Experts

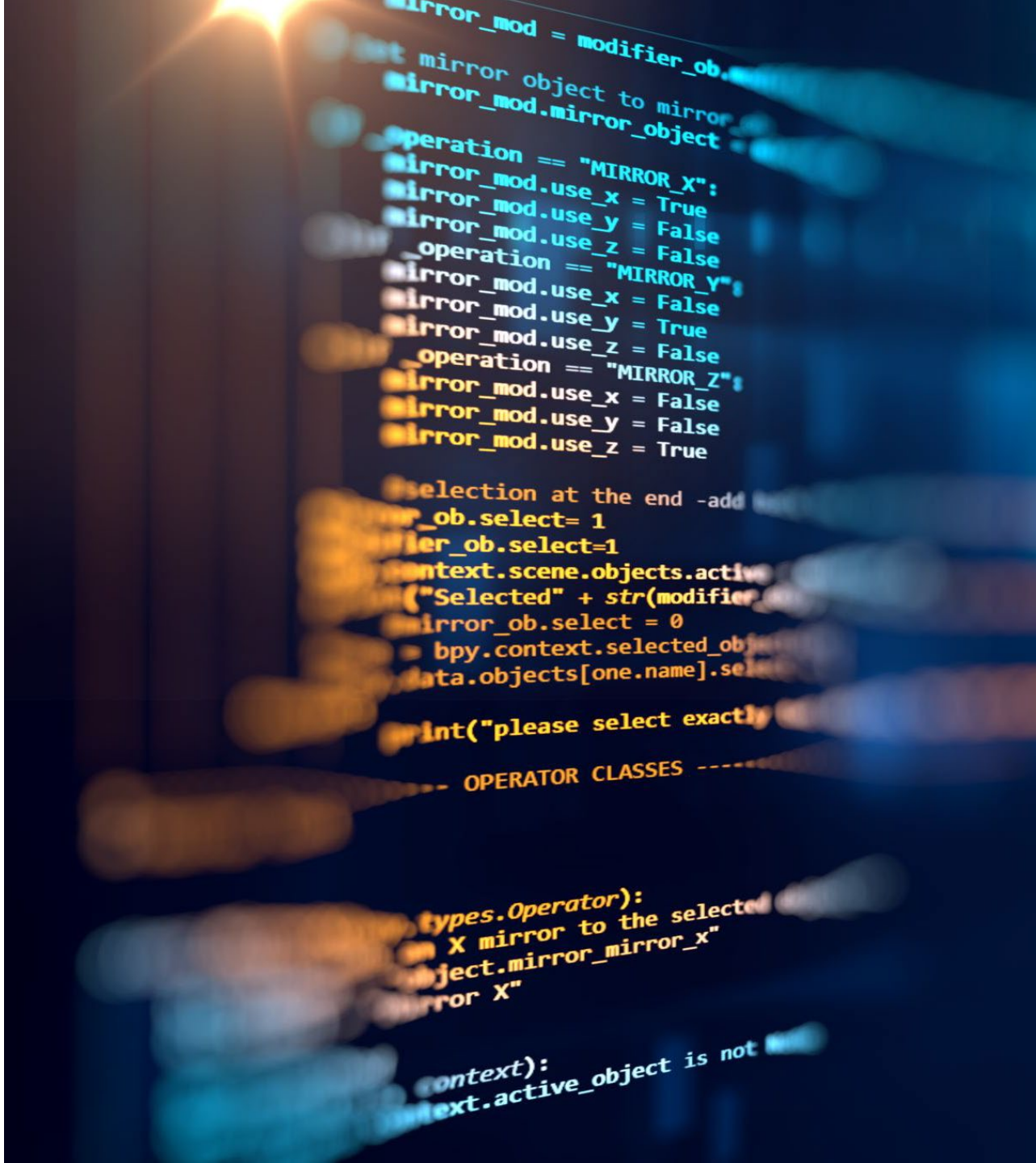
Flask Advanced (I)

By Cyrus Wong

Source:
https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world

Outline

1. Templates
2. Web Forms
3. Database
4. User Logins
5. User Profile
6. Error Handling
7. Followers
8. Pagination



Templates

app/routes.py: Use render_template() function

```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Miguel'}
    return render_template('index.html', title='Home', user=user)
```

app/templates/index.html: Main page template

```
<html>
    <head>
        <title>{{ title }} - Microblog</title>
    </head>
    <body>
        <h1>Hello, {{ user.username }}!</h1>
    </body>
</html>
```

Templates

app/routes.py: Use render_template() function

```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Miguel'}
    return render_template('index.html', title='Home', user=user)
```

The render_template() function invokes the Jinja2 template engine, substitutes {{ ... }} blocks with the corresponding values, given by the arguments (e.g. title='Home',user=user)

app/templates/index.html: Main page template

```
<html>
    <head>
        <title>{{ title }} - Microblog</title>
    </head>
    <body>
        <h1>Hello, {{ user.username }}!</h1>
    </body>
</html>
```

Templates

Templates also support **control statements**, given inside {% ... %} blocks

app/templates/index.html: Conditional statement in template

```
<html>
    <head>
        {% if title %}
        <title>{{ title }} - Microblog</title>
        {% else %}
        <title>Welcome to Microblog!</title>
        {% endif %}
    </head>
    <body>
        <h1>Hello, {{ user.username }}!</h1>
    </body>
</html>
```

Conditional Statements – if,elif,else

app/routes.py: Fake posts in view function

```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Miguel'}
    posts = [
        {
            'author': {'username': 'John'},
            'body': 'Beautiful day in Portland!'
        },
        {
            'author': {'username': 'Susan'},
            'body': 'The Avengers movie was so cool!'
        }
    ]
    return render_template('index.html', title='Home', user=user, posts=posts)
```

Templates also support **loops**, given inside {% ... %} blocks

Loops

app/templates/index.html: Inherit from base template

```
{% extends "base.html" %}

{% block content %}
    <h1>Hi, {{ user.username }}!</h1>
    {% for post in posts %}
    <div><p>{{ post.author.username }} says: <b>{{ post.body }}</b></p></div>
    {% endfor %}
{% endblock %}
```

app/routes.py: Fake posts in view function

```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Miguel'}
    posts = [
        {
            'author': {'username': 'John'},
            'body': 'Beautiful day in Portland!'
        },
        {
            'author': {'username': 'Susan'},
            'body': 'The Avengers movie was so cool!'
        }
    ]
    return render_template('index.html', title='Home', user=user, posts=posts)
```

Templates also support **loops**, given inside {% ... %} blocks

Loops

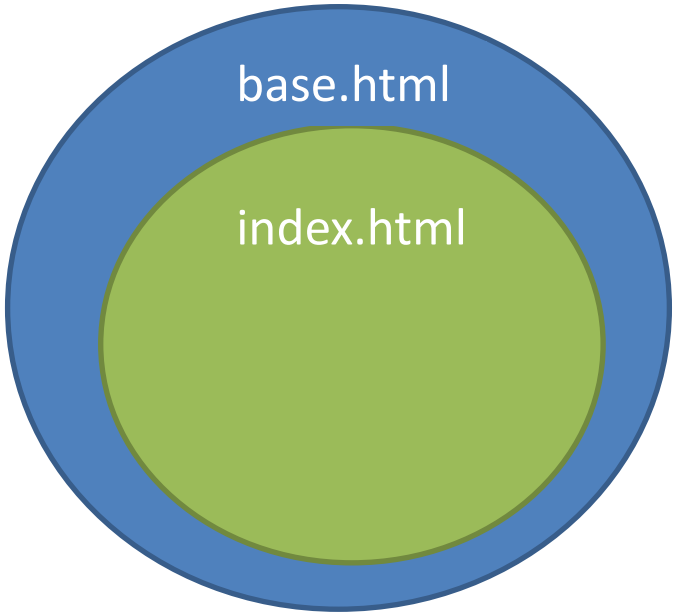
app/templates/index.html: Inherit from base template

```
{% extends "base.html" %}

{% block content %}
    <h1>Hi, {{ user.username }}!</h1>
    {% for post in posts %}
    <div><p>{{ post.author.username }} says: <b>{{ post.body }}</b></p></div>
    {% endfor %}
{% endblock %}
```

Templates Inheritance

app/templates/base.html: Base template with navigation bar

```
<html>
    <head>
      {% if title %}
      <title>{{ title }} - Microblog</title>
      {% else %}
      <title>Welcome to Microblog</title>
      {% endif %}
    </head>
    <body>
        <div>Microblog: <a href="/index">Home</a></div>
        <hr>
        {% block content %}{% endblock %}
    </body>
</html>
```



app/templates/index.html: Inherit from base template

```
{% extends "base.html" %}

{% block content %}
    <h1>Hi, {{ user.username }}!</h1>
    {% for post in posts %}
    <div><p>{{ post.author.username }} says: <b>{{ post.body }}</b></p></div>
    {% endfor %}
{% endblock %}
```

Web Forms

- Flask-WTF
  - An extension to handle web forms
  - Python classes to represent web forms

```
(venv) $ pip install flask-wtf
```

- Configuration
  - Define a class “Config”
  - Using a secret to protect web forms against a nasty attack called Cross-Site Request Forgery

config.py: Secret key configuration

```
import os

class Config(object):
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'you-will-never-guess'
```

Can be accessed with a dictionary

```
>>> from microblog import app
>>> app.config['SECRET_KEY']
'you-will-never-guess'
```

SECRET_KEY

- Used as a cryptographic key
- Useful to generate signatures or tokens.
- Should be unique in production server

config.py: Secret key configuration

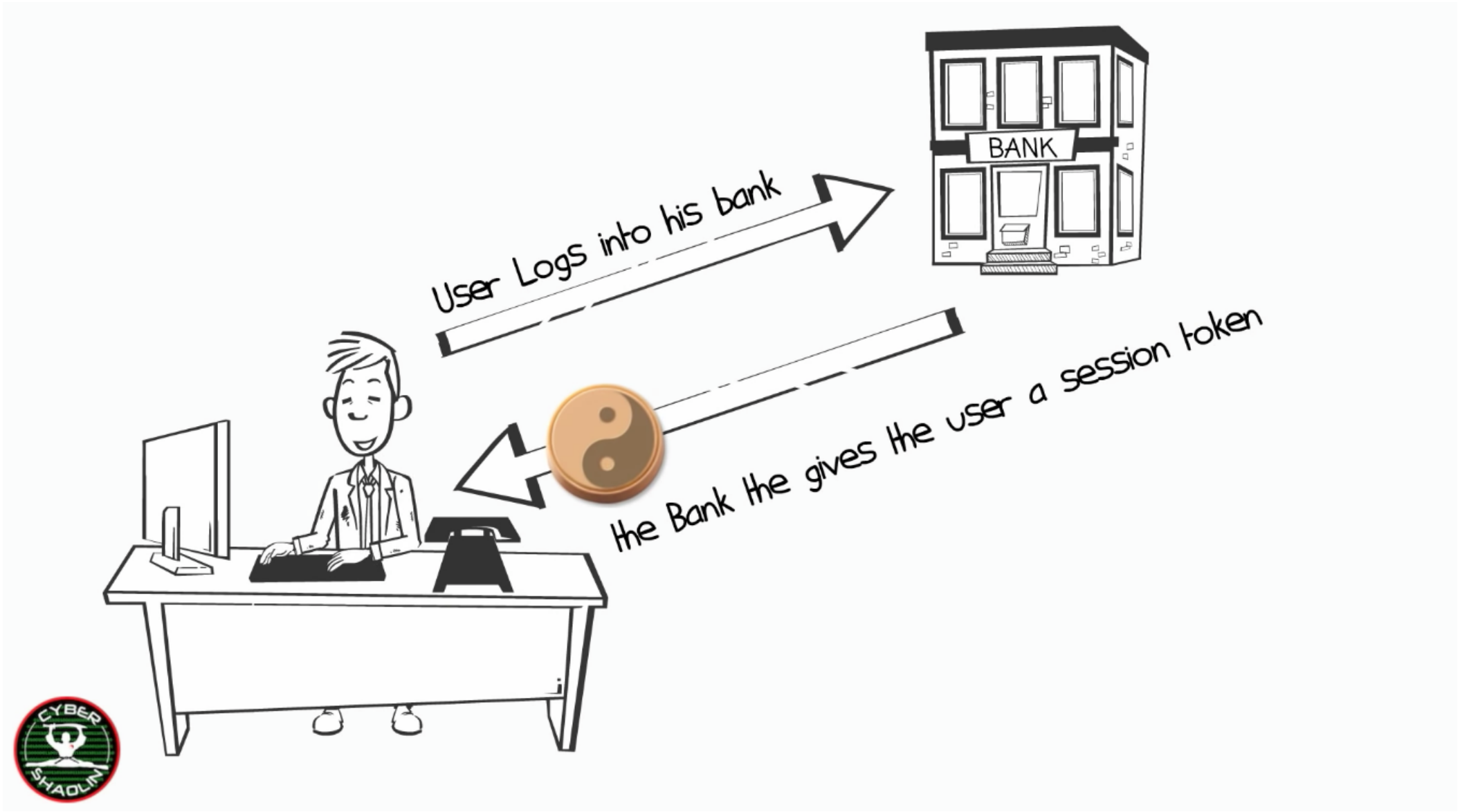
```
import os

class Config(object):
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'you-will-never-guess'
```

