Group information:

Matias Ramsland, StudID: 259150
Lukasz Pietkiewicz, StudID: 253469
Chiran Pokhrel, StudID: 259205
Jakub Mroz, StudID: 260703
Konrad Jarczyk , StudID: 242615

1. Introduction

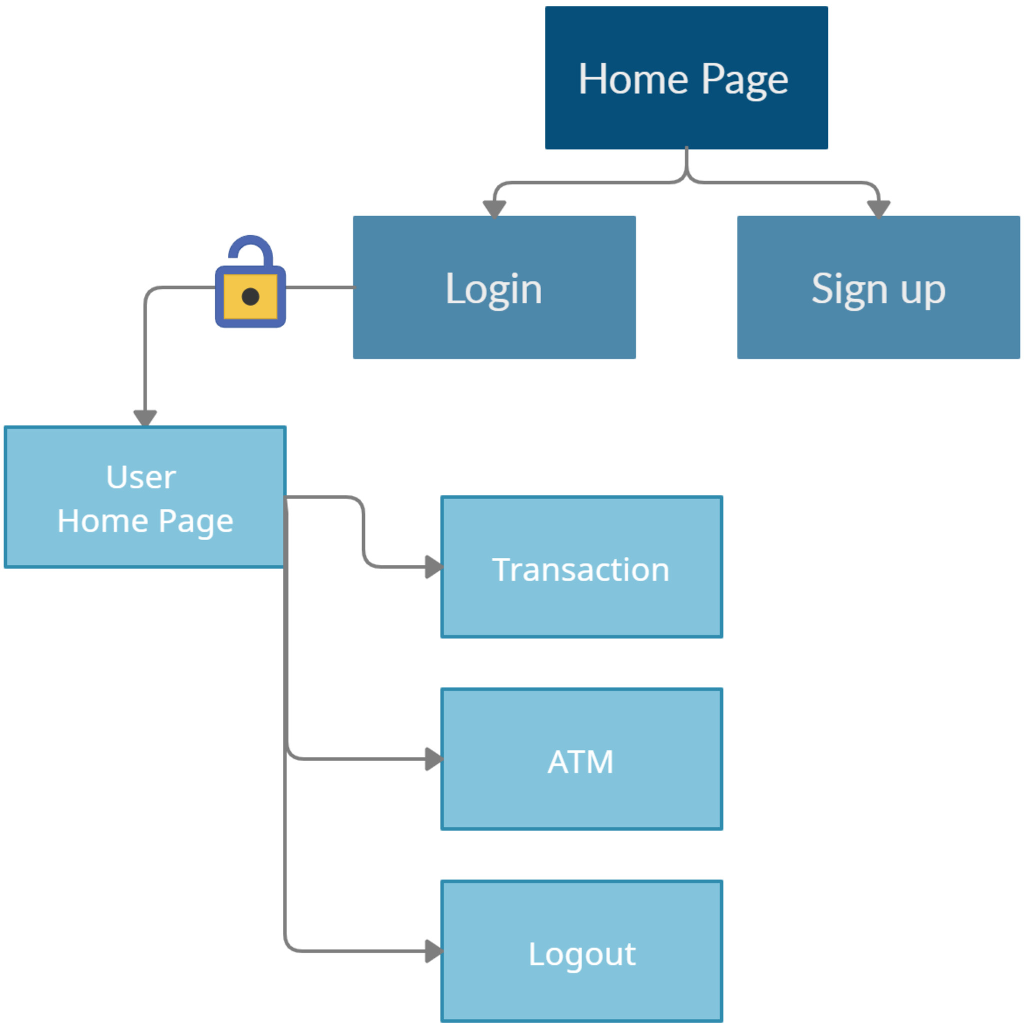
The goal of the project was to create a secure banking application, resistant to OWASP TOP10 attacks.

Our application allows users to add money to their account and send it to a different user through a webpage. Visitors cannot use banking services. To become a user, a visitor must sign up for an account first.

The application is written in python with flask framework, and is using SQLAlchemy for database.

