SecureCart Development Report

5550645

February 2025

Contents

1 Introduction

In this report I will document the concept, design and implementation process of an e-commerce website developed for SecureCart. This report details only the "verification" part of the V-Model of software development, encompassing requirement gathering, high level system design, detailed software design and implementation.

2 Requirements

In this section, I will identify key functional and non-functional requirements for users and administrators of the SecureCart e-commerce website. It has been stated by SecureCart that security and scalability are key areas on which they would like the development process to focus.

2.1 Functional Requirements

Functional requirements can be expressed as a set of user stories, identifying specific users, considering the goals they would like to achieve, and enumerating the requirements to help them achieve those goals.

These are contained in the table below. Alongside each user story is a more concrete task, describing the specific details of what needs to be implemented.

User Type	Story	Task
Customer	As a customer, I want to be able to browse products so that I can find ones I am interested in.	Customers should be able to browse a list of products in the store.
Customer	As a customer, I want to be able to search for products by name so that I can quickly find things I already know I want to buy.	Customers should be able to search products on the store page by name.
Customer	As a customer, I want to be able to filter products by price so that I can find products which are in my price range.	Customers should be able to filter products on the store page by their minimum and maximum price.

Customer	As a customer, I want to see pictures and descriptions of products in the store so that I can see easily what I am purchasing.	Images and descriptions should be prominently displayed for all items on the store page.
Customer	As a customer, I want to have the contents of my shopping cart saved so that I can quickly resume previous shopping sessions.	The contents of the user's cart should be stored persistently either locally or on the server.
Customer	As a customer, I want to be able to view the status of my previous orders, so that I can be made aware when they have been fulfilled.	The user's previous orders should be displayed on their profile page, with their status prominently displayed.
Customer	As a customer, I want to be able to update my personal information, such as my email and shipping address, so that I can make sure my emails and products get sent to me.	The user should be able to update their email and address on their profile page.
Customer	As a customer, I want to be able to use two-factor authentication in a way which is easy and familiar to me, so that I can keep my account secure without additional effort.	Standard two-factor authentication protocols should be used, and QR codes should be generated to make enabling 2FA easier.
Customer	As a customer, I want to be able to submit my card details without worrying about how secure they are being kept, so that I can easily pay for things without stress	A popular third-party payment provider should be used to give users a sense of familiarity and safety when entering their payment details.
Customer	As a customer, I want to be able to close my account at any time, so that I can delete my personal data and leave the service if I wish.	The user's account page should have a clear way to close their account and leave the service.

Customer	As a customer, I want to be able to modify my card with last minute changes, such as adding more of an item, or removing it, so that I can quickly update my order without going back to the store page.	The cart page should allow for adding duplicates and removing products currently in the cart.
Administrator	As an administrator, I want to be able to easily get an overview of the orders in the system, so that I can fulfil them and track how the store is doing.	A list of past and present orders should be shown in the administrator's dashboard.
Administrator	As an administrator, I want to be able to filter orders by their current status, so that I can focus on fulfilling unfulfilled orders.	An option to filter by order status should be present in the administrator's dashboard.
Administrator	As an administrator, I want to be able to view and search all the products in the system, so that I can manage my inventory effectively.	The administrator's dashboard should show a searchable list of all products in the database.
Administrator	As an administrator, I want to be able to view all the users in the system, so that I can fulfil orders.	The administrator's dashboard should show a list of all users in the database.
Administrator	As an administrator, I want to be able to easily add additional products to the store, so that I can expand my offerings to my customers.	The administrator's dashboard should contain an option to add new products to the store.
Administrator	As an administrator, I want to be able to update the status of orders, so that I can let the customers know that I have fulfilled their orders.	The administrator's dashboard should allow for updating the status of an order, for example, from Confirmed to Fulfilled.
Administrator	As an administrator, I want to be able to delete users from the system, so that I can moderate my platform, or remove dead accounts.	The administrator's dashboard should provide an option to delete specific users.

Administrator	As an administrator, I want to be able to promote other user accounts to be administrators, so that I can have additional people to manage to store.	There should be an option to promote specific customer accounts to administrator accounts within the administrator's dashboard.

2.2 Non-Functional Requirements

SecureCart have stated that security and scalability are key priorities for them in the development of this system. Both of these comprise of non-functional requirements, which do not aid a specific user in carrying out a specific task, but rather improve the system in these metrics overall. I have listed these below in a similar format to the functional requirements above.

Focus	Task	Implementation
Security	Credentials should be stored securely in the database and should not be retrievable.	Password hashing with a strong algorithm (e.g. Argon2).
Security	Data should be security transmitted between the frontend and the backend of the application.	TLS/SSL should be enabled at least between the edge of the service and the user.
Security	Two-factor authentication should be available and should follow relevant standards.	An RFC-6238 compliant existing implementation of 2fa should be used.
Security	Sensitive/personal data should be encrypted at rest.	Database column-level encryption.
Security	Bruteforce authentication attacks should be mitigated.	Host-based timeouts after a given number of authentication attempts.
Security	Session hijacking attacks should be mitigated.	Cookies set with SameSite, Secure and HttpOnly and TLS in use. Sessions automatically expire after a given duration.
Security	CSRF should be mitigated.	Cookies set with SameSite=Strict, CSRF token in request headers, no side-effects in GET requests, CORS same-origin policy.
Security	XSS should be mitigated.	Cookies stored with HttpOnly, all user input added to the DOM security (i.e. textContent or escaped).

Security	Secrets should be handled securely.	Docker/Docker compose builtin secrets.
Security	Denial of service attacks should be mitigated.	Use of a reverse proxy with loadbalancing capabilities (NGINX), distributed storage.
Scalability	The service should be able to be scaled horizontally.	Stateless core API, use of horizontally-scalable storage services (Redis, Postgres, MinIO).
Scalability	The core service should use minimal resources.	Lightweight, compiled programming language (Rust).
Scalability	The service should be able to be distributed across multiple physical hosts.	Service connection parameters externally configurable, horizontal scalability.
Scalability	The service should be easy to deploy.	Docker.

3 System Design

In this section I will give a high-level overview of the service's architecture and design.

3.1 Technology Stack

The choice of technology stack is an important decision in the service's design, and so I will detail here why every tool has been chosen.

Tool	Role	Justification
TypeScript	Frontend core programming language.	TypeScript adds type-safety guarantees on top of regular JavaScript, as well as allowing for new ECMAScript standards to be used on older versions of web browsers which may not yet support them. It can be compiled directly to JavaScript, and so is interoperable with it.

Bootstrap	Frontend UI library	Bootstrap handles most common UI tasks with built in classes which look consistent and heavily reduce the time and effort spent on frontend UI design. It also has a JavaScript API with TypeScript type support available.
Docker	Containerisation	Docker is the most common containerisation tool in use, and there are officially supported Docker images for most common services. Docker compose heavily simplifies deployment and architecture design.
NGINX	Reverse Proxy / HTTP Server	NGINX is simple to deploy and configure, has an officially supported docker image, allows for some useful more advanced features, such as upstream authentication sub-requests, and can handle a very large amount of traffic.

Rust	Backend core programming language	Rust provides strict type and memory safety guarantees, and the strong type system allows for a reduction in runtime errors by validating invariants at compile time. This is ideal for a web service, where there are a very large number of edge cases and invalid states which could otherwise occur. It is also very lightweight, being a compiled language with no garbage collector, meaning that it uses few resources on the server, and can be scaled even with relatively little physical hardware available. It also has a mature package ecosystem, with crates available for most common tasks a web API would be required to perform (database calls, async runtime, etc).

Axum	Backend API framework	Axum is one of the most used Rust web frameworks, with a large ecosystem and support for most common crates (e.g. serde, tokio). It is minimal and lightweight, meaning that it does not make software architecture decisions, and leaves the actual software design up to the developer, allowing more control and flexibility. This is particularly useful because it allows greater choice over other aspects of the tech stack, such as a database and temporary storage. It can also handle a lot of the more error-prone parts of writing a backend service, such as parsing and validating user input (when used with serde), and middleware-based dependency injection.
Redis	Session/temporary storage	Redis is widely deployed due to its high degree of reliability and performance. It uses a flexible key-value data model, making it ideal for tasks like storing information about currently authenticated sessions. It is also horizontally scalable by default, as it has built-in sharding when run in a Redis cluster.
PostgreSQL	Database	Postgres is strongly typed and has very good support from Rust database libraries such as SQLx. It is also very performant and reliable.

MinIO	Media object storage	MinIO is chosen to store dynamic media, such as uploaded images, since it is fully S3-compatible, meaning that the actual API service can be agnostic towards the actual storage service it is using. If the service works with MinIO, it will work with S3 or any other S3-compatible storage service too.
Stripe	Processing payments	Stripe handles most of the payment process automatically, supporting most common payment methods without any additional work. It also is architected in such a way that payment information never gets sent to the SecureCart backend server, and is handled entirely by Stripe, completely negating any risk from failure to comply with PCI-DSS and similar regulation.

3.2 Service Architecture

Since there are a significant number of components required for this system, including but not limited to a database for storing persistent data, a reverse proxy for handling routing and traffic and an HTTP server for serving the frontend, it is valuable to design the architecture of which services can interact and how, prior to designing the actual software internals. Below is a simple, high-level architecture diagram showing the services required for the application to function, and how they communicate. It also gives a rudimentary overview of the HTTP routing, so that it is clear how the services fit into the external API.

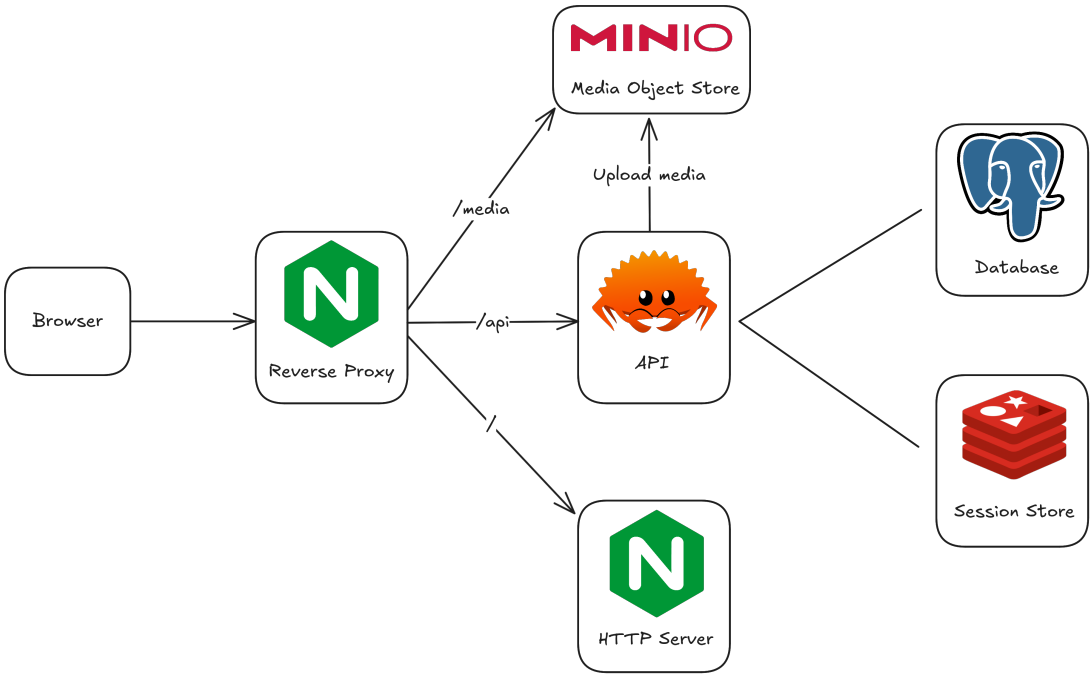

Figure 1: The application's service architecture

4 Detailed Design

In this section, I will go into more depth about the specifics of the software is designed.

4.1 Data Modelling

4.1.1 Primary Database

This application uses a database, and so it is useful to get an overview of the data modelling schema in use. Below is an entity-relationship diagram (ERD) showing how the tables in the database relate to one another, and what they contain.

