FLASK

Flask ist eine «Klasse», die man importiert from flask import Flask

app= Flask(\_\_name\_\_), app ist Variable für Flask-App.

\_\_name\_\_=Name des Models in Python, nicht Name der Datei, (damit flask weiss, wo Sachen sind.)

routes= was man im Browser eingibt.

@app.route(routes) ist Dekorater, der Backend Sachen managet und erlaubt uns Funktion zu schreiben, die Info(s) zurückgibt, die auf Website angezeigt werden, die für diese Route ist.

Diese Funktion wird oft als Applikation bezeichnet, wir erstellen viele Applikationen, die man sich aber nicht denken soll wie Applikationen auf dem Handy. Applikationen definieren die Logik und die Funktionalität der Webseiten. Z.B. Ein Link einer Webseite wird durch eine Funktion festgelegt, auf welche Seite der Link führt. (Rückgabewert der Applikation ist die Webseite auf die man von diesem Link gelangt. Default Rückgabewert der Applikation ist immer die Webseite, die bei Eingabe der Routes erscheint)

*Variante 1:*

Environment Variabel muss gesetzt werden: export FLASK\_APP= name.py (name z.b. flaskapp). Danach kann man Flask App starten mit flask run.

Debugg Variable setzen mit export FLASK\_DEBUG=1, dann werden Änderungen sichtbar

*Variante 2:*

if \_\_name\_\_= ‘\_\_main\_\_’: hier wird \_\_name\_\_=\_\_main\_\_

app.run(debug=True)

Starten über python name.py (Debugg schon gesetzt)

* Flask App startet einen Webserver, welcher über Port 5000 angesprochen wird. (localhost)

Für routes werden Templates erstellt, welche Front End Infos haben.

Die Applikation kann über die Fkt. render\_template (import render\_template) das hat html.Template dem Webserver zur Verfügung stellen.

Braucht man Flask nur für reine statische Front End Anwendung (html, css) ist dies die Hauptanwendung. (default methode=’GET’)

Heute ist eigentlich jede Webseite eine Webapplikation (nicht nur statische Infos), mit denen man interagiert, Posts und Updates, etc.

return render\_template( html-Datei, var1= Daten/Konstanten/Variablen, var2= Daten/Konstanten/Variablen, .... var\_n= Daten/Konstanten/Variablen)

Daten/Konstanten/Variablen vom Backend (Datenbank), welche man im FrontEnd anzeigen will)

In Template wird var1 – var\_n verbaut mit der Template Engine von Flask (Ginger2)

Code Block: {% ....python code...for v in var…if var\_n… %}

html….Variable: {{ v.titel }}…html

Layout Template: {% extends html %}, {% block content %},{% endblock content %}

url\_for Funktion (import url\_for): Findet exakten Ort von Routes.

1. html

href=’’{{ url\_for (‘static’, filename=’main.css’) }} main.css ist in Projektordner

1. html

href=’’{{ url\_for (‘NameApplikation’ var1}} def NameApplikation():

1. backend

return redirect (url\_for(‘NameApplikation’), var1)

1. backend

variable= url\_for (‘ordnername’, filename=’pfad der datei’)

Für Login/Register ist Klasse FlaskFormen(Extensions) in Flask geeignet (install flask-wtf, import FlaskForm) -> forms.py

(Unter-)Klassen wie StringField, IntegerField, TextField, Password Field, SubmitField sind Repräsentationen der FlaskForm, werden automatisch in html-Ausdruck umgewandelt im Template. Erster Parameter dieser Klassen (z.B. Username) ist Label in HTML. Argument validators hat eine Liste von UnterUnterKlassen wie: DataRequired()gibt es Eingabe?, Länge , EqualTo, etc... Eignet sich alles sich gut für Login/Register.

Kann mit form.validate\_on\_submit() geprüft werden

Für diese Formen braucht man einen Secret-Key, der die Registration/Login schützt (Cookies etc)

Instanz dieser Form: form= RegistrationForm() form= LoginForm()

form in Template einfügen -> <form method=»POST», action=»»> leer: wenn man form postet, wird Info zur selben route gepostet (geht’s zurück zur selben route.)

{{ form. hidden\_tag() }} (nicht so wichtig..teil der sicherheit, wie secret-key)

Felder der form Klasse werden eingefügt.

{{form.username.label}}

{{form.username}} Eingabefeld.

{% if form.username.errors%}->Eingabefehler werden angezeigt

{{form.submit(class=buttom)}}

Redirect Fkt: was geschieht nachdem eine Funktion in einer Applikation ausgeführt wurde. (Applikation führt mit return render-template html aus, wenn auf link in html und alles OK ist in Funktion, return redirect(url\_for(NameApplikation )führt zu einer neuen Seite), ist ein Interface zwischen Webserver und Flask.

@app.route(routes, methode=’GET’, POST’) ohne Post gibt es method not Allowed Fehler. Submitting Post request. Um Post request zu erlauben, muss man erlaubte Methoden in Route anpassen.

ORM -Datenbank:SQL Alchemy (sql\_lite DB für die Entwicklung). ORM: Objektrelationale Abbildung ist eine Technik der Softwareentwicklung, mit der ein in einer objektorientierten Programmiersprache geschriebenes Anwendungsprogramm seine Objekte in einer relationalen Datenbank ablegen kann.(Wiki)

Pip install flask-sqlalchemy, from flask\_sqlalchemy import SQLAlchemy

URI der Datenbank muss festgelegt werden, wo man DB findet. Wo ist Datei.db zu finden.

App.config[‘SQLALCHEMY\_DATABASE\_URI’]=sqlite:///site.db relativer Pfad zum Projektordner

db= SQLAlchemy(app)

Struktur der DB kann als Klasse representiert werden. ->models.py

Jede Klasse ist eine Tabelle in der DB. (NamederKlasse-Modell, db.Modell)

def \_\_repr(self): Wie Objekt in DB ausgedruckt wird

Objektrelational: Verbinden von Tabellen. Ein User kann mehrer Posts haben, ein Post nur ein User.

Class User:

posts=db.relationship(‘Post’, backref=autor), wie wenn man autor als Column in Klasse Post einfügen würde, dadurch bekommt man Zugriff auf Felder von User

post.autor. gibt alle Felder von User zurück, z.B. post.autor.username

user.posts gibt alle Posts von user zurück-> for post in user.posts

Class Post:

User\_id= db.Column(db.Foreignkey(‘user.id’) dadurch bekommt man Zugriff auf User.id, User.id ist Absender der Nachricht, gespeichert in User\_id im Post-Modell

Python

From app import db

From app.models import User, Post

Db.create\_all()

User\_1=User(username=….)

Db.session.add(User\_1)

Db.query.all()

Db.drop\_all() Alle löschen

Db.create\_all() neu erstellen