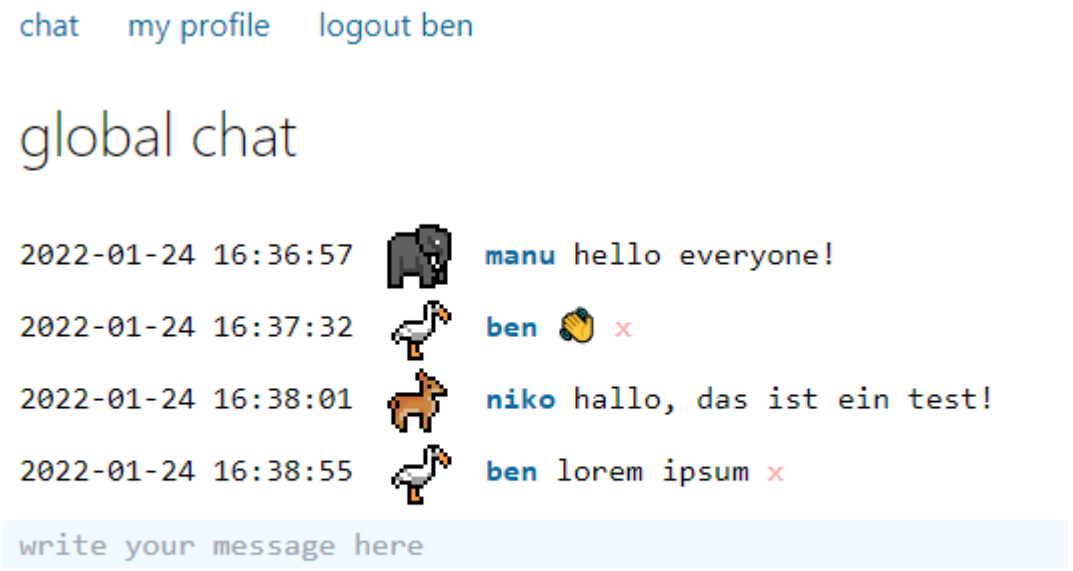
Web Application Security

For my WEB (web application security) seminar paper I have decided to write a chat-app with profile function that can be integrated in a canvas-based multiplayer game. Since the game is still very much work in progress I have focused on creating a web-app with the corresponding functionality. (See the GitHub version of this writeup. Highly recommended!)



> landing page for logged-in users

table of contents

TODO

- [ ] grep for todos and fix them (`rg -i todo`)
- [ ] write tests
- [ ] deploy app with https certificate
    - see deployment section for details
- [ ] change docstrings to flasgger format
- [ ] switch to PostGres
- [ ] use type hinting

local development setup

The app is based on Flask, a Python micro web framework. To get it running in development mode execute the following steps that will install all required dependencies. For a detailed breakdown of used libraries see the section: used libraries.

```
git clone 'git@github.com:bmedicke/MCS3_WEB_seminar_paper.git' # clone repo.
cd MCS3_WEB_seminar_paper # switch to it.

python3 -m venv env # create virtual environment.
source env/bin/activate # activate virtual environment.

pip install -r requirements.txt # install dependencies.

# optional:
docker-compose up -d # start docker-compose services in background.
```

See .flaskenv for configuration options including the bound network interface and port. By default the development server will run at: 0.0.0.0:7701.

Security note: Note, that the secret, that is used for signing session cookies, defaults to `dev` if the environment variable `SECRET_KEY` is not set. There are three ways to set this key

when deploying:

- via `export SECRET_KEY=xxxx` before starting flask
- via `SECRET_KEY=xxxx` in `.env` (recommended)
- via `SECRET_KEY=xxxx` in `.flaskenv` (not recommended since this file is commited)

Both `.flaskenv` and `.env` are automatically parsed.

You can use `flask generate-secret-key` to create your own secure key. This command uses the `token_urlsafe()` function from Python's `secret` module to generate cryptographically strong random strings (32 characters long). **This string should not be commited**!

```
root::kali:MCS3_WEB_seminar_paper:# flask create-secret-key
i1QsvPSuoWM3eFQ6ZMfGh_84j6IzUZ-u1Jvpt37XqEg
root::kali:MCS3_WEB_seminar_paper:#
```

> example run of `flask generate-secret-key` (be sure to run it yourself)

The following docker-compose services are available:

- **db**: Postgres
  - *security note:* standard password should be changed to something secure
  - *security note:* password should be removed from docker-compose file (and it should not be commited)
  - since the app is currently using `sqlite` this service is not in use
- **adminer** (localhost:7780)
  - web-base database manager/GUI

starting the app

The app uses a SQLite (file-based) database for storing user profiles and messages. Before starting the app the database schema has to be used to create the database:

```
flask init-db # apply db schema (recreates db if it exists).
sqlitebrowser instance/flask-api.sqlite # take a look at the schema.
flask run # see .flaskenv and .env for environment variables.
```

To check if configuration changes took affect you can run `flask read-config`:

```
root::kali:MCS3_WEB_seminar_paper:# flask read-config                    0 [main]
{'APPLICATION_ROOT': '/',
 'DATABASE': '/root/projects/MCS/MCS3_WEB_seminar_paper/instance/flask-api.sqlite',
 'DEBUG': True,
 'ENV': 'development',
 'EXPLAIN_TEMPLATE_LOADING': False,
 'JSONIFY_MIMETYPE': 'application/json',
 'JSONIFY_PRETTYPRINT_REGULAR': False,
 'JSON_AS_ASCII': True,
 'JSON_SORT_KEYS': True,
 'MAX_CONTENT_LENGTH': None,
 'MAX_COOKIE_SIZE': 4093,
 'PERMANENT_SESSION_LIFETIME': datetime.timedelta(days=31),
 'PREFERRED_URL_SCHEME': 'http',
 'PRESERVE_CONTEXT_ON_EXCEPTION': None,
 'PROPAGATE_EXCEPTIONS': None,
 'SECRET_KEY': 'dev',
```

> abbreviated output from `flask read-config`

used libraries

- Flask
  - relatively unopinionated Python web microframework
  - there is a default templating engine but it can be changed
  - as a microframework it aims to be simple (no ORM) but extensible
- flask-wtf
  - integration between WTForms and Flask
  - provides CSRF (Cross-Site-Request-Forgery) protection
    - can be used without WTForms (as in this project)
- bcrypt
  - password salting and hashing
    - *security note:* bcrypt truncates passwords to 72 bytes
  - no longer used for this project (switched to flask-wtf)
- python-dotenv
  - for setting environment variables in Python from dotfiles
  - can be used standalone but also acts as Flask extension when imported into a Flask app:
    - automatically parses `.env` and `.flaskenv`
- sqlite3
  - SQLite is a file-based, self-contained SQL database engine
  - easy to use during prototyping