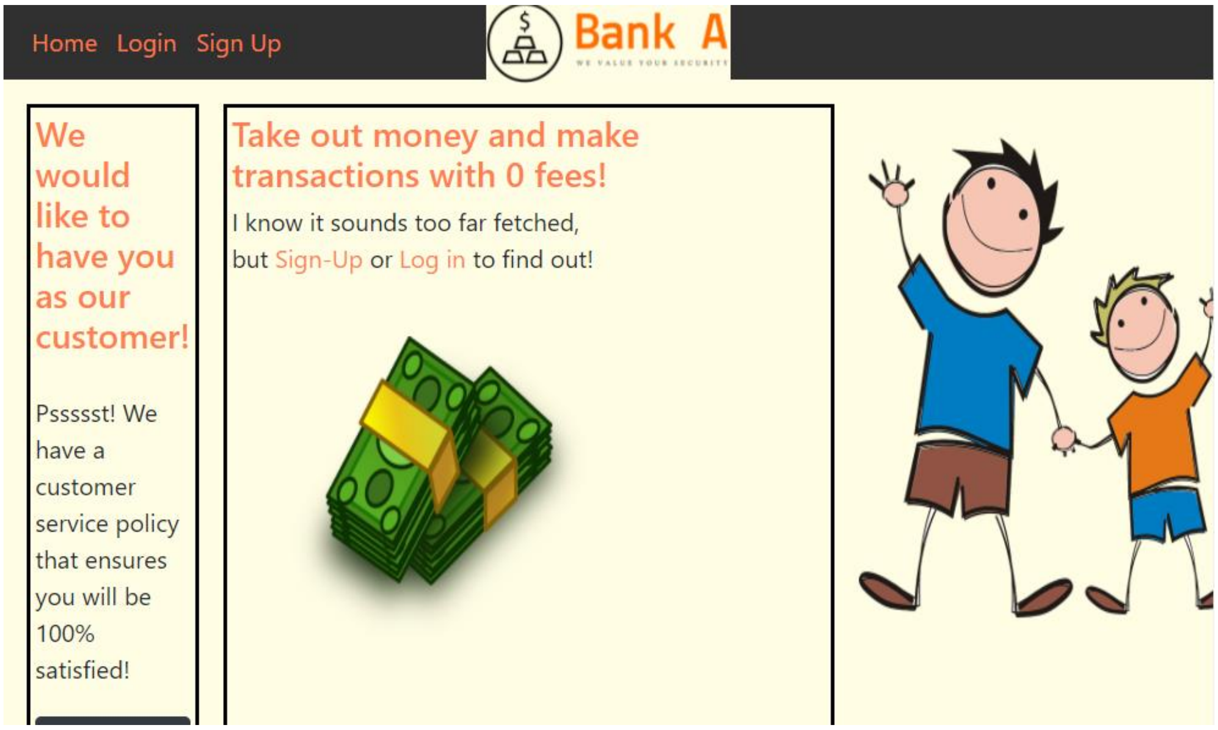
2. Layout/Functions/Front-end

[[pic1 Site map]]



A visitor can only see login and signup pages from the homepage. When an unlogged user tries to visit the URL of any other page, he gets redirected back to the homepage.

[[pic2 what visitor sees on homepage]] //todo pic close to navigation bar



We would like to have you as our customer!

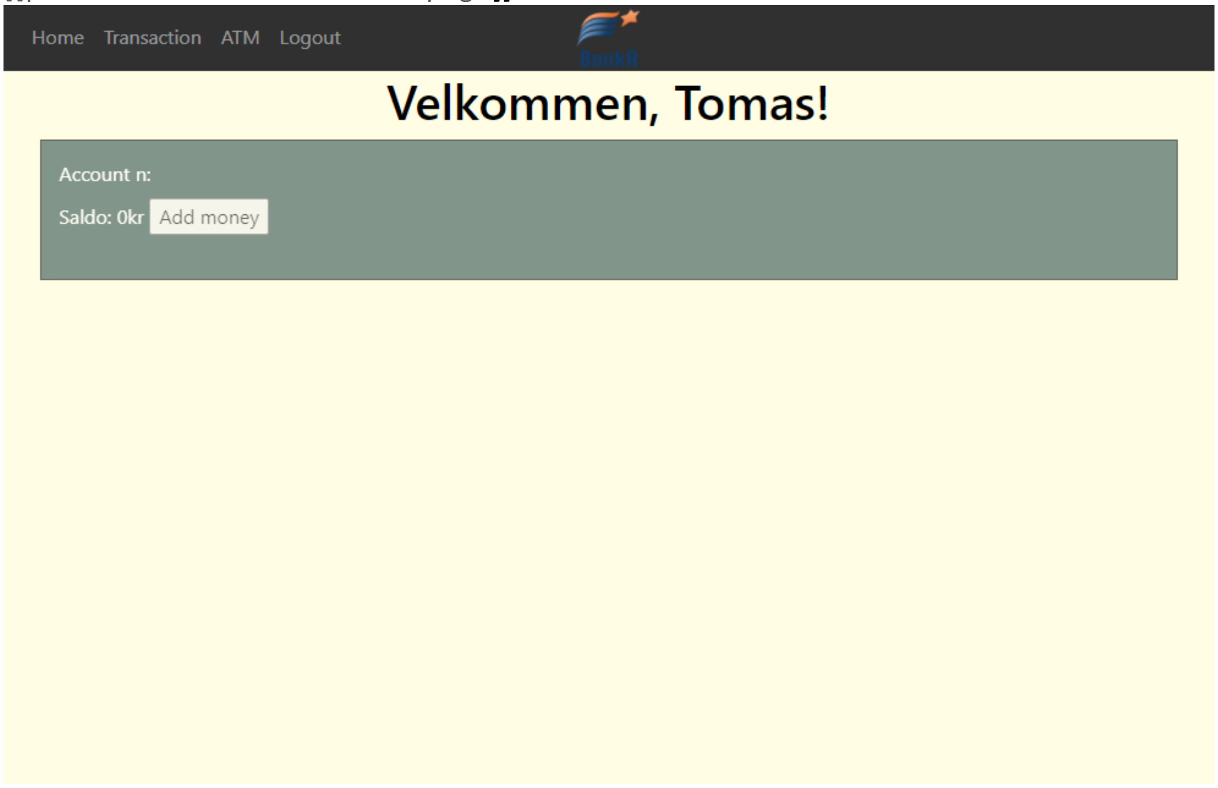
Pssssst! We have a customer service policy that ensures you will be 100% satisfied!

