/log/mongodb/mongod.log

# network interfaces
net:
  port: 27017
  bindIp: 127.0.0.1

# how the process runs
processManagement:
  timeZoneInfo: /usr/share/zoneinfo

replication:
  replSetName: rs0
```

Figure 1: Example `/etc/mongod.conf` configuration file. Based on the default config for Debian and Ubuntu

```
# /etc/systemd/system/inz-nginx.service

[Unit]
Description=nginx proxy for inz
Documentation=https://nginx.org/en/docs/
After=network-online.target remote-fs.target nss-lookup.target
Wants=network-online.target

[Service]
Type=forking
PIDFile=/run/nginx.pid
WorkingDirectory=/inz
ExecStart=/usr/sbin/nginx -p /inz/nginx -c /inz/nginx/conf/nginx.conf
ExecReload=/bin/sh -c "/bin/kill -s HUP $(/bin/cat /run/nginx.pid)"
ExecStop=/bin/sh -c "/bin/kill -s TERM $(/bin/cat /run/nginx.pid)"

[Install]
WantedBy=multi-user.target
```

Figure 2: Example unit file for the nginx proxy. Project root is assumed to be `/inz`.

```
# /etc/systemd/system/inz.service

[Unit]
Description=inz server
After=network-online.target remote-fs.target nss-lookup.target
 ↪  mongod.service inz-nginx.service
Wants=network-online.target
Requires=mongod.service inz-nginx.service

[Service]
Type=simple
WorkingDirectory=/inz
Environment="NODE_ENV=production"
# Remember to change the secret!
Environment="SECRET=SECRET"
ExecStart=/usr/bin/node /inz/index.js
ExecStop=/bin/kill -s TERM $MAINPID

[Install]
WantedBy=multi-user.target
```

Figure 3: Example unit file for the main server. Project root is assumed to be `/inz`.

```
import { MongoClient } from "mongodb";
import { DB_NAME, DB_URL } from "./config.js";

const USERNAME = "administrator_username";

const client = new MongoClient(DB_URL, { directConnection: true });
await client
        .db(DB_NAME)
        .collection("users")
        .updateOne({ _id: USERNAME }, { $set: { role: "user" } });
await client.close();
```

Figure 4: A simple Node.js script, which makes the user `administrator_username` an administrator.

```
[
    // add challenge as "challenge" array of 1 document
    { $lookup: {
            from: "challenges",
            localField: "challengeId",
            foreignField: "_id",
            as: "challenge",
    }},
    // add challenges from answers as "answerChallenges" array
    { $lookup: {
            from: "challenges",
            localField: "answers.challengeId",
            foreignField: "_id",
            as: "answerChallenges",
    }},
    // select and modify returned fields
    { $project: {
            name: 1,
            categoryId: 1,
            descriptionMd: 1,
            descriptionRendered: 1,
            question: 1,
            hints: 1,
            visible: 1,
            // Replace the "challenge" array with the only document inside it
            challenge: { $first: "$challenge" },
            // Add a challenge field (subdocument) to each answer with challengeId
            answers: {
                $map: {
                    input: "$answers",
                    as: "a",
                    in: {
                        $mergeObjects: [
                            "$$a",
                            { challenge: {
                                    $first: {
                                        $filter: {
                                            input: "$answerChallenges",
                                            cond: { $eq: ["$$this._id", "$$a.challengeId"] },
                                            limit: 1,
}}}}]}}}}}]
```

Figure 5: Aggregation pipeline for the `fullTasks` collection.

List of additional files in electronic submission (if applicable)

Additional files uploaded to the system include:

- source code of the application,
- test data,
- a video file showing how software or hardware developed for thesis is used,
- etc.

List of Figures

List of Tables