# RabbitMQ Infrastructure
Documentation

Paolo Bettelini
Scuola d'Arti e Mestieri di Trevano (SAMT)

# Contents

# 1 Introduction

## 1.1 Abstract

The goal of this project is to make a network infrastructure which extensively uses a messaging queue system (RabbitMQ). My additional personal requirement is to use the Rust programming language as much as possible.

## 1.2 Information

This is a project of the Scuola Arti e Mestieri di Trevano (SAMT) under the following circumstances

- **Section**: Computer Science
- **Year:** Fourth
- **Class:** Progetti Individuali
- **Supervisor:** Geo Petrini
- **Title:** RabbitMQ prototype
- **Start date**: 2022-09-29
- **Deadline**: 2022-12-07

and the following requirements

- **Documentation**: a full documentation of the work done
- **Diary**: constant changelog for each work session
- **Source code**: working source code of the project

All the source code and documents can be found at http://gitsam.cpt.local/2022_2023_1_semestre/prototipo-rabbitmq [**gitrepo**].

## 1.3 Structure

This document is structured as such:

1. **Introduction:** General information, requirements and scope of the project
2. **Analysis:** Analysis

# 2 Analysis

## 2.1 Requirements

| Req-00 | |
|---|---|
| **Name** | Login & Register |
| **Priority** | 1 |
| **Version** | 1.0 |
| **Notes** | none |
| **Description** | The user must be able to create an account and log in. |
| **Subrequirements** | |
| **Req-00__0** | The Authentication must be kept alive by a cookie. |
| **Req-00__1** | The keep-alive cookie must contains a randomly generated token. |
| **Req-00__2** | The password must be hashed client-side. |

| Req-01 | |
|---|---|
| **Name** | Functionality |
| **Priority** | 1 |
| **Version** | 1.0 |
| **Notes** | none |
| **Description** | The website must contain a file dropzone. The user must be able to upload an image which will be converted into a 200x200 px webp. |
| **Subrequirements** | |
| **Req-01__0** | During the conversion an async progress status must be display. |

| Req-02 | |
|---|---|
| **Name** | Message Queues |
| **Priority** | 1 |
| **Version** | 1.0 |
| **Notes** | none |
| **Description** | Every message between WebServer and Worker must be through message queues. |

| Req-03 | |
|---|---|
| **Name** | List of images |
| **Priority** | 1 |
| **Version** | 1.0 |
| **Notes** | none |
| **Description** | When the users logs in a list of the previously converted images must be display. |
| **Subrequirements** | |
| **Req-03__0** | Only the last N images are loaded. Another chunk of images is loaded if requested by the user. |

| Req-04 | |
|---|---|
| **Name** | Network Structure |
| **Priority** | 1 |
| **Version** | 1.0 |
| **Notes** | none |
| **Description** | A loadbalancer (Round Robin) is the entry point for $N$ WebServers. For each WebServer there exist a RabbitMQ server. There are $M$ workers which access the queues of the queue servers. Each worker stores data on the same database. |

| Req-05 | |
|---|---|
| **Name** | Scalability |
| **Priority** | 1 |
| **Version** | 1.0 |
| **Notes** | none |
| **Description** | The network must scale with multiple servers. |

# 3 Planning

## 3.1 Initial Gantt Chart

## 3.2 Final Gantt Chart

# 4 Infrastructure

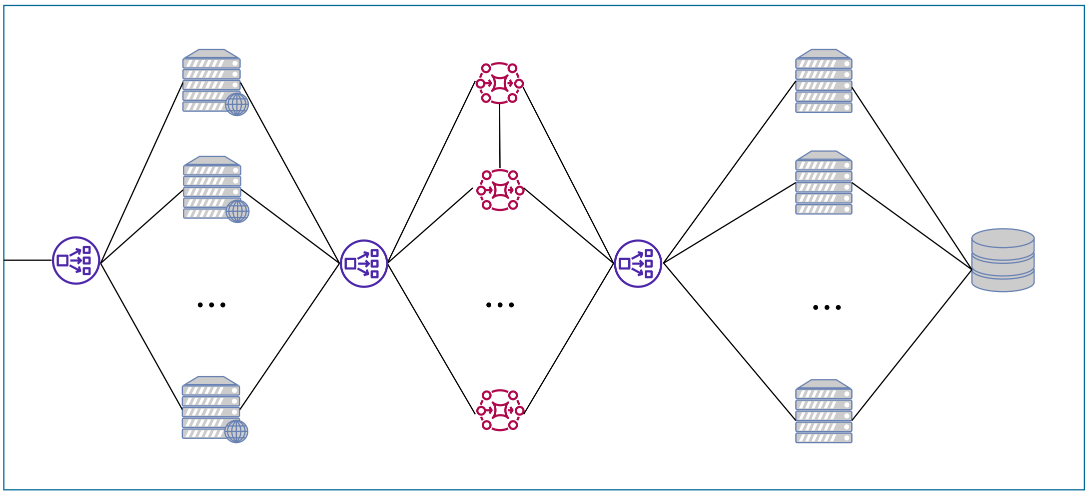


Figure 1: Network Infrastructure

# 5 Technologies

## 5.1 WebAssembly

WebAssembly (wasm) is a portable low-level language supported by all majors browsers. It can be used for a variety of things within the browser and can be mixed with HTML and JavaScript. WebAssembly does not need to be parsed by the browser since it is already in a binary format.
At the time fo writing, WebAssembly is not widely used and it is not always faster than JavaScript based applications. However, is it my opinion and hope that it will better in the future and that it will become a standard in web development.