    - this is part of the Python standard library
- click
    - library for command line parsing
    - can be used standalone but also acts as Flask extension when imported into a Flask app:
    - used for extending Flask with the custom CLI commands:
        - init_db_cli
        - gen_secret_key
        - read_config
- SQLAlchemy
    - object relational mapper
    - supports a wide range of databases
    - not yet used in `main` branch
- psycopg[pool,binary] (versions 3) and psycopg2-binary (version 2)
    - Postgres adapter (for notify/listen events)
    - not yet used in `main` branch
    - planned alternative for sqlite
- black
    - highly opinionated Python code formatter
    - code style for this project: `black -l79 **/*.py`
        - all defaults except reduce maximum linewidth to 79
- ptpython, ipython
    - ptpython is used for debugging:
        - for proper code completion in the breakpoints REPL
    - ptpython requires the (nonstandard) IronPython runtime
- flasgger
    - Flask extension that extracts OpenAPI specification from Flask views
    - adds an API endpoint (/apidocs) that serves endpoint documentation
- Jinja2
    - Flask's default template engine
    - similar to Django's templating syntax:
        - control structures `{% %}`
        - variable values `{{ }}`
        - comments `{# #}`
    - *supports Unicode*

    - *automatic HTML escaping*
    - *optional Sandbox to evaluete untrusted code*
- Tailwind CSS
    - Tailwind provides utility-class based styling

---

other interesting libraries to consider:

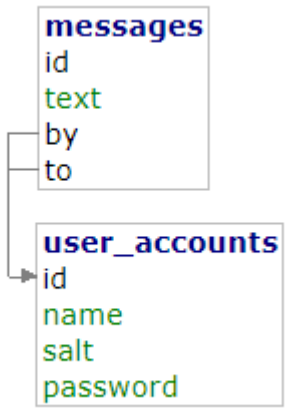
- flask-sqlalchemy SQLAlchemy extension for Flask
- flask-login for session handling

the chat app

The following sections cover a specific aspect of the chat app each:

database schema

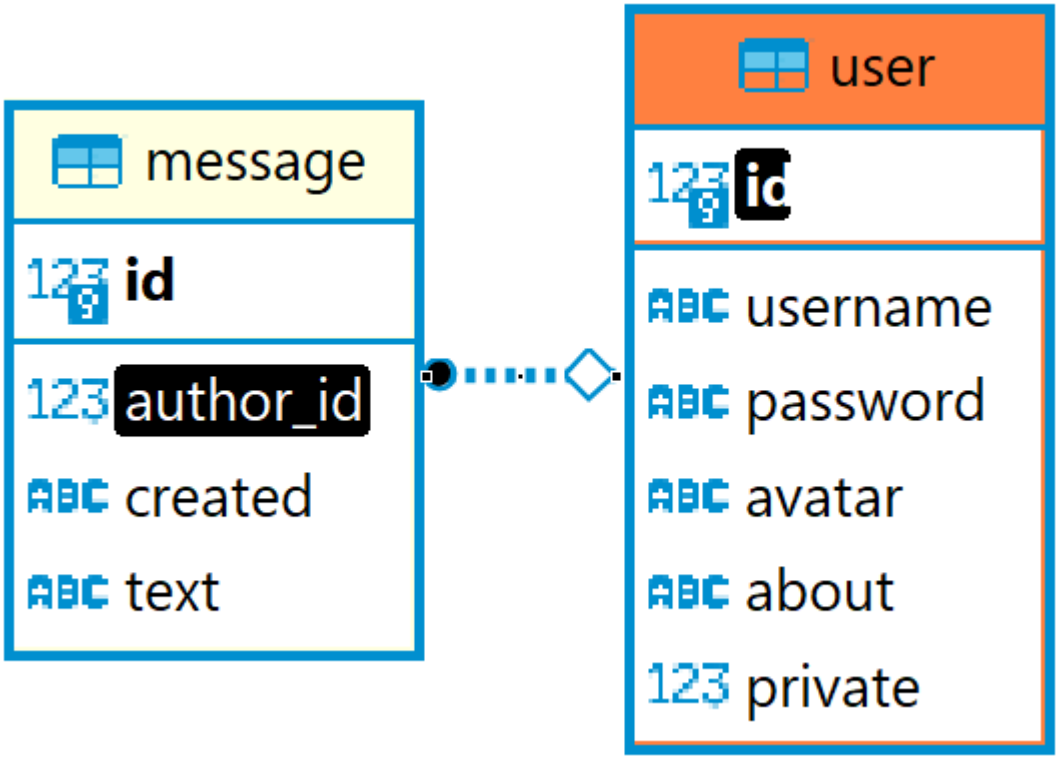
The old schema was based on a private messaging function. It also stored salt and password in seperate fields.



> old schema (PostGres)

I've since changed my mind and switched to an exclusively, global chat (planned to be a proximity-based chat in the game).

My plan is to start out with SQLite and maybe switch to PostGres later (SQLite is purely concurrent and might be too slow for larger apps but should be fine as a starting point).



> current schema (SQLite)

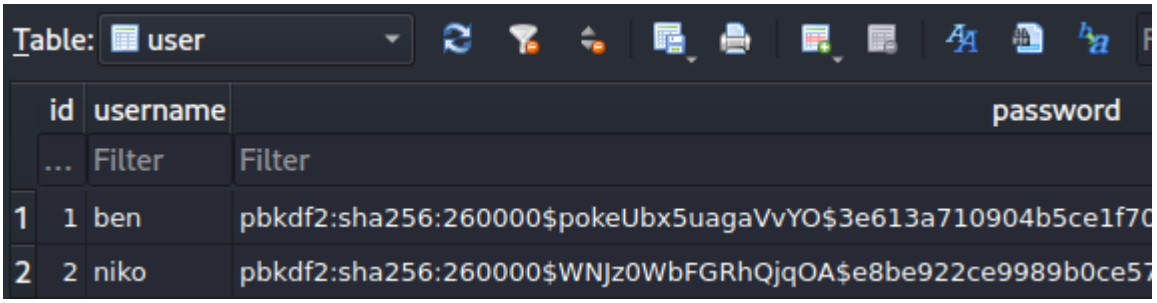
relevant sections from `schema.sql`:

```
CREATE TABLE user (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  username TEXT UNIQUE NOT NULL,
  password TEXT NOT NULL,
  avatar TEXT NOT NULL DEFAULT '0000',
  about TEXT DEFAULT '',
  private INTEGER NOT NULL DEFAULT 1
);

CREATE TABLE message (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  author_id INTEGER NOT NULL,
  created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  text TEXT NOT NULL,
  FOREIGN KEY (author_id) REFERENCES user (id)
);
```