If set in environment variable: use it
If no: use a hardcoded string

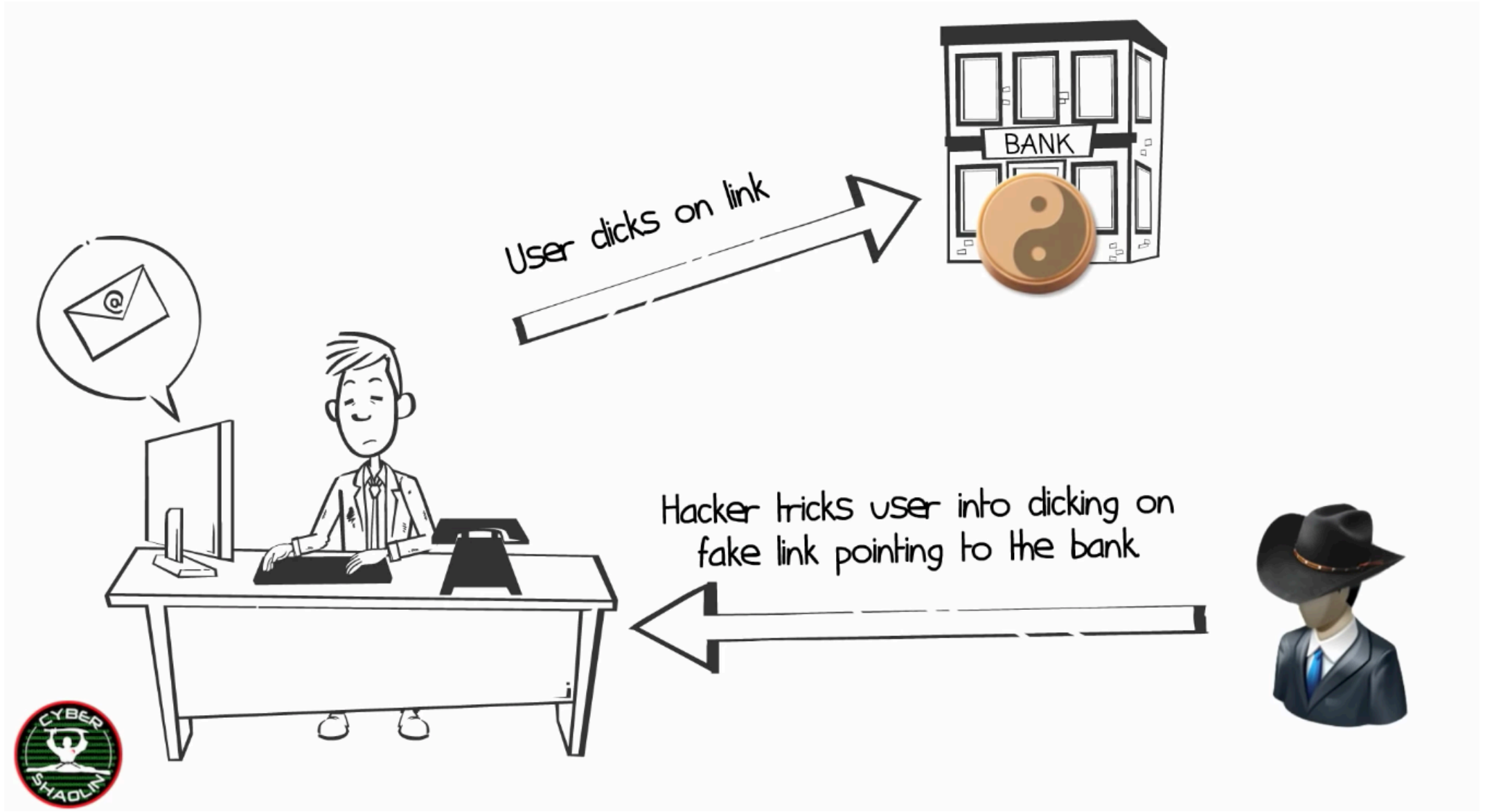
CSRF attacks

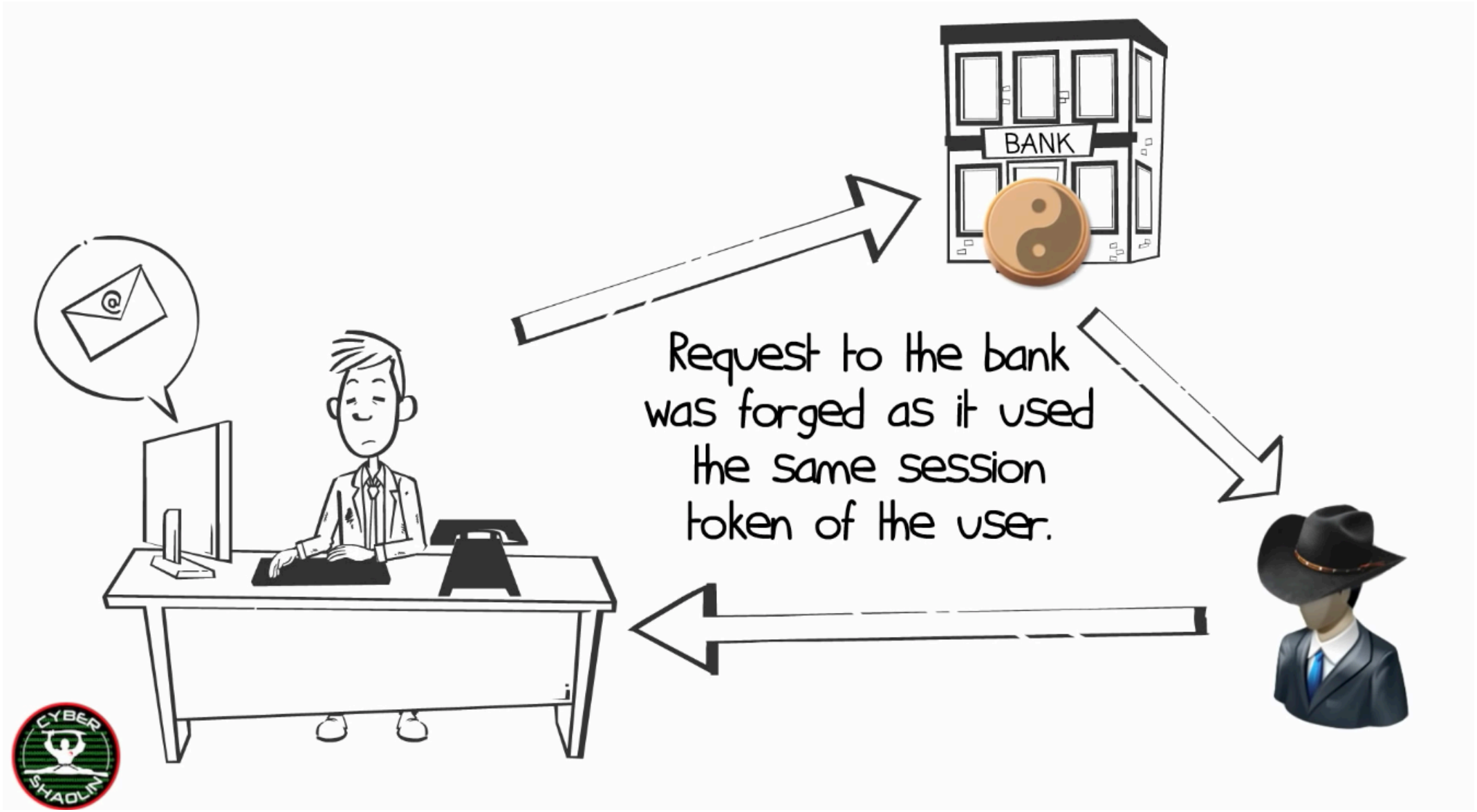
- CSRF（Cross-site request forgery）
  跨站請求偽造
- https://www.youtube.com/watch?v=m0EHlfTgGUU

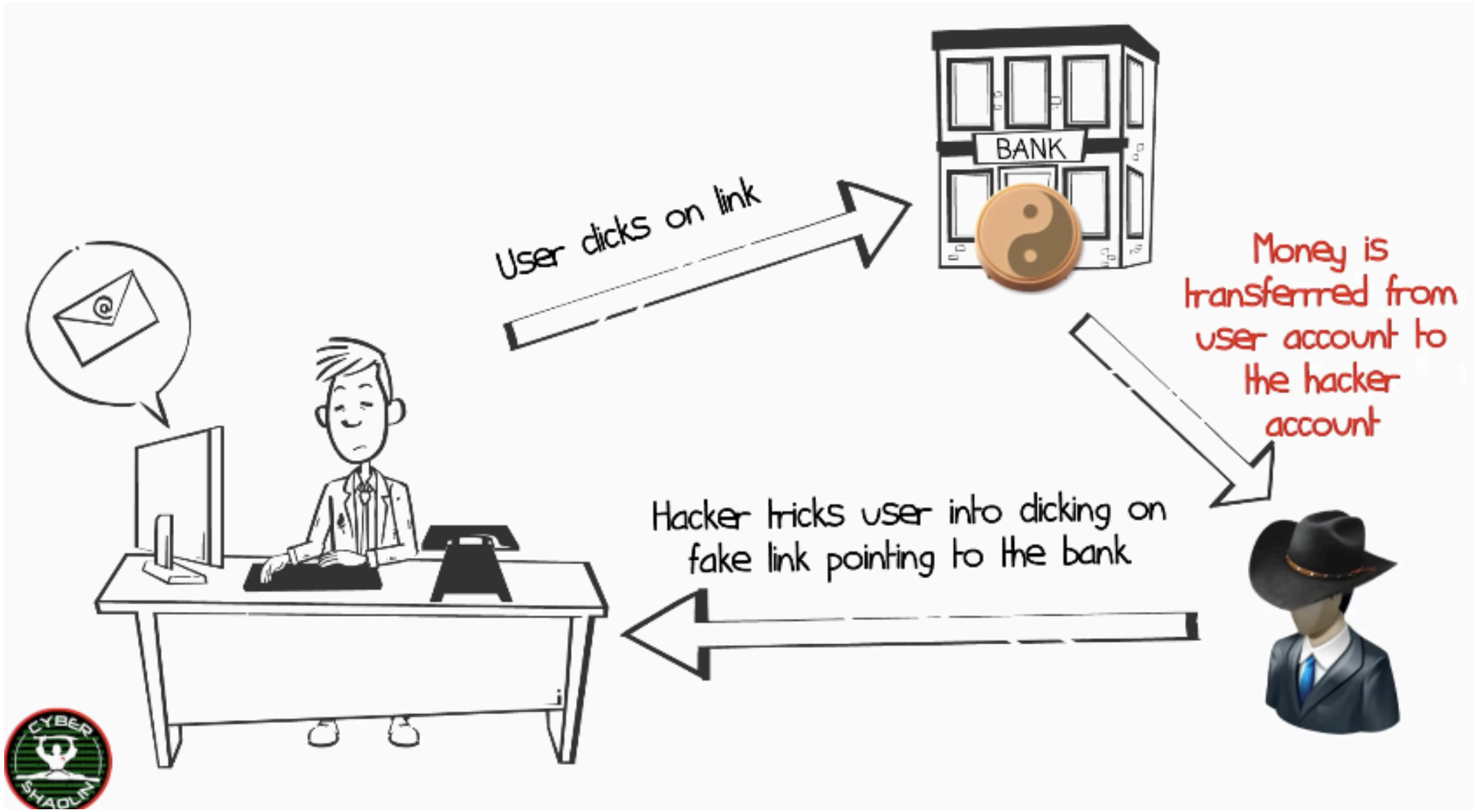

User Logs into his bank
BANK

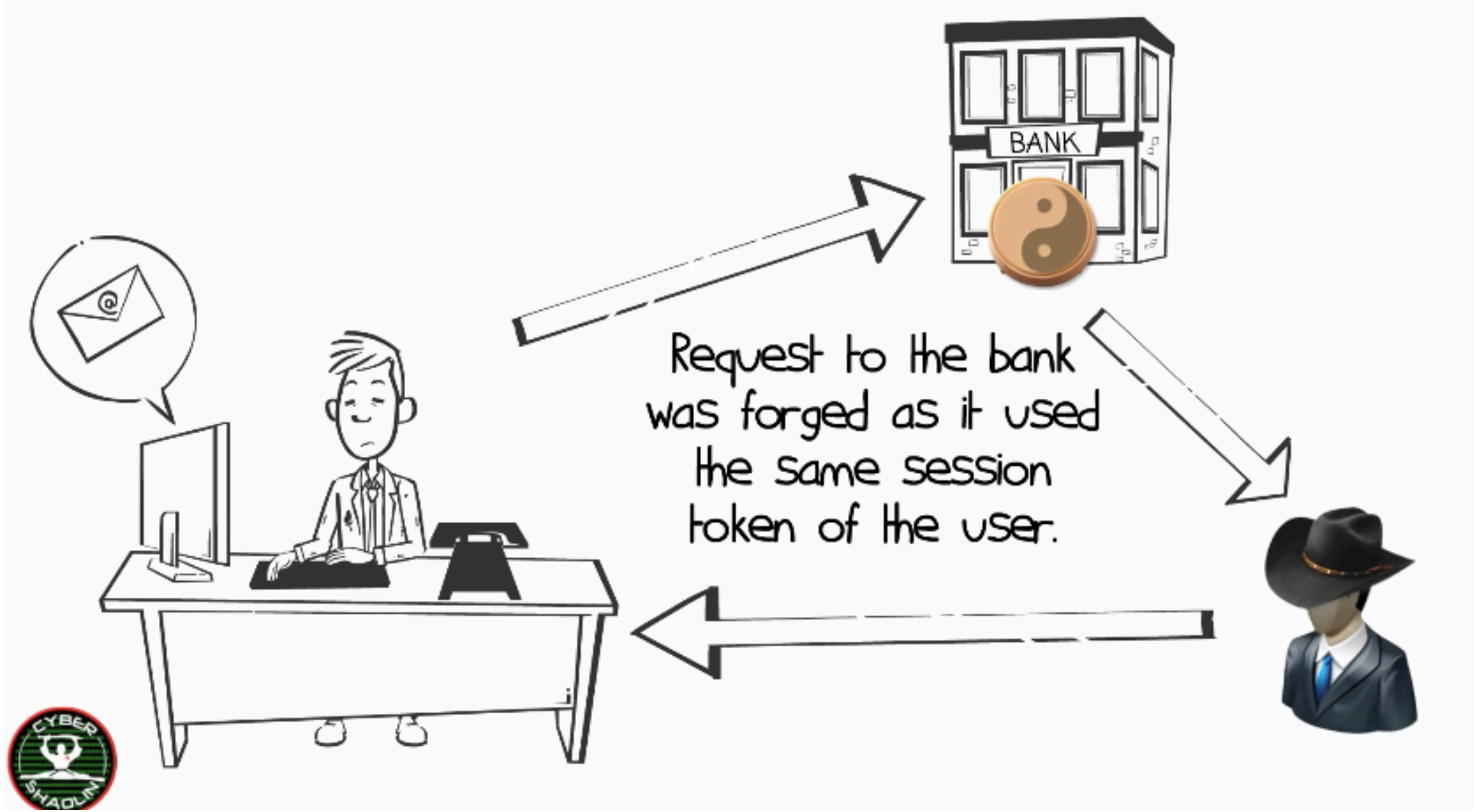

BANK
User Logs into his bank
the Bank the gives the user a session token
CYBER
SHAOLIN


