
Flask with Python

— Our first web app —

What's Flask?

- Flask is what's known as a WSGI framework. Mercifully pronounced "whiskey,"
- This stands for Web Server Gateway Interface.
- Essentially, this is a way for web servers to pass requests to web applications or frameworks

Preparations Needed:

1. Installing Flask Packages

- **Pip install flask**
- **pip install flask-login** → **Flask-Login provides user session management for Flask.** It handles the common tasks of logging in, logging out, and remembering your users' sessions over extended periods of time.
- **Pip install flask-sqlalchemy** → **Flask-SQLAlchemy is an extension for Flask that adds support for SQLAlchemy to your application.** It aims to simplify using SQLAlchemy with Flask by providing useful defaults and extra helpers that make it easier to accomplish common tasks.

SQLAlchemy

- SQLAlchemy is a library that **facilitates the communication between Python programs and databases.**
- Most of the times, this library is used as an Object Relational Mapper (ORM) tool that translates **Python classes to tables** on relational databases and automatically converts **function calls to SQL statements**.

Files to be created within your project:

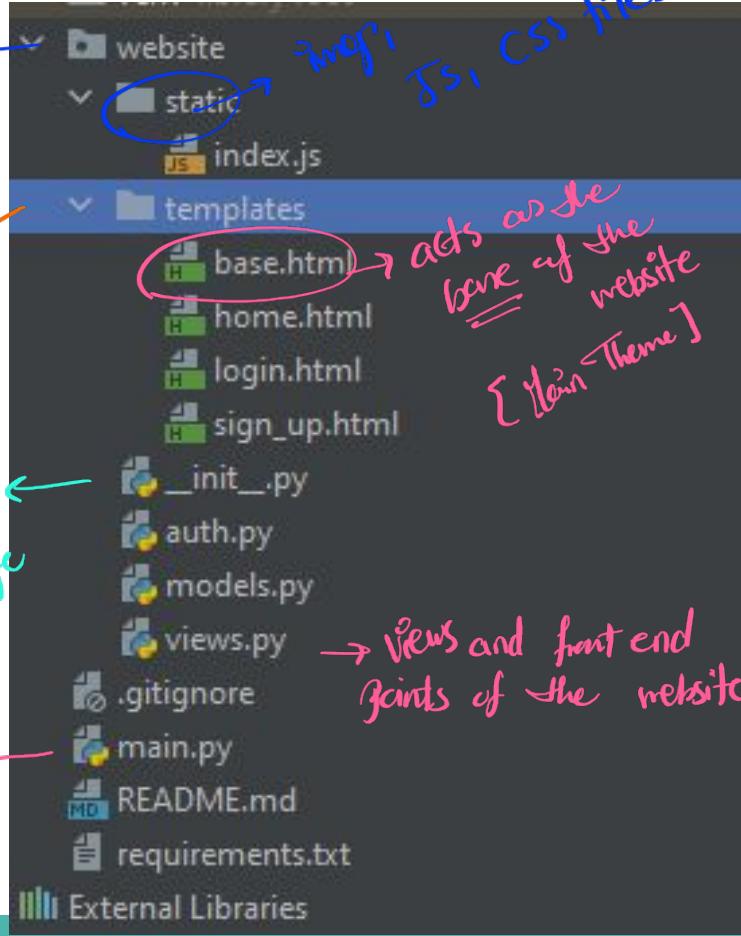
There's a templating function that's called Jinja allows us to use HTML inside our Python files

will store all the code for the website

when you run an HTML file, you call it a template

To make this Python file a Python package

will start the web server



__init__.py

```
1 from flask import Flask
2
3 def create_app():
4     app = Flask(__name__)
5     app.config['SECRET_KEY'] = 'hjshjhdjah kjshkjdhjs'
6
7     return app
```

↙ function

This is how you
initialize a
Python file

↗ app = Flask(__name__)

↗ app.config['SECRET_KEY'] = 'hjshjhdjah kjshkjdhjs'

↗ return app

→ To encrypt the cookies and
session data

Main.py

✓ Python package

```
1 from website import create_app  
2  
3 app = create_app()  
4  
5 if __name__ == '__main__':  
6     app.run(debug=True)
```

only if we run this file, we will run the server, ~~not~~ if we just imported it.

every time we change the Python code, it will automatically run the server

What is a blueprint?

A blueprint defines a collection of views, templates, static files and other elements that can be applied to an application. For example, let's imagine that we have a blueprint for an admin panel. This blueprint would define the views for routes like `/admin/login` and `/admin/dashboard`. It may also include the templates and static files that will be served on those routes. We can then use this blueprint to add an admin panel to our app, be it a social network for astronauts or a CRM for rocket salesmen.

Views.py

where we gonna save the standard routes of
our web server [anything that's not authentication
will go to that file]

te > views.py

```
1 from flask import Blueprint  
2  
3 views = Blueprint('views', __name__)
```

4 ↑ ↴

5 They don't have to be
same but it's recommended
to keep it simple

Auth.py

website > auth.py

```
1 from flask import Blueprint  
2  
3 auth = Blueprint('auth', __name__)  
4 |
```

→
same thing

↳ Now I say that i have a
different URLs

Views.py

site > views.py

```
1 from flask import Blueprint  
2  
3 views = Blueprint('views', __name__)  
4  
5 @views.route('/')  
6 def home(): This function will run whenever you go to  
7                                     that URL
```

The code shows the creation of a Blueprint named 'views' and its association with the root URL '/'. A handwritten note explains the meaning of each part: 'views' is circled and labeled 'the name of the blueprint'; '@views.route('/'') is labeled 'URL'; and the 'def home()' function is labeled 'This function will run whenever you go to that URL'.