Note the following:

- the `id` fields are auto-incrementing primary keys of type `INTEGER`
- the `message` table has an `author_id` field that is a foreign key pointing to the `id`

field of the `user` table
    - each message is owned by a user (the author)
- `created` is a timestamp that is populated by sqlite itself via `CURRENT_TIMESTAMP`
- `text` is of type `TEXT` and may not be null, empty messages don't make much sense
- `username` and `password` are of type text as well and can not be null
    - `username` is `UNIQUE`
    - **potentially dangerous bug**: `password` was unique for some time as well
        - This kind of bug might have been exploited by an attacker by creating accounts with common passwords (and unlikely usernames) and checking if a server error occurs. On a server error the attacker could have tried known usernames (from the chat) with the identified passwords.
        - Not in this instance though, since the stored passwords are hashed with a salt. (The `password` field actually stores 3 things: the algorithm, the hashed password and the salt itself)
- the `user` table has a field named `private` that stores wheter the user profile should be hidden
    - since sqlite does not have a Boolean type this is stored as `INTEGER` and cast to `bool` before usage
    - for **privacy reasons** this **defaults to** `True` when creating a new user
    - a user public profile shows: their profile picture, the customizable about text and their username
    - a private user profile only shows: `private user`
    - a non existing user profile also shows: `private user` to **avoid user enumeration** (auto-incrementing user ids would make this task trivial otherwise)
    - users have the option to set this option to `False` in their profile
- `avatar` stores an image id (from a list of options) and not the path to an image or the image itself
- **TODO**: add `coordinates` field to both the `message` and `user` table for proximity-based chatting



> user table via `sqlitebrowser`: showing the `id`, `username`, and `password` fields

The following is a short overview of available endpoints and a manual analysis of endpoints (specifically methods) that have the potential to change data:

list of `GET` endpoints

Grep sourcecode for `.route`:

- /
  - the landing page and main chat interface
- /user/<int:user_id>
  - user profile pages
- /auth/register
  - form to create a new account
- /auth/login
  - form to login
- /auth/logout
  - endpoint to logout
- /profile
  - displays own user profile (different from `/user/`)
- /profile/edit
  - edit user profile of logged-in user
- ~~/create~~
  - form to send a message
  - functionality integrated into the `/` endpoint
    - avoids duplication of code and improves maintainability

list of `POST` endpoints

Grep for `.methods`:

- `/`
  - post a message from `/` by pressing enter from the chat bar
- `/delete/<int:message_id>`
  - deletes a message (if logged-in user is the author)

    - send POST-request from `/` by clicking red cross
- `/profile/edit`
    - POST-request from same endpoint
- `/auth/register`
    - POST-request via form on same endpoint
- `/auth/login`
    - POST-request via form on same endpoint

list of endpoints that write to the db

Grep for `.commit`:

- `/` (via `message_post()`), POST
- `/profile/edit`, POST
- `/register`, POST
- `/delete/<int:message_id>`, POST

Cross reference of endpoints with functions that can change the database:

All endpoints that have the ability to change the database are POST. (As far as I know only `POST` and `GET` methods are allowed in forms, so I am limited to these for now, even if it is not quite conform with REST)

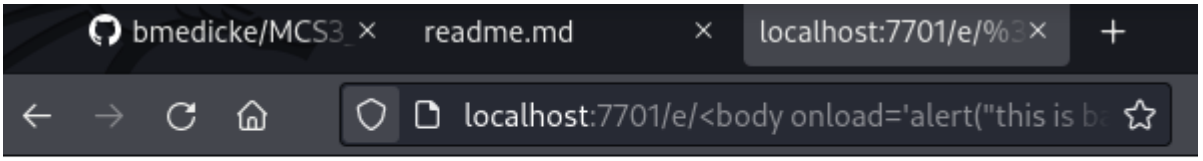
Other things to note when creating endpoints:

Security note: When creating an endpoint that extracts a variable from the url that is later used it has to be properly escaped!

Compare the following two Flask routes (inspired by a bug):

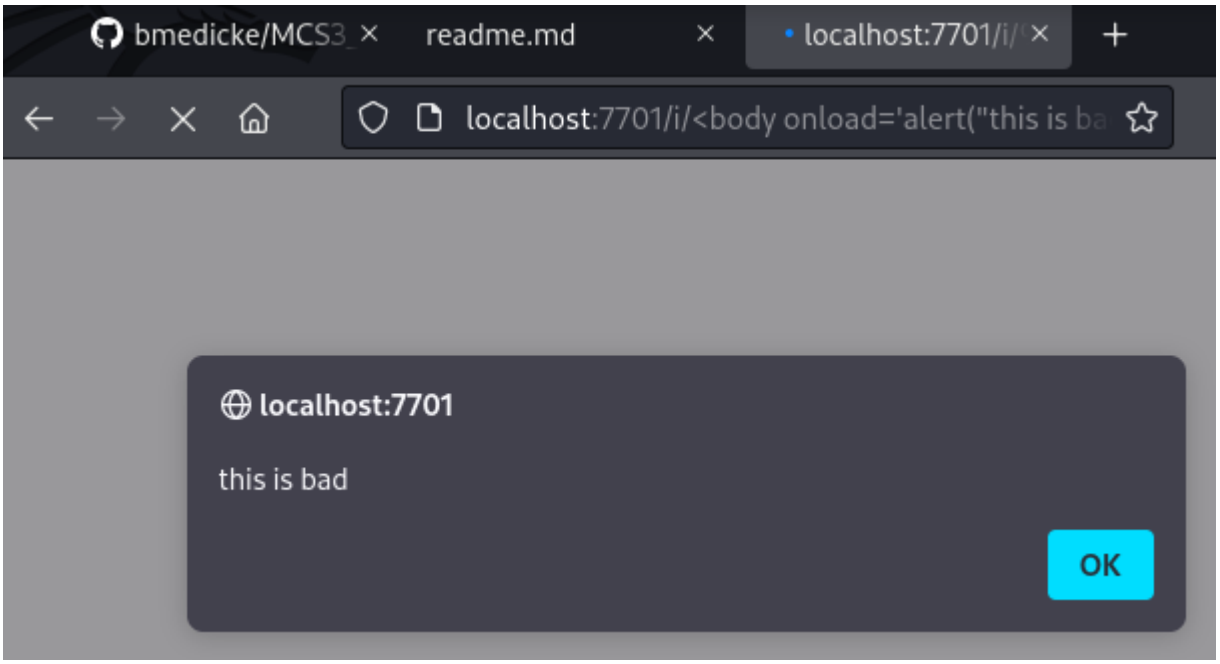
```
@app.route("/i/<unescaped>")
def injection(unescaped):
    """
    injection demo route

    localhost:7701/i/<body onload='alert("this is bad");'>
    """
    return f"{unescaped}"

@app.route("/e/<escaped>")
def no_injection(escaped):
    """
    injection-safe demo route
```

```
    localhost:7701/e/<body onload='alert("this is bad");'>
    """
    return f"{escape(escaped)}"
```