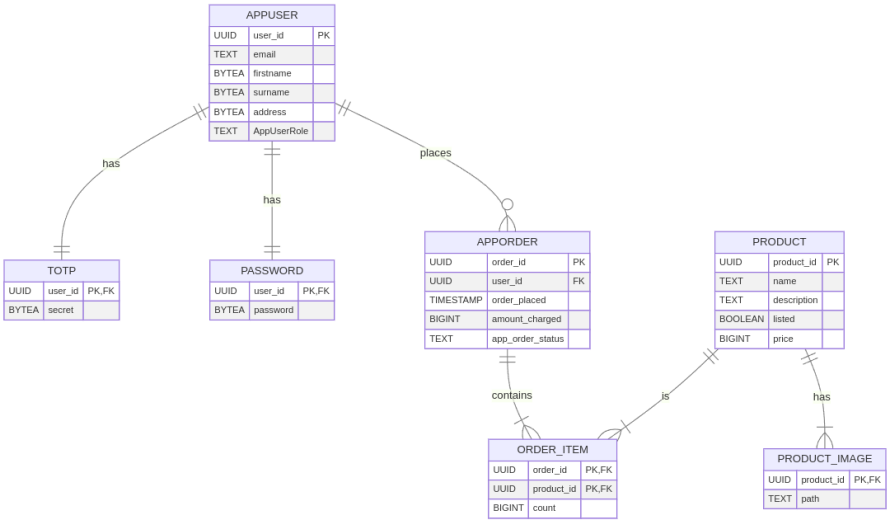

Figure 2: ERD for the primary database

Additionally some of the fields in the database are encrypted using a symmetric key stored only within the application's memory. These fields are as follows:

- AppUser.forename
- AppUser.surname
- AppUser.address
- Totp.secret

These fields specifically are encrypted since the first three are sensitive personal information and the latter is an important piece of authentication material, which could allow an attacker to take over a user's account if exposed. Unfortunately, while I would have liked to encrypt the users' email addresses as well, this is not possible since they need to be available for Postgres to query.

Credentials are stored separately from the users they are linked to in order to allow for greater flexibility during future development. A large component of the system design was building in the flexibility now to make it easy to

potentially add additional authentication mechanisms in the future (such as passkeys, or other forms of two-factor authentication besides TOTP). It is also possible that a user may wish to have multiple authentication or 2FA methods enabled on their account, although implementing that is out of scope for the initial prototype being developed in this report.

4.1.2 Session Store

The session store also has a specific data model in how it stores session information. It is a much simpler model, since Redis is primarily a key-value store, allowing for primitive data structures such as lists and hashes (sets of key-value pairs). The figure below shows the keys in the store, and the data-types which they point to. The keys should be read as being concatenated, e.g. `session:authenticated:{TOKEN}`, and values enclosed in `{ }` are effectively the "primary keys", being what would be used to query the session storage.

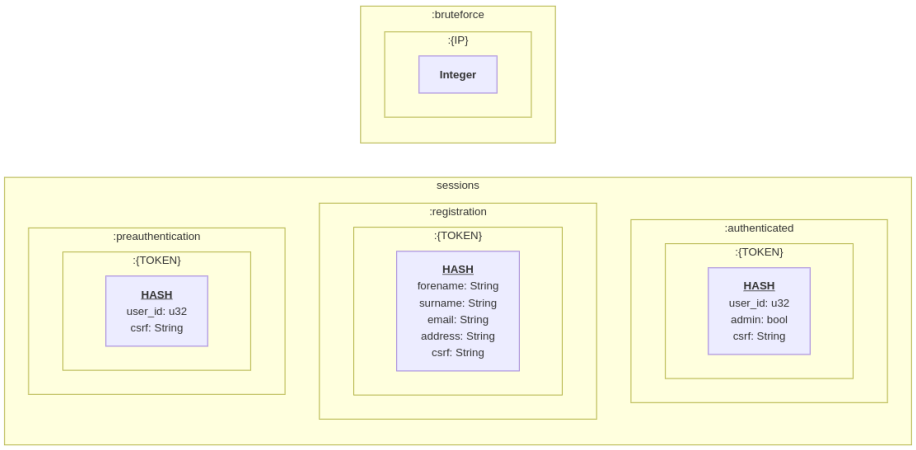

Figure 3: The data model for the session store.

As you can see, each session type holds different data (which is explained in greater depth in the Session Management section), except for the presence of a CSRF token, which is necessary for all sessions. This is explained in more depth

in the CSRF Tokens section. Furthermore, there is a mapping of IP addresses to a single integer under the key `bruteforce`. This is explained further in the Bruteforce Prevention section, but in terms of the Redis data model, this value would be accessed like so: `bruteforce:0.0.0.0`.

4.2 API

The API design is an important factor, since it governs how the rest of the service will be organised. The API follows a primarily RESTful style, with GET, POST, PUT and DELETE mapping to the CRUD operations, but some components do not fit easily into that pattern, such as authentication and registration, and so these components do not necessary strictly follow a RESTful style. Below, I have listed every route in the API, the methods it accepts, and brief description of what it does.

Route	Methods	Description
/auth	GET, POST	Lists supported authentication methods, and accepts credential to initiate the login flow.
/auth/check	GET	Returns 200 only if currently successfully authenticated.
/auth/check/customer	GET	Returns 200 only if currently authenticated as a customer.
/auth/check/admin	GET	Returns 200 only if currently authenticated as an administrator.
/auth/2fa	GET, POST	Lists supported 2FA methods, and accepts a 2FA credential to complete the login flow.
/registration	POST	Accepts a user's basic information (name, email, address, etc) to begin the onboarding flow.
/registration/credential	POST	Accepts a primary authentication credential to complete the onboarding flow.
/products	GET, POST	Queries products in the database, adds a new product to the database.
/product/{id}	GET, PUT, DELETE	Gets a specific product's information, updates a product, or deletes it.

/product/{id}/images	GET, POST	Lists images linked to a product, or uploads a new image.
/orders	GET, POST	Queries orders in the database, or creates a new order.
/orders/{id}	GET, DELETE	Gets information about a specific order, or deletes it.
/orders/{id}/fulfil	POST	Sets an order's status to Fulfilled.
/checkout	POST	Initiates the payment flow for a specific (Unconfirmed) order.
/users	GET	Queries users in the database.
/users/self	GET, PUT, DELETE	Gets or updates information about the currently authenticated user, or deletes their account.
/users/self/credential	PUT	Updates the currently authenticated user's primary authenticated method.
/users/self/2fa	POST	Adds a new 2FA method to the current user.
/users/self/2fa/new	GET	Generates a 2FA method/secret on the server, and retrieves it.
/users/{id}	GET, PUT, DELETE	Gets or updates a specific user's information, or deletes their account.
/users/{id}/promote	POST	Sets a specific user's role to Administrator.
/webhook/stripe	POST	Accepts webhook events from Stripe.

4.3 Application Structure

The structure of the actual application itself, and the way data and behaviour flows within it will be explained in this section. I will consider what layers the application needs, and how this translates to the structure of the code itself.

4.3.1 Architectural Layers

A common pattern in software architecture is to divide software into layers. This is particularly useful in the case of having a web frontend, an API, and a database, since the architecture is effectively already layered by the network structure, where the user interacts only with the frontend, which interacts with the API, which interacts with the database. The API itself can be divided

up into three layers: Presentation, Business Logic, and Data. The presentation layer is the actual HTTP interface which clients (i.e. the frontend) interact with, the business logic layer is the set of behaviours triggers by those interactions, and the data layer is what maps business logic in the application to the state stored in other storage services, such as the database. The application itself is stateless, and has no data stored persistently in memory. Instead, all state is held outside the application in the database, the session store, and the object store.

The three layers of the application can be literally separated into three modules in how the code is laid out. I have chosen to call these routes, services and models. Routes will be stored in /routes, services will be stored in /services, and models will be stored in /db/models, since there is additional logic related to connecting to the database which is placed in the /db module. Each submodule in `routes` will correspond to one route, with nested modules corresponding to nested paths in the API. I have included a screenshot below of the actual code structure once completed, to demonstrate the layout I am describing.

Figure 4: The file structure, highlighting routes, services, and models

4.3.2 Sessions and Middleware

I am using Axum to build this application, which means that I will be following Axum's recommended way of routing and handling requests. Axum has a strong focus on the use of Extractors and middleware. Extractors are functions which run on incoming requests, before your handler function is called, which enable dependency injection by allowing you to access certain elements of the request within the handler function. An example of an Extractor which I have used extensively is `Json`, which parses the body of a request as Json into a specified type (see the example below).

```rust
#[derive(Deserialize)]
struct CreateOrderRequest {
    products: Vec<CreateOrderRequestProductEntry>,
}

#[derive(Deserialize)]
struct CreateOrderRequestProductEntry {
    product: Uuid,
    count: u32,
}

async fn create_order(
    State(state): State<AppState>,
    Extension(session): Extension<CustomerSession>,
    Json(body): Json<CreateOrderRequest>,
) -> Result<Json<AppOrder>, HttpError> {
    let user_id = session.user_id();
    Ok(Json(
        orders::create_order(
            user_id,
            body.products
                .into_iter()
                .map(|entry| (entry.product, entry.count))
                .collect(),
            &state.db,
        )
        .await?,
    ))
}
```

Figure 5: An example of using the Json extractor

This then enables the creation of custom middleware, through the use of the Extension extractor. Middleware which you write can place values into an extension for the incoming request, and then the Extension extractor can be

used by a handler function to access them.

I will be using middleware to implement session handling, by extracting the session cookie from incoming requests, and providing a validated session object to handler functions.

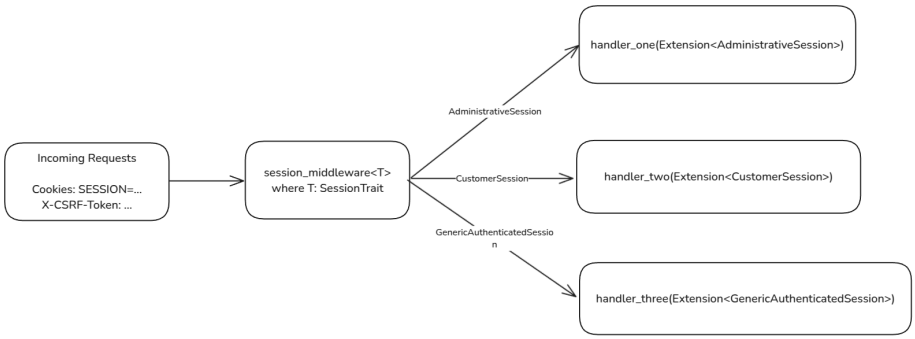

Figure 6: How a request gets validated and parsed through the middleware

This is made possible through the use of traits. Rust does not support inheritance, favouring composition and enums where possible. However, traits are effectively interfaces, allowing you to be generic over related types which implement the trait.

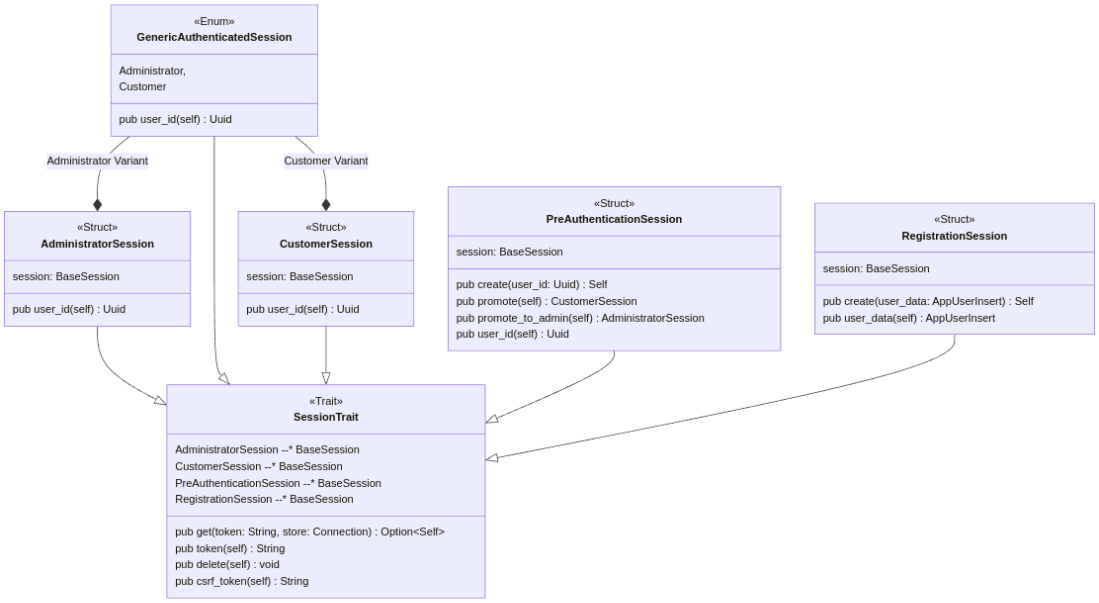

Figure 7: Struct diagram for sessions

Here, BaseSession is effectively a handle to the underlying session store, which handles serialising and deserialising data in a type-safe way.