__init__.py

Now, we have some blueprints, we need to define them into our initialization file to include them inside our packages

```
ite > _init_.py
1 from flask import Flask
2
3
4 def create_app():
5     app = Flask(__name__)
6     app.config['SECRET_KEY'] = 'hjshjhdjah kjshkjhds'
7
8     from .views import views
9     from .auth import auth
10
11
12
13     return app
14
```



we have them imported but not yet registered

Registering the blueprints on __init__.py file

```
def create_app():
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'hjshjhhdjah kjshkjhjs'

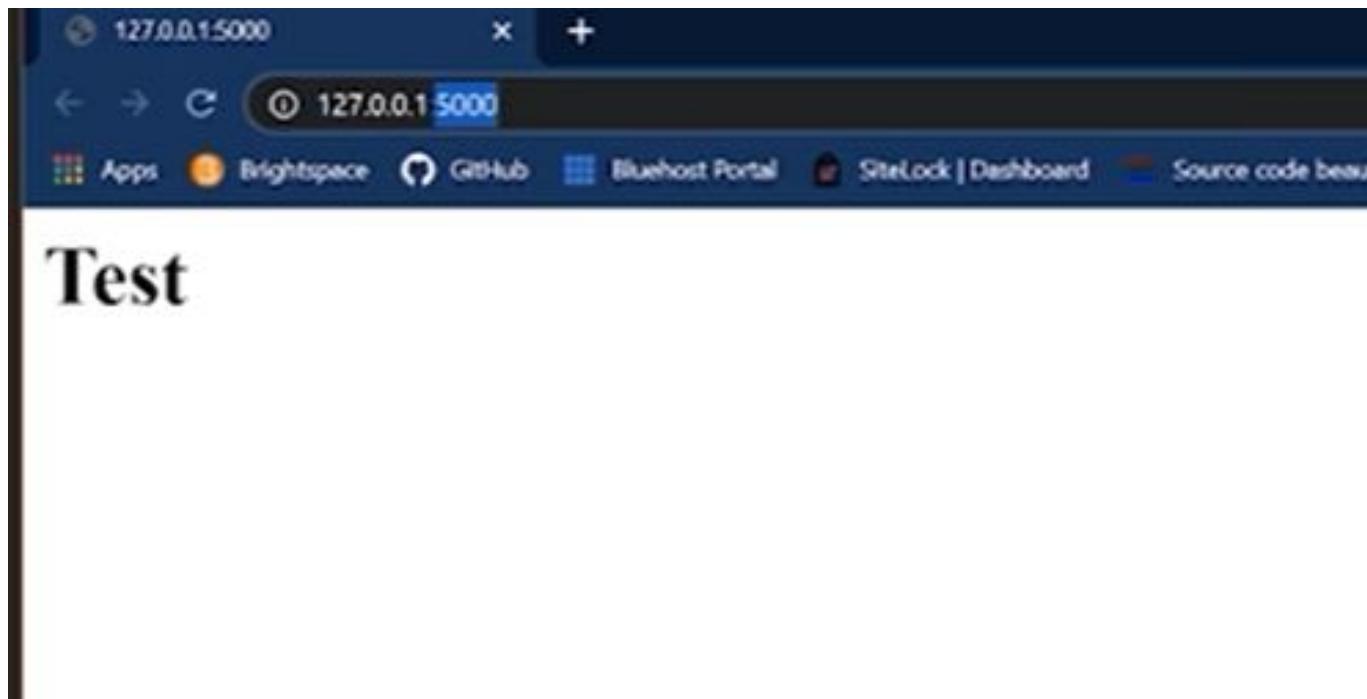
    from .views import views
    from .auth import auth

    app.register_blueprint(views, url_prefix='/')
    app.register_blueprint(auth, url_prefix='/')

    return app
```

the start of the
URL

Run and Test your app



Adding your sign in and up functions on auth.py

```
@auth.route('/login')
def login():
    return "<p>Login</p>" ↴ Just for  
Testing purposes
```

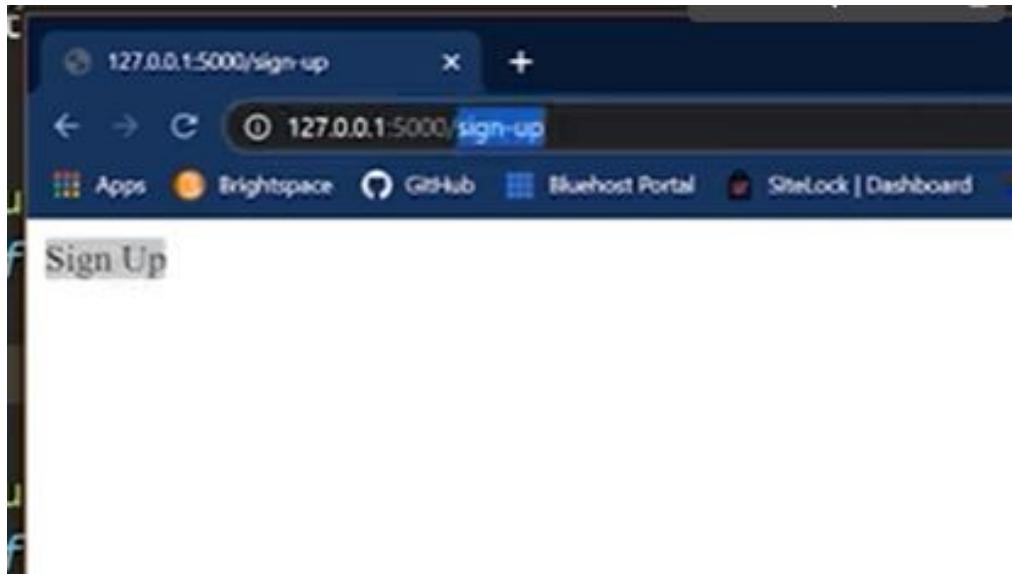


```
@auth.route('/logout')
def logout():
    return "<p>logout</p>" ↴
```