<body onload='alert("this is bad");'>

> route with proper escaping of user input



> route without proper input sanitization allows for JavaScript injection attacks

---

It is also possible to restrict the variable part of a route to a datatype, which can mitigate this kind of attack as well. See the `/user` route from profile.py for an example:

```
@blueprint.route("/user/<int:id>")
def user(id):
    """
    shows profile of user by id (if set to public)

    returns html
```

```
    """
    db = get_db()
    user = db.execute(
        """
        SELECT username, private, avatar, about
        FROM user
        WHERE id = ?
        """,
        (escape(id),),
    ).fetchone()

    return render_template("/profile/user.html", user=user)
```

Note the following:

- the user route will only trigger for integers in the variable part of the URL: `<int:id>`
- I have chosen to `escape()` the input nontheless in case the endpoint is edited in the future (or if there's a bug in the endpoint handling)
- SQL queries in this app use parameterized statements (the sqlite3 library does not support prepared statements)
- *security note:* when returning HTML (the default) user provided values must be `escape()`d to prevent injections
  - unsafe: `http://localhost:7701/i/<body onload='alert("this is bad");'>`
  - safe: `http://localhost:7701/u/<body onload='alert("this is bad");'>`
  - Jinja templates do this automatically (but you can explicitly disable this behaviour)

structure of the app

The following ASCII diagram shows the project structure:

```
├── docker
│   ├── ...
├── docker-compose.yml
├── env
│   ├── ...
├── flask_api
│   ├── auth.py
│   ├── database.py
│   ├── __init__.py
│   ├── message.py
│   ├── profile.py
│   ├── schema.sql
│   ├── static
│   │   ├── favicon.png
│   │   ├── profiles
```

```
│   │   │   ├── 0000.png
│   │   │   ├── 0001.png
│   │   │   ├── ...
│   │   └── style.css
│   └── templates
│       ├── auth
│       │   ├── login.html
│       │   └── register.html
│       ├── base.html
│       ├── index.html
│       ├── message
│       │   └── create.html
│       └── profile
│           ├── edit.html
│           ├── show.html
│           └── user.html
├── instance
│   └── flask-api.sqlite
├── readme.md
├── requirements.txt
├── .gitignore
├── .env
└── .flaskenv
```

- `docker` and `docker-compose.yml` are used for storing docker data (postgres) and the service file respectively
- `env` is the virtual environment that is used to store installed libraries (instead of the global store)
- `__init__.py` is the starting point of the Flask app and marks the encompassing folder as a Python module
    - this file imports the other scripts
- `schema.sql` is used by `flask init-db` to setup the database (see database schema)
- `static` files are served directly
- `templates` contains the served HTML/Jinja2 templates, `base.html` is inherited from by the other templates
- `instance` is created by `flask init-db` and contains the sqlite database (`flask-api.sqlite`)
- `.env` and `.flaskenv` are parsed by the app and used for environment variables

- *security note:* the `.flaskenv` file should not be commited if there are any secrets stored in it
    - you should use the `.env` file for secrets (which is in `.gitignore`)

dunder init

Abbreviated `__init__.py`, the starting point of the app:

```
from dotenv import load_dotenv  # automatically load .flaskenv
from flask import Flask
from flasgger import Swagger
from flask_wtf.csrf import CSRFProtect
import os

def create_app(test_config=None):
    """
    application factory function for the Flask app.

    returns a Flask object
    """

    # read secret key from env vars when deploying,
    # used for signing session cookies:
    SECRET_KEY = os.environ.get("SECRET_KEY", "dev")

    # name app after module name:
    app = Flask(__name__, instance_relative_config=True)

    app.config.from_mapping(
        SECRET_KEY=SECRET_KEY,
        DATABASE=os.path.join(app.instance_path, "flask-api.sqlite"),
    )

    # ...

    # views for routes are imported via blueprints:
    from . import auth
    from . import database
    from . import message
    from . import profile

    # register database functions with the app (includes cli command):
    database.init_app(app)

    # register authentication blueprint (register/login/logout):
    app.register_blueprint(auth.blueprint)

    # ...

    # require valid CSRF token for modifying requests:
    csrf = CSRFProtect()
    csrf.init_app(app)

    # generate apidocs:
    Swagger(app)
```

```
    return app
```

`__init__.py` defines a single function that in turn creates the Flask app (factory pattern). If no `SECRET_KEY` environment variable is it defaults to `dev`. After the configuration is done, the blueprints for endpoints are imported and registered with the app.

`CSRFProtect` is imported from the `flask_wtf` library (which is only used for the CSRF protection). Calling `csrf.init_app(app)` enables CSRF protection globally (for `POST`, `DELETE`, `PATCH` and `PUT` requests) by registering the Flask extension.

Since I use my own forms (as opposed to WTForms) a hidden `csrf_token` has to be added to each form. The value of the token can be be used in Jinja with `{{ csrf_token }}`, `flask_wtf` populates this variable automatically.

When receiving a form this value is expected, otherwise the request will be aborted with an error (400).

As an example here is part of the `index.html`:

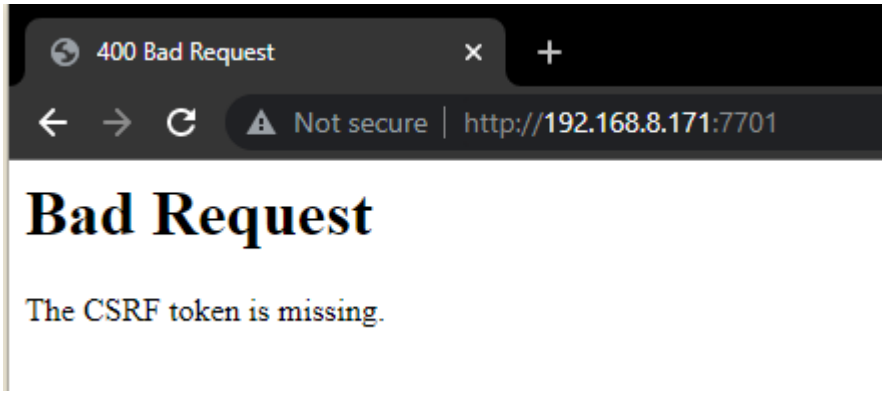
```
<!-- ... -->

<form method="post" accept-charset="utf-8">
    <input type="text" name="text"
        id="text" value="" placeholder="{{ "write your message here"
                        if g.user else "log in to start chatting" }}"
        class="chatbar w-full pl-2 mt-2 font-mono
                    {{ "bg-gray-50" if not g.user else "bg-sky-50" }}"
        required {{ "disabled" if not g.user }}>
    <input type="hidden" name="csrf_token" value="{{ csrf_token() }}">
</form>

<!-- ... -->
```