BANK
Hacker tricks user into clicking on fake link pointing to the bank
CYBER SHAOLIN

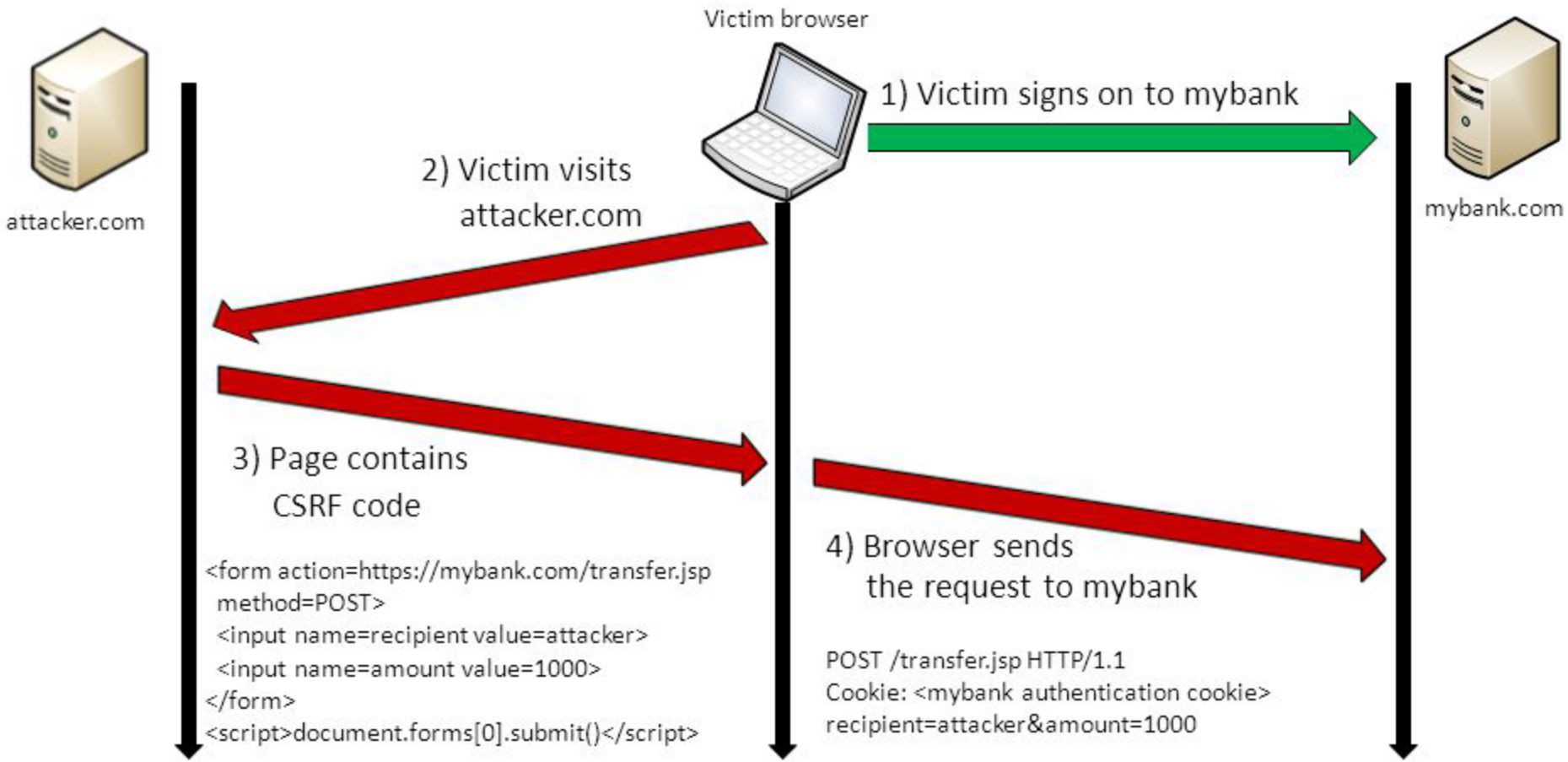

BANK
User clicks on link
Hacker tricks user into clicking on fake link pointing to the bank
CYBER SHAOLIN


BANK
Request to the bank was forged as it used the same session token of the user.
CYBER SHAOLIN


BANK
User clicks on link
Money is transferred from user account to the hacker account
Hacker tricks user into clicking on fake link pointing to the bank
CYBER SHAOLIN


BANK
Request to the bank was forged as it used the same session token of the user.
CYBER SHAOLIN

Cross-Site Request Forgery (CSRF)



Form Views

app/routes.py: Login view function

```
from flask import render_template
from app import app
from app.forms import LoginForm

# ...

@app.route('/login')
def login():
    form = LoginForm()
    return render_template('login.html', title='Sign In', form=form)
```

The form=form syntax:
simply passing the form object created in the line above (and shown on the right side) to the template with the name **form**

Web Forms

- Tell Flask to read and apply the configuration

```
# The lowercase "config" is the name of the
Python module config.py, and obviously
the one with the uppercase "C" is the
```

app/__init__.py: Flask configuration

```
from flask import Flask
from config import Config

app = Flask(__name__)
app.config.from_object(Config)

from app import routes
```

User Login Form

- 1 Class = 1 form
- 1 Class variables = 1 form fields

app/forms.py: Login form

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired

class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
    remember_me = BooleanField('Remember Me')
    submit = SubmitField('Sign In')
```

Form Templates

app/templates/login.html: Login form template

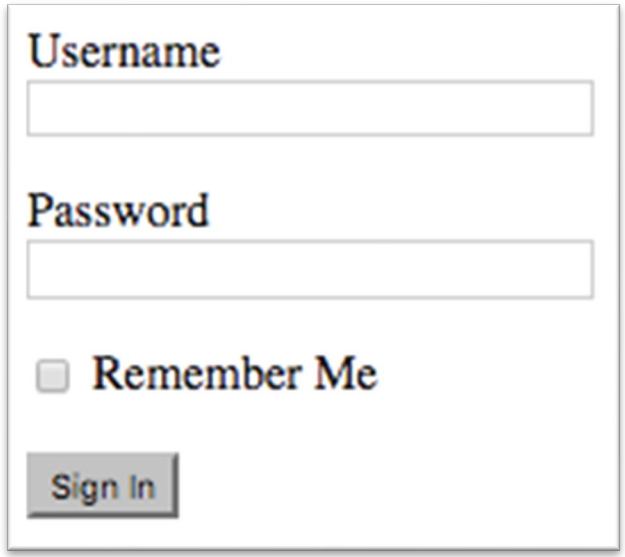
```
{% extends "base.html" %}

{% block content %}
    <h1>Sign In</h1>
    <form action="" method="post" novalidate>
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}
        </p>
        <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

The HTML <form> element is used as a container for the web form

This template expects a form object instantiated from the LoginForm class

Form Fields

Flask-WTF	HTML5
• StringField	• <input type="text">
• PasswordField	• <input type="password">
• BooleanField	• <input type="checkbox">
• SubmitField	• <input type="submit”>



NO HTML field tag

form object know how to render themselves as HTML

app/templates/login.html: Login form template

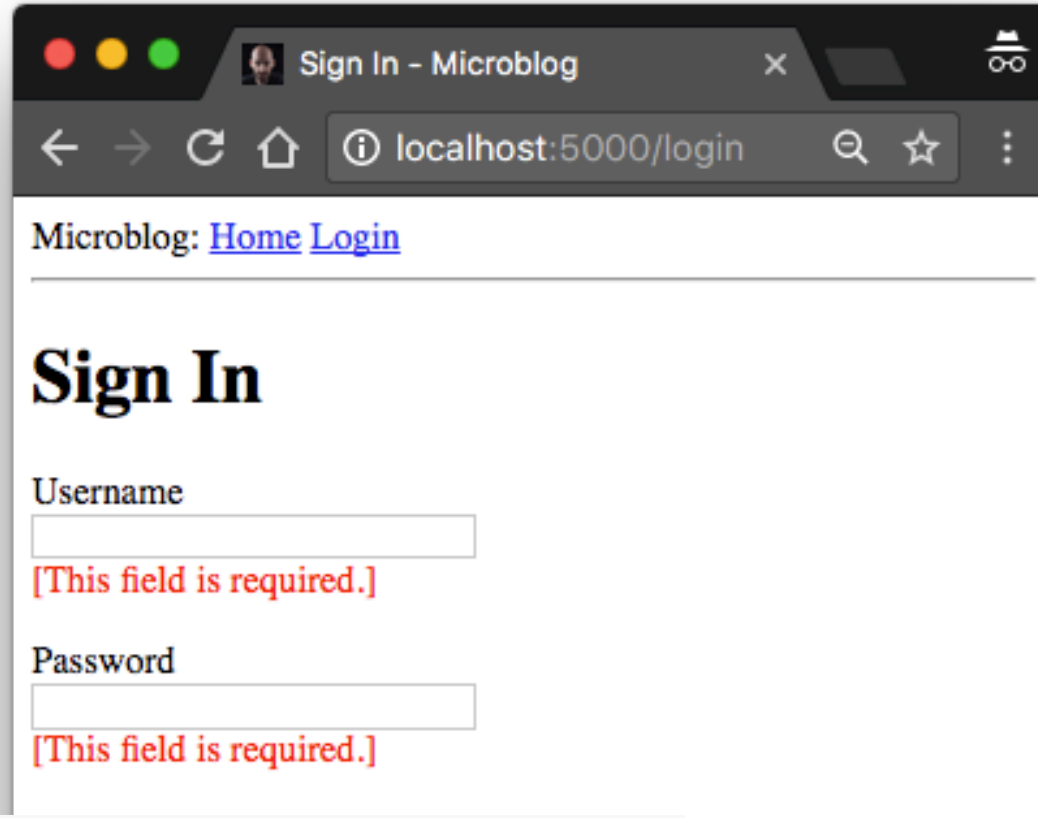
```
{% extends "base.html" %}

{
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}
        </p>
        <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

Call {{ form.<field_name>.label }} -> Get field label
Call {{ form.<field_name>() }} -> Get the field

For fields that require additional HTML attributes, those can be passed as arguments.
E.g. size = 32

User Login Form

- The **validators** argument is to attach validation behaviors to fields.
- The **DataRequired** validator simply checks that the field is not submitted empty.




```
username = StringField('Username', validators=[DataRequired()])
password = PasswordField('Password', validators=[DataRequired()])
```

CSRF Defense

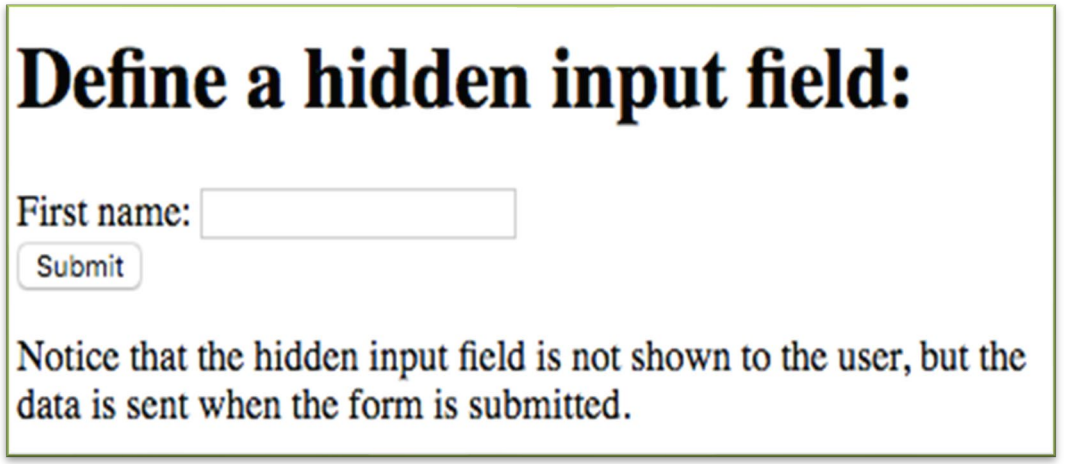
It generates a hidden field that includes a token that is used to protect the form against CSRF attacks.

```
{% block content %}
    <h1>Sign In</h1>
    <form action="" method="post" novalidate>
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}
        </p>
        <p>
            {{ form.password.label }}<br>
    </form>
{% endblock %}
```

form.hidden_tag() + SECRET_KEY = Job Done
Then Flask-WTF does the rest for you!

Hidden field

```
<!DOCTYPE html>
<html>
<body>

<h1>Define a hidden input field:</h1>

<form action="/action_page.php">
  First name: <input type="text" name="fname"><br>
  <input type="hidden" id="custId" name="custId" value="3487">
  <input type="submit" value="Submit">
</form>

<p>Notice that the hidden input field is not shown to the user, but the data is
sent when the form is submitted.</p>

</body>
</html>
```



Receiving Form Data

The HTTP protocol

- GET requests: **Returns** information to the client
- POST requests: **Submits** form data to the server

Receiving Form Data

app/routes.py: Receiving login credentials

```
from flask import render_template, flash, redirect

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        flash('Login requested for user {}, remember_me={}'.format(
            form.username.data, form.remember_me.data))
        return redirect('/index')
    return render_template('login.html', title='Sign In', form=form)
```

This view function accepts GET and POST requests (Default only GET)

form.validate_on_submit()
GET request: return False
POST request: return True

The flash() function showing a message to the user

The redirect() jumps to a different page

Improving Field Validation

- **Validators** + Meaningful error message for failed validation

- form validators can generate descriptive error messages

Improving Field Validation

app/templates/login.html: Validation errors in login form te

```
{% extends "base.html" %}

{% block content %}
    <h1>Sign In</h1>
    <form action="" method="post" novalidate>
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}<br>
            {% for error in form.username.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}<br>
            {% for error in form.password.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

Error occurs on Any fields with validators THROWS
form.<field_name>.errors

form.<field_name>.errors is a **List** since one field - multiple validators

If you submit the form with an empty password
Show error message in red

Generating Links

- Better control over links, use Flask's function : **url_for()**
  NOT redirect('/index')!!