Take out money and make transactions with 0 fees!

I know it sounds too far fetched, but Sign-Up or Log in to find out!

After logging in, the user gets access to ATM and Transaction pages, also homepage changes to show the user's saldo.

[[pic3 what user sees on homepage]]

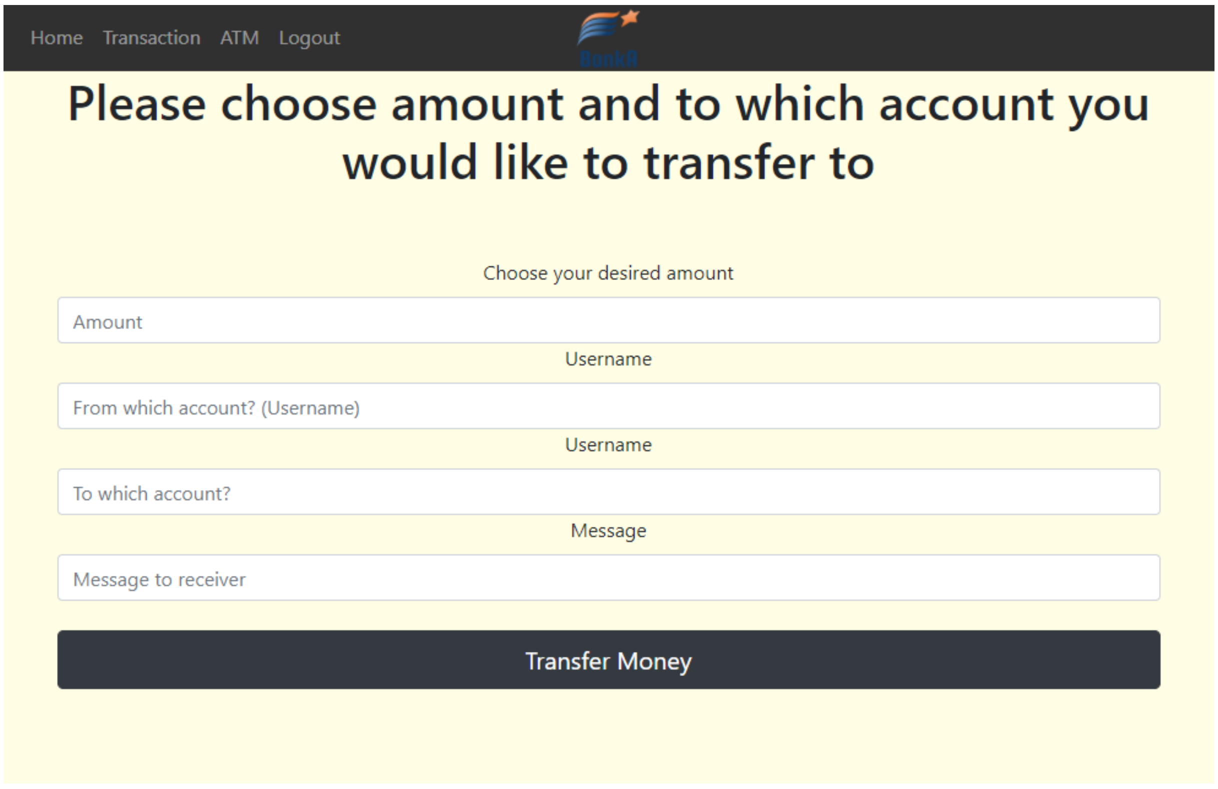


Velkommen, Tomas!