Populated `csrf_token` variable in a browser:

```
<form method="post" accept-charset="utf-8">
  <input type="text" name="text" id="text" value placeholder="write your message here"
  class="chatbar w-full pl-2 mt-2 font-mono
                    bg-sky-50" required>
  <input type="hidden" name="csrf_token" value="IjA5ZmVmOWUwZjkzMDllZjVmN2MyYjBjM2I2ZGY4
  YWYwYjNlNzAxMTAi.Ye7SzQ.f51wdTh-0DH_VsLav8vxKGac_fM"> == $0
</form>
```

> the same form rendered in a browser

And here a screenshot when creating a request without that token:



> missing CSRF token

security note: FlaskWTF uses CSRF opt-out for endpoints (all endpoints are protected, secure by default). It is possible to exempt endpoints from CSRF protection with the `@csrf.exempt` decorator. This is not recommended.

database

`database.py` file contains utility functions such as for creating and destroying connections.

The most commonly used function is `get_db()`. It connects to the SQLite database and stores the connection in the `g` object.

```
def get_db():
    """
    creates database connection or gets existing one from app context

    returns db attribute
    """
    # check for 'db' attribute in current app context:
    if "db" not in g:
        g.db = sqlite3.connect(
            current_app.config["DATABASE"],
            detect_types=sqlite3.PARSE_DECLTYPES,
        )
        g.db.row_factory = sqlite3.Row  # rows can be accessed like dicts.

    return g.db
```

The `g` ("global") object is always present for each request and each request has its own version (global in the sense of the request). This is big part of Flask's design philosophy (thread-local objects).

Session data is a good example of data that can be stored in this object. (See auth section)

It also reduces the number of variables you have to pass around and improves readability and maintainability (which in turn is an important part of writing secure code).

There are other thread-locals, among them `current_app`, which is used in this app as well.

auth

`auth.py` contains both helper functions and the `/auth` endpoints.

The following function is registered with the Flask app to run before each request. The `user_id` is read from the cookie and - if successful - user data is read from the database and stored in the `g` object (see previous section).

All SQL statements are parameterized.

```
@blueprint.before_app_request
def load_logged_in_user():
    """
    gets session data from cookie (if it exists)

    stores data in g object (for duration of request)
    """
    user_id = session.get("user_id")

    if user_id is None:
        g.user = None

    else:
        g.user = (
            get_db()
            .execute(
                """
                SELECT *
                FROM user
                WHERE id = ?
                """,
                (user_id,),
            )
            .fetchone()
        )
```

The `/login` endpoint produces the same error message no matter if a supplied password is wrong or the username does not exist to prevent leaking information to attackers:

```
@blueprint.route("/login", methods=("GET", "POST"))
def login():
    """
    allows users to log in

    returns html
    """
```

```
    if request.method == "POST":
        username = request.form["username"]
        password = request.form["password"]

        db = get_db()
        error = None

        user = db.execute(
            """
            SELECT *
            FROM user
            WHERE username = ?
            """,
            (username,),
        ).fetchone()

        # use same error message to not leak information:
        if user is None:
            error = "invalid credentials"
        elif not check_password_hash(user["password"], password):
            error = "invalid credentials"

# ...
```

The next function is a decorator. Applying this decorator to another function wraps that function with new functionality: If there is no user stored in the `g` object (no user logged in) it will redirect to the login page. This decorater is used to protect endpoints that should not be accessed anonymously.

```
def login_required(view):
    """
    decorator for views that require authentication

    returns view that redirects to login page if not logged in
    """

    @functools.wraps(view)
    def wrapped_view(**kwargs):
        if g.user is None:
            return redirect(url_for("auth.login"))

        return view(**kwargs)

    return wrapped_view
```

`validate_credentials()` is used to make sure that credentials supplied during the registration process are up to the configured standard. Personally I am annoyed by enforced special characters, mixed case and numerals. I prefer creating entropy by length (xkcd 936).

```
def validate_credentials(username, password, password_confirmation):
    """
    checks if password and username match requirements

    returns error or None
    """
    PASSWORD_MIN_LEN = int(os.environ.get("PASSWORD_MIN_LEN"))
    error = None

    if not username:
        error = "username can not be empty"

    if not password:
        error = "password can not be empty"

    if len(password) < PASSWORD_MIN_LEN:
        error = f"password too short ({PASSWORD_MIN_LEN} chars minimum)"

    if password != password_confirmation:
        error = "passwords do not match"

    return error
```

security note: for ease of development and testing the configured `PASSWORD_MIN_LEN` in `.flaskenv` is currently only 12. This should be adjusted upward.

profile

The `/profile` endpoint is one example that should only be accessible when a user is actually logged in.

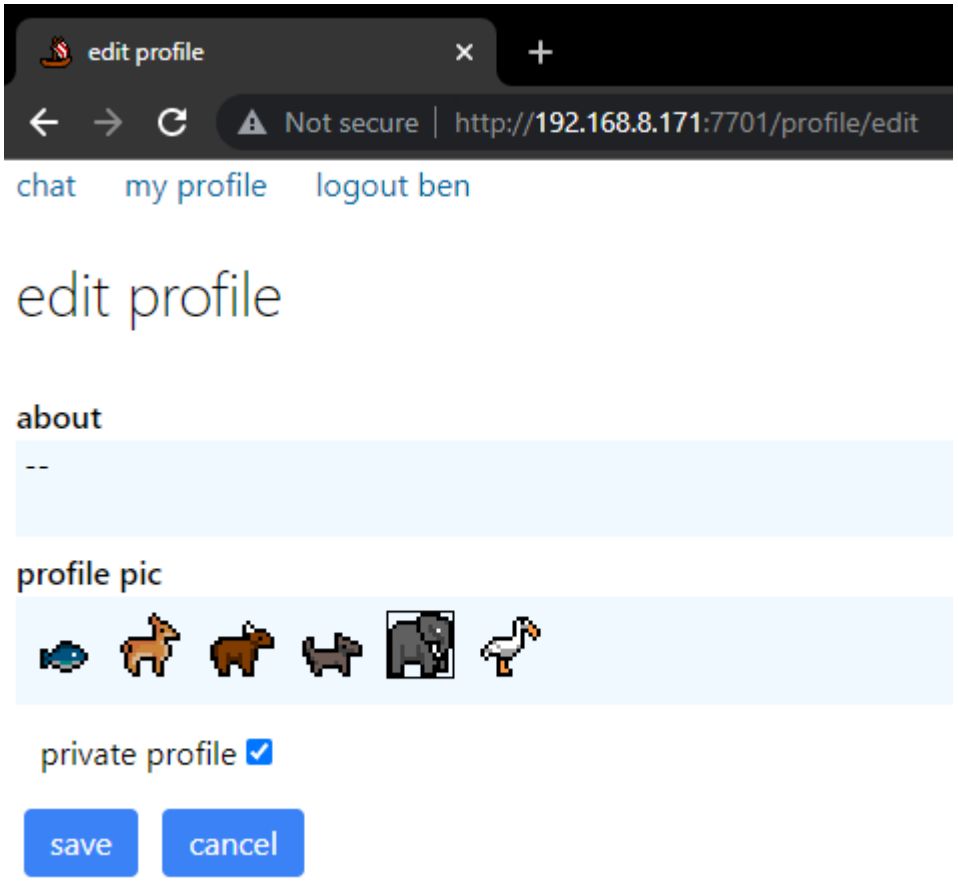
This is accomplished by applying the aforementioned `@login_required` decorator.

```
@blueprint.route("/profile")
@login_required
def profile():
    """
    displays (logged in) user profile

    returns html
    """
    return render_template("profile/show.html")
```