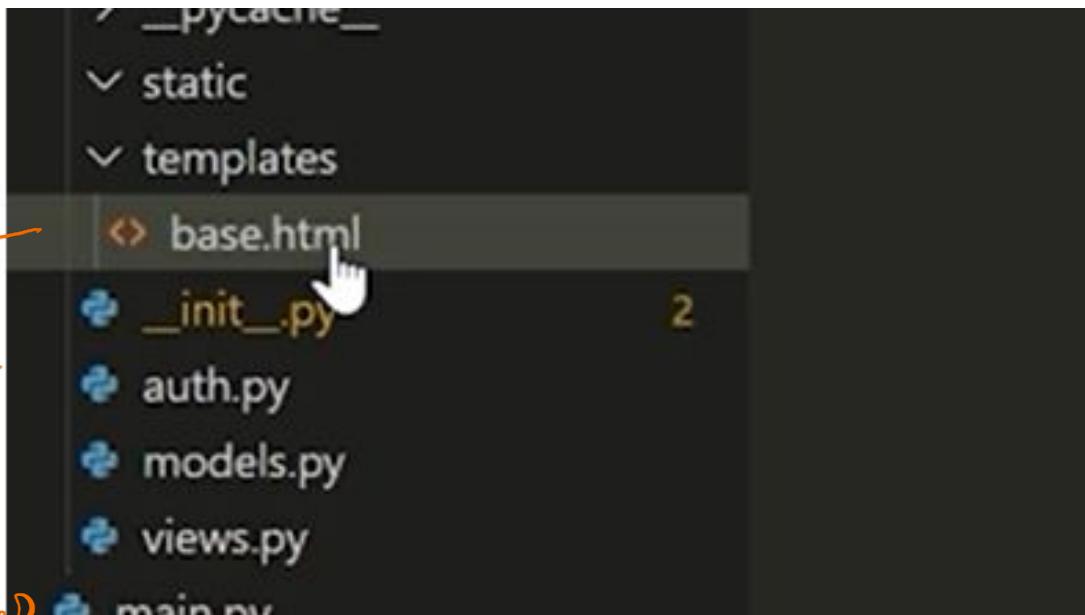
```
@auth.route('/sign-up')
def sign_up():
    return "<p>Sign Up</p>" ↴
```

Test your three routes



Create an HTML file inside your templates folder

it's like the theme of the website that will have the header - footer etc we will override it with the other specific templates



Scan this code for the HTML Files

- There are some files that we're going to import since we're using Bootstrap packages that will help us with some animations to make our app prettier



With Jinja,

The curly brackets and the percentage sign are used to surround any **python** code within your HTML File

```
<title>[% %]</title>
</head>
<body></body>
```

you can
add in
if statement's
for loops

```
    />  
  
    <title>{% block title %}Home{% endblock %}</title>  
  </head>
```

Children
will inherit
this block
but they can
override it
to change
the title

Add in your Nav Bar inside base.html file

```
<title>{% block title %}Home{% endblock %}</title>
</head>
<body>
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
    </nav>
```

Bootstrap
classe's !
Check out
the Bootstrap
documentation if
you want

Add in a button on your nav bar in order to expand the menu items if the screen is too small:

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
  <button
    class="navbar-toggler"
    type="button"
    data-toggle="collapse"
    data-target="#navbar"
  >
```

Adding our buttons inside our nav bar:

```
<span class="navbar-toggler-icon"></span>
</button>
<div class="collapse navbar-collapse" id="navbar">
  <div class="navbar-nav">
    <a class="nav-item nav-link" id="login" href="/login">Login</a>
    <a class="nav-item nav-link" id="signUp" href="/sign-up">Sign Up</a>
    <a class="nav-item nav-link" id="logout" href="/logout">Logout</a>
    <a class="nav-item nav-link" id="home" href="/">Home</a>
  </div>
```

Extending the base temp into the home.html file

So that home.HTML inherit base HTML

```
bsite > templates > home.html  
1  {% extends "base.html" %} {% block title %}Changed{% endblock %}  
2
```

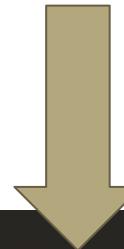
That was
home

Rendering our template on views.py file

```
Website > views.py
1 from flask import Blueprint, render_template
2
3 views = Blueprint('views', __name__)
4
5
6 @views.route('/')
7 def home():
8     return render_template("home.html")
9
```

The diagram illustrates the process of rendering a template. A large blue arrow points downwards from the title 'Rendering our template on views.py file' to the code editor. Inside the code editor, a yellow curly brace groups the 'Blueprint' and 'render_template' imports. A yellow bracket on the right side of the code editor points to the text 'To run HTML files'. A pink oval highlights the 'return render_template("home.html")' line, and a pink arrow points upwards from this line towards the bottom of the code editor.

Rendering our templates on auth.py



auth.py

```
from flask import Blueprint, render_template
```

```
auth = Blueprint('auth', __name__)
```

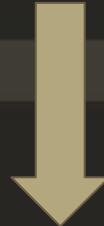
Add to your auth.py the route to your templates:

```
@auth.route('/login')
def login():
    return render_template("login.html")
```

✓ Try to add it to your sign up

website > auth.py

```
3     auth = Blueprint('auth', __name__)
4
5     |
6     @auth.route('/login')
7     def login():
8         return render_template("login.html")
9
10
11    @auth.route('/logout')
12    def logout():
13        return "<p>Logout</p>"
14
15
16    @auth.route('/sign-up')
17    def sign_up():
18        return render_template("sign_up.html")
```



Working on your sign up.html

? log^{on} · HTML

e > templates > sign_up.html > Form > div.form-group > input#email.form-control

```
1  {% extends "base.html" %} {% block title %}Sign Up{% endblock %} {% block content %}
```

content %}

```
3  <form method="POST">
```

```
4      <h3 align="center">Sign Up</h3>
```

```
5      <div class="form-group">
```

```
6          <label for="email">Email Address</label>
```

```
7          <input
```

```
8              type="email"
```

```
9                  class="form-control"
```

```
.0                  id="email"
```

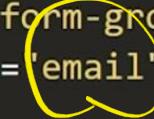
```
.1                  name="email"
```

```
.2                  placeholder="Enter email"
```

```
.3          />
```

```
.4      </div>
```

```
.5  </form>
```



Try to do
it yourself
for the rest of
the fields:
- first Name
- Last Name
- Part 1
- Part 2
Then do the
login form

Add in the methods that our functions may accept inside our auth.py file:

website > auth.py

```
1  from flask import Blueprint, render_template
2
3  auth = Blueprint('auth', __name__)
4
5  ↓ what type of request will it use?
6  @auth.route('/login', methods=['GET', 'POST'])
7  def login():
8      return render_template("login.html", boolean=True)
9
10
11 @auth.route('/logout')
12 def logout():
13     return "<p>Logout</p>"
14
15
16 @auth.route('/sign-up')
```

flask.blueprints.Blueprint.
**options)

Returns → FunctionType

HTTP requests

what type
at request
are sent to
your website

Change to
DB or
the website



Add it to your sign up method as well:

```
1
2
3
4
5
6 @auth.route('/login', methods=['GET', 'POST'])
7 def login():
8     return render_template("login.html", boolean=True)
9
10
11 @auth.route('/logout')
12 def logout():
13     return "<p>Logout</p>"
14
15
16 @auth.route('/sign-up', methods=['GET', 'POST'])
17 def sign_up():
18     return render_template("sign_up.html")
```



flask.blueprints.Blueprint
**options)
Returns → FunctionType

On our auth.py file, we need to import the request