## 5.2 Rust

Rust is a generic compiled programming language. The code is compiled using LLVM to machine code and its speed is comparable to C and C++. Rust is the first programming language to guarantee memory safety; memory is not manually freed nor garbage collected. It is not possible to dereference a null pointer, cause memory segfaults, core dumps and memory leaks. Code that could cause undefined behavior can still be written, but it is strictly bounded in blocks where the compiler is relaxed. This relaxation implies that the language is also low-level. Another key feature to the performance of Rust is zero cost abstraction, which means that generic types and function abstractions are resolved at compile-time. Conditional compilation and compile-time computations are also extensively used.
There are also many features concerning the programming experience, such as advanced metaprogramming and code generation using macros, intelligent compiler, dependency system (Cargo), modern syntax and many tools to ease development.

**Note**: a Rust *crate* refers to a library. A *feature* is an optional component of library. A *module* is a logical section of a program or library.

## 5.3 RabbitMQ

RabbitMQ is a popoular message broker implementing many messaging protocols. Message brokers such as RabbitMQ can make an infrastructure to route messages, validate them and transform them. Messages queue are used to store messages. Multiple consumers may consume messages from a queue.
RabbitMQ servers can also form a cluster. `todo`

# 6 Implementation

## 6.1 Frontend

### 6.1.1 Generating WebAssembly

WebAssembly code is copiled from Rust code using the `wasm-pack` tool. The rust code uses the `wasm_bindgen` crate to bind to WebAssembly. A function to export into the module might be written as

```rust
#[wasm_bindgen]
pub fn hash(value: String) -> String {
    let data = value.as_bytes().to_vec();

    let digest = sha256(&data);

    to_base64(digest)
}
```

Compiling using

```
wasm-pack build
```

will produce a folder named `pkg/` which contains the wasm module.

### 6.1.2 Importing the module

I used `webpack` to integrate the wasm module in the website and be able to call wasm function from JavaScript. `npm` is used to handle the dependencies.

**package.json**

```json
{
    "name": "webapp-frontend",
    "version": "0.1.0",
    "description": "Frontend",
    "main": "index.js",
    "scripts": {
        "build": "webpack --config webpack.config.js"
    },
    "author": "Paolo Bettelini",
    "devDependencies": {
        "webpack": "^5.74.0",
        "webpack-cli": "^4.10.0",
        "copy-webpack-plugin": "^11.0.0"
    },
    "dependencies": {
        "frontend": "file:../pkg"
    }
}
```

**webpack.config.js**

```js
const CopyWebpackPlugin = require("copy-webpack-plugin");
const path = require('path');

module.exports = {
  entry: {
    login: "./www/login.js",
    register: "./www/register.js",
    upload: "./www/upload.js",
```

```
    gallery: "./www/gallery.js"
  },
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "[name].bundle.js",
  },
  mode: "development",
  plugins: [
    new CopyWebpackPlugin({
      patterns: [ "www" ],
    })
  ],
  experiments: {
    asyncWebAssembly: true
  }
};
```

To compile the website to static files run

```
npm run build
```

To call a wasm function within the file `login.js` we can do the following.

```
import { hash } from 'frontend'

console.log(hash('Hello World'));
```

The compiled file is called `login.bundle.js` which is what the HTML page will need to include (see webpack config).

### 6.1.3 Website

The following image shows the index page when the user is logged in.

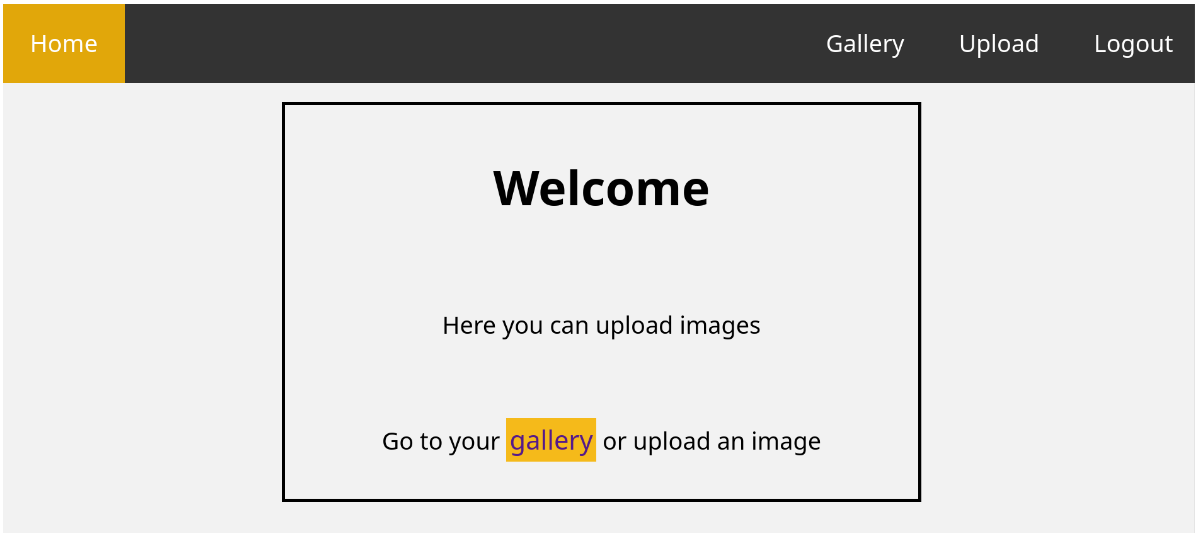


Figure 2: Index page - user logged in

The following image shows the index page when the user is not logged in.