> Flask protects you against one of the most common security problems of modern web applications: cross-site scripting (XSS). Unless you deliberately mark insecure HTML as secure, Flask and the underlying Jinja2 template engine have you covered. But there are many more ways to cause security problems.

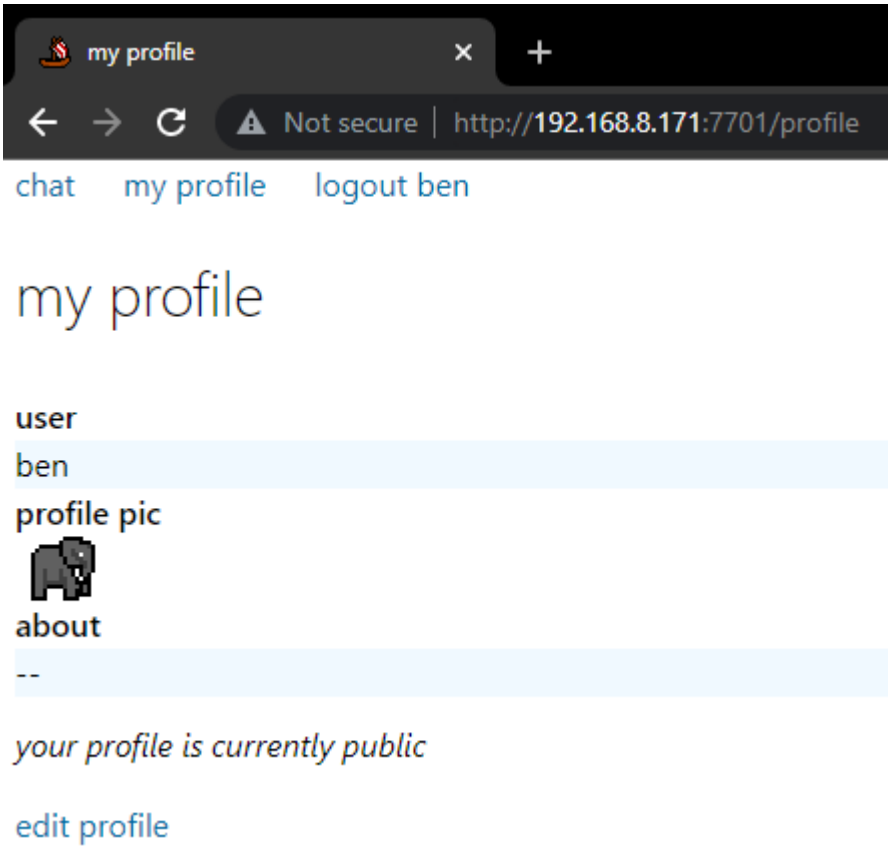
> **via:** https://flask.palletsprojects.com/en/1.0.x/advanced_foreword/

Since the profile page allows users to write a long string and save it to their profile (the about field/biography) it is a self-evident target for injection attacks (the same test was performed for the other POST endpoints).



> enumerating strings over the about field

Since Jinja escapes variable input and output by default none of the endpoints are vulnerable.



> not vulnerable to injection attacks

I have used SQL injections strings from PayloadsAllTheThings and Injecting SQLite database based application by Manish Kishan Tanwar.

```
def get_profile_pics():
    """
    gets names of available profile pics

    returns list of basenames without file ending
    """
    profiles_path = url_for("static", filename="profiles")
    files = glob(
        os.path.join(current_app.root_path + profiles_path + "/*.png")
    )
    profile_pics = list()

    for file in files:
        profile_pics.append(os.path.basename(file.strip(".png")))

    return profile_pics
```

The profile picture field is not a direct file upload and can only contain ids of pictures returned by the function above:

```
if avatar not in get_profile_pics():
```

```
    error = "invalid profile picture choice"
```

message

The heart of the application.

The following is the part of the index Jinja template that displays all messages:

```
{% for message in messages %}
    <li class="font-mono font-thin mt-1">
        <span class="ml-2">{{ message.created }}</span>
        <a href="/user/{{ message.author_id }}"
           class="">
            <img src="{{ profile_pic(message.avatar) }}"
                class="avatar inline ml-2"
                alt="avatar" />
            <span class="ml-2 font-bold text-sky-700 hover:text-gray-700">{{ m
        </a>
        <span>{{ message.text }}</span>
        {% if g.user.id == message.author_id %}
          <form class="inline" action="/delete/{{message.id}}" method="post" a
            <input type="hidden" name="csrf_token" value="{{ csrf_token() }}">
            <button type="submit" class="text-red-300">x</button>
          </form>
        {% endif %}
    </li>
{% endfor %}
```