The session middleware function (middleware in Axum can be created from a single function) can take anything which implements SessionTrait as a type parameter, and so can use the specific implementation of the static `get` constructor method to validate the session. A CustomerSession can return an error if the session is not linked to a customer, a RegistrationSession can return an error if it is an authenticated session, etc.

4.3.3 Database Models

I am not using an ORM to interact with the database, but rather SQLx, a Rust crate which provides type-safe, compile-time checked SQL queries. Queries are checked during compilation, optionally against a real live database, to en-

sure both that they are valid, and that all types are correct. Despite the type safety already provided by SQLx, writing dedicated structs for each table in the database is a far more convenient approach. For brevity, I will not detail every struct for all the tables in the database, but will focus on one specifically, the model for an AppUser. I have designed each of the structs to operate in the same way, with consistent usage and methods, so the design of this model is the same as all the others.

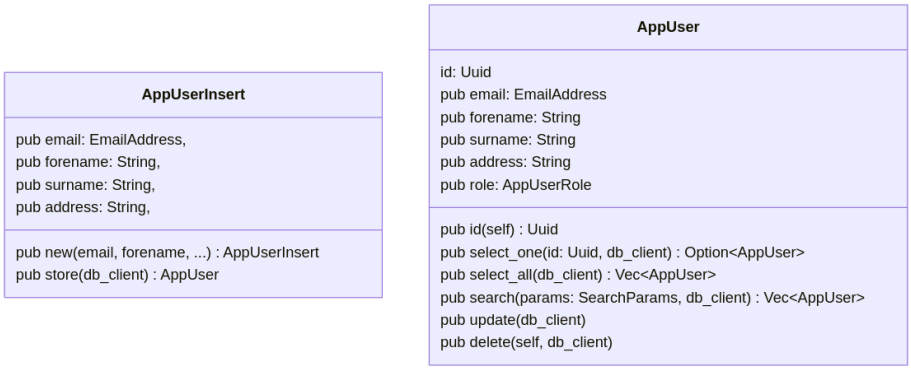

Figure 8: The AppUserInsert and AppUser models

As can be seen, AppUser has no constructor method, and its ID field is private. This means that it is impossible to directly construct outside the `db::models::appuser` module. The only two ways to get an instance of AppUser are to create an AppUserInsert and call its `store` method, which will save it to the database, or to use the `select_one`, `select_all` and `search` static methods. This design allows expressing at compile time whether we are operating on an existing database model or not. It enforces that AppUser instances are only ever constructed by first constructing an AppUserInsert, which enables strict control over how and where objects are instantiated and database calls are made.

4.3.4 Media Storage

This application processes a small amount of dynamic media, in the form of images uploaded for different products. These images will not be stored within the API, as this would make the API stateful and non-scaleable, while also using bandwidth which could be used for API requests on serving images. Instead, a separate media object store is used, in the form of MinIO. MinIO is an S3-compatible object store, which stores artifacts in a similar way to a traditional filesystem, but with abstraction over where/how the data is actually being stored. The rest of the application stores a relative path (within the storage bucket) to images whenever they are referenced (e.g. in the product_image table), and adds the URI (most likely it is local so `/media`) to the object store when sending data to the client. This way, the client can just immediately load the URI they receive to display the image.

The sequence diagrams below show how an image might be uploaded, and then subsequently fetched.

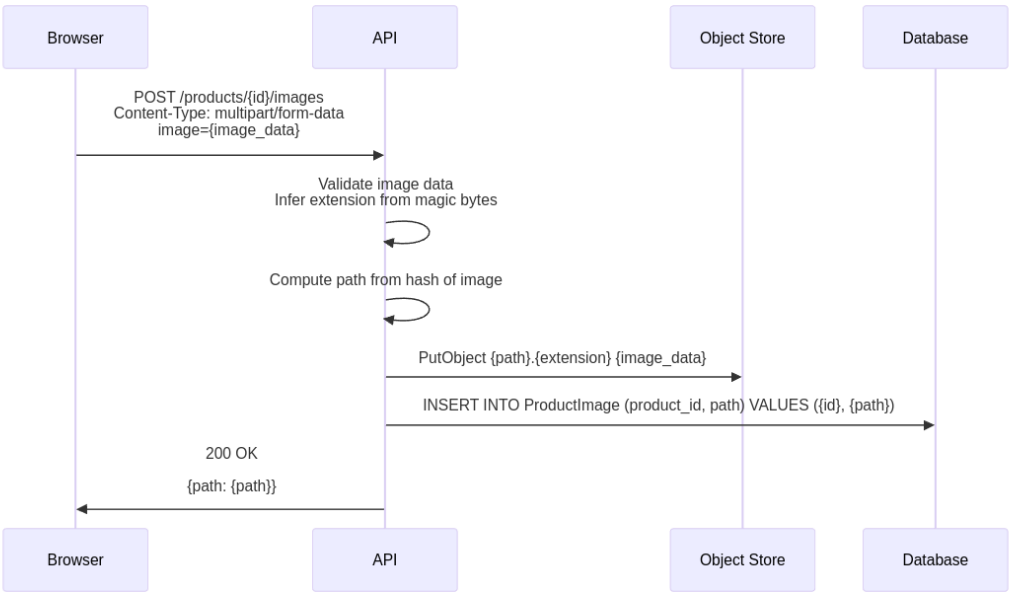

Figure 9: Image upload

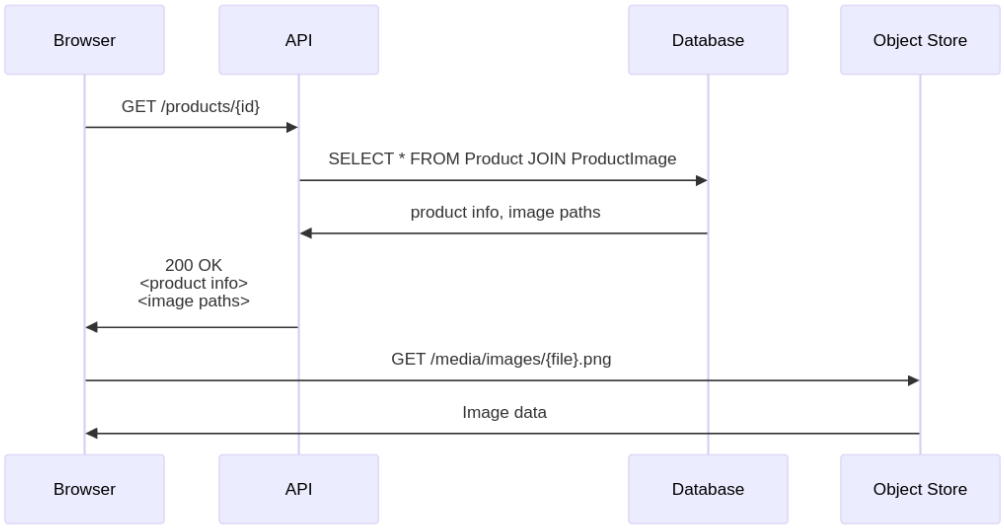

Figure 10: Fetching a product and its images

In terms of scalability, we can reduce data duplication by naming the uploaded files with their own SHA256 hash. The way that S3 works by default is that it will insert a new object if another one doesn't exist with the same name, and will otherwise update the existing object, which can be more performant. This means that we get a potential performance enhancement when uploading product images, and also completely avoid duplication of images within the object store.

4.4 Registration

Below is a sequence diagram showing the registration process.

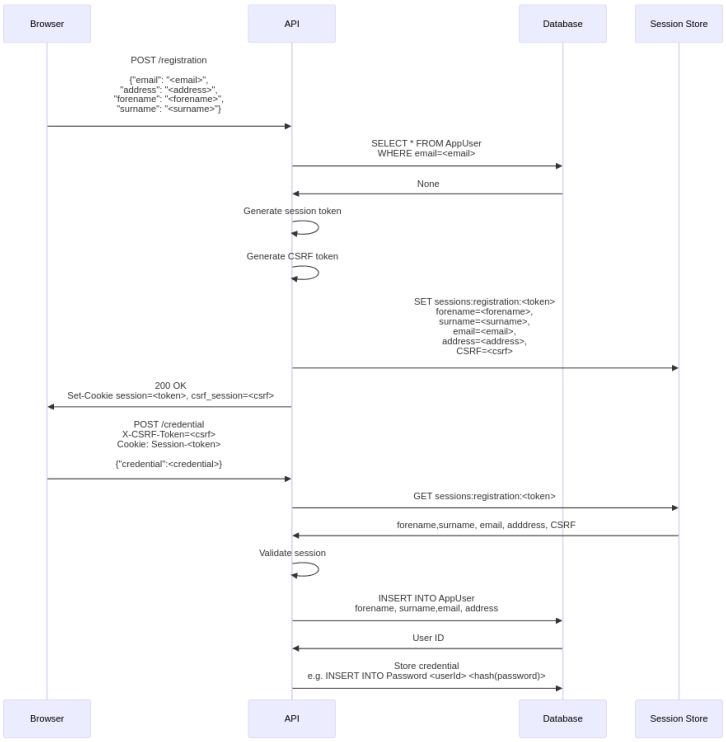

Figure 11: The registration process

4.5 Authentication

Authentication has been designed with support for multiple authentication and 2fa methods from the group up. The sequence diagrams below show the process.

The process differs when 2FA is enabled to when it is not. Here is the process when it is not enabled:

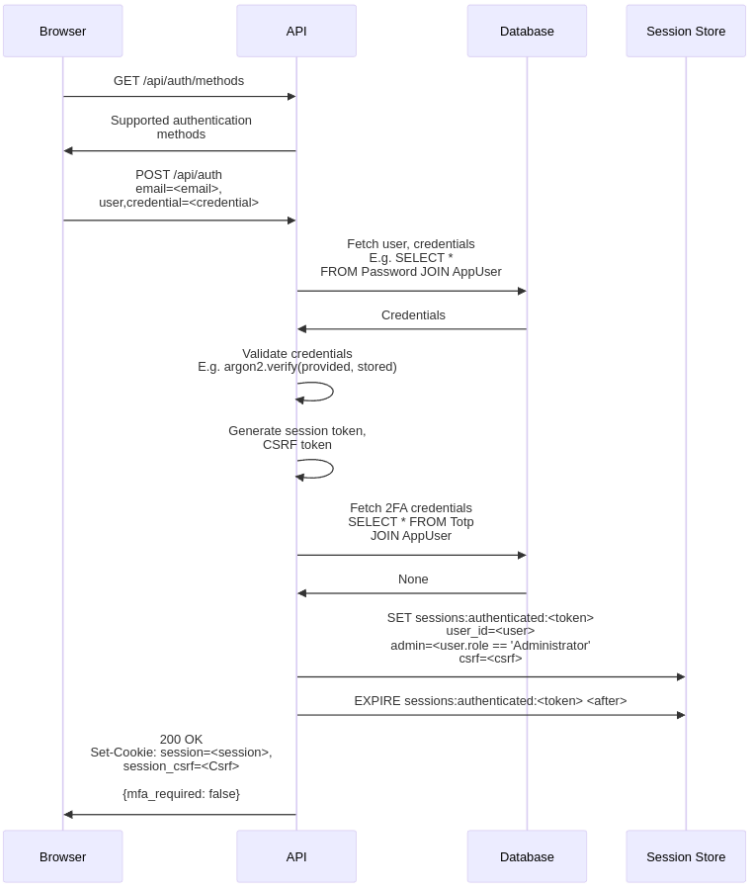

Figure 12: Authentication without 2FA

If 2FA is enabled, the process differs slightly. The below sequence diagram shows the rest of the process for 2FA after the paths diverge.