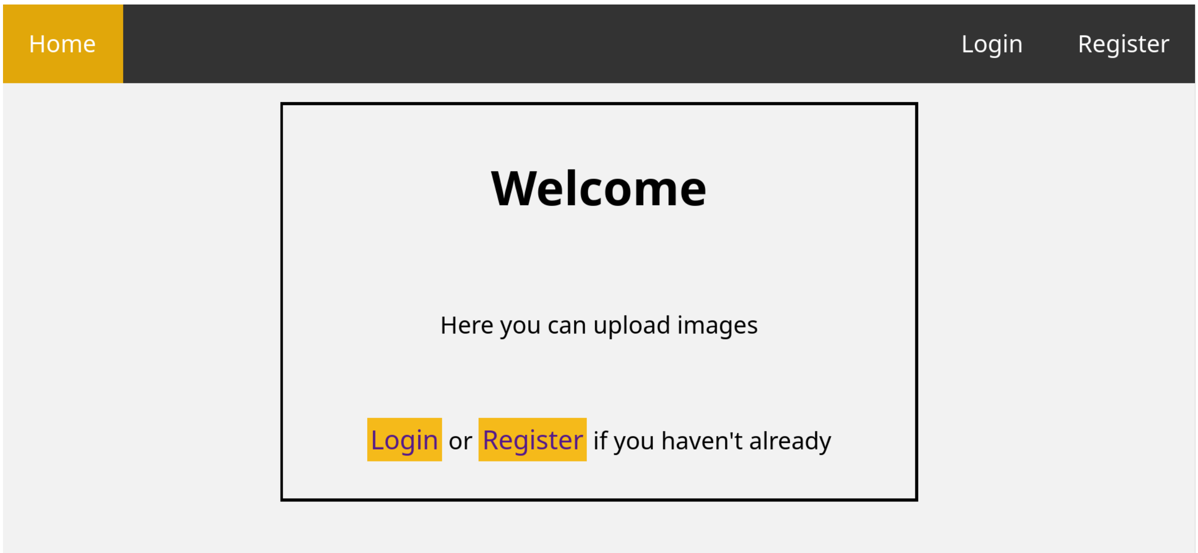


Figure 3: Index page - user not logged in

The following image shows the logout page.

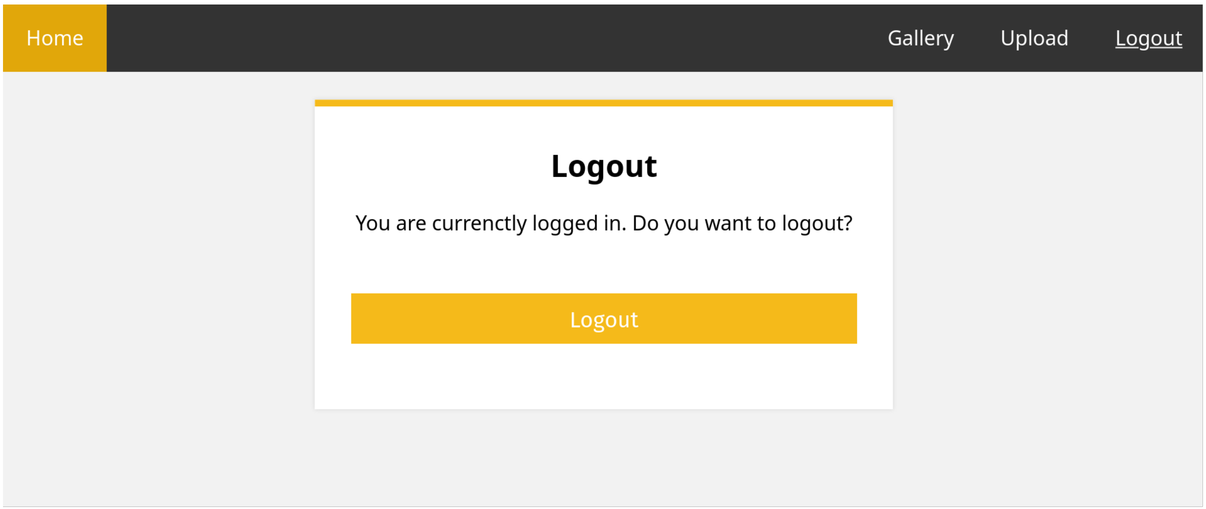


Figure 4: Logout page

The following image shows the login page.

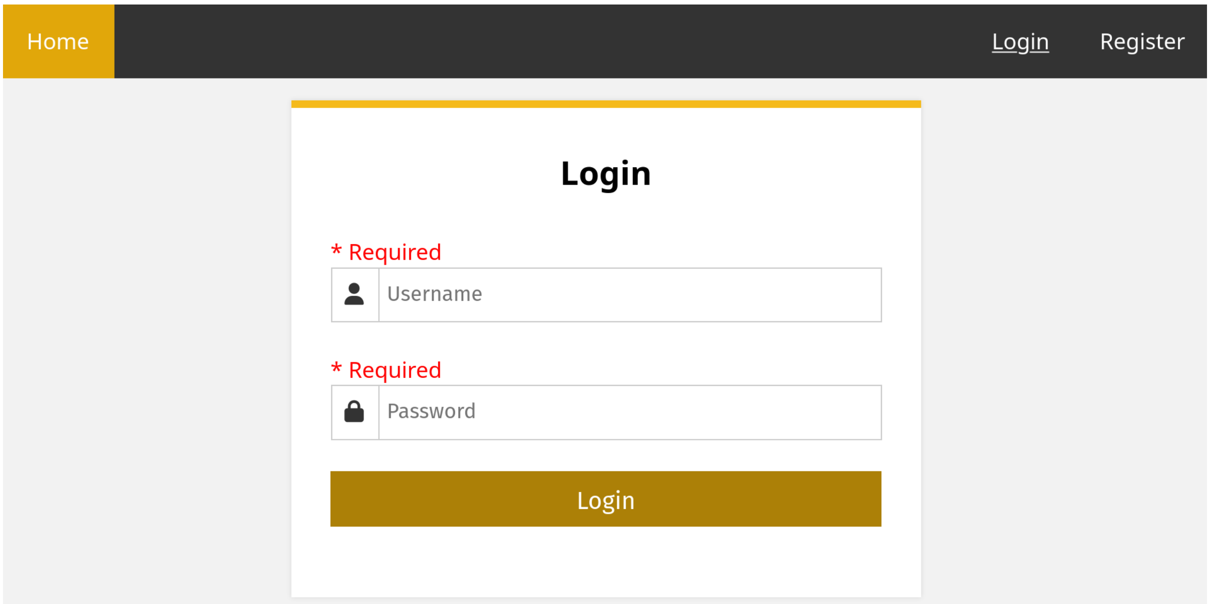


Figure 5: Login page

The following image shows the register page.