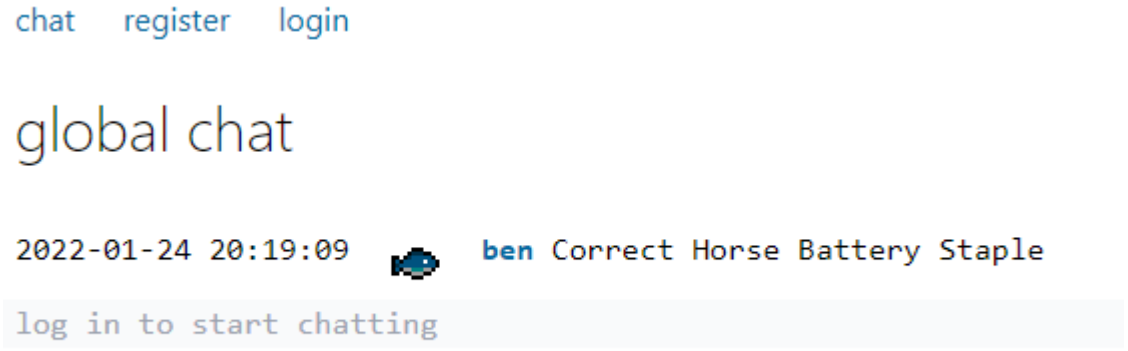
If the logged in user is the same as the author of a message, a red cross will be displayed that allows users to delete their own messages. Of course this is only client side protection. Here is the corresponding server side check in `message.py`:

```
@blueprint.route("/delete/<int:id>", methods=("POST",))
@login_required
def delete(id):
    """
    deletes a message (if it exists and is owned by user)

    redirects to index
    """
    error = None
    db = get_db()

    message_author = db.execute(
        """
        SELECT author_id
        FROM message
        WHERE id = ?
```

```
        """,
        (id,),
    ).fetchone()

    # invalid message id:
    if message_author is None:
        error = "denied"

    else:
        # logged in user did not author message:
        if g.user["id"] != message_author["author_id"]:
            error = "denied"

    if error:
        flash(error)
    else:
        db.execute(
            """
            DELETE FROM message
            WHERE id = ?
            """,
            (id,),
        )
        db.commit()

    return redirect(url_for("message.index"))
```

The variable endpoint is secured by limiting the datatype of `id` to an integer.

Same goes for the chat bar itself that is greyed out (client side) and protected by server side checks.



> disabled chat bar

analysis

Static analysis:

I have looked for a static analysis tool for Python and Flask. So far I have found:

- https://github.com/python-security/pyt
  - which no longer works with recent versions of Python
- https://github.com/FHPythonUtils/PyTaintX
  - a (more recently) maintained fork
  - which I could also not get to work consistently with Python 3.9

```
(env) root::kali:flask_api:# pytaintx *.py                                  0 [main]
Traceback (most recent call last):
  File "/root/projects/MCS/MCS3_WEB_seminar_paper/env/bin/pytaintx", line 8, in <mod
ule>
    sys.exit(main())
  File "/root/projects/MCS/MCS3_WEB_seminar_paper/env/lib/python3.9/site-packages/py
taintx/__main__.py", line 101, in main
    FrameworkAdaptor(cfg_list,
  File "/root/projects/MCS/MCS3_WEB_seminar_paper/env/lib/python3.9/site-packages/py
taintx/web_frameworks/framework_adaptor.py", line 33, in __init__
    self.run()
  File "/root/projects/MCS/MCS3_WEB_seminar_paper/env/lib/python3.9/site-packages/py
taintx/web_frameworks/framework_adaptor.py", line 91, in run
    function_cfgs.extend(self.find_route_functions_taint_args())
```

> exceptions with Python 3.9

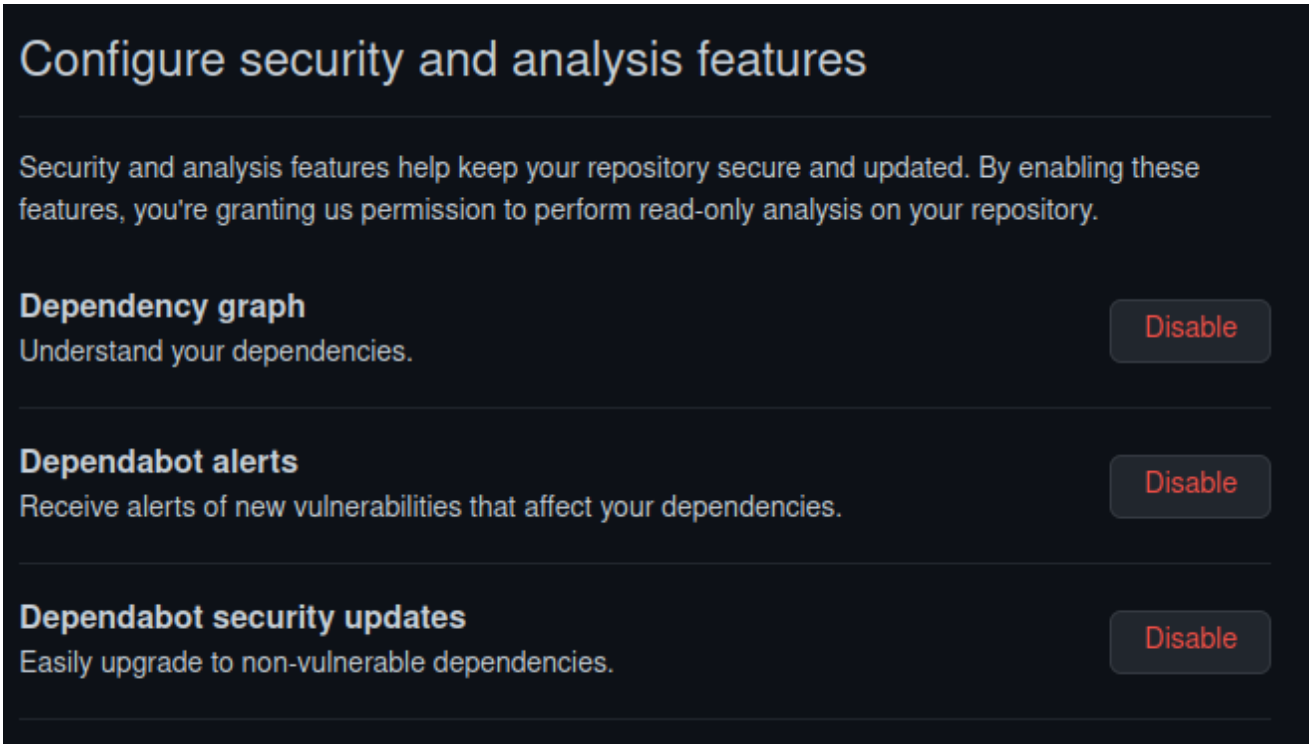
Lines of code:

```
root::kali:flask_api:# cloc *.py **/*.html
      14 text files.
      14 unique files.
       0 files ignored.

github.com/AlDanial/cloc v 1.90  T=0.01 s (977.1 files/s, 60231.7 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           cod
-------------------------------------------------------------------------------
Python                           6            123            156            28
HTML                             8             54              0            24
-------------------------------------------------------------------------------
SUM:                            14            177            156            53
-------------------------------------------------------------------------------
```

Additionally the source code is automatically styled with `black` and linted with `pyflakes`.