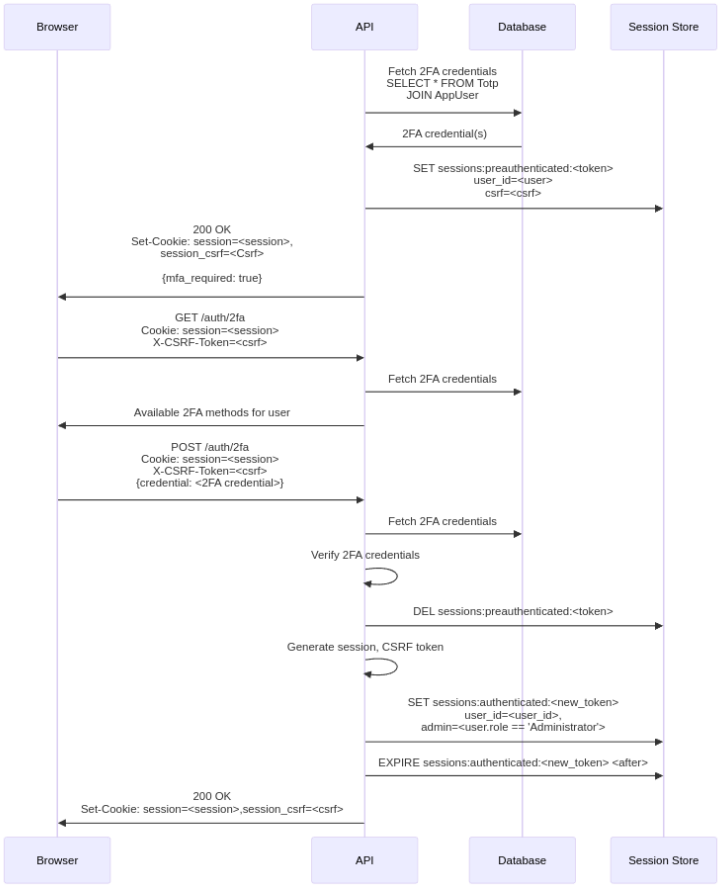

Figure 13: Authentication with 2FA

If at any point on of the steps fail, an HTTP error code will be returned the client, such as 401 for an authentication failure.

4.6 Checkout

Payments will be handled via Stripe, to avoid processing payment information. I will be using the Stripe PaymentIntents API for this, and the sequence diagram below demonstrates how the payment process works.

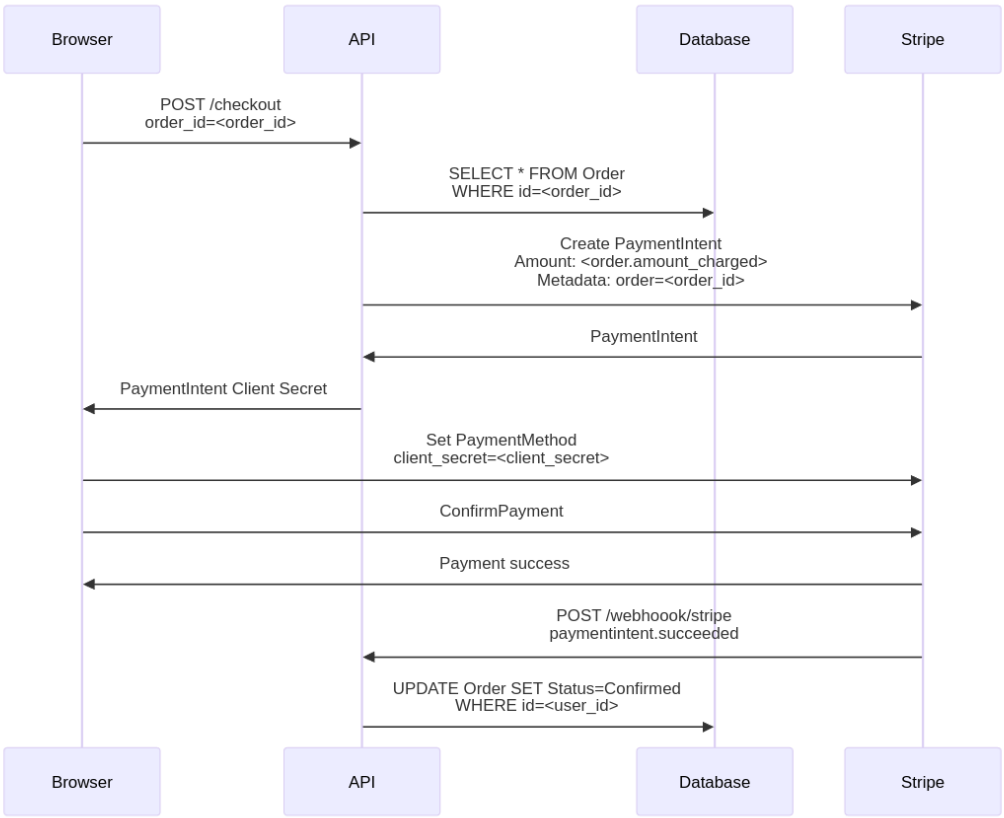

Figure 14: Stripe PaymentIntent flow

4.7 Security Features

A list of security features which will be implemented and how is explained below.

Feature	Explanation	Implementation

CSRF Tokens	A separate value, tied to the current session, which the browser must submit as an HTTP header	Randomly generate tokens when creating new sessions, have a function on the frontend to set the HTTP header (see figure 15, validate CSRF during session validation)
Session Hijacking	It should not be possible for an attacker to access a user's session token and use it to make requests.	CSRF tokens, cookies set to SameSite=Strict, HttpOnly=true, Secure=true.
Password Hashing	Passwords cannot be recovered from the Password.password field in the Database	Passwords are salted and hashed with Argon2 before being stored in the database, and are checked by doing an argon2 verify.
Database Encryption	Sensitive personal data cannot be recovered from the database without the key	The pgcrypto `pgp_sym_encrypt/decrypt` functions are built into Postgres, so use them when updating/selecting sensitive information
SQL Injection Prevention	SQL injection attacks should not be possible through any input	Use exclusively prepared statements, SQLx validates most queries at compile time which already enforces the use of prepared statements.
Rate Limiting/Bruteforce Mitigation	An attacker should not be able to bruteforce users' credentials.	Every login request increments a counter, until a max attempts threshold is reached and the client is banned for a fixed period of time.
DOS Prevention	An attacker should not be able to overload the server with work or data.	Set a (large) maximum password length to prevent Argon2-related DOS attacks, set a max request size in NGINX. Store all uploaded media in the object store outside the application.
XSS Prevention	An attacker should not be able to inject JavaScript code into the browser of another use.	Use exclusively .textContent to add user-supplied data to frontend pages.

```
async function fetch_csrf(uri: string, params?: RequestInit) {
  const csrf_token = document.cookie
    .split(";")
    .filter((c) => c.startsWith("session_csrf="))
    .map((c) => c.substring("session_csrf=".length, c.length))
    .pop();
  if (csrf_token === undefined) {
    throw new Error("CSRF token cookie is not set");
  }
  const headers = params ? new Headers(params.headers) : new Headers();
  headers.set("X-CSRF-Token", csrf_token);
  return fetch(uri, {
    ...params,
    headers,
  });
}
```

Figure 15: Setting the CSRF header on the frontend

Secure Transmission	An attacker should not be able to sniff traffic as it travels between the browser and the application.	Secure the connection by setting up TLS in NGINX.
Role-based Authorisation	Role-based authorisation should be in place across the application, to prevent a user gaining privileges they should not have.	Create a generic middleware which accepts a type parameter of the required session type, as detailed above.
SSTI Prevention	An attacker should not be able to inject template code which will run server-side.	Do not use templates.

4.8 UI Flow

The diagram below shows how users can transition between pages in the web UI.

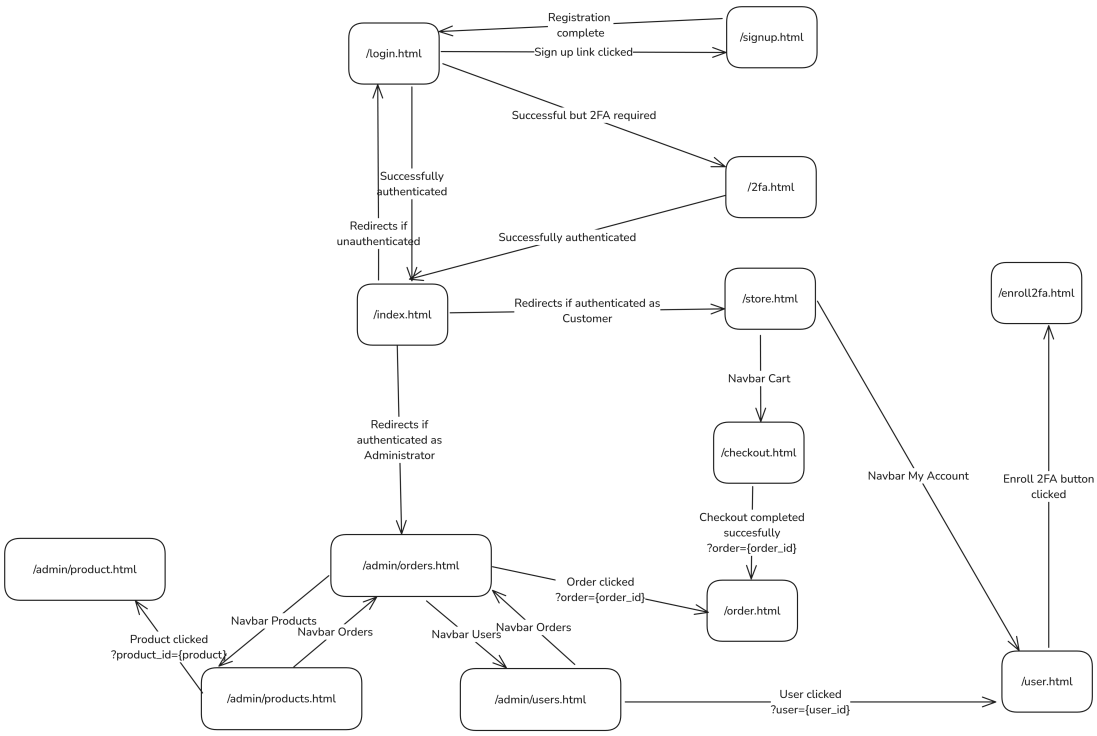

Figure 16: The transitions between pages in the UI

5 Implementation

5.1 NGINX

Here is the NGINX config used to route traffic to the correct parts of the API. It also restricts access to the /admin and /media domains by making a request upstream to /api/auth/check. This is not strictly necessary for the security of the application, but it simplifies the frontend since it won't have to handle the case where a non-admin user is loading webpages which make API calls requiring admin authentication. Restricting unauthenticated access to /media reduces the likelihood of denial of service attacks and very heavy load on the whole system.

```
events {
    worker_connections 2000;
}

http {
    include mime.types;
    server {
        listen 8443 ssl;
        ssl_certificate /ssl/securecart.crt;
        ssl_certificate_key /ssl/securecart.key;
        ssl_protocols TLSv1.3;
        ssl_ciphers HIGH:!aNULL:!MD5;
        location /api/ {
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_pass http://api/;
            client_max_body_size 2M;
        }
        location /media/ {
            auth_request /api/auth/check;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-Host $host;
            proxy_pass http://minio:9000/media/;
        }
        location / {
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-Host $host;
            proxy_redirect off;
            proxy_pass http://frontend/;
        }
        location /admin/ {
            auth_request /api/auth/check/admin;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Host $host;
            proxy_pass http://frontend/admin/;
        }
        location /js/admin/ {
            auth_request /api/auth/check/admin;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Host $host;
            proxy_pass http://frontend/js/admin/;
        }
    }
}
```

5.2 User UI

These are the pages which a normal user will see while using the application.