Account n:

Saldo: 0kr

[[pic3.1 transfer]]



Please choose amount and to which account you would like to transfer to

Choose your desired amount

Amount

Username

From which account? (Username)

Username

To which account?

Message

Message to receiver

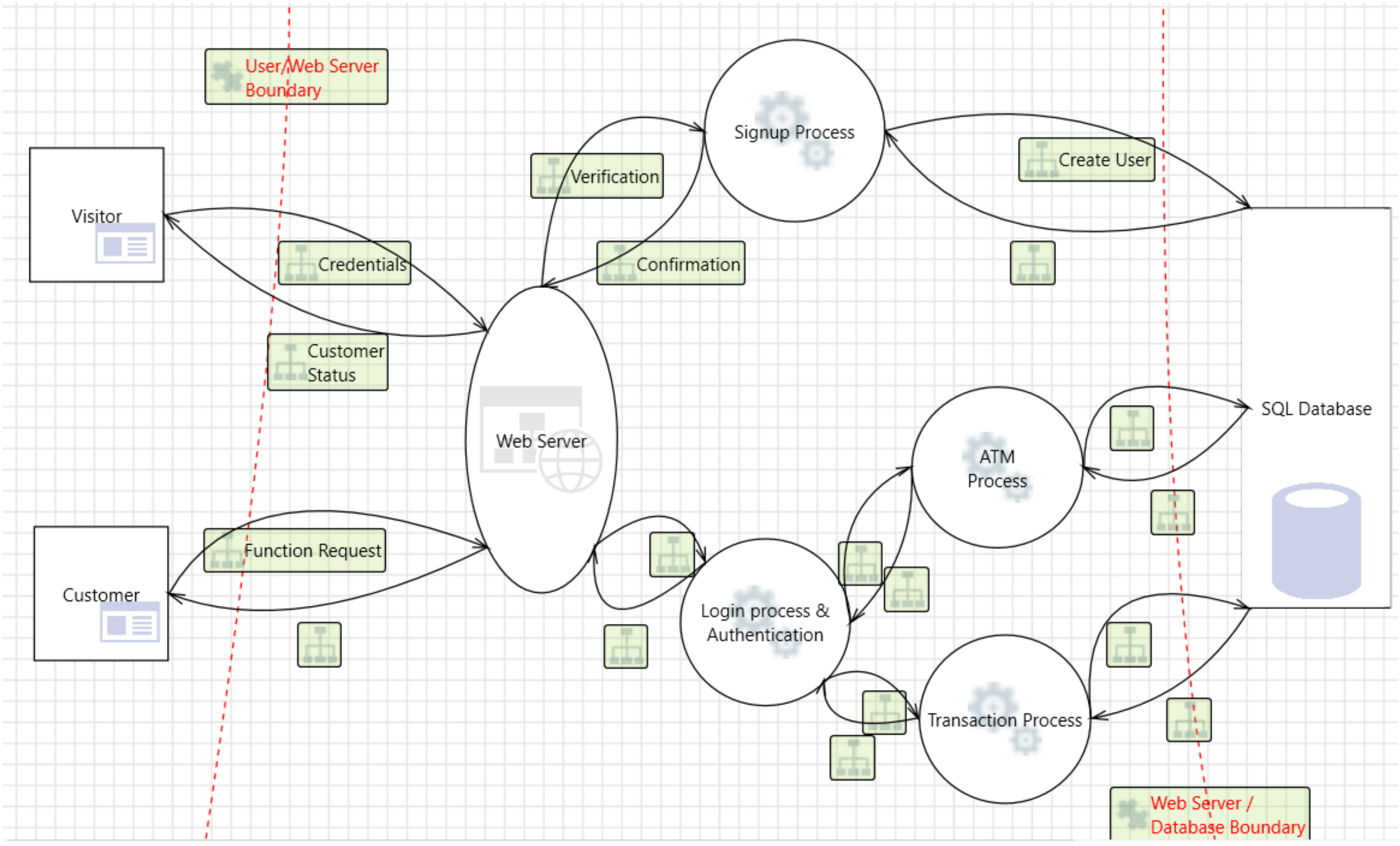
Transfer Money

To transfer cash from your own bank account in the Transaction page, the user must know the username of the receiver, and in addition confirm his own.

ATM service simulates depositing cash in a local ATM to fill your account.