Generates URLs mapped to name of view functions

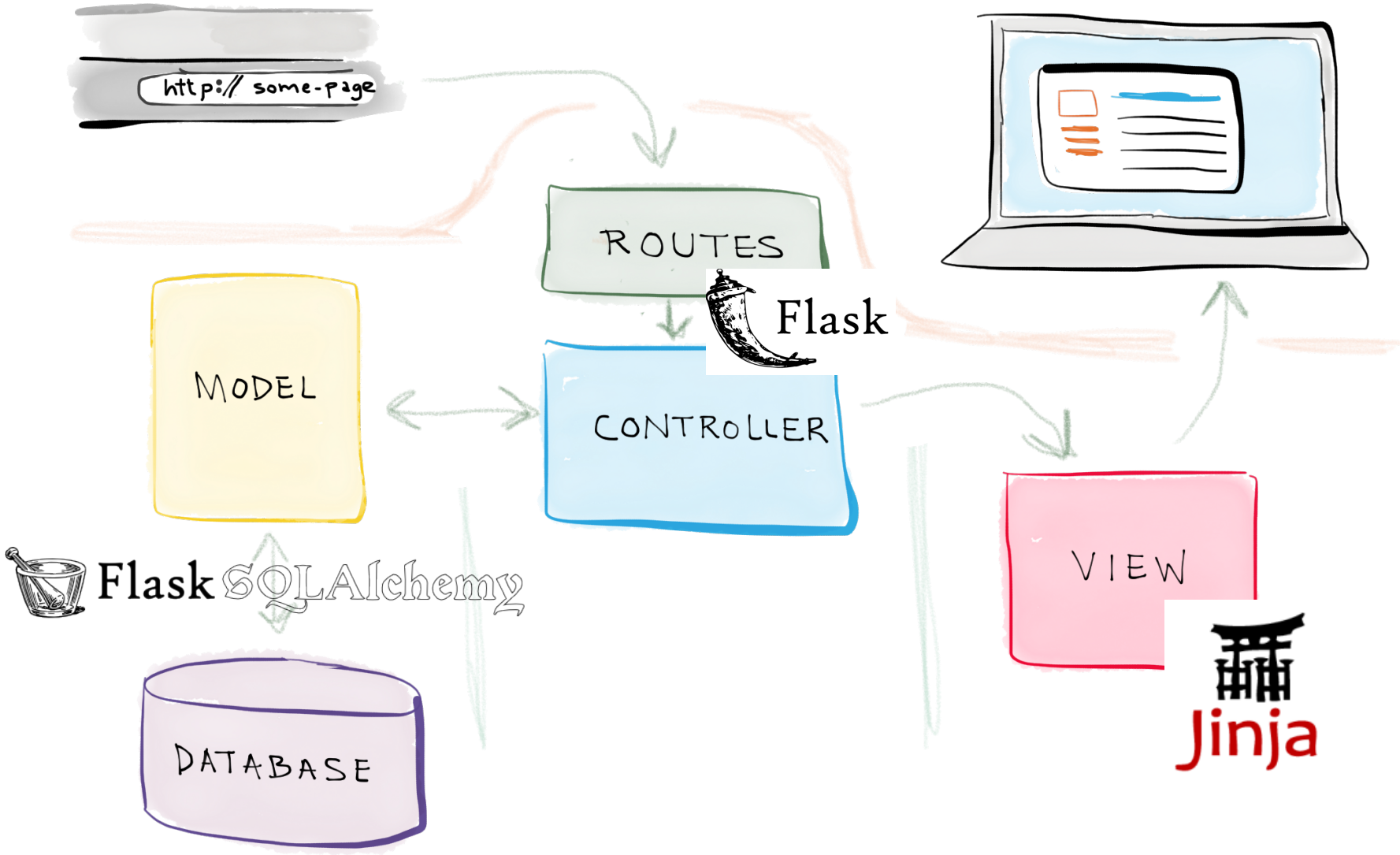
```
@app.route('/login')
def login():
    form = LoginForm()
    return render_template('
```

url_for('login') returns /login

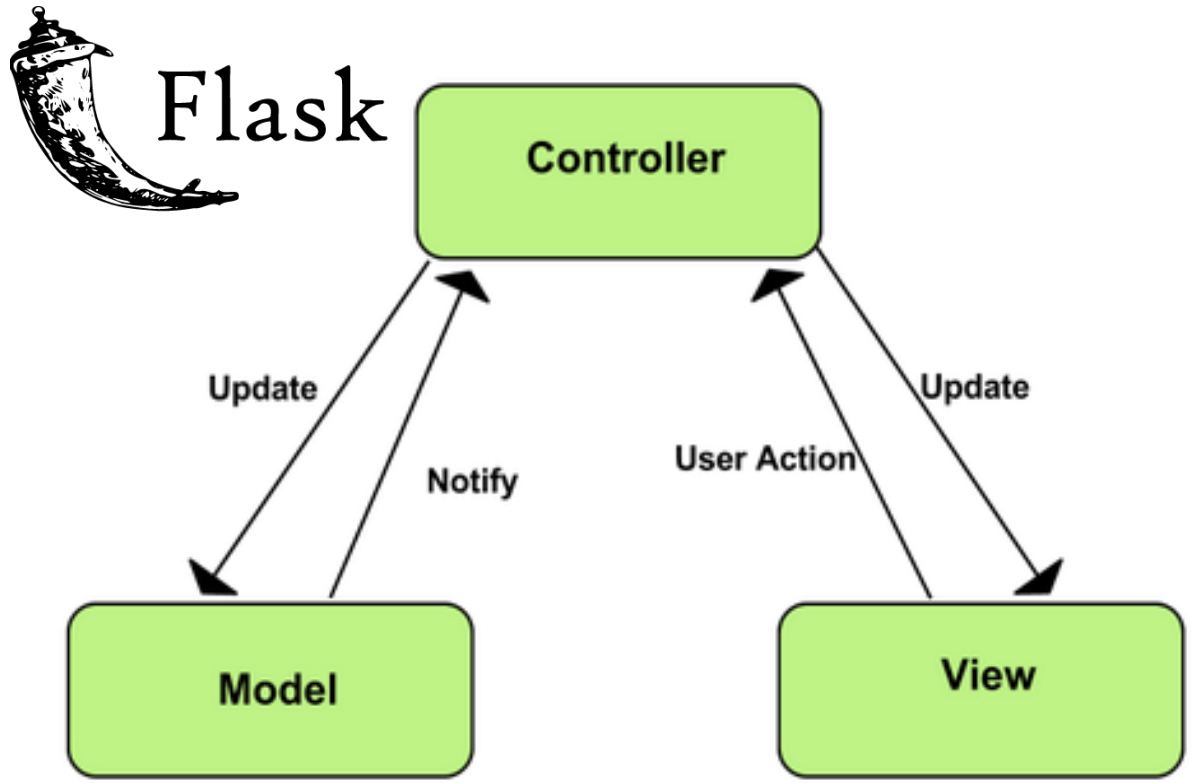
app/templates/base.ht

```
<div>
    Microblog:
    <a href="{{ url_for('index') }}">Home</a>
    <a href="{{ url_for('login') }}">Login</a>
</div>
```

MVC Model


http:// some-page
ROUTES
Flask
MODEL
CONTROLLER
VIEW
Flask SQLAlchemy
DATABASE
Jinja

MVC Model

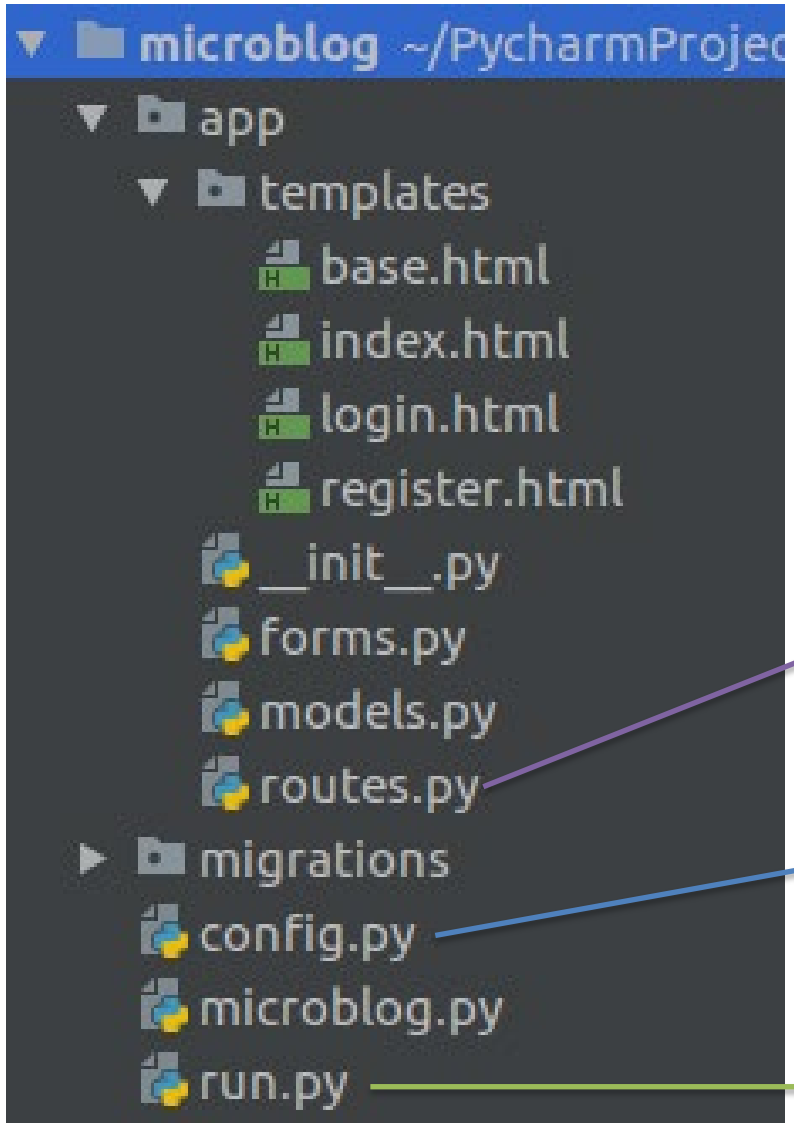


Flask

Flask SQLAlchemy

Jinja

File Structure



Handle all routes
(url request by client)

contains Config variable
throughout the whole site

Starting point

Database

Using Flask-SQLAlchemy

- Extension that provides a Flask-friendly wrapper to SQLAlchemy package
- Support different database (SQLite, MySQL, Postgre ….)

Flask + SQLAlchemy = Flask-SQLAlchemy

```
(venv) $ pip install flask-sqlalchemy
```

Define database

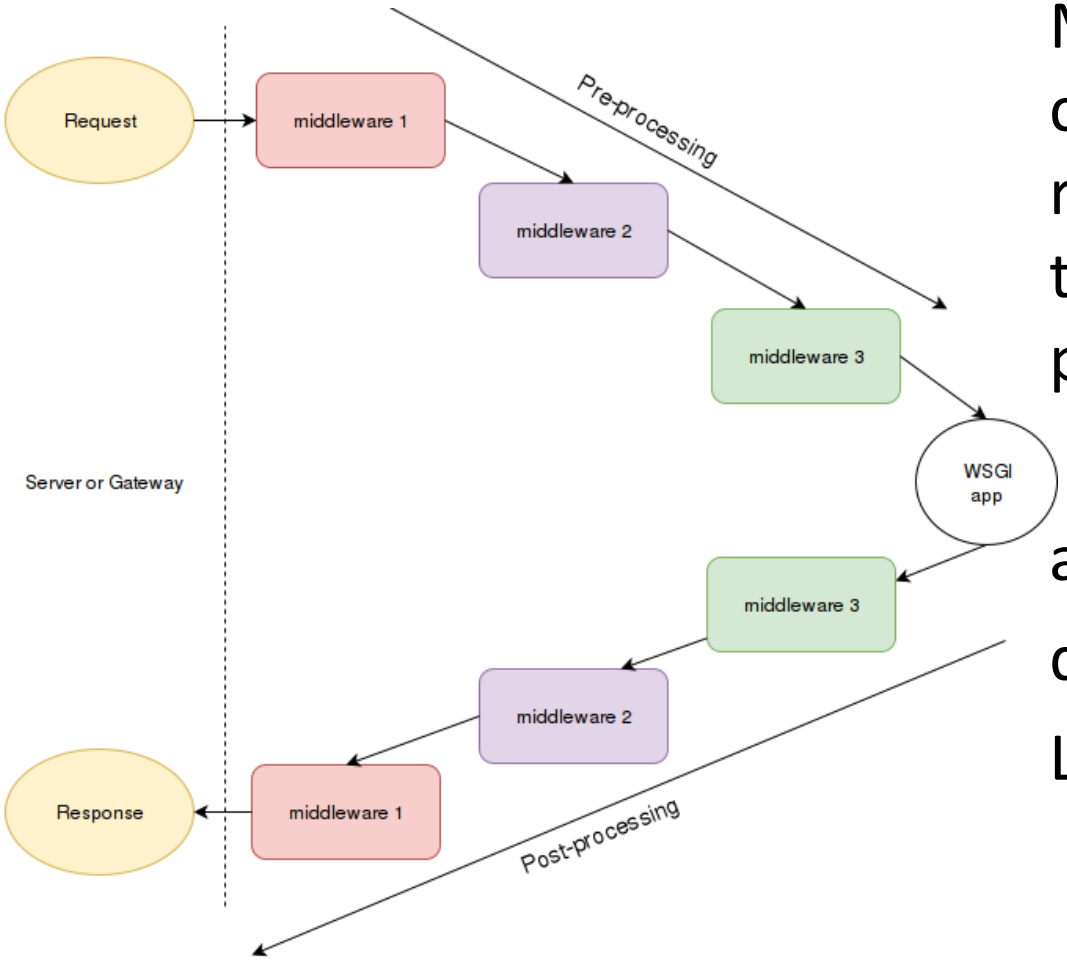
config.py: Flask-SQLAlchemy configuration