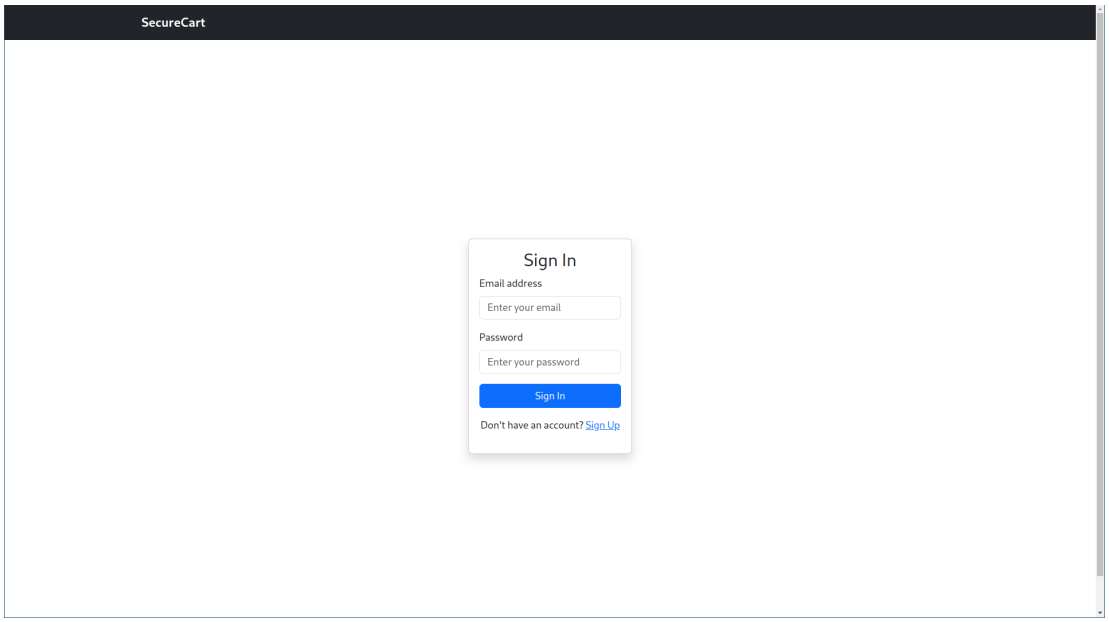

Figure 17: login.html

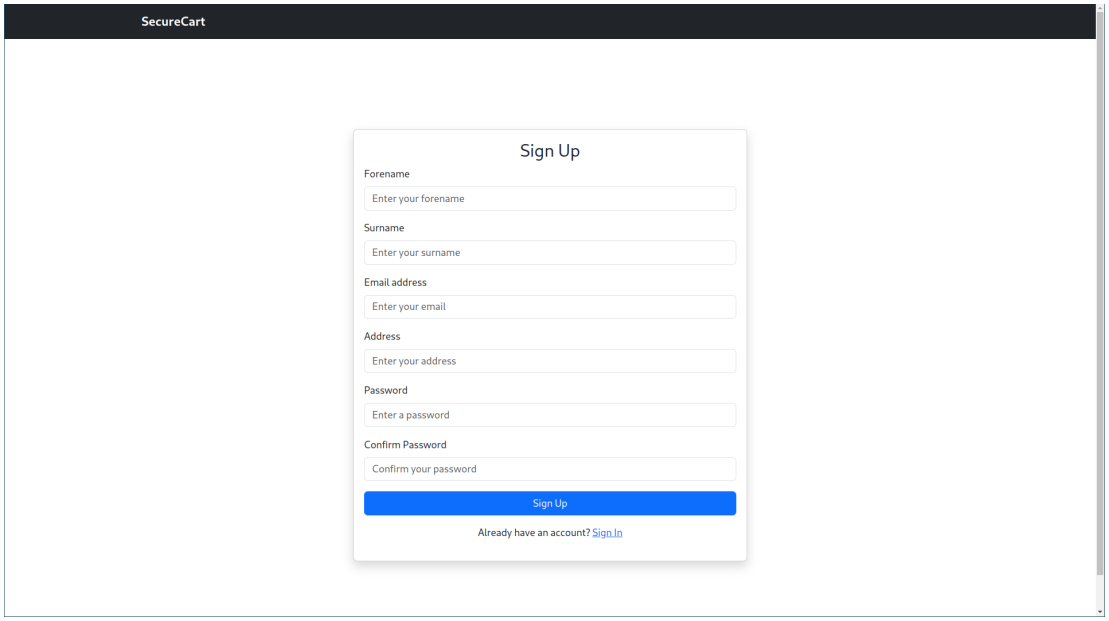

Figure 18: signup.html

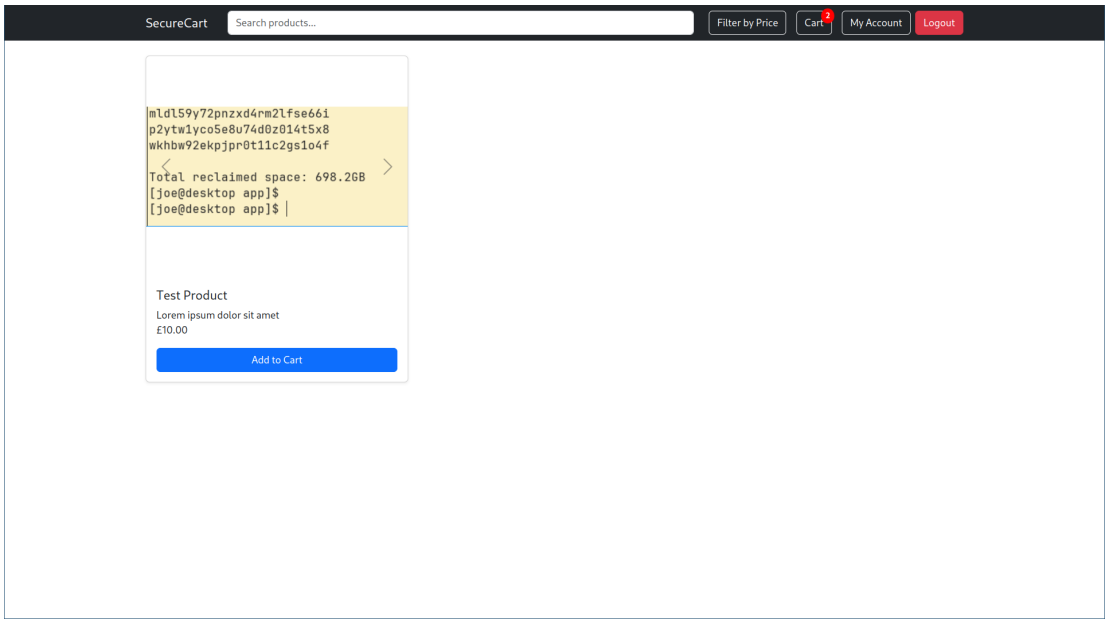

Figure 19: store.html (apologies for the random image, didn't have any others)