```
te > auth.py
1 from flask import Blueprint, render_template, request
2
```



Sign up function on your auth. Py file

The screenshot shows a code editor with a dark theme. The file being edited is `auth.py`, which contains the following Python code:

```
14
15
16 @auth.route('/sign-up', methods=['GET', 'POST'])
17 def sign_up():
18     if request.method == 'POST': if the received method is post
19         email = request.form.get('email') get the requests
20         firstName = request.form.get('firstName')
21         password1 = request.form.get('password1')
22         password2 = request.form.get('password2')
23
24     return render_template("sign_up.html")
25
```

Handwritten yellow annotations are present in the code:

- An annotation for the `if request.method == 'POST':` line reads: "if the received method is post". It has arrows pointing from "if" to "request.method" and from "post" to "request.method".
- An annotation for the `return render_template("sign_up.html")` line reads: "get the requests". It has an arrow pointing from "get" to "render_template".

What if the user entered a wrong pass or didn't add in an email? You have to have a warning msg [Flashing]

```
auth.py
from flask import Blueprint, render_template, request, flash
auth = Blueprint('auth', __name__)


```

```
15
16 @auth.route('/sign-up', methods=['GET', 'POST'])
17 def sign_up():
18     if request.method == 'POST':
19         email = request.form.get('email')
20         firstName = request.form.get('firstName')
21         password1 = request.form.get('password1')
22         password2 = request.form.get('password2')
23
24     if len(email) < 4:
25         flash('Email must be greater than 3 characters.', category='error')
26     elif len(firstName) < 2:
27         flash('First name must be greater than 1 characters.', category='error')
28     elif password1 != password2:
29         flash('Passwords don\'t match.', category='error')
30     elif len(password1) < 7:
31         flash('Password must be at least 7 characters.', category='error')
32     else:
33         flash('Account created!', category='success')
34     pass
35
```

Now your msgs are flashed but they're not displayed on the front-end of your app. So we need to go to our base.html file to display them:

```
28         <span class="navbar-toggler-icon"></span>
29     </button>
30     <div class="collapse navbar-collapse" id="navbar">
31         <div class="navbar-nav">
32             <a class="nav-item nav-link" id="login" href="/login">Login</a>
33             <a class="nav-item nav-link" id="signUp" href="/sign-up">Sign Up</a>
34             <a class="nav-item nav-link" href="#">Logout</a>
35             <a class="nav-item nav-link" href="#">Profile</a>
36         </div>
37     </div>
38 </nav>
39
40     {% with messages = get_flashed_messages(with_categories=true) %}
41
42     {% endwith %}
```

That is done below the nav-bar, the with messages is a function that's built in in flask that allows us to receive those msgs

To error | success

The msgs are now received but they're not yet displayed.

```
{% with messages = get_flashed_messages(with_categories=true) %} {% if
messages %} {% for category, message in messages %} {% if category ==
'error' %}
<div class="alert alert-danger alter-dismissible fade show" role="alert">
    {{ message }}
    <button type="button" class="close" data-dismiss="alert">
        <span aria-hidden="true">x</span>
    </button>
</div>
{% else %}
<div class="alert alert-success alter-dismissible fade show" role="alert">
    {{ message }}
    <button type="button" class="close" data-dismiss="alert">
        <span aria-hidden="true">x</span>
    </button>
</div>
{% endif %} {% endfor %} {% endif %} {% endwith %}
```

→ Python

Initializing our DB inside our `__init__.py`

```
in.py      _init_.py X  auth.py  base.html  home.html  models.py  
te > _init_.py  
1 from flask import Flask  
2 from flask_sqlalchemy import SQLAlchemy  
3     The name of the DB object that we will use  
4 db = SQLAlchemy()  
5 DB_NAME = "database.db"
```

The code shows the initialization of a Flask application and its database. A handwritten note explains that the variable 'db' is used whenever we want to interact with the database, such as creating or deleting users.

Setting up our database

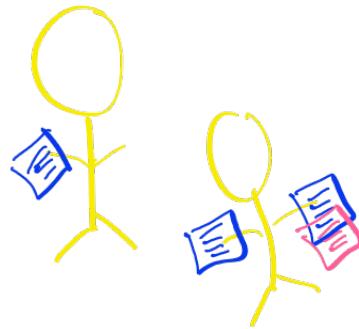
```
main.py      _init_.py X  auth.py  base.html  home.html  models.py  views.py
website > _init_.py
1  from flask import Flask
2  from flask_sqlalchemy import SQLAlchemy
3
4  db = SQLAlchemy()
5  DB_NAME = "database.db"
6
7
8  def create_app():
9      app = Flask(__name__)
10     app.config['SECRET_KEY'] = 'hjshjhdjah kjshkjdhjs' ↴
11     app.config['SQLALCHEMY_DATABASE_URI'] = f'sqlite:///{{DB_NAME}}' ↴
12     db.init_app(app)           ↴ initialize the DB   ↴ The location of the
13                               ↴
14     from .views import views
15     from .auth import auth
```

The main website folder

The location of the DB

Let's head to our `models.py` file to draft out the schema of our database

- We gonna have a db for our **Users** and another for our **Notes**



Models.py

current Package {website}

```
1 from . import db
2 from flask_login import UserMixin
3
4 |
```

↳ To login our users

Defining the users table:

```
main.py      __init__.py    auth.py    base.html    home.html    models.py X  views.py
website > models.py
```

1 from . import db
2 from flask_login import UserMixin
3
4 class User(db.Model, UserMixin):
5 id = db.Column(db.Integer, primary_key=True)
6 email = db.Column(db.String(150), unique=True)
7 password = db.Column(db.String(150))
8 first_name = db.Column(db.String(150))

Columns

inherit from UserMixin

Type at the column

No user can have the same email

Defining the notes table:

```
main.py _init_.py auth.py base.html home.html models.py X views.py
website > models.py

1 from . import db
2 from flask_login import UserMixin
3 from sqlalchemy.sql import func ↵ for saving the date
4
5
6 class Note(db.Model):
7     id = db.Column(db.Integer, primary_key=True) ↵ ids are always unique and automatically incremented
8     data = db.Column(db.String(10000))
9     date = db.Column(db.DateTime(timezone=True), default=func.now())
10    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
11
12    ↳ association with the users table ↳ database column that references to another column in a diff table
13
14    class User(db.Model, UserMixin):
15        id = db.Column(db.Integer, primary_key=True)
16        email = db.Column(db.String(150), unique=True)
```

Linking the 2 tables together:

```
4  
5  
6 class Note(db.Model):  
7     id = db.Column(db.Integer, primary_key=True)  
8     data = db.Column(db.String(10000))  
9     date = db.Column(db.DateTime(timezone=True), default=func.now())  
10    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))  
11  
12 class User(db.Model, UserMixin):  
13     id = db.Column(db.Integer, primary_key=True)  
14     email = db.Column(db.String(150), unique=True)  
15     password = db.Column(db.String(150))  
16     first_name = db.Column(db.String(150))  
17     notes = db.relationship('Note')  
18  
19
```

↑ Primary Key

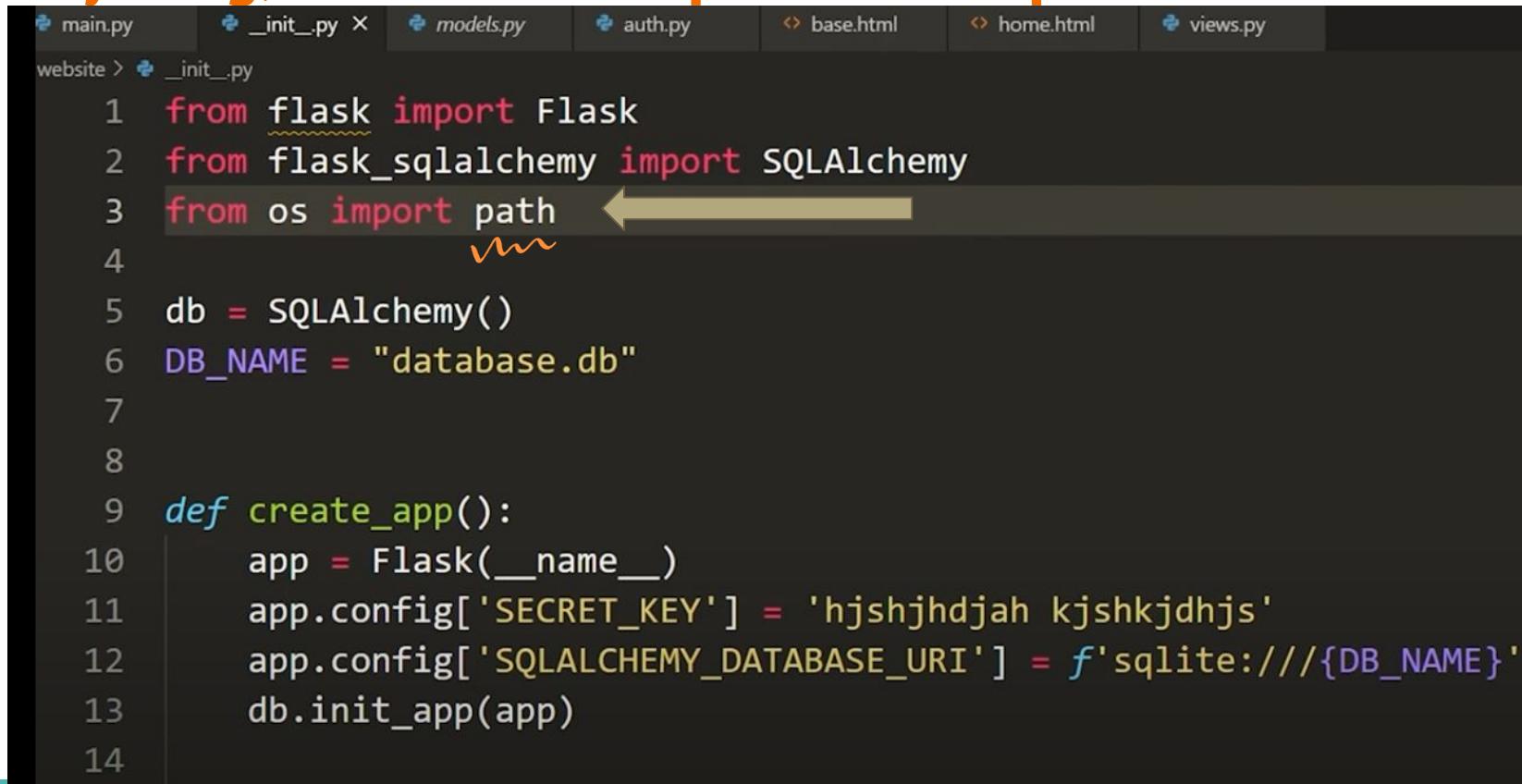
→ Link the Note Table with Users

Importing the models into our `__init__.py` file

```
main.py      __init__.py X  models.py  auth.py  base.html  home.html  views.py
website > __init__.py
15     from .views import views
16     from .auth import auth
17
18     app.register_blueprint(views, url_prefix='/')
19     app.register_blueprint(auth, url_prefix='/')
20
21
22
23     create_database(app)
24
25     return app
26
27
28 def create_database(app):
29     if not path.exists('website/' + DB_NAME):
30         db.create_all(app=app)
```

checks if the DB already exists. If it doesn't, then create it

To check whether or not the DB was created before anything, we need to import the OS path:



```
main.py _init_.py ✘ models.py auth.py base.html home.html views.py  
website > _init_.py  
1 from flask import Flask  
2 from flask_sqlalchemy import SQLAlchemy  
3 from os import path ←  
4  
5 db = SQLAlchemy()  
6 DB_NAME = "database.db"  
7  
8  
9 def create_app():  
10     app = Flask(__name__)  
11     app.config['SECRET_KEY'] = 'hjshjhdjah kjshkjdhjs'  
12     app.config['SQLALCHEMY_DATABASE_URI'] = f'sqlite:///{{DB_NAME}}'  
13     db.init_app(app)  
14
```

Importing the tables from the DB models:

```
site > _init_.py
15     from .views import views
16     from .auth import auth
17
18     app.register_blueprint(views, url_prefix='/')
19     app.register_blueprint(auth, url_prefix='/')
20
21     from .models import User, Note ←
22
23     create_database(app)
24
25     return app
26
```

Now, our DB is created and linked. Let's work on signing up the users.

On our auth.py file, we need to configure:

website > auth.py

