SLH
Lab #2

- This lab will be graded.
- The quality of your code will be graded.
- Your submission has to be in Rust.
- Test your code whenever you can, but we will **not** put the focus on testing.
- We provide you with a backend template and a fully implemented frontend.

1 Description

The objective of this lab is to build a web backend that supports two methods of authentication. A user can register a new account and login with an email/password combination. The second method of authentication is OAuth2 so that the user can login with his Google account. We chose Google OAuth 2.0 as an example as it is widely used. However, you are free to use another provider.
The website only provides basic functionalities. If you are not logged in, only the `/home` and `/login` pages can be accessed. A logged in user has access to two new pages; they can either change their password or logout.

2 Explanations on the template

The template that we provide has most of the server code already implemented. In addition to this template, you will need a running PostgreSQL database. You can either use a local database, or start the container with the provided `docker-compose.yml`.
Our template requires several libraries to support the web application. The main web library that we use is `axum` for the endpoints, we do a bit of HTML templating with `handlebars` and we use the `diesel` ORM to interact with the database.

2.1 axum

`axum`[1] is the web application framework used to implement the backend. We start the server in `main.rs` and listen on `localhost:8000`. We show the function used to provide the `/home` endpoint below.

[1] `https://github.com/tokio-rs/axum`

```rust
async fn get_home(State(hbs): State<Handlebars<'_>>, user: Option<UserDTO>) -> impl IntoResponse {
    Html(hbs.render("home", &user).unwrap())
}
...
impl<S> FromRequestParts<S> for UserDTO
```

First, we declare an `async` function that takes two arguments and returns a struct implementing the `IntoResponse` trait. The first argument is a `State` parameter; this is essentially used to share objects across all endpoints. The second argument is used to get the current authenticated user or `None` otherwise. For endpoints that can only be accessed by authenticated users, we omit the `Option` and simply use `user: UserDTO`. The authentication is checked by implementing the `FromRequestParts` trait.
Cookies can be created with `axum-extra`'s `CookieJar`. Temporary values may be stored in the server's memory using `axum-sessions`'s `MemoryStore`.

2.2 handlebars

`handlebars`[2] is used to do some templating. Essentially, the home page displays different information if the user is authenticated or not. You should not have to use this crate; the code is already implemented.

2.3 diesel

`diesel`[3] is used to interact with the PostgreSQL database in order to store the users. All the code is already provided in `db.rs`. The DB configuration is present in the `.env` file.

3 Main tasks

3.1 Standard authentication

The first step is the implementation of the standard authentication. You need to implement the register/login endpoints inside `web_auth.rs`. The register function should ensure that the email address is valid by sending a verification link. You can use the `lettre`[4] crate. You will need to implement the email verification endpoint yourself.
Once the password has been verified, you must use a JWT (JSON Web Token) to authenticate the user and verify the token in `impl FromRequestParts for UserDTO` . We recommend the `jsonwebtoken`[5] crate. You must not use `axum-extra`'s `SignedCookieJar` or `PrivateCookieJar`.

3.2 OAuth2

In addition to the standard authentication, you also need to implement an OAuth2 authentication mechanism using the `oauth2`[6] crate. You are free to use the provider and the authenti-

[2] https://github.com/sunng87/handlebars-rust
[3] https://github.com/diesel-rs/diesel
[4] https://github.com/lettre/lettre
[5] https://github.com/Keats/jsonwebtoken
[6] https://github.com/ramosbugs/oauth2-rs

cation method you want (e.g. GitHub, Google, Microsoft). However, the lab was tested with Google OAuth 2.0.
In order to setup OAuth 2.0, you need to configure it in the Google Cloud console. You can use the guide here or in Appendix A.

4 Starting the application

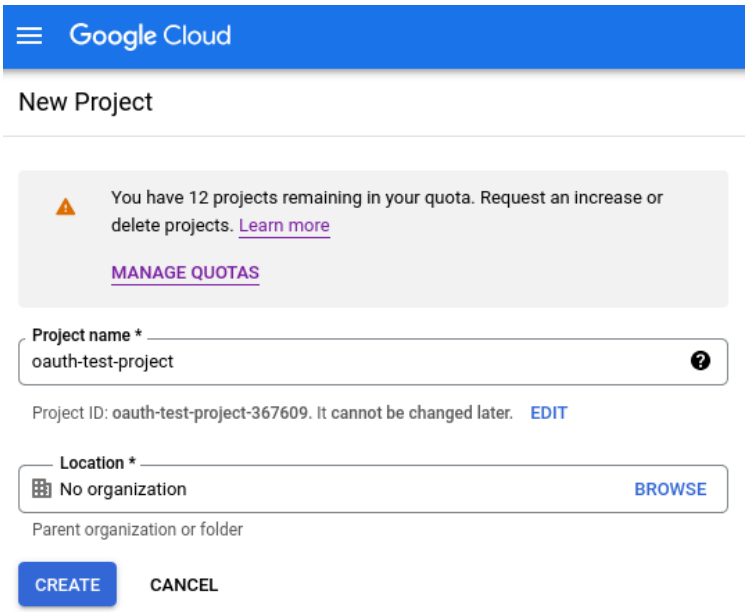
If you are on Linux / macOS, you will have to install the PostgreSQL client libraries. On Ubuntu / Debian, install `libpq-dev`. On Arch, install `postgresql-libs`. On macOS, install `postgresql` (e.g. using `brew install postgresql`). The Windows libraries are pre-packaged in the `win_libs` folder and automatically copied in the build folder by `build.rs`.
Start the PostgreSQL container with `docker compose up`. Then, you can start the server with `cargo run`. Once the many dependencies have been downloaded and compiled (sorry!), the server should start and display the following:

```
$ cargo run
Finished dev [unoptimized + debuginfo] target(s) in 0.05s
    Running `target/debug/auth`
Setting up DB pool...
Executing DB migrations
DB migrations successful
listening on 127.0.0.1:8000
```

You can now access the website via `http://localhost:8000`.

A Google Cloud OAuth 2.0

First, go to `https://console.cloud.google.com/projectselector2/apis/credentials` and login with your Google account. You will need to accept the Google Cloud Platform's ToS. Once this is done, press create a new project. You should have the following:



Under APIs & Services, go to Credentials. Select configure consent screen and select User Type External. Specify the app name, your Google account email address, add the `/auth/userinfo.mail` scope. The summary should like that:

- User type: External
- App name: Whatever
- Support email: your_email@google.com
- App*: Not provided
- Authorized domains: Not provided
- Contact email addresses: your_email@google.com

Go back to the Credentials page. Select create credentials, OAuth client ID, choose a web application type, specify a name. The authorized redirect URIs should be `http://localhost:8000/_oauth`. Change the port / hostname as needed. You get the client ID and the client secret, those values will be needed by the Rust application.