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config(object):
    # ...
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'app.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

app/__init__.py: Flask-SQLAlchemy and Flask-Migrate initialization

```
from flask import Flask
from config import Config
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

app = Flask(__name__)
app.config.from_object(Config)
db = SQLAlchemy(app)
migrate = Migrate(app, db)

from app import routes, models
```

Middleware



Middleware can be chained, allowing our response or request to go through multiple phases of processing.

app = Flask(__name__)

db = SQLAlchemy(app)

Login=LoginManager(app)

```
from werkzeug.wrappers import Request, Response, ResponseStream

class middleware():
    def __init__(self, app):
        self.app = app
        self.userName = 'Tony'
        self.password = 'IamIronMan'

    def __call__(self, environ, start_response):
        request = Request(environ)
        userName = request.authorization['username']
        password = request.authorization['password']

        # these are hardcoded for demonstration
        # verify the username and password from some database or env config variable
        if userName == self.userName and password == self.password:
            environ['user'] = { 'name': 'Tony' }
            return self.app(environ, start_response)

        res = Response(u'Authorization failed', mimetype= 'text/plain', status=401)
        return res(environ, start_response)
```

```
from flask import Flask, request
from middleware import middleware

app = Flask('DemoApp')

# calling our middleware
app.wsgi_app = middleware(app.wsgi_app)

@app.route('/', methods=['GET', 'POST'])
def hello():
    # using
    user = request.environ['user']
    return "Hi %s"%user['name']

if __name__ == "__main__":
    app.run('127.0.0.1', '5000', debug=True)
```

Database models

1 Class
1 Table

SQLAlchemy translates:
Database models (Class) -> Table

app/models.py: User database model

```
from app import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    password_hash = db.Column(db.String(128))

    def __repr__(self):
        return '<User {}>'.format(self.username)
```

Inherits from db.Model, **a base class for all models** from Flask-SQLAlchemy.

Tells Python how to print objects of this class Useful for debugging

Create Database

Creating The Migration Repository:
(venv) $ flask db init

```
(venv) $ flask db init
  Creating directory /home/miguel/microblog/migrations ... done
  Creating directory /home/miguel/microblog/migrations/versions ... done
  Generating /home/miguel/microblog/migrations/alembic.ini ... done
  Generating /home/miguel/microblog/migrations/env.py ... done
  Generating /home/miguel/microblog/migrations/README ... done
  Generating /home/miguel/microblog/migrations/script.py.mako ... done
  Please edit configuration/connection/logging settings in
  '/home/miguel/microblog/migrations/alembic.ini' before proceeding.
```

Operate Database

Whenever there is create/update to database model, two steps are a MUST:

1. Generate a migration class (Migrate)

2. Apply the migration to database (Upgrade)

Database Migration

Generate a migration class :

-m option stands for message

(venv) $ flask db migrate -m "users table"

```
(venv) $ flask db migrate -m "users table"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'user'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_user_email' on '['email']'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_user_username' on '['username']'
  Generating /home/miguel/microblog/migrations/versions/e517276bb1c2_users_table.py ... done
```

Apply the migration to database :

(venv) $ flask db upgrade

```
(venv) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> e517276bb1c2, users table
```

Database Migration Version(s)

Run “flask db migrate” 1 time, 1 migration file will be automatically generated

1st version
../microblog/migrations/versions/780739b227a7_posts_table.py

2nd version
../microblog/migrations/versions/e517276bb1c2_posts_table.py

.

.

nth version
../microblog/migrations/versions/exxxxxxsddsc2_posts_table.py

Database Upgrade and Downgrade

- flask db upgrade:
  When database schema **updated**

- flask db downgrade
  Rollback to previous version of database schema

Database Upgrade and Downgrade

```
def upgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.create_table('user',
    sa.Column('id', sa.Integer(), nullable=False),
    sa.Column('username', sa.String(length=64), nullable=True),
    sa.Column('email', sa.String(length=120), nullable=True),
    sa.Column('password_hash', sa.String(length=128), nullable=True),
    sa.PrimaryKeyConstraint('id')
    )
    op.create_index(op.f('ix_user_email'), 'user', ['email'], unique=True)
    op.create_index(op.f('ix_user_username'), 'user', ['username'], unique=True)
    # ### end Alembic commands ###


def downgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.drop_index(op.f('ix_user_username'), table_name='user')
    op.drop_index(op.f('ix_user_email'), table_name='user')
    op.drop_table('user')
    # ### end Alembic commands ###
```

User Logins

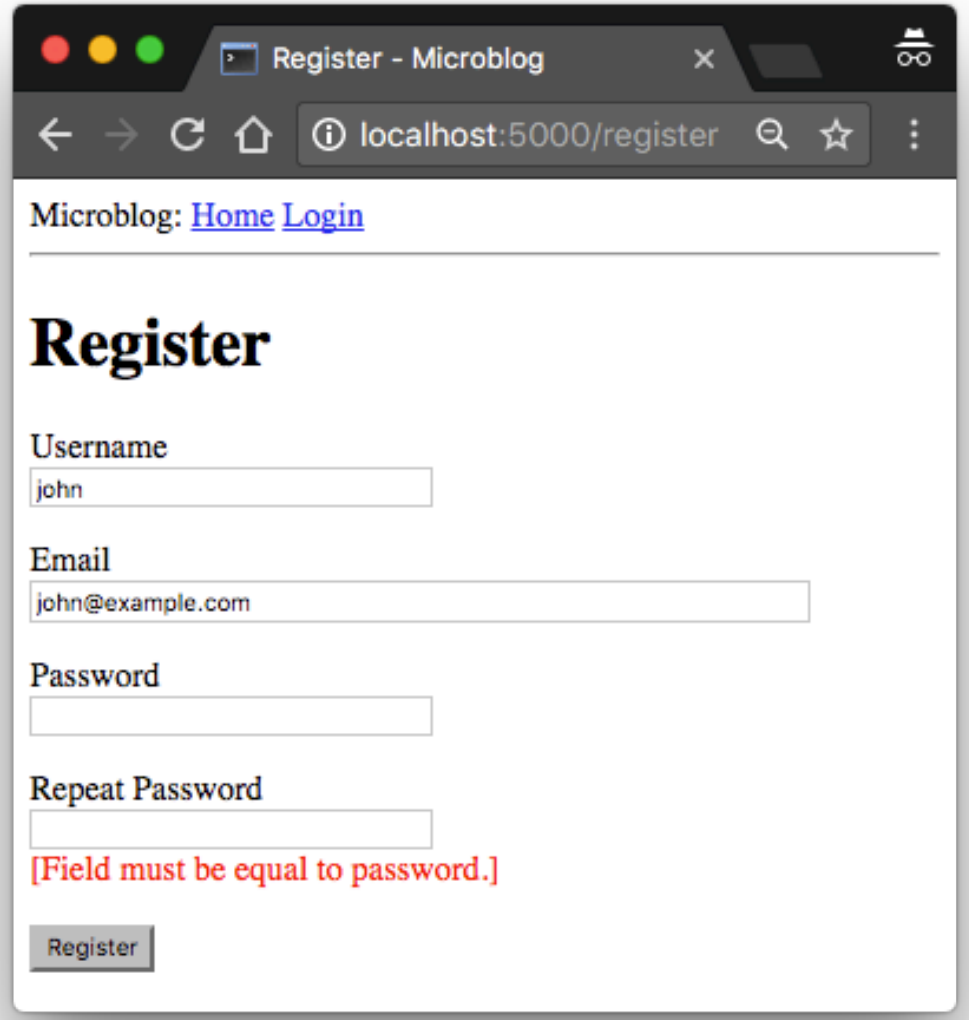
Flask-Login

- extension manages the user logged-in state
- Remember users logged in, no need keep re-login

```
(venv) $ pip install flask-login
```

app/__init__.py: Flask-Login initialization

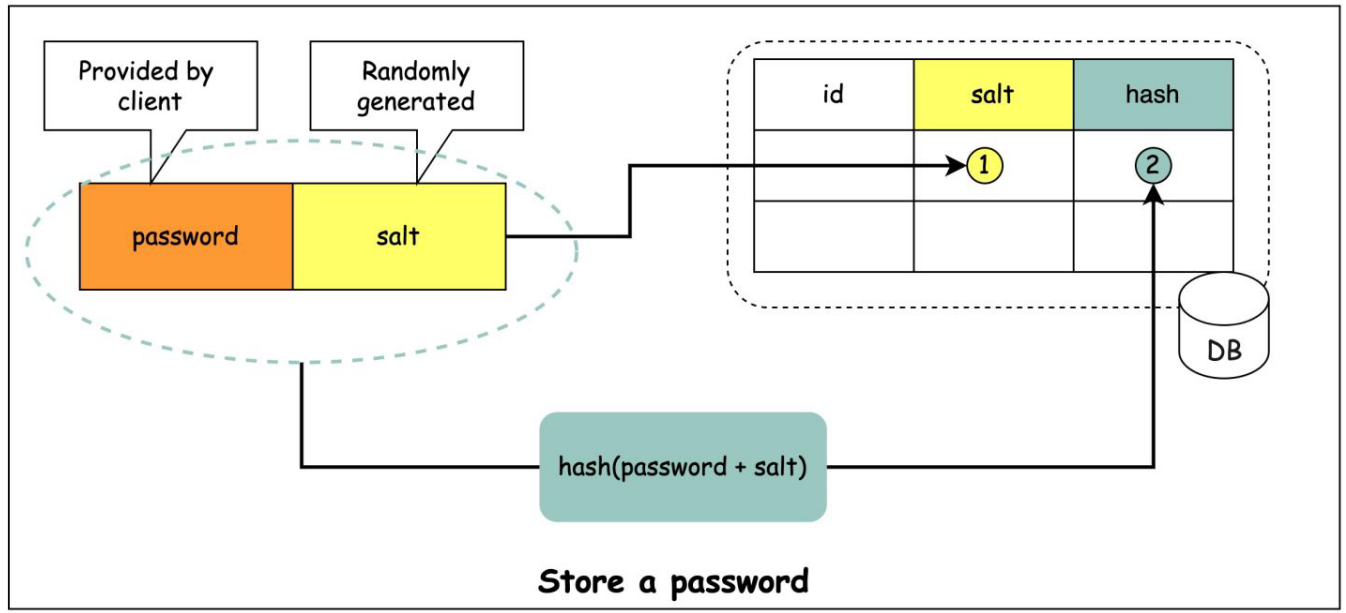
```
# ...
from flask_login import LoginManager

app = Flask(__name__)
# ...
login = LoginManager(app)

# ...
```

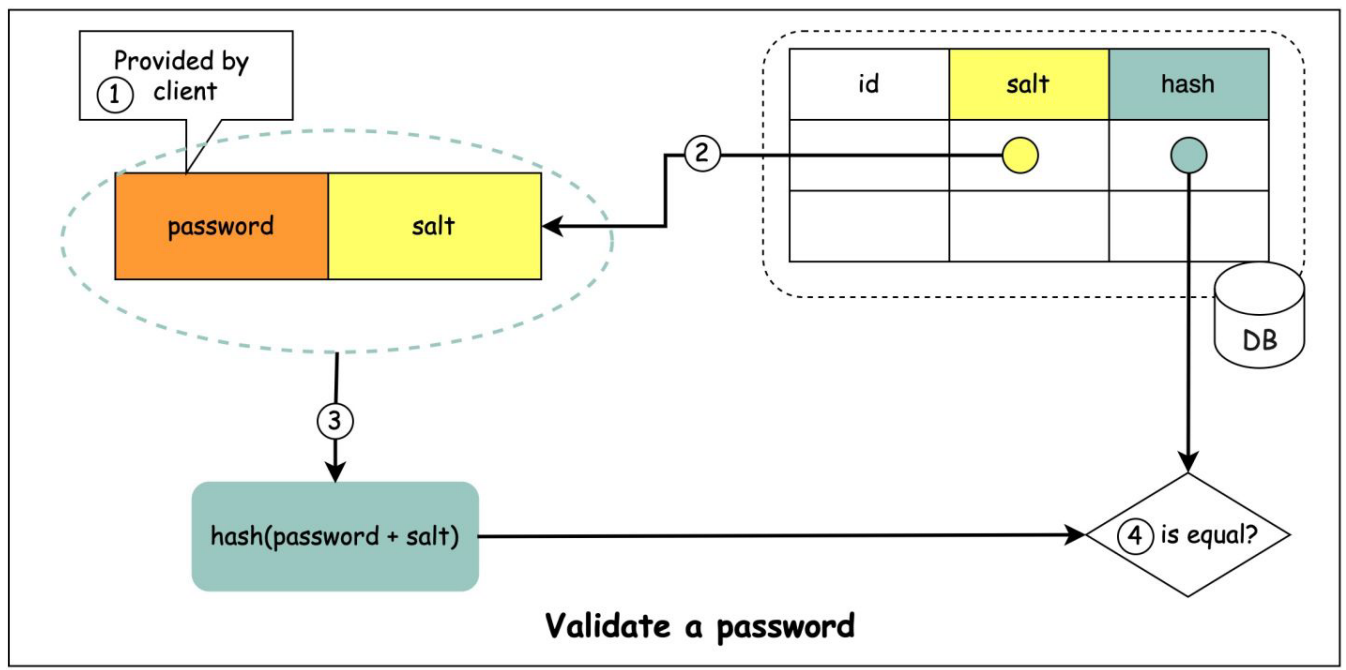


Password Hashing

How to store passwords in DB?

blog.bytebytego.com





Password Hashing

1. Generate Hash

* no known reverse operation

```
>>> from werkzeug.security import generate_password_hash
>>> hash = generate_password_hash('foobar')
>>> hash
'pbkdf2:sha256:50000$vT9fkZM8$04dfa35c6476acf7e788a1b5b3c35e217c78
dc04539d295f011f01f18cd2175f'
```

* After Hash -> Long encoded string
If same password multiple times,
Different result!!