```
1 from flask import Blueprint, render_template, request, flash
2 from .models import User ←
3
4 auth = Blueprint('auth', __name__)
5
6
7 @auth.route('/login', methods=['GET', 'POST'])
8 def login():
9     return render_template("login.html", boolean=True)
10
11
12 @auth.route('/logout')
13 def logout():
14     return "<p>Logout</p>"
15
16
```

We need to import a few things that will help us hash our password:

```
from werkzeug.security import generate_password_hash, check_password_hash
```

hashing is a way of encryption
that makes the pass has no
inverse

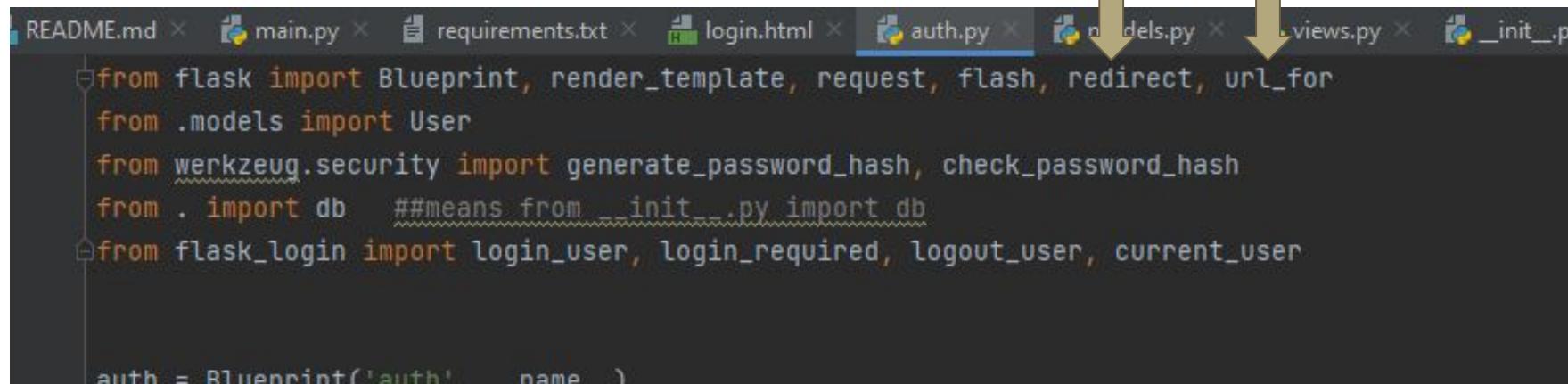
Adding our new users to our auth.py file

```
if user:  
    flash('Email already exists.', category='error')  
elif len(email) < 4:  
    flash('Email must be greater than 3 characters.', category='error')  
elif len(first_name) < 2:  
    flash('First name must be greater than 1 character.', category='error')  
elif password1 != password2:  
    flash('Passwords don\'t match.', category='error')  
elif len(password1) < 7:  
    flash('Password must be at least 7 characters.', category='error')  
else:  
    new_user = User(email=email, first_name=first_name, password=generate_password_hash(  
        password1, method='sha256'))  
    db.session.add(new_user)  
    db.session.commit()  
    login_user(new_user, remember=True)  
    flash('Account created!', category='success')  
return redirect(url_for('views.home'))
```

→ hashing algorithm
→ update DB

Redirecting the user into
the homepage after they
sign up

After a new user is added, we should redirect them to the home page:



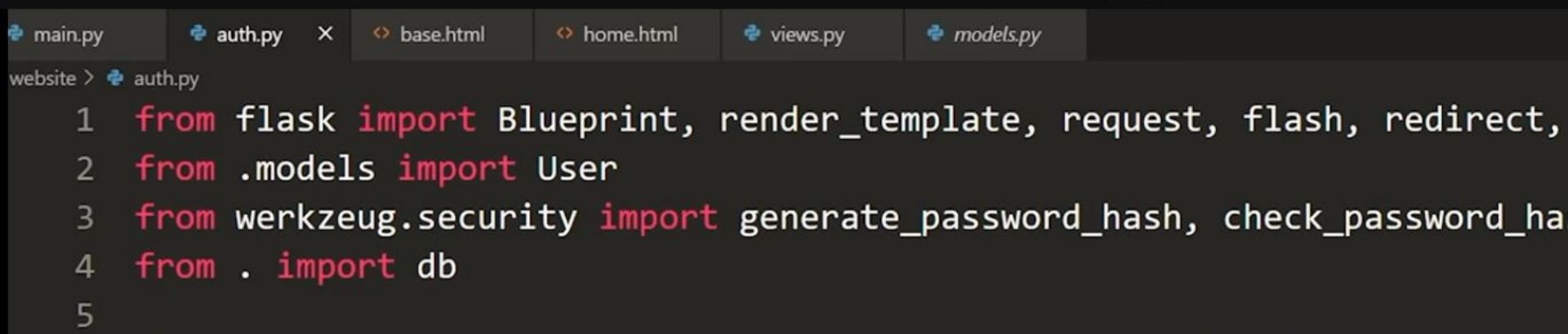
```
README.md × main.py × requirements.txt × login.html × auth.py × models.py × views.py × __init__.py
from flask import Blueprint, render_template, request, flash, redirect, url_for
from .models import User
from werkzeug.security import generate_password_hash, check_password_hash
from . import db    ##means from __init__.py import db
from flask_login import login_user, login_required, logout_user, current_user

auth = Blueprint('auth', __name__)
```

```
    flash('Password must be at least 7 characters.', category='error')
else:
    new_user = User(email=email, firstName=firstName, password=password)
    db.session.add(new_user)
    db.session.commit()
    flash('Account created!', category='success')
    return redirect(url_for('views.home'))
```

```
return render_template("sign_up.html")
```

Don't forget about importing the db to your auth.py file



A screenshot of a code editor showing the contents of the auth.py file. The file is part of a project structure named 'website' under 'auth.py'. The code imports several modules from the Flask framework and the application's models:

```
1  from flask import Blueprint, render_template, request, flash, redirect,  
2  from .models import User  
3  from werkzeug.security import generate_password_hash, check_password_ha  
4  from . import db  
5
```

Handling the sign-in inside our auth.py file

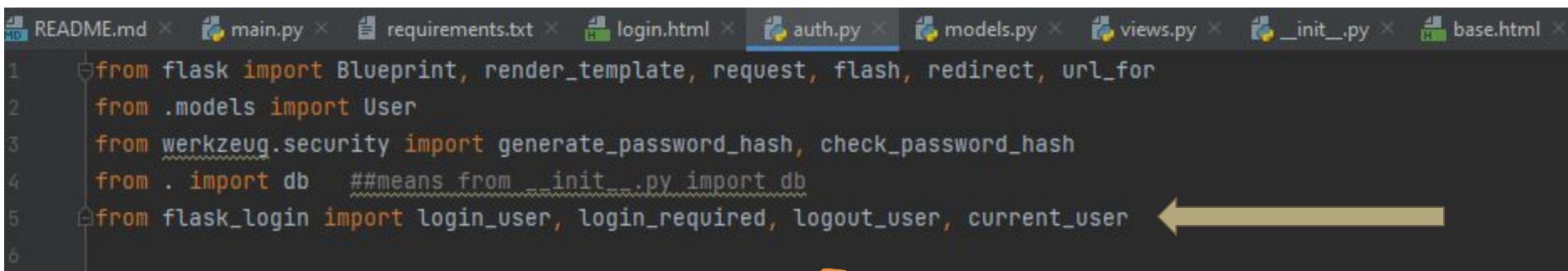
↳ Try it yourself!

Def login():

```
If request.method=='post':  
    email =request.form.get('email')  
    password =request.form.get('password')
```

```
user = User.query.filter_by(email=email).first()  
if user:    ↗ return only the  
            ↗ The hashed pass 1st result if we  
            ↗ have many [ we  
            ↗ should not ]  
    if check_password_hash(user.password, password):  
        flash('Logged in successfully!', category='success')  
        login_user(user, remember=True) → Browser will remember the  
        ↗ user until they clear  
        ↗ their browser.  
        return redirect(url_for('views.home'))  
    else:  
        flash('Incorrect password, try again.', category='error')  
else:  
    flash('Email does not exist.', category='error')
```

Libraries that will help us safe the user sessions:



```
1 from flask import Blueprint, render_template, request, flash, redirect, url_for
2 from .models import User
3 from werkzeug.security import generate_password_hash, check_password_hash
4 from . import db ##means from __init__.py import db
5 from flask_login import login_user, login_required, logout_user, current_user
```

To ensure that no user can access the homepage unless they're logged in.

Redirecting the users after they log out:

```
30  
31 @auth.route('/logout')  
32 @login_required → makes sure that the user logged out only  
33 def logout():           if they're logged in.  
34     logout_user()  
35     return redirect(url_for('auth.login'))  
36  
37
```

Require the users to be logged in before they access the homepage: (views.py)

website > views.py

```
1  from flask import Blueprint, render_template
2  from flask_login import login_required, current_user
3
4  views = Blueprint('views', __name__)
5
6
7  @views.route('/')
8  @login_required
9  def home():
10     return render_template("home.html")
11
```

You cannot
go to the
homepage
until you're
logged in

Flask Login Manager: (-- init.py --)

```
README.md × main.py × requirements.txt × login.html × au
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from os import path
from flask_login import LoginManager ←
```

Where should the flask take us if we're not logged in?

Back to the login screen: (Still on `--init--.py`)

```
create_database(app)
login_manager = LoginManager()
login_manager.login_view = 'auth.login'
login_manager.init_app(app)

@login_manager.user_loader
def load_user(id):
    return User.query.get(int(id))

return app
```

where should flask
redirect us if we're
not logged in?

How it Works

You will need to provide a [user_loader](#) callback. This callback is used to reload the user object from the user ID stored in the session. It should take the [str](#) ID of a user, and return the corresponding user object. For example:

```
@login_manager.user_loader  
def load_user(user_id):  
    return User.get(user_id)
```

On your views.py, check if there's a logged in user

The screenshot shows a code editor with several tabs at the top: main.py, auth.py, base.html, home.html, views.py (which is the active tab), and __init__.py. Below the tabs, the file structure is shown as website > views.py. The code in views.py is:

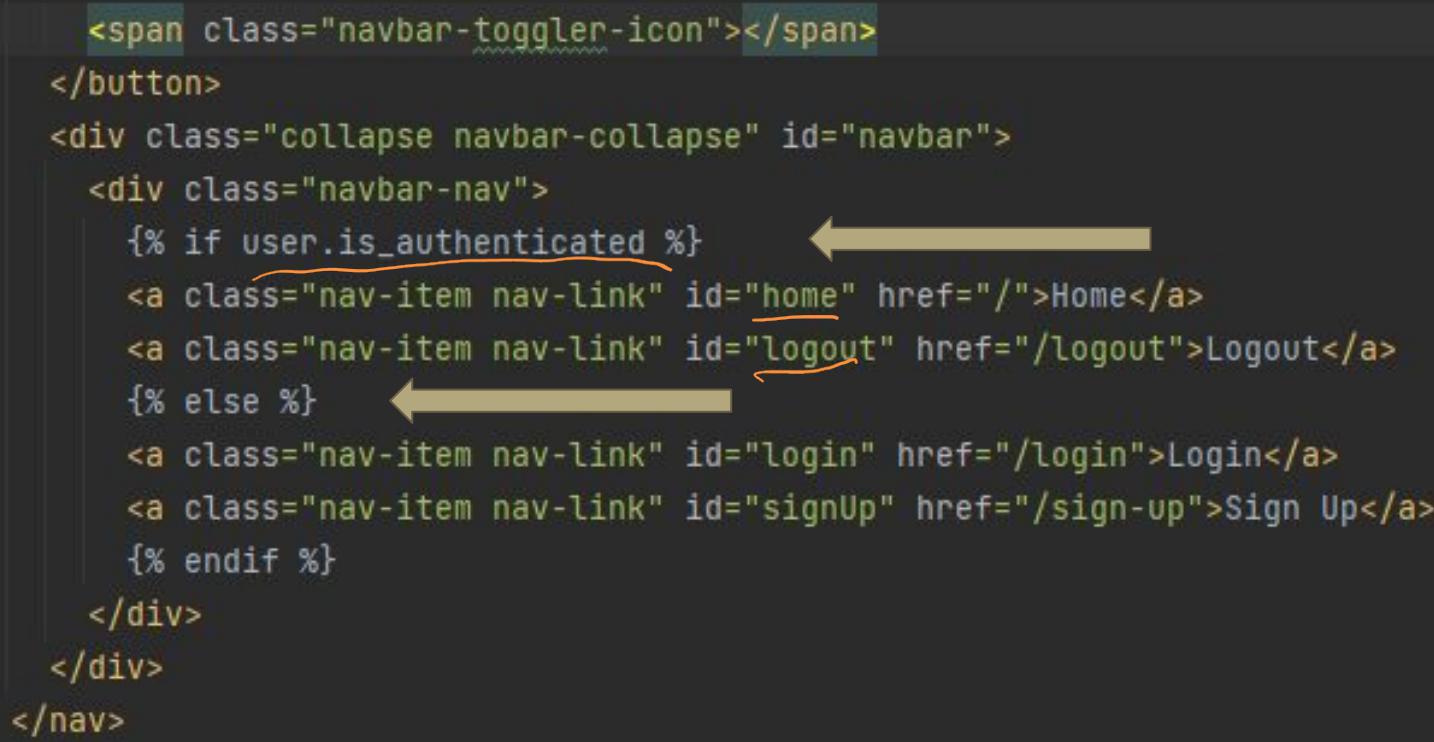
```
1 from flask import Blueprint, render_template
2 from flask_login import login_required, current_user
3
4 views = Blueprint('views', __name__)
5
6
7 @views.route('/')
8 @login_required
9 def home():
10     return render_template("home.html", user=current_user)
```

Annotations include:

- An orange arrow points from the word "current_user" in the imports section to the "current_user" parameter in the render_template call.
- A wavy orange line underlines the "current_user" parameter in the render_template call.
- A handwritten note in the bottom right corner says: "check if the logged in user is authenticated".
- A tooltip on the right side of the screen provides documentation for render_template: "flask.render_t **context) Returns → str".

We need to ensure that even the html nav-bar is only shown to the logged in users: [back to base.html]