3. Security/Design

[[pic4 Data Flow model]]



[[pic5 Attack tree]] //TODO

Possible threats are: //maybe unnecessary when attack tree is made

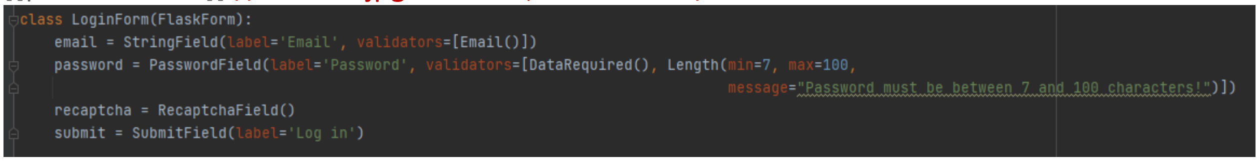
- SQL injection
- Broken Authentication
- Spoofing
- Broken Access Control
- Information disclosure

3.1 How it’s done/ with pics of code

>Getting user input

WTForms extension is used to gather and validate input from users. Depending on the need, built-in or custom validators are used.

[[pic6 wtforms]] //code as jpgs for now, later as text/code in Latex

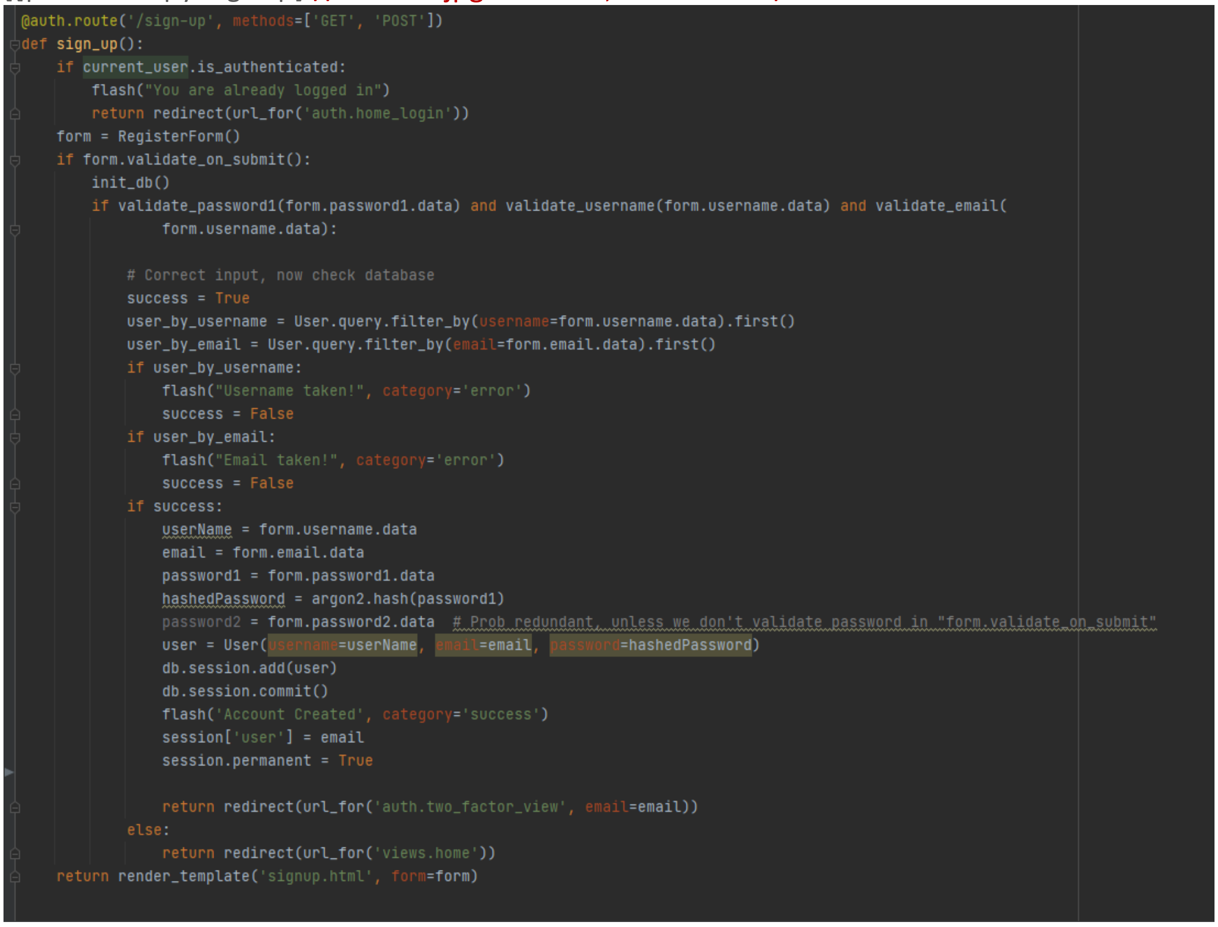


```
class LoginForm(FlaskForm):
    email = StringField(label='Email', validators=[Email()])
    password = PasswordField(label='Password', validators=[DataRequired(), Length(min=7, max=100,
                                                                                  message="Password must be between 7 and 100 characters!")])

    recaptcha = RecaptchaField()
    submit = SubmitField(label='Log in')
```

