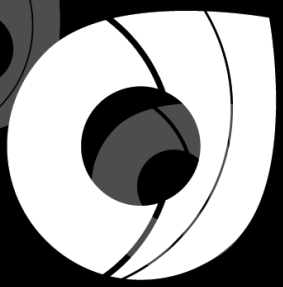
Corndel DevOps Engineering Programme

in association with Softwire

Module 10: Project Exercise



Corndel Digital.

DB31

Module 10

Project Exercise Brief

In this exercise we will add authentication and authorisation to the app. This will avoid all our to-dos being publicly accessible and restrict who can add new ones.

To achieve this we will use Github authentication using an OAuth flow.

Part 1: Adding authentication

Step 1: Preparation

Before writing the code we need register an app for Github and install some helpful libraries.

Register an app on Github

Follow the Github documentation to create your oauth app.

For the homepage URL field enter the address for accessing the website locally. For the callback add a particular path to this URL for example `/login/callback`.

> *You will need both a client-id and client-secret for your `.env` file. The client-secret once generated will only be shown once, so take a note of it to avoid needing to regenerate one later.*

Installing oauthlib and flask-login

There are multiple libraries that can be used to help implement authentication. For this exercise we will use oauthlib and flask-login. You can add this dependency to your project with `poetry` (or if your project uses `pip` to manage dependencies, you can use `pip install` instead of `poetry add`):

```
poetry add oauthlib flask-login
```

Step 2: Adding a redirect to Github

First you need to implement the code to trigger a redirect to Github to start the authentication flow. To do this we will need to use the LoginManager from flask-login.

flask-login LoginManager

There are two handlers needed for flask-login to work successfully, for the time being the user_loader can be left returning `None`.

```
login_manager = LoginManager()

@login_manager.unauthorized_handler
def unauthenticated():
    pass # Add logic to redirect to the
         # Github OAuth flow when unauthenticated

@login_manager.user_loader
def load_user(user_id):
    return None

login_manager.init_app(app)
```

Now you should be able to add a `@login_required` decorator to any route, then when visited your `unauthenticated` method will be called.

Implementing the redirect code

Now flask-login is integrated you can implement the code to perform the redirect. Github has documentation available for authorizing oauth apps.

> *Hint: You may want to take a look at the prepare_request_uri method to help with this.*

Step 3: Implementing the callback route

After completing Step 2 and checking you are redirected to Github to authorize using the app the next step is to implement the callback route.

- Parse the code that's given to us by the request GitHub sends to us
- Use that code to obtain an Access Key (hint - take a look at the prepare_token_request and parse_request_body_response methods)
- Use the Access Key to call the user-info endpoint to find details of our user (hint - take a look at the add_token method)
- Construct a new user object and create a new session for them using the login_user method. For the time being the user object only needs the user_id and should extend UserMixin.

For now do not store the users in the database this is a stretch goal in Part 3. Finally, you should update the user_loader in Step 2 to create the user object from the user_id in the same way as the login callback.

> *You should ensure to set the secret_key on the flask app. If locally not using HTTPS include `OAUTHLIB_INSECURE_TRANSPORT=1` in the environment file. The return value from `parse_request_body_response` is not needed, the token is stored in the `client` and is automatically used in later calls to `add_token`.*

Step 4: Final steps

You should now check all the previous endpoints require authentication and your changes haven't broken anything.

> *Setting 'LOGIN_DISABLED' to 'True' in the app config will prevent the 'login_required' decorator from redirecting in tests.*

Part 2: Adding authorisation

Now that we have prevented everyone from being able to access the app (although they only need a Github account!) we should only allow some users to alter to-dos.

Introduce two different roles:

- `reader` - These users can view to-dos but not change or create new ones
- `writer` - These users can also change existing to-dos or create new ones

Step 1: Add mapping to determine role from user

As with Part 1 we do not need to store the role for a user in the database, instead we can hardcode a mapping from a user_id to their role. (This is not best practice, just a shortcut for a training exercise.)

You should be careful to ensure that the same user will consistently have the same role.

Step 2: Add checks to endpoints for role

For each endpoint needing the `writer` role you should check the user has the required permission and reject the request if not. You can access the user object for the logged in user via current_user.

> *You may want to use a decorator to avoid duplicating logic for role checks.*

Step 3: Alter frontend based on role

After Step 2 the user should be rejected from doing any action they do not have permission to. This leads to a poor user experience for readers who are shown options they cannot use.

Update the frontend to avoid showing options to create or alter existing to-dos for users who have the `reader` role.

Stretch Goals

(Stretch goal) Part 3: Add admin role and management of roles

In both Part 1 and Part 2 we avoided storing the users in our database, while this requires less code it prevents us to easily adjust roles for users of the app.

Now we can resolve that and add an `admin` role for a user who can additionally alter the permissions of other users.

Step 1: Store users and their roles in the database

Using MongoDB as from the previous exercise store the authenticated users and their role in the database. The first user should be an `admin` and subsequent users should have the `reader` role.

You will need to update the callback endpoint to store any new users in the database as well as the user loader to load the user from the database.

Step 2: Add endpoints for admins to alter roles

Admins will need to use new endpoints to be able to alter the role of other users. These endpoints should have correct authorisation in place.

Step 3: Interface

To have a complete app the admins will need to be able to alter the roles using a web interface rather than knowing what HTTP requests to make.

Implement a page for admins to manager users, it will need:

- The ability to see all users and their current role
- To change the role of a particular user to one of the other roles (including `admin`)