```
<span class="navbar-toggler-icon"></span>
</button>
<div class="collapse navbar-collapse" id="navbar">
  <div class="navbar-nav">
    {% if user.is_authenticated %}
      <a class="nav-item nav-link" id="home" href="/">Home</a>
      <a class="nav-item nav-link" id="logout" href="/logout">Logout</a>
    {% else %}
      <a class="nav-item nav-link" id="login" href="/login">Login</a>
      <a class="nav-item nav-link" id="signUp" href="/sign-up">Sign Up</a>
    {% endif %}
  </div>
</div>
</nav>
```



In our auth.py file, we need to modify the login method to check the current user:

```
0     flash('Logged in successfully!', category='success')
1     login_user(user, remember=True)
2     return redirect(url_for('views.home'))
3 else:
4     flash('Incorrect password, try again.', category='error')
5 else:
6     flash('Email does not exist.', category='error')
7
8 return render_template("login.html", user=current_user) ←
```

Same for signup:

```
|     |     return redirect(url_for('views.home'))  
  
return render_template("sign_up.html", user=current_user)
```

Adding & Displaying notes: [home.html]

```
{% extends "base.html" %} {% block title %}Home{% endblock %} {% block content %}  
%}  
<h1 align="center">Notes</h1>    ↴ Some bootstrap code  
<ul class="list-group list-group-flush" id="notes">  
  {% for note in user.notes %} → ensuring that the notes are related to the user  
    <li class="list-group-item">{{ note.data }}</li>  
  {% endfor %}  
</ul>                                ↴ To get the notes from the user.  
<form method="POST">    ↴  
  <textarea name="note" id="note" class="form-control"></textarea>  
  <div align="center">  
    <button type="submit" class="btn btn-primary">Add Note</button>  
  </div>  
</form>  
{% endblock %}
```

Views.py

```
@views.route('/', methods=['GET', 'POST'])
@login_required
def home():
    if request.method == 'POST':
        note = request.form.get('note')#Gets the note from the HTML

        if len(note) < 1:
            flash('Note is too short!', category='error')
        else:
            new_note = Note(data=note, user_id=current_user.id) #providing the schema for the note
            db.session.add(new_note) #adding the note to the database
            db.session.commit() || update
            flash('Note added!', category='success')

    return render_template("home.html", user=current_user)
```

Don't forget about importing the DB into the views.py to link both the front and back end of the app:

The screenshot shows a code editor with a dark theme. The top navigation bar includes tabs for main.py, auth.py, base.html, home.html, models.py, views.py (which is the active tab), and __init__.py. Below the tabs, the file structure is shown as website > views.py. The code in views.py is as follows:

```
1 from flask import Blueprint, render_template, request, flash
2 from flask_login import login_required, current_user
3 from .models import Note    ↗
4 from . import db    ↗
5 DB_NAME           str kite
6 views = Blueprint('views', __name__)
```

Two specific imports are highlighted with orange arrows pointing to them: 'from .models import Note' and 'from . import db'. A tooltip for 'DB_NAME' shows the text 'str kite'.

Deleting the note from home.html

```
<h1 align="center">Notes</h1>
<ul class="list-group list-group-flush" id="notes">
  {% for note in user.notes %}           ←
  <li class="list-group-item">
    {{ note.data }}
    <button type="button" class="close" onClick="deleteNote({{ note.id }})">
      <span aria-hidden="true">x</span>
    </button>
  </li>
  {% endfor %}
</ul>
```

Js function that will send out the data to the back end

We'll use our index.js file to handle the deletion function:

The screenshot shows a code editor with multiple tabs at the top: main.py, auth.py, base.html, home.html, index.js (which is the active tab), views.py, and __init__.py. Below the tabs, the file index.js contains the following code:

```
1 function deleteNote(noteId) {① get the note
2   fetch("/delete-note", {
3     method: "POST",
4     body: JSON.stringify({ noteId: noteId }),
5   }).then((res) => {
6     window.location.href = '/';③ refresh the page
7   });
8 }
```

Handwritten annotations in orange are overlaid on the code:

- ① get the note: points to the first line of the function.
- ② delete it through this endpoint: points to the fetch call.
- ③ refresh the page: points to the window.location.href assignment.

```
5 import json, jsonify
```

Deletion inside our views.py file:

```
@views.route('/delete-note', methods=['POST'])
def delete_note():
    note = json.loads(request.data) # this function expects a JSON from the INDEX.js file
    noteId = note['noteId']
    note = Note.query.get(noteId)
    if note: → if the note does exist
        if note.user_id == current_user.id:
            db.session.delete(note)
            db.session.commit()

    return jsonify({})
```