>Creating user

When the signup form is sent, the input is validated, then a check is made to see if a user with a given name or email already exists. If not - then the given password is hashed and user data is sent to the database and saved. App then redirects to the homepage. If something goes wrong during the process, a red error message will appear.

[[pic7 auth.py signup]] //code as jpgs for now, later as text/code in Latex



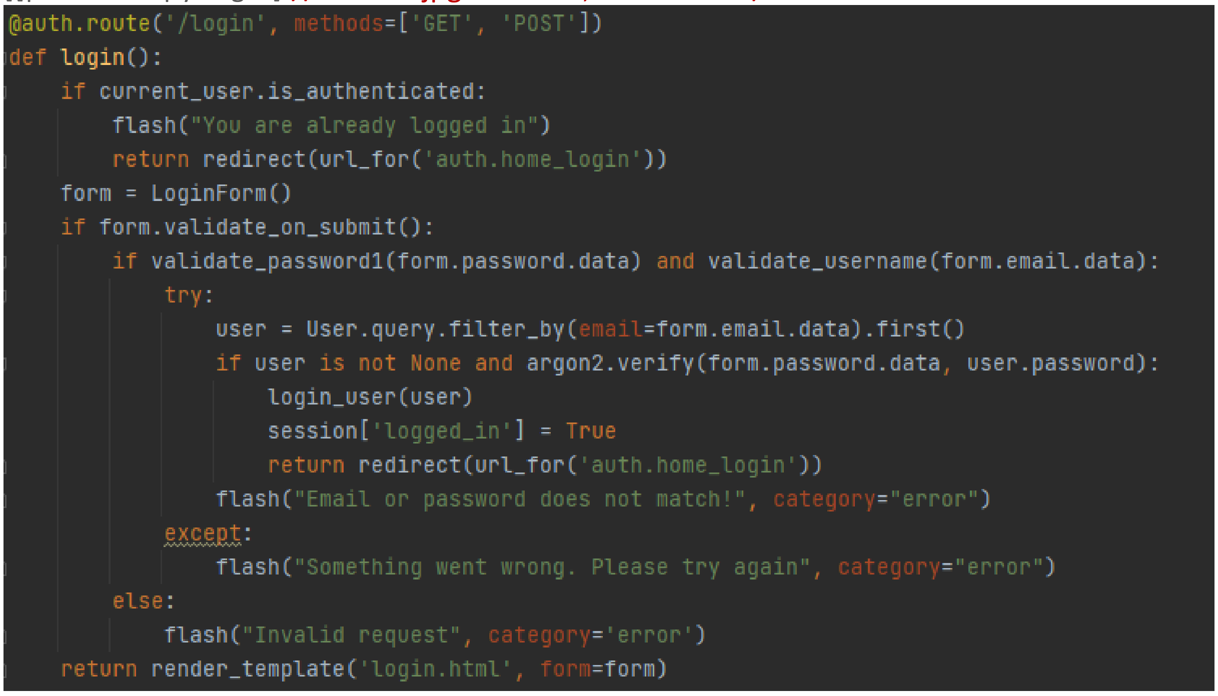
```
@auth.route('/sign-up', methods=['GET', 'POST'])
def sign_up():
    if current_user.is_authenticated:
        flash("You are already logged in")
        return redirect(url_for('auth.home_login'))
    form = RegisterForm()
    if form.validate_on_submit():
        init_db()
        if validate_password1(form.password1.data) and validate_username(form.username.data) and validate_email(
                form.username.data):

            # Correct input, now check database
            success = True
            user_by_username = User.query.filter_by(username=form.username.data).first()
            user_by_email = User.query.filter_by(email=form.email.data).first()
            if user_by_username:
                flash("Username taken!", category='error')
                success = False
            if user_by_email:
                flash("Email taken!", category='error')
                success = False
            if success:
                userName = form.username.data
                email = form.email.data
                password1 = form.password1.data
                hashedPassword = argon2.hash(password1)
                password2 = form.password2.data  # Prob redundant, unless we don't validate password in "form.validate_on_submit"
                user = User(username=userName, email=email, password=hashedPassword)
                db.session.add(user)
                db.session.commit()
                flash('Account Created', category='success')
                session['user'] = email
                session.permanent = True

                return redirect(url_for('auth.two_factor_view', email=email))
            else:
                return redirect(url_for('views.home'))
    return render_template('signup.html', form=form)
```

Argon2 function is being used to hash passwords.

