Secure UserManagement

Struktur

```
.
├── api                    // Eine Rust Applikation, läuft auf localhost:8181 (backend)
├── docker-compose.yml     // Enthält Instruktionen zum Aufstarten der Datenbank
└── frontend               // Eine Elm Applikation, läuft auf localhost:8000 (frontend)
```

API

Routes

```
Mounting /api/v1/:
    => POST /api/v1/users/all application/json (all_users)
    => POST /api/v1/users/add application/json (new_user)
    => POST /api/v1/users/<id> application/json (show_user)
    => POST /api/v1/login application/json (login)
```

Die API stellt kann verschiedene Abfragen behandeln (siehe Bild), wovon *new_user* und *login* ohne Login (bzw. JWT Token) aufgerufen werden können. Den anderen (*all_users* und *show_user*) muss ein valides Token im json mitgegeben werden.

Hashing der Passwörter

Beim Registrieren werden die Passwörter vor dem Einsetzen in die Datenbank mit dem Algorithmus/Standard *Argon2* gehashed.
Ein zufälliger Salt wird für jeden neuen Nutzer generiert und ebenfalls in der *users*-Tabelle gespeichert.
Ein zusätzlicher Schlüssel befindet sich nur auf dem Server und ist für jeden neuen User gleich.
Beim Login wird der Salt wieder aus der Datenbank genommen und der Hash in der Datenbank wird mit einem neuen Hash aus dem eingegebenen Passwort und dem Salt verglichen:

```
match user_by_name {
    None => None,
    Some(user_by_name) => {
        // compare hashed values
        let pw_ok = pw_hashing::compare_input_to_hashed_value(password, &user_by_name.pw_hash, &user_by_name.pw_salt);
        // if equal, the user with given password and username exists
        if pw_ok {
```

JWT

Die JWT-Tokens werden wie folgt erstellt:

```
encode(&Header::default(), &payload, KEY).unwrap()
```

Wenn die Validierung erfolgreich ist, wird der Login-Anfrage das generierte JWT-Token mitgesendet:

```
json!({
    "status": validation_ok,
    "result": if validation_ok { jwt_token } else { "".to_string() }
})
```

Bei Anfragen, welche ein Login erfordern, muss das Token vom User mitgeschickt werden und die Validität wird geprüft:

```rust
pub fn has_access_from_jwt_token_str(token: &String) -> bool {
    let result = decode::<UserRolesToken>(&token, KEY.as_ref(),
      &Validation::default());
    match result {
        Ok(_) => true,
        Err(_) => false
    }
}
```

Datenbankabfragen

Die Datenbankabfragen wurden mit der ORM Library *diesel* durchgeführt, in welcher Prepared Statements bereits integriert sind.
Abfragen sehen (am Beispiel eines Inserts) folgendermassen aus:

```rust
pub fn insert(user: NewUser, conn: &PgConnection) -> bool {
    diesel::insert_into(users::table)
        .values(&user)
        .execute(conn)
        .is_ok()
}
```

CORS

Damit das Frontend und Backend problemlos miteinander kommunizieren können, musste ich auf dem Server das *Cross Origin Resource Sharing* aktivieren.

```rust
let allowed_origins = AllowedOrigins::all();

    rocket_cors::CorsOptions {
        allowed_origins: allowed_origins,
        allow_credentials: true,
        ..Default::default()
    }
```

Logging

Zum Erkennen von Abläufen werden fortlaufend Nachrichten auf dem *stdout* ausgegeben:

```
POST /api/v1/login application/json:
    => Matched: POST /api/v1/login application/json (login)
They're equal!
Success: Created a valid cookie on login!!
    => Outcome: Success
    => Response succeeded.
```

Frontend

Wenn der Benutzer nicht eingeloggt ist, stehen ihm die LoginPage und die RegisterPage zur Verfügung.
Nach einem erfolgreichen Login wird das JWT Token beim Benutzer gespeichert.
Wenn der Benutzer eingeloggt ist, stehen ihm die Logout Funktion, die HomePage und die UsersPage zur verfügung.
In der HomePage wird dem Benutzer das Token angezeigt und in der UsersPage werden alle Benutzer abgebildet, welche vom Server nur abgefragt werden können, wenn ein valides Token mitgesendet wurde.