workflow

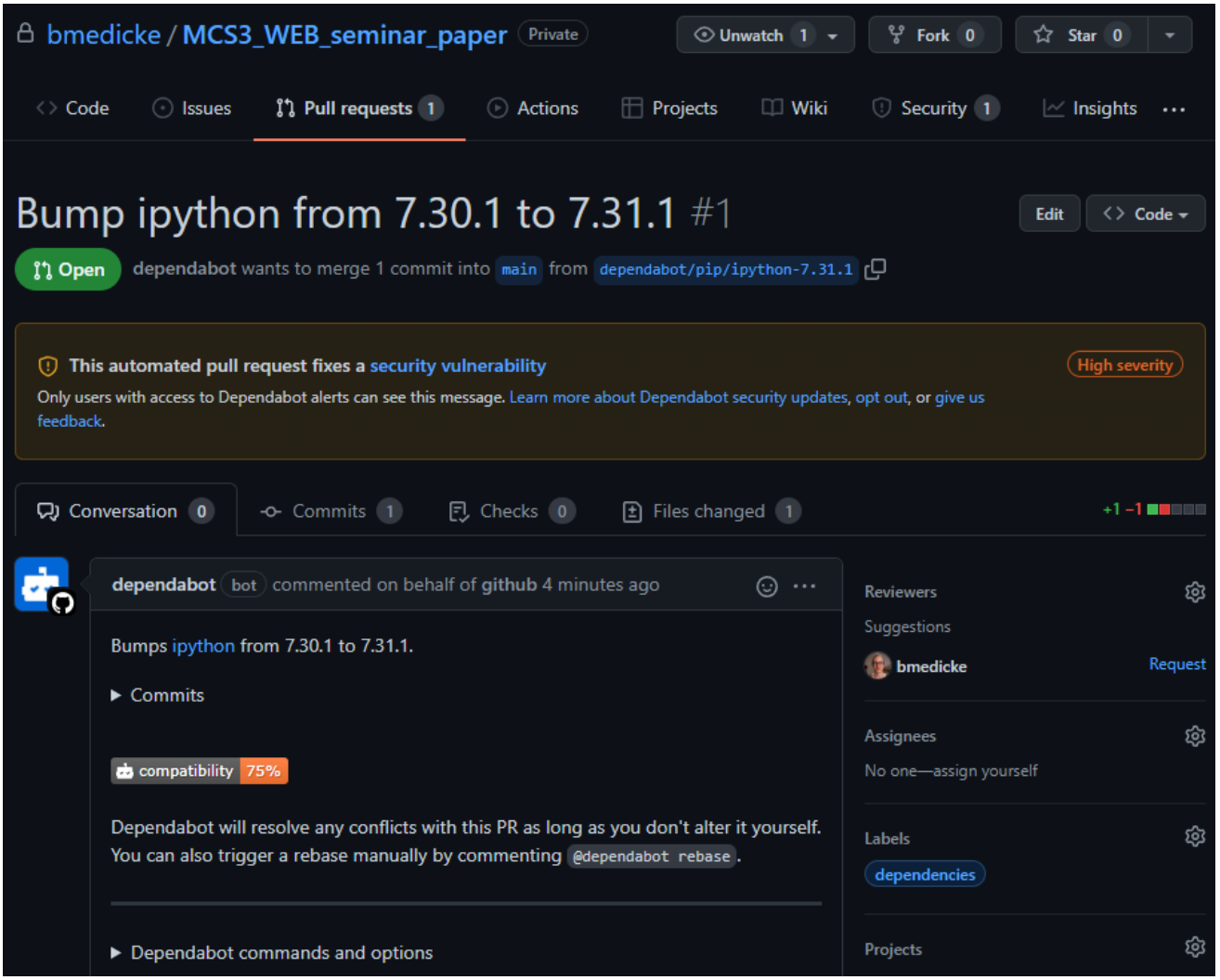
Next to local pre/post-commit hooks there are also serverside solutions. GitHub provides several security and analysis features that should be enabled:



> GitHub Security Checks

This particular dependency is not critical as IronPython is only used in the debugging workflow but the alerts work (and are near instant after activating the above option):

The Dependabot security updates are a great alternative to the Node-only `npm audit`. The `depandabot` bot even sends pull requests with updates:



> Pull request to update dependency

Interacting with the GitHub servers via `git` also provides warnings:

```
(env) root::kali:MCS3_WEB_seminar_paper:# gp                                0 [main]
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 566 bytes | 566.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
remote:
remote: GitHub found 1 vulnerability on bmedicke/MCS3_WEB_seminar_paper's default br
anch (1 high). To find out more, visit:
remote:      https://github.com/bmedicke/MCS3_WEB_seminar_paper/security/dependabot/
requirements.txt/ipython/open
```

> CLI notification from the GitHub server when pushing

deployment

The following points should be considered when deploying the app to production:

- [ ] werkzeug (the shipped WSGI server) is to be used only during development, not

production
    - gunicorn can be used instead, among others
    - https://werkzeug.palletsprojects.com/en/2.0.x/serving/
    - https://flask.palletsprojects.com/en/2.0.x/tutorial/deploy/
- [ ] adjusting the `app.config` options
    - [ ] `SECRET_KEY` https://flask.palletsprojects.com/en/2.0.x/config/#SECRET_KEY
        - set via env vars or `.env`
        - used for signing session cookies
    - [ ] `PERMANENT_SESSION_LIFETIME`
        - the default is 31 days
    - [ ] `SESSION_COOKIE_SECURE` and `SESSION_COOKIE_SAMESITE` are `False` by default!
    - [ ] adjust `WTF_*` options if required

```
{
    'WTF_CSRF_ENABLED': True,
    'WTF_CSRF_CHECK_DEFAULT': True,
    'WTF_CSRF_METHODS': {'POST', 'DELETE', 'PATCH', 'PUT'},
    'WTF_CSRF_FIELD_NAME': 'csrf_token',
    'WTF_CSRF_HEADERS': ['X-CSRFToken', 'X-CSRF-Token'],
    'WTF_CSRF_TIME_LIMIT': 3600,
    'WTF_CSRF_SSL_STRICT': True
}
```

- [ ] the app uses `from werkzeug.security import check_password_hash, generate_password_hash`
    - uses default algorithm `pbkdf2:sha256` (https://en.wikipedia.org/wiki/PBKDF2)
    - uses default salt length of `16`
    - https://werkzeug.palletsprojects.com/en/2.0.x/utils/#werkzeug.security.generate_password_hash
- [ ] additional git hooks and GitHub webhooks
- [ ] semgrep
- [ ] set security HTTP header
- [ ] set session cookie flags
- [ ] consider JWT
    - more useful for microservices
    - harder to get correct than sessions
- [ ] consider two-factor authentification

useful resources

- https://github.blog/2021-12-06-write-more-secure-code-owasp-top-10-proactive-controls/