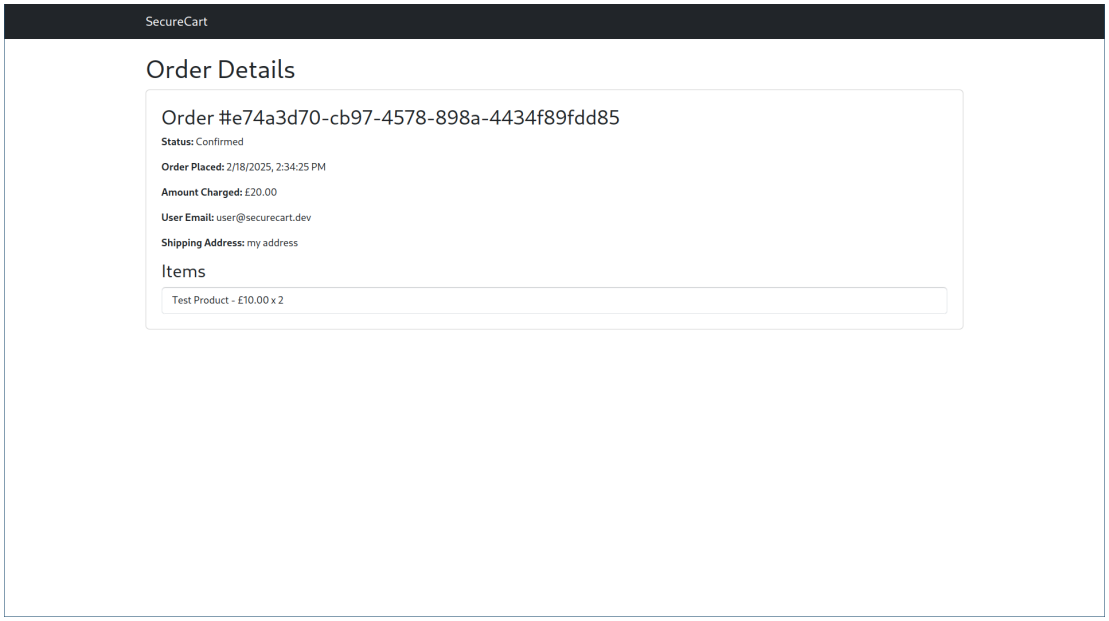

Figure 20: order.html

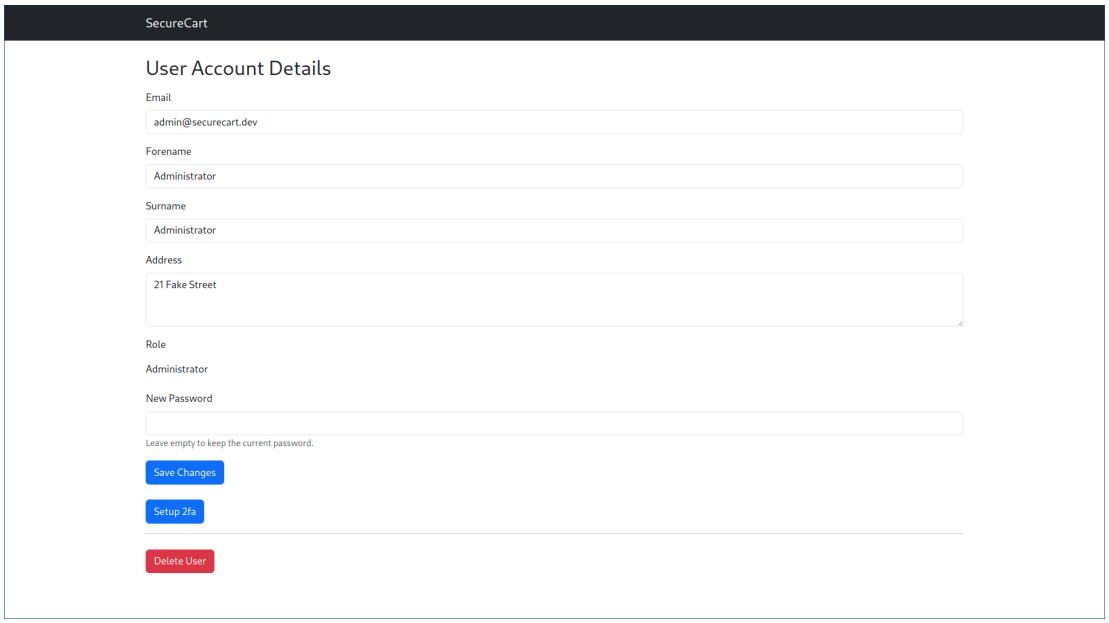

Figure 21: /user.html

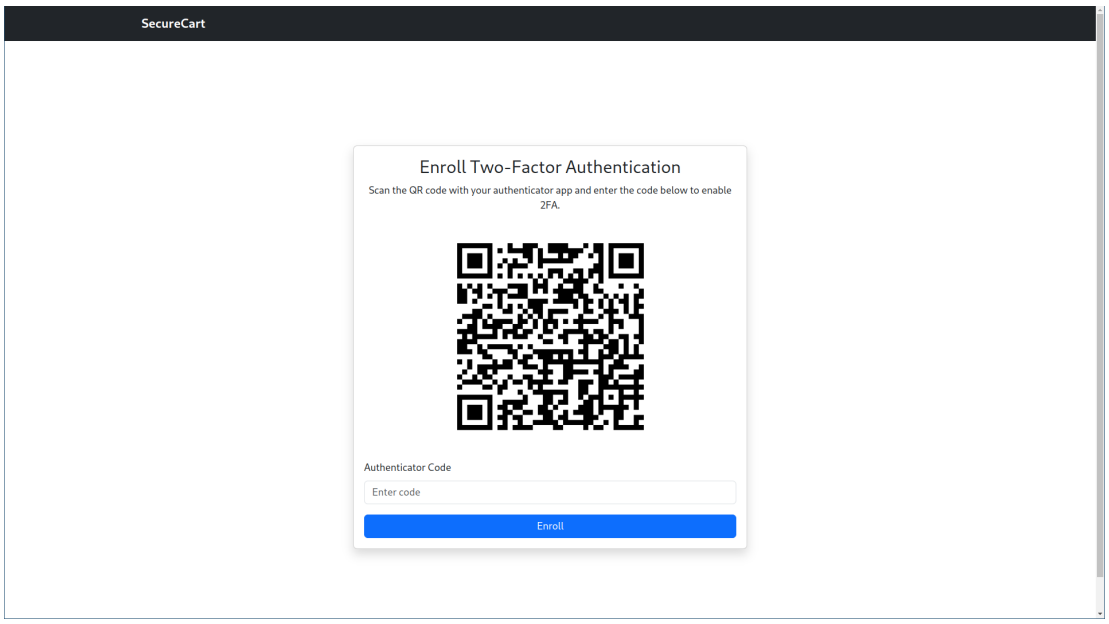

Figure 22: /enroll2fa.html

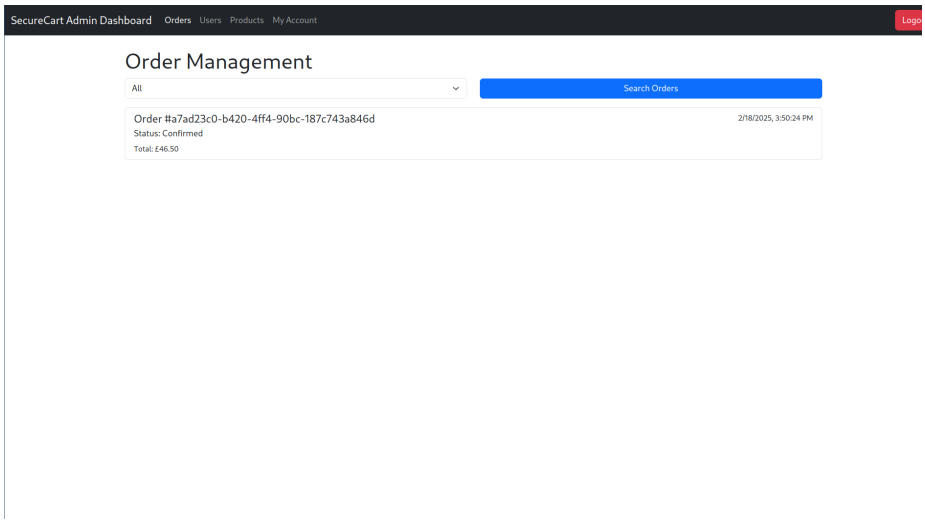

Figure 24: order.html

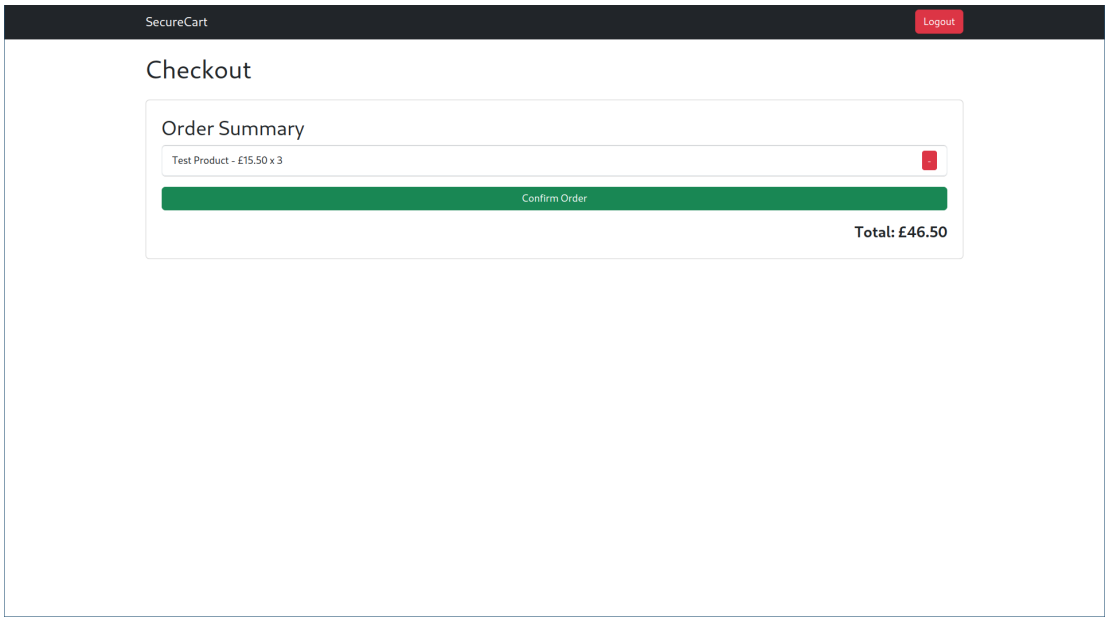

Figure 23: /checkout.html

5.3 Admin UI

These are the pages an admin will see while using the application.

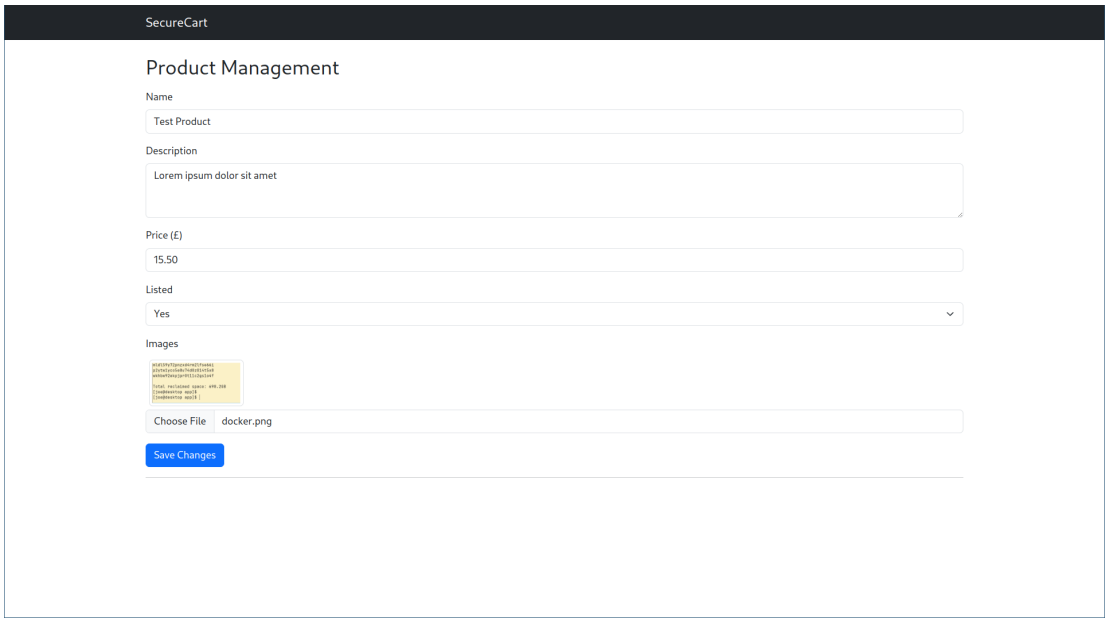

Figure 25: /admin/product.html

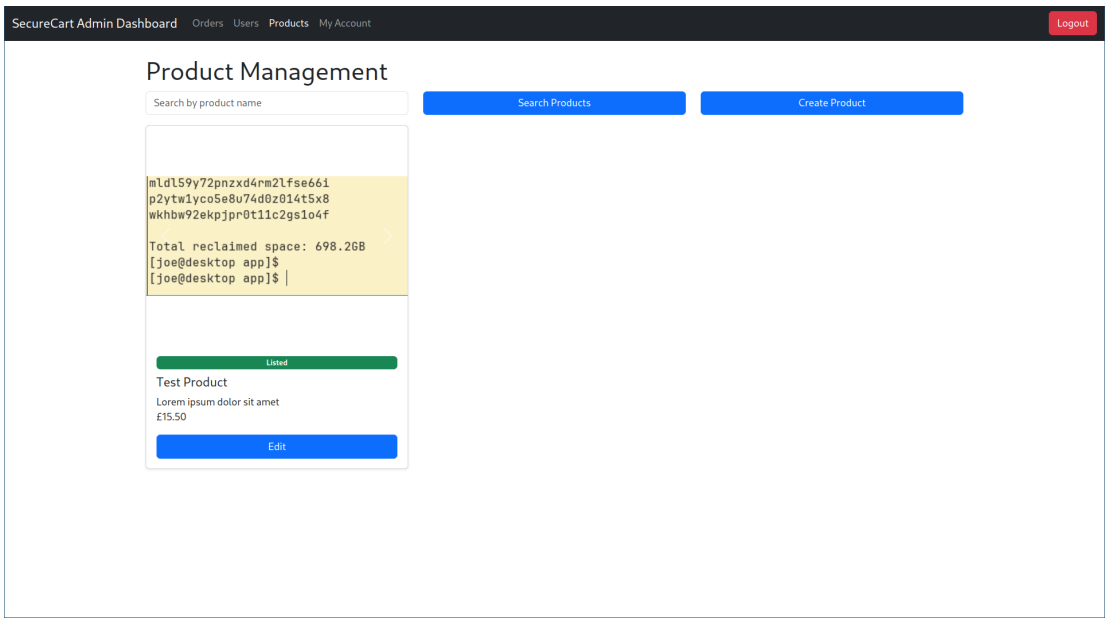

Figure 26: /admin/products.html

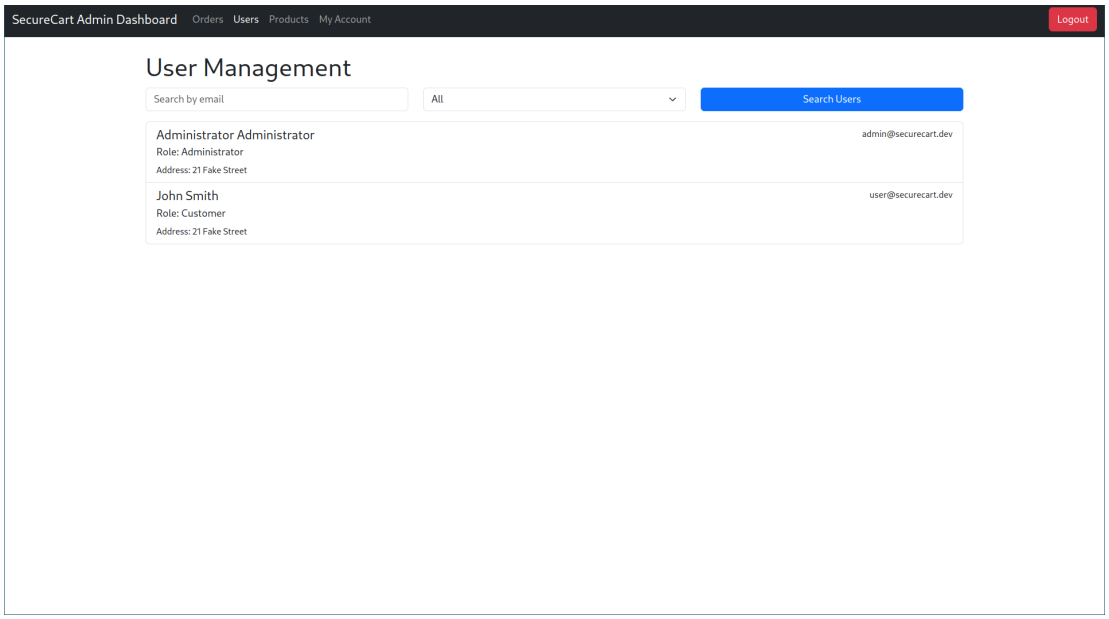

Figure 27: /admin/users.html

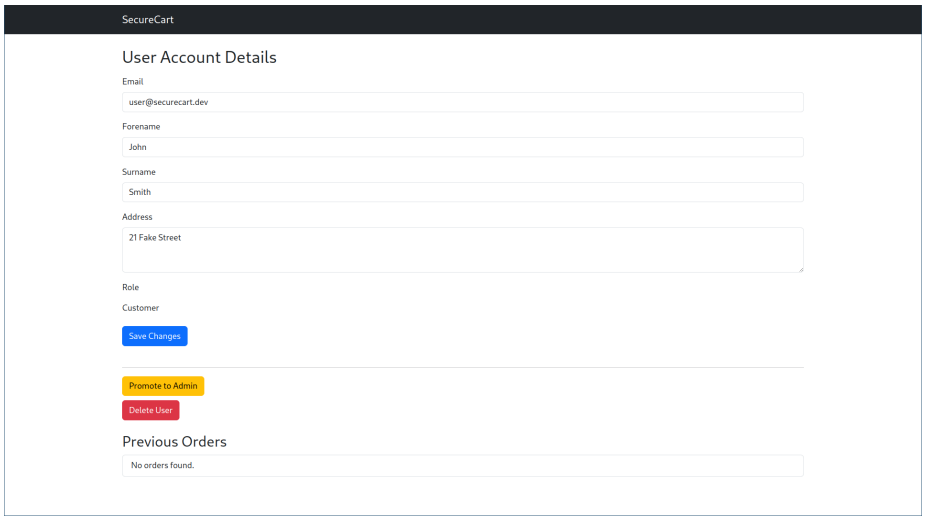

Figure 28: /user.html

5.4 Session Middleware

Here is the middleware described in the Sessions and Middleware section.

```rust
pub async fn session_middleware<T: SessionTrait + 'static>(
    State(state): State<AppState>,
    cookie_jar: CookieJar,
    mut req: Request,
    next: Next,
) -> Result<Response, StatusCode> {
    let session_cookie = cookie_jar
        .get("session")
        .ok_or(StatusCode::UNAUTHORIZED)?
        .value();
    let session = T::get(session_cookie, &mut state.session_store.clone())
        .await
        .map_err(|err| {
            eprintln!("Error loading session from store: {err}");
            StatusCode::INTERNAL_SERVER_ERROR
        })?
        .ok_or_else(|| {
            eprintln!("Invalid session token.");
            StatusCode::UNAUTHORIZED
        })?;
    let csrf_token = req
        .headers()
        .get("X-CSRF-Token")
        .ok_or_else(|| {
            eprintln!("Request is missing X-CSRF-Token");
            *STATUS_CODE_BAD_CSRF
        })?
        .to_str()
        .map_err(|_err| {
            eprintln!("CSRF token contains non-ASCII.");
            StatusCode::BAD_REQUEST
        })?;
    if csrf_token != session.csrf_token() {
        eprintln!("Incorrect X-CSRF-Token in request");
        return Err(*STATUS_CODE_BAD_CSRF);
    }
    req.extensions_mut().insert(session);
    Ok(next.run(req).await)
}
```