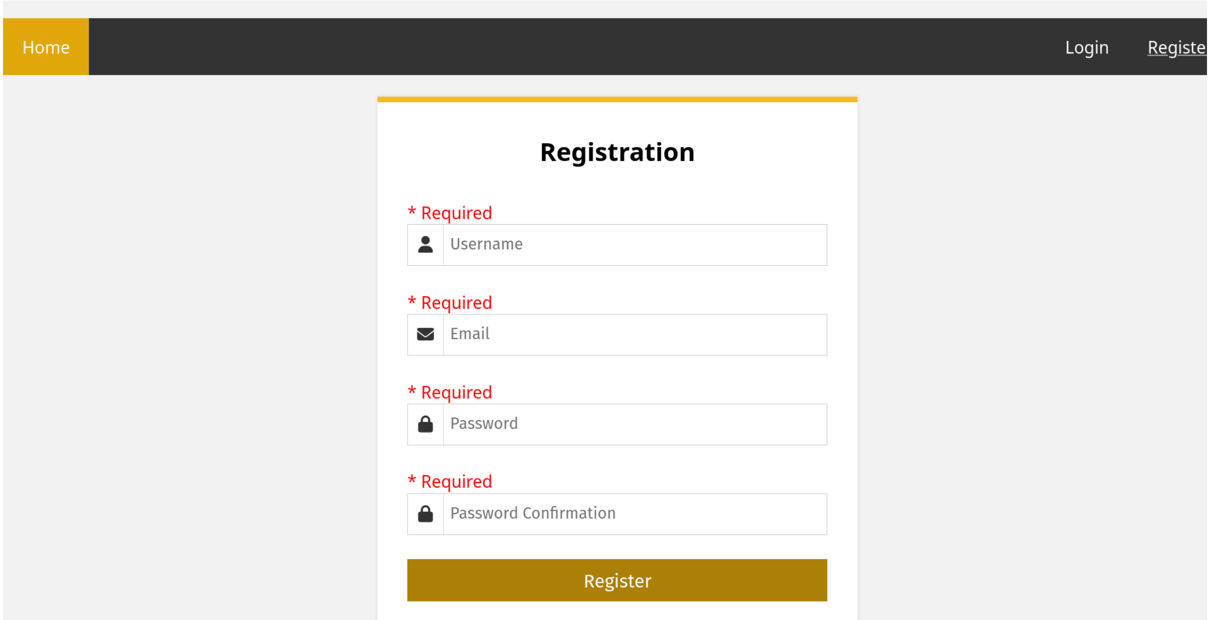


Figure 6: Register page

The following image shows the upload page. No images have been uploaded.

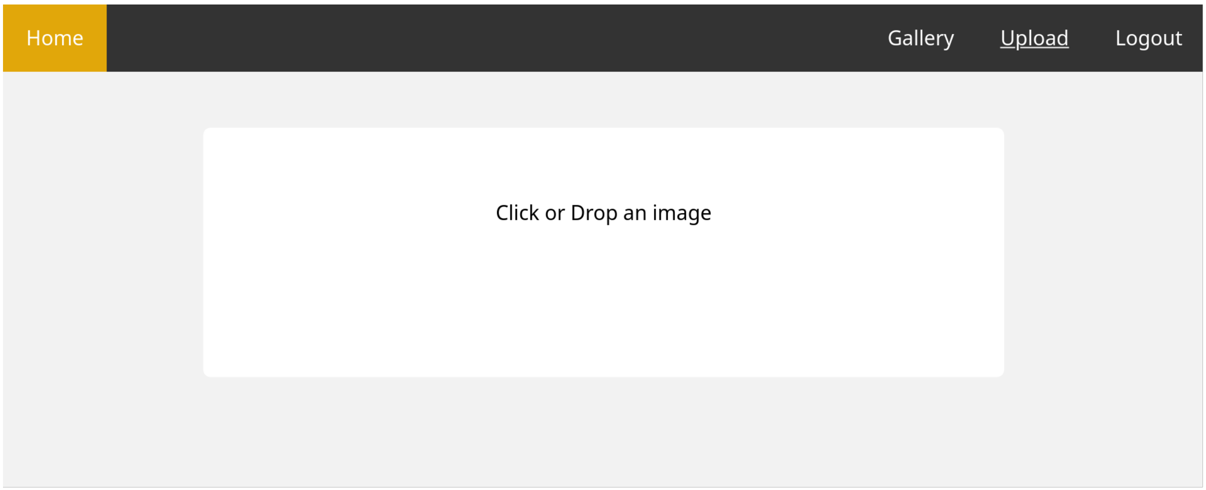


Figure 7: Upload page - empty

The following image shows the upload page. Three images have been uploaded.