2. Check Hash

```
>>> from werkzeug.security import check_password_hash
>>> check_password_hash(hash, 'foobar')
True
>>> check_password_hash(hash, 'barfoo')
False
```

Password Hashing

app/models.py: Password hashing and verification

```
from werkzeug.security import generate_password_hash, check_password_hash

# ...

class User(db.Model):
    # ...

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)
```

User Model for Flask-Login

The Flask-Login extension works with the application's user model with the following required items:

app/models.py: Flask-Login user mixin class

```
# ...
from flask_login import UserMixin

class User(UserMixin, db.Model):
    # ...
```

is_authenticated: a property that is True if the user has valid credentials or False otherwise.
is_active: a property that is True if the user's account is active or False otherwise.
is_anonymous: a property that is False for regular users, and True for a special, anonymous user.
get_id(): a method that returns a unique identifier for the user as a string (unicode, if using Python 2).

Keep User Logged in

Flask-Login keeps track of the logged in user by storing its unique identifier in Flask's user session, a storage space assigned to each user who connects to the application.

Store ID

app/models.py: Flask-Login user loader function

```
from app import login
# ...

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

Remember the user using **Session**

Retrieve ID

Each time the logged-in user navigates to a new page, Flask-Login retrieves the ID of the user from the session, and then loads that user into memory.

User Loader Function

Flask-Login knows nothing about databases
So we need a function to load the user.

app/models.py: Flask-Login user loader function

```
from app import login
# ...

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

@login.user_loader decorator to register user loader

id in string

Logging Users In

app/routes.py: Login view function logic

```
# ...
from flask_login import current_user, login_user
from app.models import User

# ...

@app.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.username.data).first()
        if user is None or not user.check_password(form.password.data):
            flash('Invalid username or password')
            return redirect(url_for('login'))
        login_user(user, remember=form.remember_me.data)
        return redirect(url_for('index'))
    return render_template('login.html', title='Sign In', form=form)
```

Flask-Login's login_user() function:
Register the user as logged in

Logging Users Out

app/routes.py: Logout view function

```
# ...
from flask_login import logout_user

# ...

@app.route('/logout')
def logout():
    logout_user()
    return redirect(url_for('index'))
```

Flask-Login's logout_user() function:
Logout the user

Requiring Users to Login

Flask-Login can force users to log in before they can view certain pages of the application.

Not logged in -> redirect to login form

Step1. Register view function that handles logins:
Added in app/__init__.py

```
# ...
login = LoginManager(app)
login.login_view = 'login'
```

'login’ name for the login view
(the name you would use in a url_for() call to get the URL)

Requiring Users to Login

Step2: Using decorator called @login_required to protect a certain view function

app/routes.py: @login_required decorator

```
from flask_login import login_required

@app.route('/')
@app.route('/index')
@login_required
def index():
    # ...
```

How Decorator works?

```
from functools import wraps
from flask import g, request, redirect, url_for

def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if g.user is None:
            return redirect(url_for('login', next=request.url))
        return f(*args, **kwargs)
    return decorated_function

@app.route('/secret_page')
@login_required
def secret_page():
    pass
```

A lot of useful cases:

- Caching
- Logging
- Before Request
- Error handler

https://flask.palletsprojects.com/en/1.1.x/patterns/viewdecorators/

“Next” Query String

app/routes.py: @login_required decorator

```
from flask_login import login_required

@app.route('/')
@app.route('/index')
@login_required
def index():
    # ...
```

app/routes.py: Redirect to "next" page

```
from flask import request
from werkzeug.urls import url_parse

@app.route('/login', methods=['GET', 'POST'])
def login():
    # ...
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.username.data).first()
        if user is None or not user.check_password(form.password.data):
            flash('Invalid username or password')
            return redirect(url_for('login'))
        login_user(user, remember=form.remember_me.data)
        next_page = request.args.get('next')
        if not next_page or url_parse(next_page).netloc != '':
            next_page = url_for('index')
        return redirect(next_page)
    # ...
```

It will also add a query string argument to this URL, making the complete redirect URL /login?next=/index. The next query string argument is set to the original URL, so the application can use that to redirect back after login.

Showing The Logged In User in Templates

app/routes.py: Do not pass user to template anymore

```
@app.route('/')
@app.route('/index')
def index():
    # ...
    return render_template("index.html", title='Home Page', posts=posts)
```

user template argument NO longer needed!!

app/templates/index.html: Pass current user to template

```
{% extends "base.html" %}

{% block content %}
    <h1>Hi, {{ current_user.username }}!</h1>
    {% for post in posts %}
    <div><p>{{ post.author.username }} says: <b>{{ post.body }}</b></p></div>
    {% endfor %}
{% endblock %}
```

Flask-Login's current_user can be used to represent logged in User

Different UI
upon logged in / anonymous Users

有Login 同 無Login
可以出唔同介面

Flask-Login‘s current_user – is_anonymous property can be used to generate the correct URL

app/templates/base.html: User profile template

```
<div>
  Microblog:
  <a href="{{ url_for('index') }}">Home</a>
  {% if current_user.is_anonymous %}
  <a href="{{ url_for('login') }}">Login</a>
  {% else %}
  <a href="{{ url_for('user', username=current_user.username) }}">Profile</a>
  <a href="{{ url_for('logout') }}">Logout</a>
  {% endif %}
</div>
```

User Registration Form Example

app/forms.py: User registration form

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import ValidationError, DataRequired, Email, EqualTo
from app.models import User

# ...

class RegistrationForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    email = StringField('Email', validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired()])
    password2 = PasswordField(
        'Repeat Password', validators=[DataRequired(), EqualTo('password')])
    submit = SubmitField('Register')

    def validate_username(self, username):
        user = User.query.filter_by(username=username.data).first()
        if user is not None:
            raise ValidationError('Please use a different username.')

    def validate_email(self, email):
        user = User.query.filter_by(email=email.data).first()
        if user is not None:
            raise ValidationError('Please use a different email address.')
```

ValidationError

app/forms.py: User registration form

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import ValidationError, DataRequired, Email, EqualTo
from app.models import User

# ...

class RegistrationForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    email = StringField('Email', validators=[DataRequ
    password = PasswordField('Password', validators=[
    password2 = PasswordField(
        'Repeat Password', validators=[DataRequired()
    submit = SubmitField('Register')

    def validate_username(self, username):
        user = User.query.filter_by(username=username.data).first()
        if user is not None:
            raise ValidationError('Please use a different username.')

    def validate_email(self, email):
        user = User.query.filter_by(email=email.data).first()
        if user is not None:
            raise ValidationError('Please use a different email address.')
```

A validation error is triggered by raising ValidationError. And display message to user

```
<p>
    {{ form.username.label }}<br>
    {{ form.username(size=32) }}<br>
    {% for error in form.username.errors %}
    <span style="color: red;">[{{ error }}]</span>
    {% endfor %}
</p>
```

User Profile

** dynamic component surround by '<' & '>'

view function that maps to the /user/<username> URL

app/routes.py: User profile view function

```
@app.route('/user/<username>')
@login_required
def user(username):
    user = User.query.filter_by(username=username).first_or_404()
    posts = [
        {'author': user, 'body': 'Test post #1'},
        {'author': user, 'body': 'Test post #2'}
    ]
    return render_template('user.html', user=user, posts=posts)
```

If URL : /user/susan
-> username = 'susan'

User Profile

app/routes.py: User profile view function

```
@app.route('/user/<username>')
@login_required
def user(username):
    user = User.query.filter_by(username=username).first_or_404()
    posts = [
        {'author': user, 'body': 'Test post #1'},
        {'author': user, 'body': 'Test post #2'}
    ]
    return render_template('user.html', user=user, posts=posts)
```

first_or_404():
Case1 – (with result): works exactly like first()
Case2 – (NO result):
Automatically sends a 404 error back to the client.

MD5 Hash

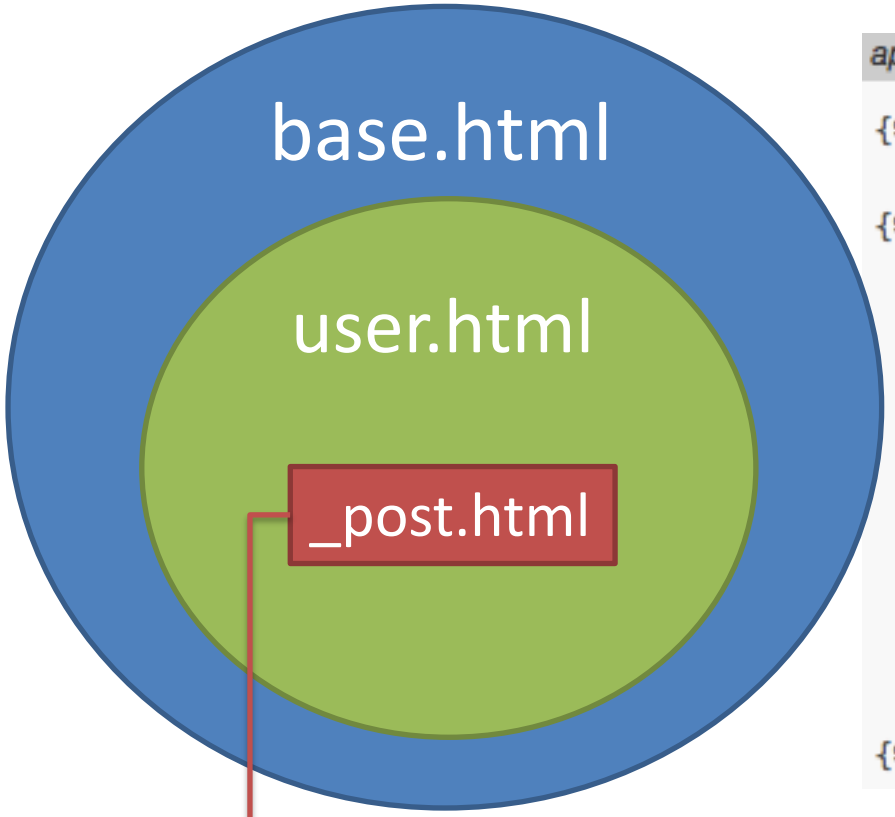
- Hash function accepts sequence of bytes
- returns 128 bit hash value
- usually used to check data integrity

app/models.py: User avatar URLs

```
from hashlib import md5
# ...

class User(UserMixin, db.Model):
    # ...
    def avatar(self, size):
        digest = md5(self.email.lower().encode('utf-8')).hexdigest()
        return 'https://www.gravatar.com/avatar/{}?d=identicon&s={}'.format(
            digest, size)
```

MD5 support in Python works on bytes and not on strings

Jinja2 Sub-Templates



app/templates/user.html: User avatars in posts

```
{% extends "base.html" %}

{% block content %}
    <table>
        <tr valign="top">
            <td><img src="{{ user.avatar(128) }}"></td>
            <td><h1>User: {{ user.username }}</h1></td>
        </tr>
    </table>
    <hr>
    {% for post in posts %}
        {% include '_post.html' %}
    {% endfor %}
{% endblock %}
```

A sub-template is also HTML markup page

Jinja2 Sub-Templates

You can use _ prefix as a naming convention to recognize sub-templates

_post.html

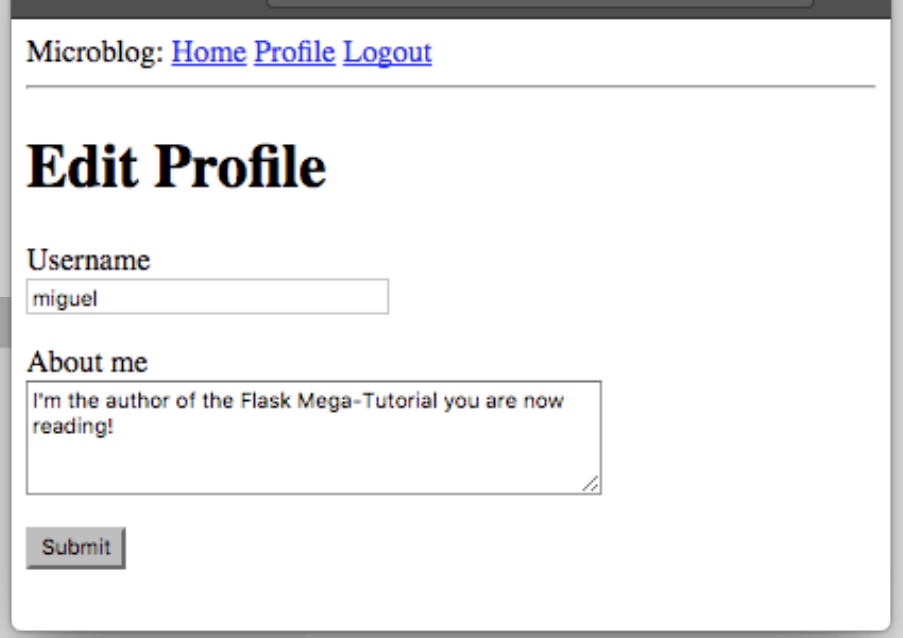
app/templates/user.html: User avatars in posts

```
{% extends "base.html" %}

{% block content %}
    <table>
        <tr valign="top">
            <td><img src="{{ user.avatar(128) }}"></td>
            <td><h1>User: {{ user.username }}</h1></td>
        </tr>
    </table>
    <hr>
    {% for post in posts %}
        {% include '_post.html' %}
    {% endfor %}
{% endblock %}
```

app/templates/_post.html: Post sub-template

```
<table>
    <tr valign="top">
        <td><img src="{{ post.author.avatar(36) }}"></td>
        <td>{{ post.author.username }} says:<br>{{ post.body }}</td>
    </tr>
</table>
```