5.5 Middleware Example

One example of the session middleware being used is in the /products API, where operations which modify the products can only be performed by an Administrator, but retrieval operations can be performed by any authenticated user.

```rust
/// Create a router for routes under the product service.
pub fn create_router(state: &AppState) -> Router<AppState> {
    let authenticated = Router::new()
        .route("/", get(search_products))
        .route("/{product_id}", get(get_product))
        .route("/{product_id}/images", get(list_product_images))
        .layer(from_fn_with_state(
            state.clone(),
            session_middleware::<GenericAuthenticatedSession>,
        ));
    let admin_authenticated = Router::new()
        .route("/", post(create_product))
        .route("/{product_id}", put(update_product))
        .route("/{product_id}", delete(delete_product))
        .route("/{product_id}/images", post(add_product_image))
        .route("/{product_id}/images/{path}", delete(delete_product_image))
        .layer(from_fn_with_state(
            state.clone(),
            session_middleware::<AdministratorSession>,
        ));
    authenticated.merge(admin_authenticated)
}
```

5.6 Session Types

These are the types illustrated in the Sections and Middleware section.

```rust
/// A session which is guaranteed to have been fully authenticated. Can be
/// constructed either infallibly using `PreAuthenticationSession::promote`,
/// or fallibly by using `CustomerSession::get` like usual.
#[derive(Clone)]
pub struct CustomerSession {
    /// The inner session used to interact with the session store.
    session: BaseSession,
}

/// A session generated prior to full authentication of a user. This should
/// be generated once a user has successfully submitted primary credentials,
/// and used to keep track of that user as they continue with MFA.
#[derive(Clone)]
pub struct PreAuthenticationSession {
    /// The inner session used to interact with the session store.
    session: BaseSession,
}

/// A session used for onboarding a new user. Created when the registration
/// process begins, and deleted once it is complete. Used to store submitted
/// user data between phases of onboarding.
#[derive(Clone)]
pub struct RegistrationSession {
    /// The inner session used to interact with the session store.
    session: BaseSession,
}

/// A session which has been fully authenticated and authorized to have
/// administrative access. Note that this is mutally exclusive with
/// having recular authenticated user access.
#[derive(Clone)]
pub struct AdministratorSession {
    /// The inner session used to interact with the session store.
    session: BaseSession,
}

/// A generic authenticated session, which may either be a customer
/// or administrator session.
#[derive(Clone)]
pub enum GenericAuthenticatedSession {      Joe, 1 week ago
    /// A customer session.
    Customer(CustomerSession),
    /// An administrator session.
    Administrator(AdministratorSession),
}
```

5.7 Session Trait

This is the definition of the trait included in the diagram in the Sections and Middleware section.

```rust
pub trait SessionTrait: Send + Sync + Clone + Sized {
    /// Get an instance of this session type given the corresponding session token.
    async fn get(
        token: &str,
        session_store_conn: &mut store::Connection,
    ) -> Result<Option<Self>, errors::SessionStorageError>;
    /// Get the session token which identifies this session.
    fn token(&self) -> String;
    /// Delete this session, immediately invalidating it.
    async fn delete(
        self,
        session_store_conn: &mut store::Connection,
    ) -> Result<(), errors::SessionStorageError>;
    /// Get this session's CSRF token.
    fn csrf_token(&self) -> String;
}
```

5.8 Session Trait Example impl

This is an example of how a Session type implements the SessionTrait trait, taken from AuthenticatedSession.

```rust
impl SessionTrait for AdministratorSession {
    async fn get(
        token: &str,
        session_store_conn: &mut store::Connection,
    ) -> Result<Option<Self>, errors::SessionStorageError> {
        Ok(BaseSession::get(token, store::SessionType::Authenticated, session_store_conn).await?.and_then(
            |session| {
                session
                    .info()
                    .as_auth()
                    .expect("Got non-authenticated session back from get with SessionType::Authenticated. Major bug in session store.")
                    .admin
                    .then_some(Self { session })
            }
        ))
    }
    async fn delete(
        self,
        session_store_conn: &mut store::Connection,
    ) -> Result<(), errors::SessionStorageError> {
        session_store_conn
            .delete(&self.token(), store::SessionType::Authenticated)
            .await
    }
    fn token(&self) -> String {
        self.session.token.clone()
    }

    fn csrf_token(&self) -> String {
        self.session.info().csrf_token()
    }
}
```

5.9 Application Initialisation

```rust
#[tokio::main]
async fn main() {
    let s3 = AmazonS3Builder::new()
        .with_endpoint(format!(
            "http://{}:{}",
            &*constants::s3::S3_HOST,
            &*constants::s3::S3_PORT
        ))
        .with_bucket_name(&*constants::s3::S3_BUCKET)
        .with_access_key_id(&*constants::s3::S3_ACCESS_KEY)
        .with_secret_access_key(&*constants::s3::S3_SECRET_KEY)
        .with_allow_http(true)
        .build()
        .expect("Could not connect to S3-compatible object storage");
    println!("CONNECTED TO S3: {s3}");
    let db_conn = db::connect()
        .await
        .expect("Could not connect to primary database");
    let session_store_conn = services::sessions::store::Connection::connect()
        .await
        .expect("Could not connect to session store");
    let state = state::AppState {
        db: db_conn,
        session_store: session_store_conn,
        media_store: Arc::new(s3),
    };
    let app = axum::Router::new()
        .route("/", get(root))
        .nest("/auth", routes::auth::create_router(&state))
        .nest("/registration", routes::registration::create_router(&state))
        .nest("/products", routes::products::create_router(&state))
        .nest("/orders", routes::orders::create_router(&state))
        .nest("/webhook", routes::webhook::create_router(&state))
        .nest("/checkout", routes::checkout::create_router(&state))
        .nest("/users", routes::users::create_router(&state))
        .with_state(state);
    let listener = TcpListener::bind("0.0.0.0:80")
        .await
        .expect("Failed to bind listener");
    axum::serve(listener, app)
        .await
        .expect("Failed to init Axum service");
}
```

5.10 Stripe PaymentIntent Creation

```rust
impl CheckoutToken {
    #[cfg(feature = "stripe")]
    pub async fn create(
        user_id: Uuid,
        order_id: Uuid,
        db_conn: &db::ConnectionPool,
    ) -> Result<Self, errors::CheckoutTokenCreateError> {
        use core::iter;
        let order = AppOrder::select_one(order_id, db_conn)
            .await?
            .ok_or(errors::CheckoutTokenCreateError::OrderNonExistent { user_id, order_id })?;
        if order.user_id() != user_id {
            return Err(errors::CheckoutTokenCreateError::Unauthorized { user_id, order_id });
        }
        let stripe_client = stripe::Client::new(&*STRIPE_SECRET_KEY);
        let mut create_intent =
            stripe::CreatePaymentIntent::new(order.amount_charged, stripe::Currency::GBP);
        create_intent.payment_method_types = Some(vec!["card".to_owned()]);
        create_intent.metadata = Some(iter::once(("order_id".to_owned(), order_id)).collect());
        Ok(Self(
            stripe::PaymentIntent::create(&stripe_client, create_intent).await?,
        ))
    }
```

5.11 Stripe Webhook

```rust
pub async fn stripe_webhook_event(
    State(state): State<AppState>,
    StripeEvent(event): StripeEvent,
) -> Result<(), StatusCode> {
    #[expect(
        clippy::wildcard_enum_match_arm,
        reason = "There are over 400 possible stripe webhook events. I refuse to list them all."
    )]
    match event.type_ {
        EventType::PaymentIntentSucceeded => {
            if let EventObject::PaymentIntent(data) = event.data.object {
                let order_id: Uuid = data.metadata.get("order_id").ok_or_else(|| {
                    eprintln!("Stripe webhook paymentintent.succeeded did not contain order_id metadata");
                    StatusCode::BAD_REQUEST
                })?
                .parse().map_err(|_parse| {
                    eprintln!("Stripe webhook paymentintent order_id not an integer");
                    StatusCode::UNPROCESSABLE_ENTITY
                })?;
                orders::confirm_order(order_id, &state.db)
                    .await
                    .map_err(|error| match error {
                        OrderConfirmationError::DatabaseError(err) => {
                            eprintln!("Error raised by database while confirming order: {err}");
                            StatusCode::INTERNAL_SERVER_ERROR
                        }
                        OrderConfirmationError::OrderNonExistent => {
                            eprintln!(
                                "Stripe attempted to confirm order {order_id}, which does not exist."
                            );
                            StatusCode::NOT_FOUND
                        }
                    })?;
            }
            Ok(())
        }
        _ => Ok(()),
    }
}
```

5.12 Database Encryption Example

```rust
/// Store this INSERT model in the database and return a complete `AppUser` model.
pub async fn store(self, db_client: &ConnectionPool) -> Result<AppUser, DatabaseError> {
    #[expect(clippy::as_conversions, reason="Used in query_as! macro for Postgres coersion")]
    Ok(query_as!(
        AppUser,
        r#"INSERT INTO appuser
        (email, forename, surname, address, role)
        VALUES ($1, pgp_sym_encrypt($2, $5), pgp_sym_encrypt($3, $5), pgp_sym_encrypt($4, $5), 'Customer')
        RETURNING id, email AS "email: _", pgp_sym_decrypt(forename, $5) AS "forename!",
        pgp_sym_decrypt(surname, $5) AS "surname!",
        pgp_sym_decrypt(address, $5) AS "address!", role AS "role!: AppUserRole""#,
        String::from(self.email),
        self.forename,
        self.surname,
        self.address,
        *DB_ENCRYPTION_KEY
    ).fetch_one(db_client).await?)
}
```

Figure 29: Database Encryption Example

5.13 Authentication Logic

```rust
/// The outcome of an authentication attempt.
pub enum AuthenticationOutcome {
    /// The authentication was successful, and an ``AuthenticatedSession`` was created.
    Success(CustomerSession),
    /// The authentication was succesful, but further authentication is required. A
    /// ``PreAuthenticationSession`` was created.
    Partial(PreAuthenticationSession),
    /// The authentication was unsuccessful.
    Failure,
    /// The authentication was successful, and an ``AdministrativeSession`` was created.
    SuccessAdministrative(AdministratorSession),
}
/// Authenticate with a primary authentication method, and return a session
/// if successful. The session is not guaranteed to be fully authenticated,
/// and checking that `AuthenticatedSession::try_from_session` is successful
/// is recommended. If the session is not authenticated, then further action
/// (most likely MFA) is required.
pub async fn authenticate(
    email: EmailAddress,
    credential: PrimaryAuthenticationMethod,
    db_conn: &db::ConnectionPool,
    session_store_conn: &mut sessions::store::Connection,
) -> Result<AuthenticationOutcome, super::errors::StorageError> {
    let mut res = AppUser::search(
        AppUserSearchParameters {
            email: Some(email),
            role: None,
        },
        db_conn,
    )
    .await?;
    let Some(user) = res.pop() else {
        return Ok(AuthenticationOutcome::Failure);
    };
    if !credential.authenticate(user.id(), db_conn).await? {
        return Ok(AuthenticationOutcome::Failure);
    }
    let user_id = user.id();
    let session = PreAuthenticationSession::create(user_id, session_store_conn).await?;
    if Totp::select(user_id, db_conn).await?.is_none() {
        match user.role {
            AppUserRole::Customer => Ok(AuthenticationOutcome::Success(
                session.promote(session_store_conn).await?,
            )),
            AppUserRole::Administrator => Ok(AuthenticationOutcome::SuccessAdministrative(
                session.promote_to_admin(session_store_conn).await?,
            )),
        }
    } else {
        Ok(AuthenticationOutcome::Partial(session))
    }
}
```