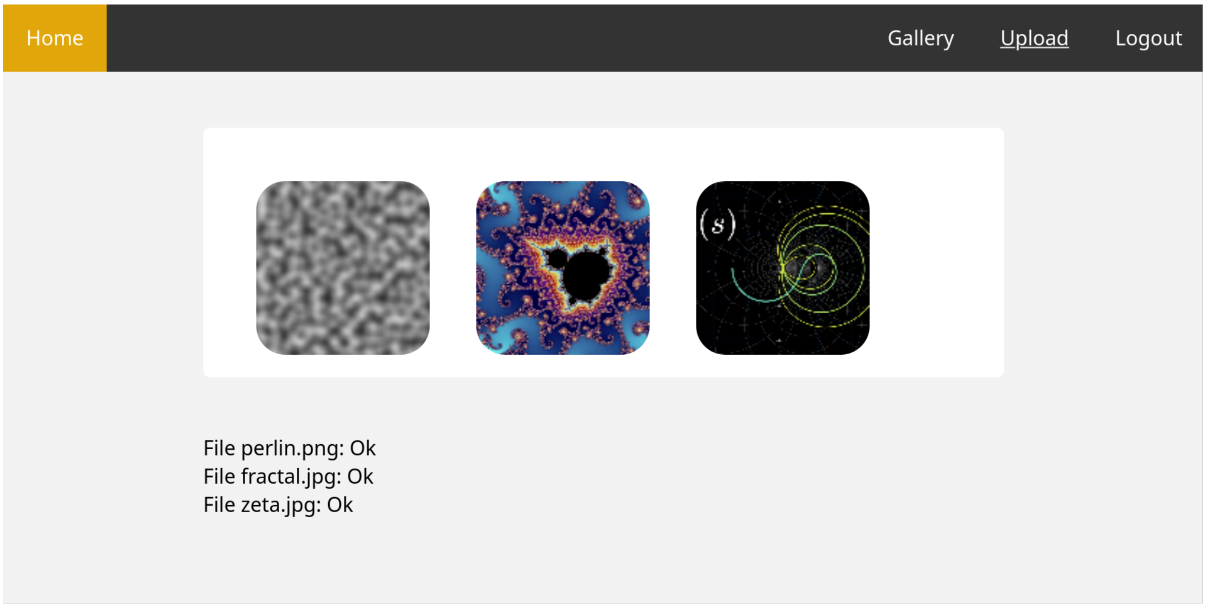


Figure 8: Upload page - full

The following image shows the gallery page. 6 images are loaded at a time. There is a button to load more images. If the are no images remaining the button disappears.

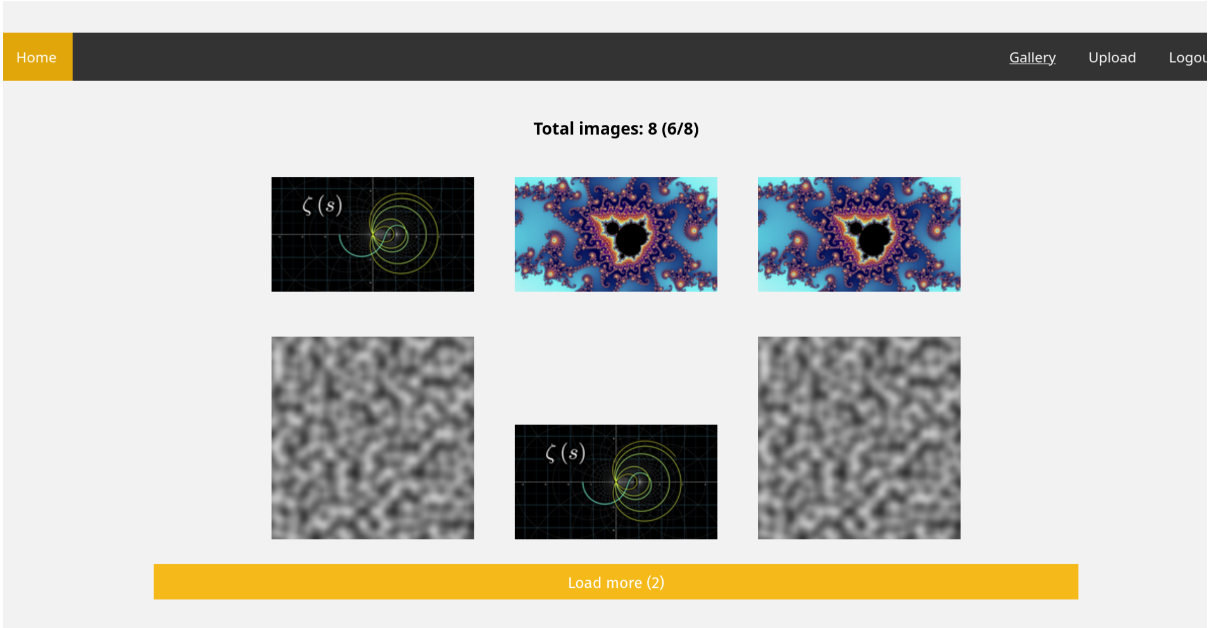


Figure 9: Gallery page

## 6.2 Webserver

### 6.2.1 Templating

Templating is used to programmatically serve HTML content based on some logic. To do so a template engine is needed. The template engine renders the HTML content when needed.

I used a template engine library for Rust called tera. Logic blocks can be integrated in the HTML file like so

```
<ul>
{% for user in users %}
    <li><a href="{{ user.url}}">{{ user.url }}</li>
{% endfor %}
</ul>
```

HTML files containing templating needs to be stored in RAM. When the webservers starts it loads from the `www` folder every file containing templating code.

### 6.2.2 Routing using warp

The webserver needs to respond to different routes. I used a composable Rust framework called warp [**warp**].

The routes are the following:

- / → Serve index page
- /**register** → Serve register page
- /**login** → Serve login page
- /**logout** → Serve logout page
- /**upload** → Serve upload page
- /**gallery** → Serve gallery page
- /**api**/**register** → Register action

- **/api/login** → Login action
- **/api/logout** → Logout action
- **/api/image/<id>** → Get image action
- **/<file>** → Serve static file
- **/index.html** → Block action
- **/register.html** → Block action
- **/login.html** → Block action
- **/logout.html** → Block action
- **/upload.html** → Block action
- **/gallery.html** → Block action

## 6.3 Database

The database is an instance of `MariaDB`.

### 6.3.1 Diesel

`diesel` is an ORM library for the Rust programming language. It supports MySQL, Postgres and SQLite and can manage migrations.

Diesel comes with a CLI tool to manage migrations. A configuration file (`diesel.toml`) may be placed in the cargo project.

```
[migrations_directory]
dir = "migrations" # folder containing the migrations
```

A table with the name `__diesel_schema_migrations` is automatically created on the database to keep track of all the migrations run.