Profile Editor



app/forms.py: Profile editor form

```
from wtforms import StringField, TextAreaField, SubmitField
from wtforms.validators import DataRequired, Length

# ...

class EditProfileForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    about_me = TextAreaField('About me', validators=[Length(min=0, max=140)])
    submit = SubmitField('Submit')
```

Length() to set the text entered is between 0 and 140 characters

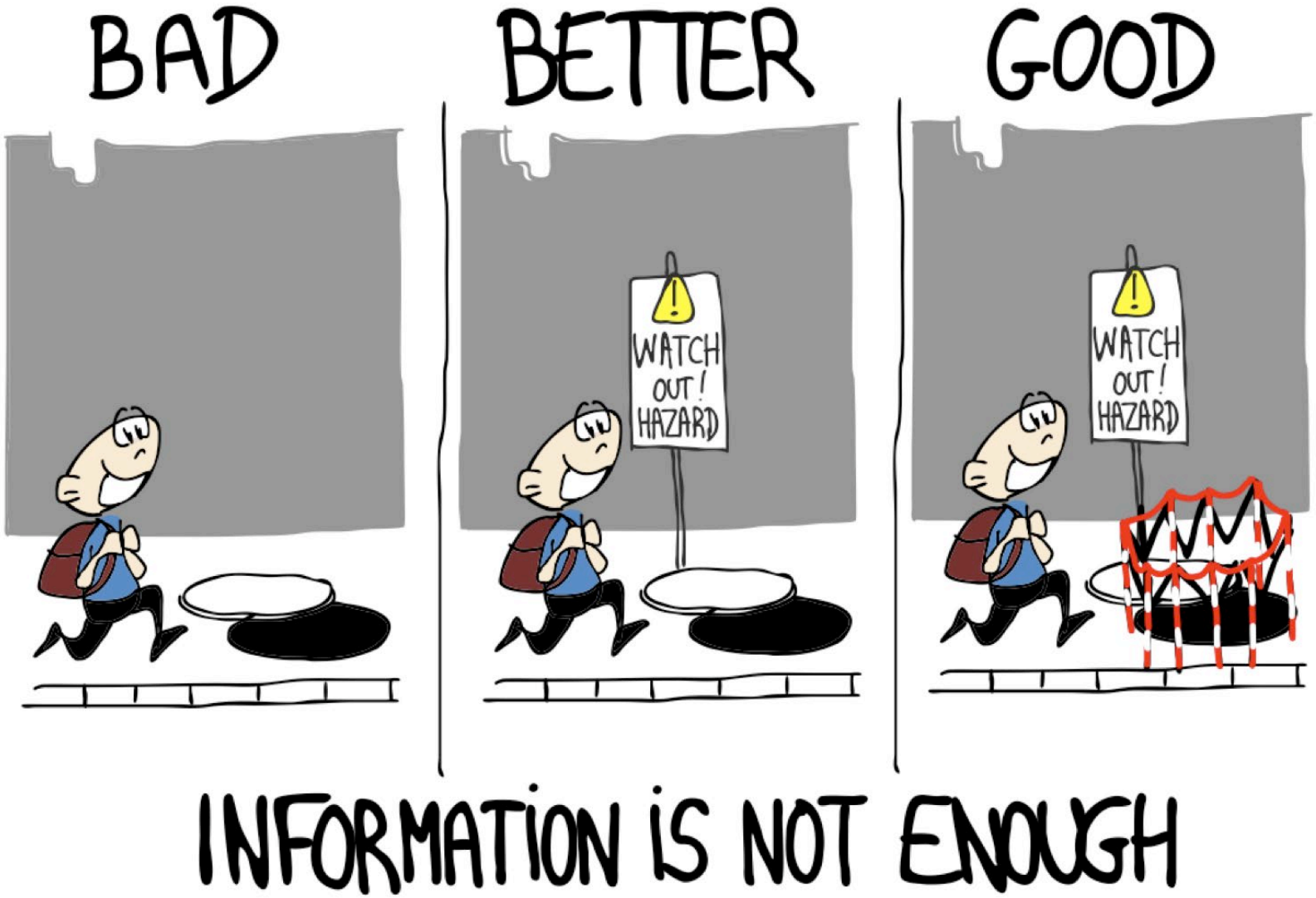
Flask-WTF

- TextAreaField(….[Length(max=0, max=140)])

HTML5

- <textarea maxlength="140" minlength="0"></textarea>

Error Handling



Debug Mode

The Best Practice is to activate debug mode, by set environment variable (NOT hardcode in source code):

1) Setting in Pycharm run configuration:

or

2) Run in Terminal

```
(venv) $ export FLASK_DEBUG=1
```

Debug Mode supports reloader feature:
Any time you save a file, the application will restart to pick up the new code

Custom Error Page

app/errors.py: Custom error handlers

```
from flask import render_template
from app import app, db

@app.errorhandler(404)
def not_found_error(error):
    return render_template('404.html'), 404

@app.errorhandler(500)
def internal_error(error):
    db.session.rollback()
    return render_template('500.html'), 500
```

Custom Error Page

app/errors.py: Custom error handlers

```
from flask import render_template
from app import app, db

@app.errorhandler(404)
def not_found_error(error):
    return render_template('404.html'), 404

@app.errorhandler(500)
def internal_error(error):
    db.session.rollback()
    return render_template('500.ht
```

app/templates/404.html: Not found error template

```
{% extends "base.html" %}

{% block content %}
    <h1>File Not Found</h1>
    <p><a href="{{ url_for('index') }}">Back</a></p>
{% endblock %}
```

Custom 404 Error Page

app/templates/500.html: Internal server error template

```
{% extends "base.html" %}

{% block content %}
    <h1>An unexpected error has occurred</h1>
    <p>The administrator has been notified. Sorry for the inco
    <p><a href="{{ url_for('index') }}">Back</a></p>
{% endblock %}
```

Custom 500 Error Page

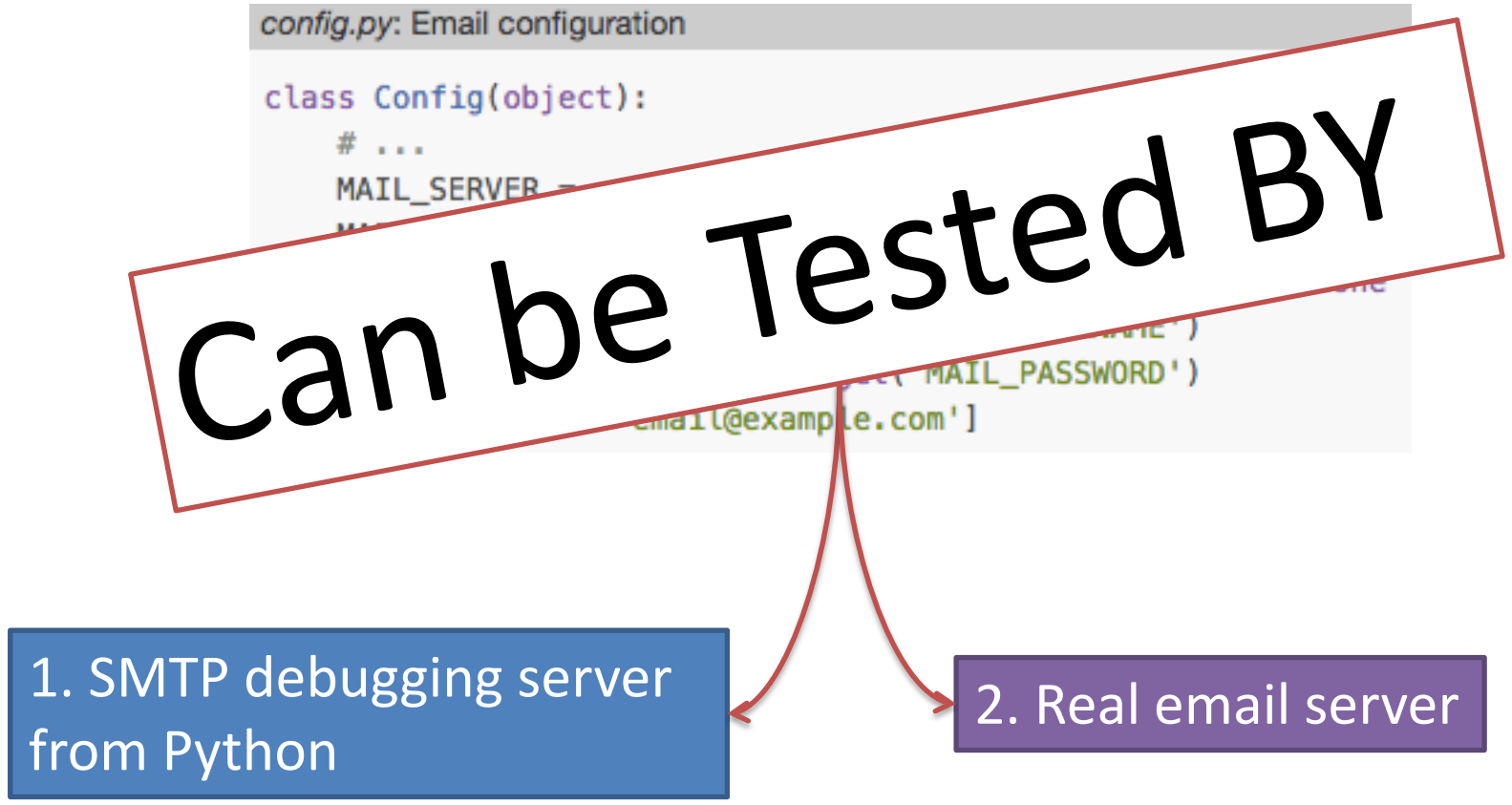
Sending Error By Emails

Need setup email server details to the configuration file

config.py: Email configuration

```
class Config(object):
    # ...
    MAIL_SERVER = os.environ.get('MAIL_SERVER')
    MAIL_PORT = int(os.environ.get('MAIL_PORT') or 25)
    MAIL_USE_TLS = os.environ.get('MAIL_USE_TLS') is not None
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD')
    ADMINS = ['your-email@example.com']
```

Sending Error By Emails



Sending Error By Emails

app/__init__.py: Log errors by email

```
import logging
from logging.handlers import SMTPHandler

# ...

if not app.debug:
    if app.config['MAIL_SERVER']:
        auth = None
        if app.config['MAIL_USERNAME'] or app.config['MAIL_PASSWORD']:
            auth = (app.config['MAIL_USERNAME'], app.config['MAIL_PASSWORD'])
        secure = None
        if app.config['MAIL_USE_TLS']:
            secure = ()
        mail_handler = SMTPHandler(
            mailhost=(app.config['MAIL_SERVER'], app.config['MAIL_PORT']),
            fromaddr='no-reply@' + app.config['MAIL_SERVER'],
            toaddrs=app.config['ADMINS'], subject='Microblog Failure',
            credentials=auth, secure=secure)
        mail_handler.setLevel(logging.ERROR)
        app.logger.addHandler(mail_handler)
```

Need Logging package

Need SMTPHandler instance to the Flask logger object (app.logger)

Sending Error By Emails

app/__init__.py: Log errors by email

```
import logging
from logging.handlers import SMTPHandler

# ...

if not app.debug:
    if app.config['MAIL_SERVER']:
        auth = None
        if app.config['MAIL_USERNAME'] or app.config['MAIL_PASSWORD']:
            auth = (app.config['MAIL_USERNAME'], app.config['MAIL_PASSWORD'])
        secure = None
        if app.config['MAIL_USE_TLS']:
            secure = ()
        mail_handler = SMTPHandler(
            mailhost=(app.config['MAIL_SERVER'], app.config['MAIL_PORT']),
            fromaddr='no-reply@' + app.config['MAIL_SERVER'],
            toaddrs=app.config['ADMINS'], subject='Microblog Failure',
            credentials=auth, secure=secure)
        mail_handler.setLevel(logging.ERROR)
        app.logger.addHandler(mail_handler)
```

ONLY send email when it is NOT debug mode

Logging to a File

app/__init__.py: Logging to a file

```
# ...
from logging.handlers import RotatingFileHandler
import os

# ...

if not app.debug:
    # ...

    if not os.path.exists('logs'):
        os.mkdir('logs')
    file_handler = RotatingFileHandler('logs/microblog.log', maxBytes=10240,
                                       backupCount=10)
    file_handler.setFormatter(logging.Formatter(
        '%(asctime)s %(levelname)s: %(message)s [in %(pathname)s:%(lineno)d]'))
    file_handler.setLevel(logging.INFO)
    app.logger.addHandler(file_handler)

    app.logger.setLevel(logging.INFO)
    app.logger.info('Microblog startup')
```

Auto create log folder if it doesn't exist

Log Rotate

app/__init__.py: Logging to a file

```
# ...
from logging.handlers import RotatingFileHandler
import os

# ...

if not app.debug:
    # ...

    if not os.path.exists('logs'):
        os.mkdir('logs')
    file_handler = RotatingFileHandler('logs/microblog.log', maxBytes=10240,
                                       backupCount=10)
    file_handler.setFormatter(logging.Formatter(
        '%(asctime)s %(levelname)s: %(message)s [in %(pathname)s:%(lineno)d]'))
    file_handler.setLevel(logging.INFO)
    app.logger.addHandler(file_handler)

    app.logger.setLevel(logging.INFO)
    app.logger.info('Microblog startup')
```