>Login process/ Session management

When the login form is sent, the input is also validated. Then the database is searched for a user with a given email and given password is compared with hashed database password. If everything is okay, the app redirects the user to his homepage. If something goes wrong during the process, a red error message will appear.

[[pic8 auth.py login] //code as jpgs for now, later as text/code in Latex

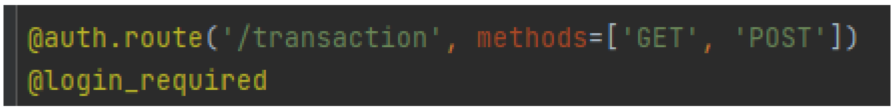


```
@auth.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        flash("You are already logged in")
        return redirect(url_for('auth.home_login'))
    form = LoginForm()
    if form.validate_on_submit():
        if validate_password1(form.password.data) and validate_username(form.email.data):
            try:
                user = User.query.filter_by(email=form.email.data).first()
                if user is not None and argon2.verify(form.password.data, user.password):
                    login_user(user)
                    session['logged_in'] = True
                    return redirect(url_for('auth.home_login'))
                flash("Email or password does not match!", category="error")
            except:
                flash("Something went wrong. Please try again", category="error")
        else:
            flash("Invalid request", category='error')
    return render_template('login.html', form=form)
```

Login process includes CAPTCHA feature to make brute-forcing passwords harder. Later user can log out simply by clicking on the logout button in the navigation bar.

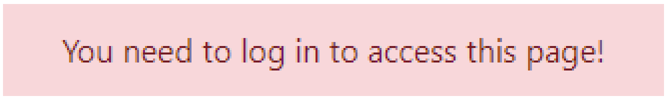
The application uses the Flask-Login extension for session management. It allows us to restrict some pages to be visible only for logged-in users.

[[pic9.1 auth.py @loginrequired]



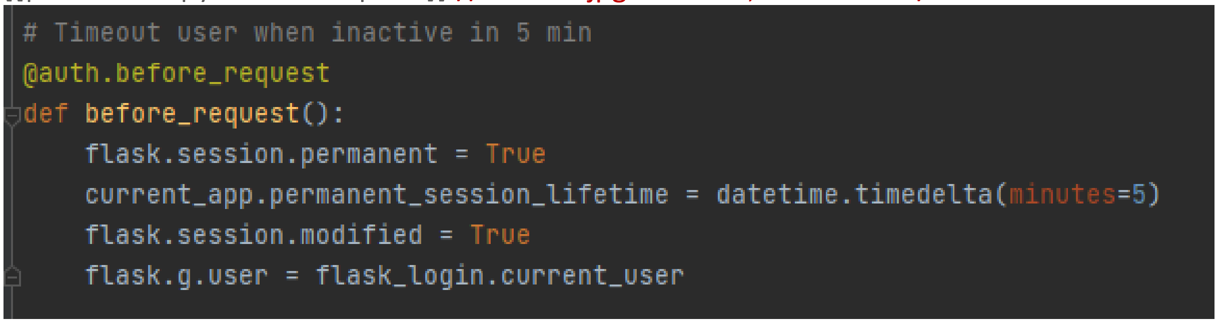
```
@auth.route('/transaction', methods=['GET', 'POST'])
@login_required
```

[[pic9.2 no access]



You need to log in to access this page!

When user becomes inactive for 5 minutes he gets logged out automatically
[[pic10 auth.py before request]] //code as jpgs for now, later as text/code in Latex



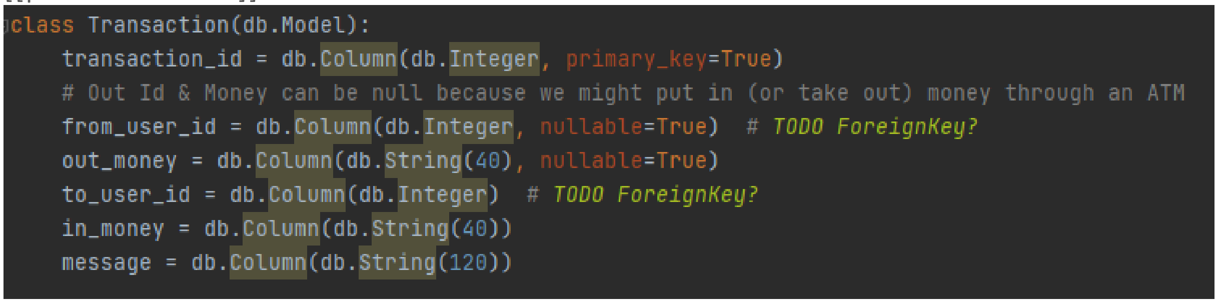
```
# Timeout user when inactive in 5 min
@auth.before_request
def before_request():
    flask.session.permanent = True
    current_app.permanent_session_lifetime = datetime.timedelta(minutes=5)
    flask.session.modified = True
    flask.g.user = flask_login.current_user
```