#### Creating a migration

```
diesel migration generate <name>
```

This command will generate a migration in the migration folder with the current timestamp. The files `up.sql` and `down.sql` created.

#### Executing migrations

```
diesel migration <run|redo|revert>
```

This command will run, redo or revert the migration on the database. The database service address must be passed using the `--database-url` parameter or by setting the `DATBASE_URL` enviroment variable.

#### Generating schema file

```
diesel print-schema > src/schema.rs
```

This command will generate the `schema.rs` file. This file is produced from the databased and is used to perform compiled-time checked queries. The database service address must be passed using the `--database-url` parameter or by setting the `DATBASE_URL` enviroment variable.

`up.sql`

```sql
CREATE TABLE user (
    id INT PRIMARY KEY AUTO_INCREMENT,
    mail VARCHAR(50) NOT NULL,
    username VARCHAR(25) NOT NULL,
    password BINARY(32) NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE image (
    id INT PRIMARY KEY AUTO_INCREMENT,
    user_id INT NOT NULL,
    uploaded_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    data BLOB NOT NULL,
    FOREIGN KEY (user_id)
        REFERENCES user(id)
            ON UPDATE CASCADE
            ON DELETE CASCADE
);
```

`down.sql`

```sql
DROP TABLE image;
DROP TABLE user;
```

The migration can be included in the code at compile time using a macro and run at the start of the program, like so

```
fn run_embedded_migrations(connection: &mut MysqlConnection) {
    const MIGRATIONS: EmbeddedMigrations = embed_migrations!();

    connection.run_pending_migrations(MIGRATIONS).unwrap();
}
```

The traits `Queryable` and `Insertable` can be automatically derived for structures, such that diesel can execute queries and inserts directly with the structures themselves.

```
#[derive(Queryable, Debug)]
#[diesel(table_name = user)]
pub struct User {
    pub id: i32,
    pub mail: String,
    pub username: String,
    pub password: Vec<u8>,
    pub token: Vec<u8>,
    pub created_at: NaiveDateTime,
}

#[derive(Insertable)]
#[diesel(table_name = user)]
pub struct NewUser<'a> {
    pub mail: &'a str,
    pub username: &'a str,
    pub password: &'a Vec<u8>,
    pub token: &'a Vec<u8>,
}

#[derive(Queryable, Debug)]
#[diesel(belongs_to(User))]
#[diesel(table_name = image)]
pub struct Image {
    pub id: i32,
    pub user_id: i32,
    pub uploaded_at: NaiveDateTime,
    pub data: Vec<u8>,
}

#[derive(Insertable)]
#[diesel(belongs_to(User))]
#[diesel(table_name = image)]
pub struct NewImage<'a> {
    pub id: i32,
    pub user_id: i32,
    pub data: &'a Vec<u8>,
}
```

Queries and inserts are executed using the schema file.

```
use crate::schema::user::{username, token, dsl::user};
use diesel::select;
// select token of user with a given username
let result: Result<Vec<u8>, _> = user
```

```
        .select(token)
        .filter(username.eq(name))
        .first(connection);
```

## 6.4 Load Balancer

## 6.5 Backend

## 6.6 Messaging

### 6.6.1 RabbitMQ

### 6.6.2 Request/Reply Pattern

A common requirement within a messaging system is a request/reply pattern. A client must be able to publish a message in a queue and *await* a response from a consumer.

**Method 1** The most intuitive method is to generate a temporary queue for each request. A client will declare a queue with a random name. Before publishing the message to the main queue, it will set the `reply_to` field. When a consumer consumes this message it will also read the `reply_to` field and send the reply to the specified queue. After publishing the client will start consume from the temporary queue. Upon arrival of the message it will stop consuming and delete the queue. This approach is rather inefficient since we need to declare a new queue for each request.

**Method 2** Instead of generating a new queue per request we might create a long-lived queue just for this purpose. Like before, the client sets the `reply_to` field and the consumer replies to this queue. The client awaits the message in the reply queue. However, if multiple clients are await a response from some consumer, the reply messages may overlap in the reply queue and cause a malfunction. This can be resolved by settings the `correlation_id` field in the message (UUID). This value is copied over by the consumer to the `correlation_id` field of the response. The awaiting clients will start to sequentially receive the replies, they will check the `correlation_id` field and if it is not theirs their will ignore it. If the message is the one they have been awaiting the will consume it and send an acknowledgment.

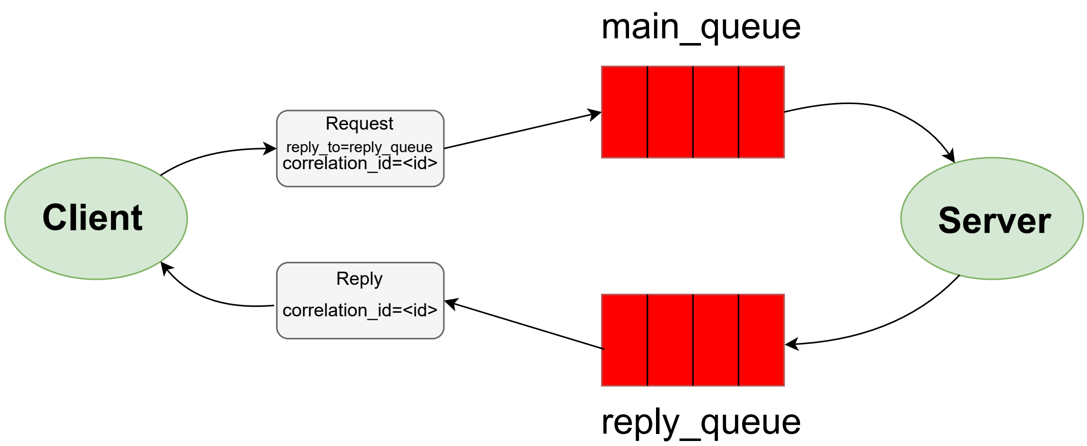


Figure 10: Request Reply Infastructure

**Method 3** RabbitMQ has a *built-in* request/reply pattern which is easier to implement and more efficient. The client will set the `reply_to` field to **amq.rabbitmq.reply-to**. This is a pseudo-queue known by the RabbitMQ server. When the server processes the message it will change the `reply_to` field to **amq.rabbitmq.reply-to.<token>** where **<token>** is a randomly generated token. The consumer will consume the message and publish the response to the **amq.rabbitmq.reply-to.<token>** pseudo-queue. The client will await the reply in no-ACK mode by consuming from the **amq.rabbitmq.reply-to** pseudo-queue. This method does not require the client to send an acknowledgment for the reply and the reply is directly sent back to the client.