Figure 30: Authentication

5.14 Two-Factor Authentication Logic

```rust
/// Outcome of a 2-factor authentication attempt.
pub enum AuthenticationOutcome2fa {
    /// The authentication was successful, an `AuthenticatedSession` was created.
    Success(CustomerSession),
    /// The authentication was successful, an `AdministrativeSession` was created.
    SuccessAdministrative(AdministratorSession),
    /// The authentication was unsuccessful.
    Failure,
}

/// Authenticate a partially authenticated user using an MFA method.
pub async fn authenticate_2fa(
    session: PreAuthenticationSession,
    method: MfaAuthenticationMethod,
    db_conn: &db::ConnectionPool,
    session_store_conn: &mut sessions::store::Connection,
) -> Result<AuthenticationOutcome2fa, super::errors::StorageError> {
    let user = AppUser::select_one(session.user_id(), db_conn)
        .await?
        .expect("User was deleting while authenticating session. Bailing.");
    if validate_2fa(session.user_id(), method, db_conn).await? {
        match user.role {
            AppUserRole::Customer => Ok(AuthenticationOutcome2fa::Success(
                session.promote(session_store_conn).await?,
            )),
            AppUserRole::Administrator => Ok(AuthenticationOutcome2fa::SuccessAdministrative(
                session.promote_to_admin(session_store_conn).await?,
            )),
        }
    } else {
        Ok(AuthenticationOutcome2fa::Failure)
    }
}
```

Figure 31: Two Factor Authentication

5.15 TOTP 2FA Generation

```rust
impl TotpInsert {
    /// Store this INSERT model in the database and return a complete `Totp` model.
    pub async fn store(&self, db_client: &ConnectionPool) -> Result<Totp, DatabaseError> {
        Ok(query_as!(
            Totp,
            "INSERT INTO totp (user_id, secret) VALUES ($1, pgp_sym_encrypt_bytea($2, $3)) RETURNING *",
            self.user_id,
            self.secret,
            *DB_ENCRYPTION_KEY
        )
        .fetch_one(db_client)
        .await?)
    }

    pub fn validate(&self, code: &str) -> bool {
        let totp = totp_rs::TOTP::from_rfc6238(
            totp_rs::Rfc6238::with_defaults(self.secret.clone())
                .expect("Non-Rfc6238-compliant secret in TOTP validation"),
        )
        .expect("Invalid URL in TOTP validation");
        totp.check_current(code)
            .expect("System time error while validating Totp code")
    }
}
```

Figure 32: TOTP 2FA Generation

5.16 TOTP 2FA Validation

```rust
impl Totp {
    /// Select a Totp record from the database by the associated user ID.
    pub async fn select(
        user_id: Uuid,
        db_client: &ConnectionPool,
    ) -> Result<Option<Self>, DatabaseError> {
        Ok(query_as!(
            Self,
            r#"SELECT user_id, pgp_sym_decrypt_bytea(secret, $2) AS "secret!" FROM totp WHERE user_id = $1"#,
            user_id,
            *DB_ENCRYPTION_KEY
        )
        .fetch_optional(db_client)
        .await?)
    }
    /// Delete the model from the database. Also consumes the model for the sake
    /// of consistency.
    pub async fn delete(self, db_client: &ConnectionPool) -> Result<(), DatabaseError> {
        Ok(query!("DELETE FROM totp WHERE user_id = $1", self.user_id)
            .execute(db_client)
            .await
            .map(|_| ())?)
    }

    /// Validate that a TOTP code is correct.
    pub fn validate(&self, code: &str) -> bool {
        let totp = totp_rs::TOTP::from_rfc6238(
            totp_rs::Rfc6238::with_defaults(self.secret.clone())
                .expect("Non-Rfc6238-compliant secret in TOTP validation"),
        )
        .expect("Invalid URL in TOTP validation");
        totp.check_current(code)
            .expect("System time error while validating Totp code")
    }
}
```

Figure 33: TOTP 2FA Validation

5.17 Image Upload Validation

Uploaded images are validated by looking at their magic bytes.

```rust
/// Supported image file types.
enum ImageFileType {
    /// A PNG image
    Png,
    /// A JPEG image
    Jpg,
    /// A GIF image
    Gif,
}

impl ImageFileType {
    /// Get the file type from the file's magic bytes, returns None if the
    /// file does not match any of the supported types.
    const fn from_bytes(bytes: &[u8]) -> Option<Self> {
        match bytes {
            &[0x89, 0x50, 0x4e, 0x47, 0x0d, 0x0a, 0x1a, 0x0a, ..] => Some(Self::Png),
            &[0xff, 0xd8, 0xff, 0xe0 | 0xee, ..]
            | &[0xff, 0xd8, 0xff, 0xe1, _, _, 0x45, 0x78, 0x69, 0x66, 0, 0, ..] => Some(Self::Jpg),
            &[0x47, 0x49, 0x46, 0x38, 0x37 | 0x39, 0x61, ..] => Some(Self::Gif),
            _ => None,
        }
    }
    /// Get the file extension typically associated with this file type.
    const fn extension(&self) -> &str {
        match *self {
            Self::Png => "png",
            Self::Jpg => "jpg",
            Self::Gif => "gif",
        }
    }
    /// Get the mimetype associated with this file type.
    const fn mimetype(&self) -> &str {
        match *self {
            Self::Png => "image/png",
            Self::Jpg => "image/jpeg",
            Self::Gif => "image/gif",
        }
    }
}
```

5.18 Image Upload Storage

This is how images are stored in the object store (MinIO). They are named by their own SHA256 hash, which means that if the same image is uploaded twice, even for different products, it will not be duplicated within the object store.

```rust
/// Store an image in the media store. Will return the path under the storage bucket
/// at which the image has been stored, and will error if the image file is of an
/// unsupported type or the networked storage access fails.
pub async fn store_image(
    store: Arc<dyn ObjectStore>,
    image: Vec<u8>,
) -> Result<String, errors::StoreImageError> {
    let mut hasher = Sha256::new();
    hasher.update(&image);
    let hash = hasher.finalize();
    let file_type =
        ImageFileType::from_bytes(&image).ok_or(errors::StoreImageError::InvalidFileType)?;
    let object_name = format!("{hash:x}");
    let object_path = PathBuf::new()
        .join(IMAGE_PREFIX)
        .join(object_name)
        .with_extension(file_type.extension())
        .to_string_lossy()
        .into_owned();
    // object_store will upsert by default, and since we use hashes, this will implicitely
    // dedup image storage.
    let mut object_attributes = Attributes::with_capacity(1);
    object_attributes.insert(
        Attribute::ContentType,
        file_type.mimetype().to_owned().into(),
    );
    object_attributes.insert(Attribute::ContentDisposition, "inline".into());
    let put_opts = PutOptions {
        attributes: object_attributes,
        ..Default::default()
    };
    store
        .put_opts(
            &Path::from(object_path.as_str()),
            PutPayload::from(image),
            put_opts,
        )
        .await
        .map_err(errors::StorageError::from)?;
    Ok(object_path)
}
```

5.19 Bruteforce Rate Limiting Usage

Here is the code or the /auth POST endpoint which handles logging in. You can see how to prevent bruteforce attempts, it accesses the X-Real-IP header, which is set by NGINX to contain the original sender's IP address.

```rust
/// Login using a credential method, and set a session cookie.
async fn login(
    headers: HeaderMap,
    cookies: CookieJar,
    State(state): State<AppState>,
    Json(body): Json<AuthenticateRequest>,
) -> Result<(CookieJar, Json<AuthenticateResponse>), HttpError> {
    let client_ip = headers
        .get("x-real-ip")
        .ok_or_else(|| {
            eprintln!("X-Real-IP header not set, I should be running behind a reverse proxy.");
            HttpError::new(
                StatusCode::BAD_REQUEST,
                Some(String::from("X-Real-IP not set")),
            )
        })?
        .to_str()
        .map_err(|err| {
            eprintln!("Failed to parse X-Real-IP header value: {err}");
            HttpError::new(
                StatusCode::BAD_REQUEST,
                Some(String::from("X-Real-IP value unparseable")),
            )
        })?;
    if state
        .session_store
        .clone()
        .bruteforce_timeout(client_ip)
        .await?
    {
        eprintln!(
            "Client {client_ip} is rate-limited for suspected bruteforce authentication attempt."
        );
        return Err(HttpError::new(
            StatusCode::TOO_MANY_REQUESTS,
            Some(String::from("Too many authentication attempts.")),
        ));
    }
    let mut session_store = state.session_store.clone();
    let outcome = auth::authenticate(
        body.email.clone(),
        body.credential,
```

5.20 Bruteforce Rate Limiting Implementation

This is the actual implementation of the rate limiting, making use of Redis' built-in key expiry. While the client continues retrying the request, the expiry time keeps getting reset back to the penalty period. Only once the timeout finishes and the key expires can another request be made.

```rust
/// Increments an internal counter to indicate an authentication attempt, and returns whether the user is timed out
pub async fn bruteforce_timeout(
    &mut self,
    client: &str,
) -> Result<bool, errors::SessionStorageError> {
    let key = format!("bruteforce:{client}");
    let attempts: u32 = self.0.incr(&key, 1u32).await?;
    if attempts < AUTH_TIMEOUT_ATTEMPTS {
        let _: () = self.0.expire(&key, i64::from(AUTH_TIMEOUT_PERIOD)).await?;
        Ok(false)
    } else {
        let _: () = self.0.expire(&key, i64::from(AUTH_PENALTY_PERIOD)).await?;
        Ok(true)
    }
}
```

5.21 Input Validation

Here are a few examples of input validation implemented in the application. I tried to make use of types which already had constraints and custom deserialisation methods implemented, such as the Uuid type from the uuid crate, to reduce the amount of manual input validation I had to do.

This is a custom "newtype" (a single element tuple struct which opaquely wraps an existing type) which represents an email address guaranteed to be validly formatted. There is no way to construct an instance of the EmailAddress type which is invalidly formatted.

```rust
impl TryFrom<String> for EmailAddress {
    type Error = ();
    fn try_from(string: String) -> Result<Self, Self::Error> {
        if EMAIL_REGEX.is_match(&string) {
            Ok(Self(string))
        } else {
            Err(())
        }
    }
}

impl From<EmailAddress> for String {
    fn from(addr: EmailAddress) -> Self {
        let EmailAddress(inner) = addr;
        inner
    }
}

impl<'de> Deserialize<'de> for EmailAddress {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
    where
        D: serde::Deserializer<'de>,
    {
        let str = String::deserialize(deserializer)?;
        Self::try_from(str).map_err(|_err| de::Error::custom("malformed email address"))
    }
}
```

```rust
/// Regex used to validate email address format. Non-comprehensive but good enough.
static EMAIL_REGEX: LazyLock<regex::Regex> = LazyLock::new(|| {
    regex::Regex::new(r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+(\.[a-zA-Z0-9-]+)+$")
        .expect("Email regex invalid")
});
```

Figure 34: The regex used to validate email address format

```rust
#[derive(Clone, sqlx::Type)]
#[sqlx(transparent)]
pub struct EmailAddress(String);
```

Figure 35: The definition of the EmailAddress newtype struct

Password are checked to be within a given length range to avoid very insecure passwords and denial of service attacks.

```rust
match credential {
    PrimaryAuthenticationMethod::Password { password } => {
        if password.len() < PASSWORD_MIN_LENGTH {
            return Err(errors::AddCredentialError::PasswordTooShort);
        }
        if password.len() > PASSWORD_MAX_LENGTH {
            return Err(errors::AddCredentialError::PasswordTooLong);
        }
        let password_model = PasswordInsert::new(stored_user.id(), &password);
```

When creating a new account, none of the fields can be blank.

```rust
/// Errors returned while initiating an onboarding session.
#[derive(Error, Debug)]
pub enum SignupInitError {
    /// An error in the underlying storage
    #[error(transparent)]
    StorageError(#[from] StorageError),
    /// The signup attempt uses an email which is already registered.
    #[error("Email is already is use")]
    DuplicateEmail(String),
    #[error("The signup address field is empty")]
    EmptyAddress,
    #[error("The signup surname field is empty")]
    EmptySurname,
    #[error("The signup forename field is empty")]
    EmptyForename,
}
```