>ATM & Transaction

These services first validate data, then authenticate given data to check if said username exists. If all is correct, a transaction is made. If something goes wrong during the process, a red error message will appear. //TODO write more

> Database

Flask-sqlalchemy extension provides support for SQLALchemy. We use 2 database models - user and transaction.
[[pic11 db model]]



```
class Transaction(db.Model):
    transaction_id = db.Column(db.Integer, primary_key=True)
    # Out Id & Money can be null because we might put in (or take out) money through an ATM
    from_user_id = db.Column(db.Integer, nullable=True)  # TODO ForeignKey?
    out_money = db.Column(db.String(40), nullable=True)
    to_user_id = db.Column(db.Integer)  # TODO ForeignKey?
    in_money = db.Column(db.String(40))
    message = db.Column(db.String(120))
```

3.2 OWASP TOP 10

which attacks are we prepared against + how //TODO

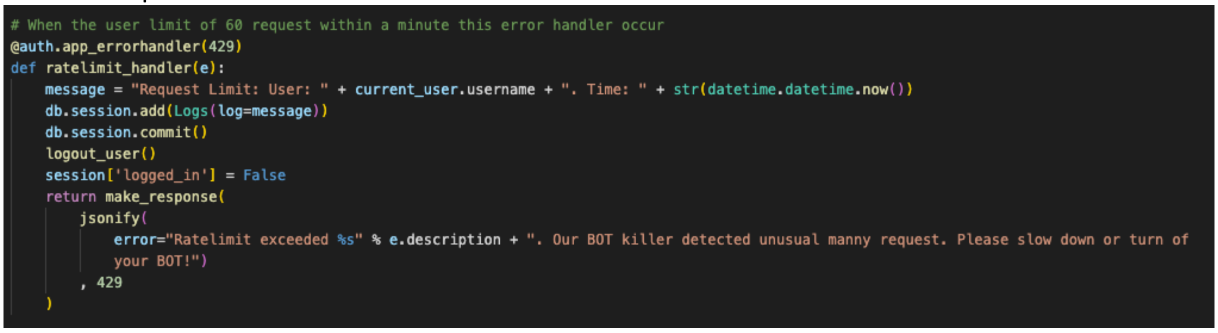
3.2.1 sql injection

etc...

3.2.2 A07:2021 Identification and Authentication Failures

This security fault is known as Broken Authentication, which is about user´s identity is breached with weaknesses with the authentication in our application, for example using automated attacks. We have implemented a few things to mitigate this, as reCAPTCHA, ratelimit , two factor authentication, strong password and deleting cookies after unused.

Firstly to avoid brute forcing and automated attacks, reCAPTCHA and ratelimit has a huge impact on this. The way we implanted this is making a reCAPTCHA user on google and verifying this in our flask WTforms. <code> reCAPTCHA = recaptchaField() <code>. It is possible to bypass wtforms validators/reCAPTCHA so we also made a request limiter, which makes a response and blocks out the user, which will reset in a couple of limits. The code we used to implement this is:



```
# When the user limit of 60 request within a minute this error handler occur
@auth.app_errorhandler(429)
def ratelimit_handler(e):
    message = "Request Limit: User: " + current_user.username + ". Time: " + str(datetime.datetime.now())
    db.session.add(Logs(log=message))
    db.session.commit()
    logout_user()
    session['logged_in'] = False
    return make_response(
        jsonify(
            error="Ratelimit exceeded %s" % e.description + ". Our BOT killer detected unusual manny request. Please slow down or turn of
            your BOT!")
        , 429
    )
```

If an attacker manages to get ahold of an users password we uses two factor authentication which is compatible with google authenticator. We uses this verification after each transaction, deposit and when we log-in. This makes the attacker not being able to access that users account without also having access to their google authenticator app. The code we used to implement this is <code>

Lastly to mitigate this security fault is checking if the user has been inactive for more than 5 minutes. This is important if a user forgets to exit the website or logging out, because then the session cookie will still be available

//TODO make better subsection names