### 6.6.3 Messages

**RabbitMessage (enum)**

| Field | Content | Description |
|---|---|---|
| LoginRequest | LoginRequestData | Login request packet |
| LoginResponse | LoginResponseData | Login response packet |
| RegisterRequest | RegisterRequestData | Register request packet |
| RegisterResponse | RegisterResponseData | Register response packet |
| GetImage | GetImageData | Get image data packet |
| ShrinkAndUpload | ShrinkAndUploadData | Shrink and upload image packet |
| GetTotalImages | GetTotalImagesData | Get total images packet |
| GetTotalImagesResponse | GetTotalImagesResponseData | Get total images response |
| ErrorResponse | ErrorResponseData | Error packet |

**LoginRequestData (struct)**

| Field | Type | Description |
|---|---|---|
| mail | String | The mail |
| username | String | The username |
| password | Vec<u8> | The password |

**LoginResponseData**

| Field | Content | Description |
|---|---|---|
| Ok | LoginResponseDataOk | Positive login response |
| Err | LoginResponseDataErr | Negative login response |

**LoginResponseDataOk**

| Field | Type | Description |
|---|---|---|
| token | Vec<u8> | The authentication token |

**LoginResponseDataErr**

| Field | Content | Description |
|---|---|---|
| NotFound | () | User was not not |
| WrongPassword | () | Password was incorrect |

**RegisterRequestData**

| Field | Type | Description |
|---|---|---|
| mail | String | The mail |
| username | String 6 The username | |
| password | Vec<u8> | The password |

**RegisterResponseData**

| Field | Content | Description |
|---|---|---|
| Ok | (RegisterResponseDataOk) | Positive register response |
| Err | (RegisterResponseDataErr) | Negative register response |

**RegisterResponseDataOk**

| Field | Type | Description |
|---|---|---|
| token | Vec<u8> | The authentication token |

**RegisterResponseDataErr**

| Field | Content | Description |
|---|---|---|
| AlreadyExists | () | User already exists |

**GetImageData**

| Field | Type | Description |
|---|---|---|
| username | String | The username |
| token | Vec<u8> | The auth token |
| index | u16 | The image index |

**ShrinkAndUploadData**

| Field | Type | Description |
|---|---|---|
| username | String | The username |
| token | Vec<u8> | The auth token |
| image | Image 6 The image | |

**GetTotalImagesData**

| Field | Type | Description |
|---|---|---|
| username | String | The username |
| token | Vec<u8> | The auth token |

**GetTotalImagesResponseData**

| Field | Type | Description |
|---|---|---|
| amount | u32 | The amount of images |

**ErrorResponseData**

| Field | Content | Description |
|---|---|---|
| AuthenticationRequired | () | Authentication failed |
| UnknownUsername | () | Username is unknown |

### 6.6.4 Binding

The binding to RabbitMQ is done via the `lapin` crate.

## 6.7 Dependencies

Here's a list of all the libraries used within the project

| **Dependency table (worker)** | | | |
|---|---|---|---|
| **Name** | **Description** | **Vesion** | **Features** |
| clap | CLI Parser | 3.2.20 | derive |
| tokio | Asynchronous runtime | 1 | full |
| log | Logging inerface | 0.4 | |
| env_logger | Logging implementation | 0.9.0 | |
| sha2 | SHA-2 hash function family | 0.10.5 | |
| image | Imaging library | 0.24.5 | |
| futures | Future and streams | 0.3.17 | |
| rand | Random number generators | 0.8.5 | |
| database | (Own) database library | - | |
| messaging | (Own) messaging library | - | |
| config | (Own) config library | - | |
| protocol | (Own) protocol library | - | |

| Dependency table (webserver) | | | |
|---|---|---|---|
| **Name** | **Description** | **Vesion** | **Features** |
| log | Logging inerface | 0.4 | |
| env_logger | Logging implementation | 0.9.0 | |
| clap | CLI Parser | 3.2.20 | derive |
| tokio | Asynchronous runtime | 1 | full |
| warp | Web server framework | 0.3.3 | |
| serde | Serialization/deserialization framework | 1.0 | derive |
| tower | client and server components | 0.4 | |
| tower-http | HTTP middleware | 0.3 | full |
| futures | Future and streams | 0.3.25 | - |
| bytes | Bytes utilities | 1.2.1 | |
| tera | Template engine | 1.17.1 | |
| lazy_static | Lazily evaluated statics | 1.4.0 | |
| once_cell | Single assignment cells | 1.16.0 | |
| base64 | Base64 encoder/decoder | 0.13.1 | |

| Dependency table (frontend) | | | |
|---|---|---|---|
| **Name** | **Description** | **Vesion** | **Features** |
| wasm-bindgen | JS and Rust interaction | 0.2.83 | |
| console_error_panic_hook | Logs panics to wasm32 | 0.1.7 | |
| wee_alloc | Allocator | 0.4.5 | |
| js_sys | JS objects binding | 0.3.60 | |
| base64 | Base64 encoder/decoder | 0.13.1 | |
| web-sys | Binding to Web APIs | 0.3.60 | |
| protocol | (Own) protocol library | - | |

| Dependency table (config) | | | |
|---|---|---|---|
| **Name** | **Description** | **Vesion** | **Features** |
| serde | Serialization/deserialization framework | 1 | derive |
| toml | TOML parser | 0.5.9 | |