RotatingFileHandler class allows inputing
1) maxBytes -> Max Size of log file
2) backupCount -> Num backup log file kept

size of the log file is 10KB, only keeping last ten log files as backup

Log Format

app/__init__.py: Logging to a file

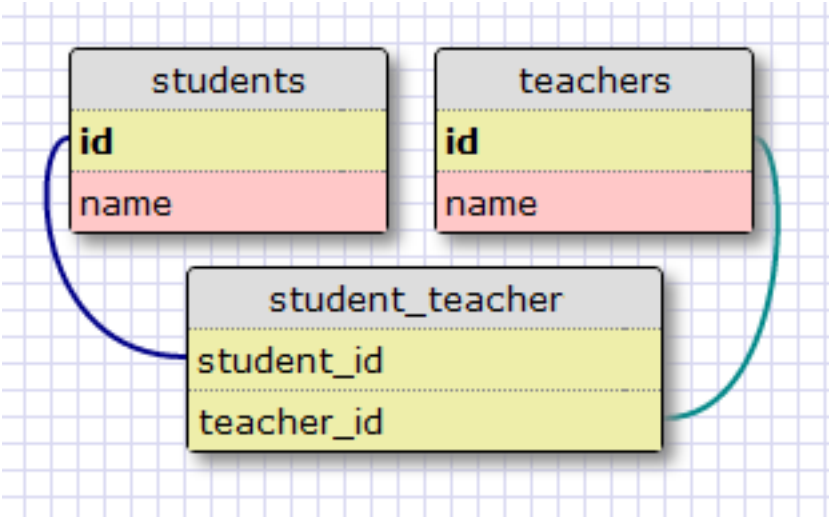
```
# ...
from logging.handlers import RotatingFileHandler
import os

# ...

if not app.debug:
    # ...

    if not os.path.exists('logs'):
        os.mkdir('logs')
    file_handler = RotatingFileHandler('logs/microblog.log', maxBytes=10240,
                                       backupCount=10)
    file_handler.setFormatter(logging.Formatter(
        '%(asctime)s %(levelname)s: %(message)s [in %(pathname)s:%(lineno)d]'))
    file_handler.setLevel(logging.INFO)
    app.logger.addHandler(file_handler)

    app.logger.setLevel(logging.INFO)
    app.logger.info('Microblog startup')
```

logging.Formatter class provides custom formatting

Logging Level
DEBUG < INFO < WARNING < ERROR < CRITICAL
increasing order of severity

Many-Many Relationship

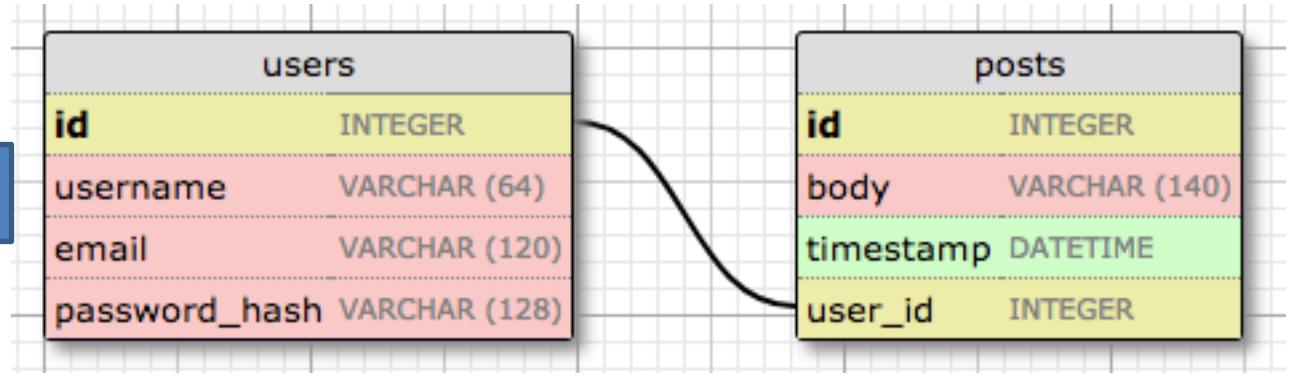
Many – Many
(Students - Teachers)



We need an Association Table “student_teacher”

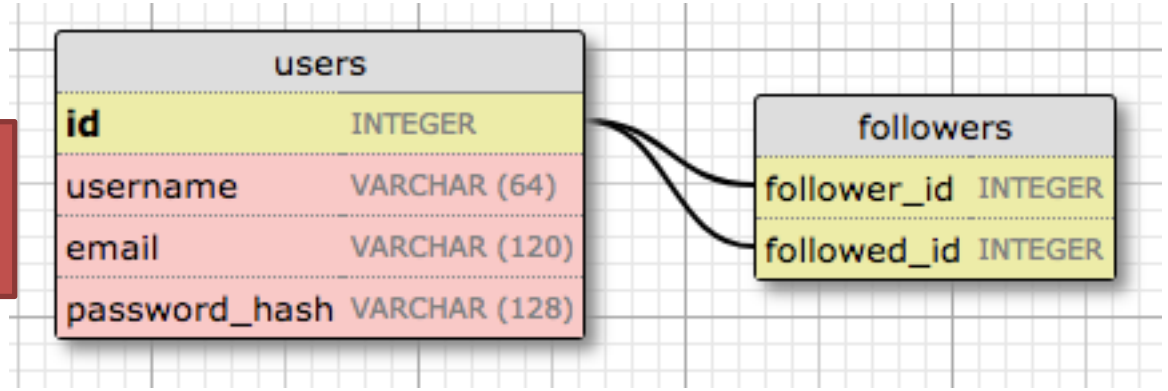
Followers Relationship

1 - Many



Relationship in a Blog

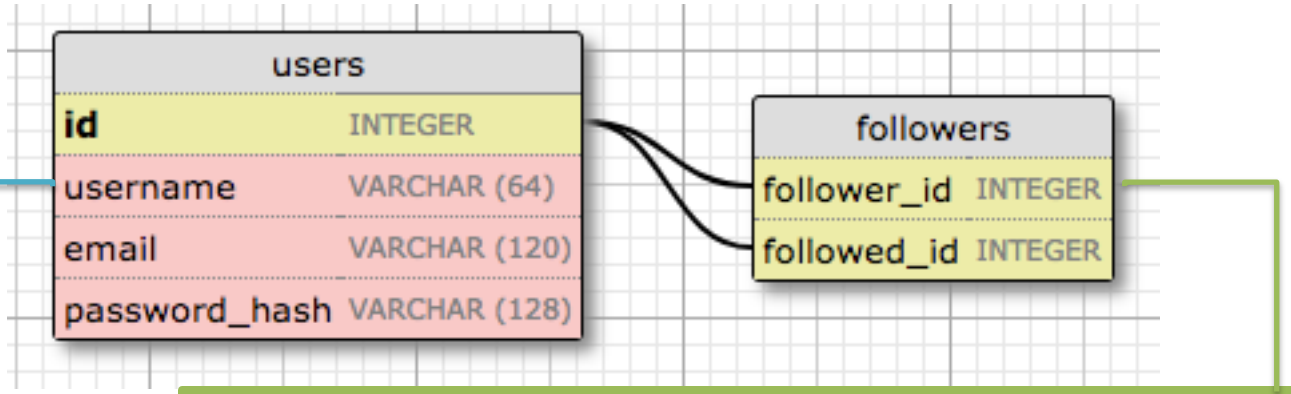
Many – Many
(self-referential)



Followers Relationship

app/models.py: Many-to-many followers relationship

```
class User(UserMixin, db.Model):
    # ...
    followed = db.relationship(
        'User', secondary=followers,
        primaryjoin=(followers.c.follower_id == id),
        secondaryjoin=(followers.c.followed_id == id),
        backref=db.backref('followers', lazy='dynamic'), lazy='dynamic')
```



app/models.py: Followers association table

```
followers = db.Table('followers',
    db.Column('follower_id', db.Integer, db.ForeignKey('user.id')),
    db.Column('followed_id', db.Integer, db.ForeignKey('user.id'))
)
```

Unit Testing the User Model

unittest package from Python
-> Write & Execute unittest

tests.py: User model unit tests.

```
from datetime import datetime, timedelta
import unittest
from app import app, db
from app.models import User, Post

class UserModelCase(unittest.TestCase):
    def setUp(self):
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'
        db.create_all()

    def tearDown(self):
        db.session.remove()
        db.drop_all()

    def test_password_hashing(self):
        u = User(username='susan')
        u.set_password('cat')
        self.assertFalse(u.check_password('dog'))
        self.assertTrue(u.check_password('cat'))
```

* Prevent using production database

Using sqlite://
To get SQLAlchemy to use an in-memory SQLite database

Unit Testing the User Model

test.py

```
def test_password_hashing(self):
    u = User(username='susan')
    u.set_password('cat')
    self.assertFalse(u.check_password('dog'))
    self.assertTrue(u.check_password('cat'))

def test_avatar(self):
    u = User(username='john', email='john@example.com')
    self.assertEqual(u.avatar(128), ('https://www.gravatar.com/avatar/'
                                     'd4c74594d841139328695756648b6bd6'
                                     '?d=identicon&s=128'))

def test_follow(self):
    u1 = User(username='john', email='john@example.com')
```

.
.
.

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

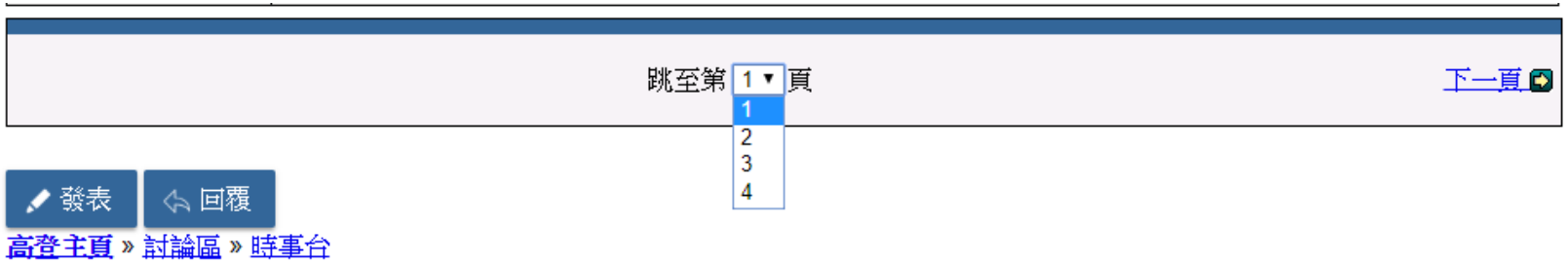
UnitTest Result:

```
(venv) $ python tests.py
test_avatar (__main__.UserModelCase) ... ok
test_follow (__main__.UserModelCase) ... ok
test_follow_posts (__main__.UserModelCase) ... ok
test_password_hashing (__main__.UserModelCase) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.494s

OK
```

Pagination

What happens if a post with a thousand replies? Or even a million?

A large list of posts or replies will be extremely slow and inefficient



So, pagination!

Pagination with Flask

Flask-SQLAlchemy supports pagination with the paginate() query method, can be called on any query object :

```
paginate(page=None, per_page=None, error_out=True, max_per_page=None)
```

- **page**: the page number, starting from 1
- **per_page**: the number of items per page
- **error_out**: An error flag
  True, when an out of range page is requested a 404 error will be returned to the client
  False, page and per_page default to 1 and 20

Get first twenty followed posts of the user

```
>>> user.followed_posts().paginate(1, 20, False).items
```

The return value from paginate is a Pagination object. The items attribute of this object contains the list of items in the requested page.

Pagination with Flask

- Page 1: *http://localhost:5000/index?page=1*
- Page 2: *http://localhost:5000/index?page=2*
- Page 3: *http://localhost:5000/index?page=3*

app/routes.py: Followers association table

```
@app.route('/', methods=['GET', 'POST'])
@app.route('/index', methods=['GET', 'POST'])
@login_required
def index():
    # ...
    page = request.args.get('page', 1, type=int)
    posts = current_user.followed_posts().paginate(
        page, app.config['POSTS_PER_PAGE'], False)
    return render_template('index.html', title='Home', form=form,
                           posts=posts.items)

@app.route('/explore')
@login_required
def explore():
    page = request.args.get('page', 1, type=int)
    posts = Post.query.order_by(Post.timestamp.desc()).paginate(
        page, app.config['POSTS_PER_PAGE'], False)
    return render_template("index.html", title='Explore', posts=posts.items)
```

get(key, default=None, type=None)

Pagination with Flask

config.py: Posts per page configuration.

```
class Config(object):
    # ...
    POSTS_PER_PAGE = 3
```

3 is just an example, pratically should be larger

app/routes.py: Followers association table

```
@app.route('/', methods=['GET', 'POST'])
@app.route('/index', methods=['GET', 'POST'])
@login_required
def index():
    # ...
    page = request.args.get('page', 1, type=int)
    posts = current_user.followed_posts().paginate(
        page, app.config['POSTS_PER_PAGE'], False)
    return render_template('index.html', title='Home', form=form,
                           posts=posts.items)

@app.route('/explore')
@login_required
def explore():
    page = request.args.get('page', 1, type=int)
    posts = Post.query.order_by(Post.timestamp.desc()).paginate(
        page, app.config['POSTS_PER_PAGE'], False)
    return render_template("index.html", title='Explore', posts=posts.items)
```

- Good idea to have these application-wide variable in the configuration file
- Easier to adjust and test

Page Navigation

app/routes.py: Next and previous page links.

```
@app.route('/', methods=['GET', 'POST'])
@app.route('/index', methods=['GET', 'POST'])
@login_required
def index():
    # ...
    page = request.args.get('page', 1, type=int)
    posts = current_user.followed_posts().paginate(
        page, app.config['POSTS_PER_PAGE'], False)
    next_url = url_for('index', page=posts.next_num) \
        if posts.has_next else None
    prev_url = url_for('index', page=posts.prev_num) \
        if posts.has_prev else None
    return render_template('index.html', title='Home', form=form,
                           posts=posts.items, next_url=next_url,
                           prev_url=prev_url)

@app.route('/explore')
@login_required
def explore():
    page = request.args.get('page', 1, type=int)
    posts = Post.query.order_by(Post.timestamp.desc()).paginate(
        page, app.config['POSTS_PER_PAGE'], False)
    next_url = url_for('explore', page=posts.next_num) \
        if posts.has_next else None
    prev_url = url_for('explore', page=posts.prev_num) \
        if posts.has_prev else None
    return render_template("index.html", title='Explore', posts=posts.items,
                           next_url=next_url, prev_url=prev_url)
```

- **has_next**: True if there is at least one more page after the current one
- **has_prev**: True if there is at least one more page before the current one

- **next_num**: page number for the next page
- **prev_num**: page number for the previous page