| Dependency table (database) | | | |
| --- | --- | --- | --- |
| **Name** | **Description** | **Vesion** | **Features** |
| serde | Serialization/deserialization framework | 1 | derive |
| diesel | Database ORM | 2.0.0 | mysql, chrono, r2d2 |
| chrono | Datetimes | 0.4.19 | |
| diesel_migrations | Diesel migrations | 2.0.0 | |

| Dependency table (protocol) | | | |
| --- | --- | --- | --- |
| **Name** | **Description** | **Vesion** | **Features** |
| protocol | Protocol definitions | 3.4.0 | |
| protocol-derive | Protocol definitions | 3.4.0 | |
| lazy_regex | Lazily evaluated regex | 2.3.1 | |

| Dependency table (messaging) | | | |
| --- | --- | --- | --- |
| **Name** | **Description** | **Vesion** | **Features** |
| lapin | AMQP client | 2.1.1 | |
| amq-protocol-types | AMQP specifications | 7.0.1 | |
| tokio | Asynchronous runtime | 1 | full |
| tokio-amqp | lapin integration with tokio | 2.0.0 | |
| deadpool | Connection pool | 0.9.5 | |
| deadpool-lapin | Connection pool for lapin | 0.10.0 | |
| futures | Future and streams | 0.3.17 | |
| uuid | UUID utils | 1.1.2 | v4, fast-rng, macro-diagnostics |
| threadpool | Thread pools | 1.8.1 | |

## 6.8 Config files

The *worker* and *worker* require a configuration file which is passed through a CLI argument. The configuration files look as follows:

**worker** `config.toml`

```toml
# Alternately, comment the [database] section and
# set the enviroment variable
#
# DATABASE_URL="mysql://worker:root@192.168.1.10:3306/service"
[database]
address = "192.168.1.10"
port = 3306
username = "worker"
password = "root"
name = "service"

# Alternately, comment the [rabbit] section and
# set the enviroment variable
#
# AMQP_URL=amqp://worker:root@192.168.1.11:5672/vhost
[rabbit]
address = "192.168.1.11"
port = 5672
username = "worker"
password = "root"
vhost = "vhost"

# Alternately, comment the [log] section and
# set the enviroment variables (only RUST_LOG is fine)
#
# RUST_LOG="info"
# RUST_LOG_STYLE="auto"
#
# See https://doc.servo.org/env_logger/index.html
[log]
log = "debug"
style = "auto"
```

**webserver** `config.toml`

```toml
[http]
www = "/path/to/dist"
ip = "0.0.0.0"
port = 8080

# Alternately, comment the [rabbit] section and
# set the enviroment variable
#
# AMQP_URL=
[rabbit]
address = "192.168.1.11"
port = 5672
username = "worker"
password = "root"
vhost = "vhost"

# Alternately, comment the [log] section and
# set the enviroment variables (only RUST_LOG is fine)
#
# RUST_LOG="info"
# RUST_LOG_STYLE="auto"
#
# See https://doc.servo.org/env_logger/index.html
[log]
log = "info"
style = "auto"
```

# 7 Structure

## 7.1 mandate

The `mandate` folder contains all the documents regarding the project (documentation and diary).

## 7.2 worker

`worker/` is the Rust project for the backend service. The worker consumes messages from the AMQP server and processes them by interacting with the database.

## 7.3 webapp

`webapp/` contais the softwre regarding the webserver and frontend.

### 7.3.1 webserver

*webserver* is the Rust project for the webserver. This program serves the web page and handles the API routes. It send messages to the AMQP server and awaits the responses once the requests are processed by a backend server.

### 7.3.2 frontend

*frontend* is the program containing Rust code which is compiled to WebAssembly. This project be compield to a WASM module and interacted with with JavaScript on the frontend.

## 7.4 common

`common/` is a collection of own Rust libraries shared accross the programs.

### 7.4.1 config

*common* is a library tohelp parse the TOML configuration files. This library is used by the webserver and worker programs.

### 7.4.2 database

*database* is a library wrapper around the database structure of this project. It is only used by the worker server.

### 7.4.3 messaging

*messaging* is a library to publish and consume messages to an AMQP server. The request/reply pattern is also available. This library is used both by the websever and worker.

# 8 Compilation and usage

## 8.1 Frontend

### 8.1.1 Compilation

The frontend project is compiled into WASM and then static files for the website are generated.

```
cd webapp/frontend/website
wasm-pack build --release
npm install
npm run build
```

The `wasm-pack` commands compiles the Rust code into a WASM module in the folder `./pkg`. The npm script *build* generates the static website files from the files of the website (*www* folder) and the WASM module (*pkg* folder). The static files are placed in `./dist`.

## 8.2 Webserver

### 8.2.1 Compilation

The compilation is done using cargo.

```
cd webapp/webserver
cargo build --release
```

The executable is now at `./target/release/webserver`.

### 8.2.2 Usage

```
USAGE:
    webserver --config <CONFIG>

OPTIONS:
    -c, --config <CONFIG>    Configuration file
    -h, --help               Print help information
```

## 8.3 Worker

### 8.3.1 Compilation

The compilation is done using cargo.
Note that you need to have the *diesel* command installed.

```
cd worker
cargo build --release
```

The executable is now at `./target/release/worker`.

### 8.3.2 Usage

```
USAGE:
    worker --config <CONFIG>

OPTIONS:
    -c, --config <CONFIG>    Configuration file
    -h, --help               Print help information
```

# 9 Testing

## 9.1 Test protocol

## 9.2 Test results

| **ID** | Result | Note |
|---|---|---|
| **Test-00** | Failed | Someting |
| **Test-01** | Failed | Someting |
| **Test-02** | Failed | Someting |
| **Test-03** | Failed | Someting |
| **Test-03** | Failed | Someting |

# 10 Conclusion

## 11 References