# Úvod

Flask je microweb framework pro tvorbu webových stránek s využitím Pythonem

Většinou se používá v rámci backendu

Importování flasku – from flask import Flask (Pozor na case!)

Při tvorby Flask aplikace je nutné definovat aplikaci jako proměnnou pomocí app=Flask(\_\_name\_\_)

Pro funkčnost a spuštění Flask aplikace se používá podmínika if \_\_name\_\_==“\_\_main\_\_“: app.run(), pro debug se do paramteru run vloží debug=True

Jednotlivé stránky jsou v python souboru definovány jako funkce – např. def index():

Tyto funkce navrací stránku pomocí return

K routování se používá speciální metoda @app.route(„jmeno v routovani“), pro kořen se používá root, tedy lomeno /

V routování lze zadávat i uživatelské parametry – např. app.route(„/<name>“) převezme jakýkoliv text zadaný do oblasti lomených závorek a uloží ho jako proměnnou name

Pro redirectování na jinou stránku je nutné tuto funkci importovat – from flask import redirect

Argumentem funkce redirect může být např. url\_for (což je také nutné importovat from flask import url\_for) a argumentem je název funkce, která reprezentuje nějakou stránku – return redirect(url\_for(„user“))

Do url\_for lze za specifikace funkce vkládat další parametry, které specifikují proměnné, se kterými se pracuje v rámci odkazované funkce – např. url\_for(„user“,name=“Jarda“)

V rámci routování lze jako cestu uvést lomeno i za specifikaci koncového bodu, což umožňuje při zadávání stránky používat variantu s lomítkem i bez lomítka – např. /admin/ přijme v prohlížeči /admin i /admin/

# Kombinace s HTML

Pro vykreslování HTML stránek je nutné importovat render\_template tedy from flask import render\_template

V rámci flasku je nutné ctít přesnou adresářovou strukturu – v domoském adresáři projektu je nutné mít všechny python skripty, do stejného adresáře je nutné vložit adresář se jménem templates, do něhož se budou vkládat jednotlivé stránky

Render\_template se poté dává do return statementu funkce, přičemž se nemusí uvádět cesta do adresáře template, protože flask si tam html soubory hledá sám – stačí pouze return render\_template(„index.html“)

Do render\_template lze za název html souboru vkládat opět další parametry jako např. content=name

V cílovém html souboru se na ně adresuje pomocí Jinja, tedy např. <p>{{content}}</p>

Když je ale několik proměnných, nemusí se předávat jen pomocí jednotlivého adresování, ale v samotném listu

Pro to je nutné v html souboru psát python kód pro vypsání jednotlivých prvků v listu

{% for element in list %}

<p>{{element}}</p>

{% endfor %}

Kromě for tam lze dělat i if statement, elif statement či i else statement stejně jako v Python kódu, ale elif a else bloky se neukončují, pouze se na konec dává endif

{% if x==2 %}

<p>Ano</p>

{% endif %}

# Template inheritance – komponenty

Komponenta se ve flasku založí pomocí specifikace v title tagu v daném html souboru, do nějž se vloží <title>{% block jmeno %}{% endblock %}</title>

Do jakékoliv stránky, která využívá blocky ze souboru nazev.html, v němž jsou uložené, je nutné uvést na začátek {% extends „nazev.html“ %}

# http metody a data z nich

Při vytvoření app.route, v níž se provádí login či registrace, je nutné zde zahrnout i podporované metody např. @app.route(„/login“,methods=[„POST“,“GET“])

Také je nutné v rámci flask knihovny importovat request – from flask import rquest

V rámci login metody je konvencí uvést podmínku if request.method==“POST“ či „GET“: a až za ní uvést proceduru při loginu

Např. lze získat jméno z formu a redirectnout uživatele na jinou stránku, kde se to jméno využívá – return redirect(url\_for(„usershow“,username=username\_retreived))

# Session

Session obecně slouží k uchování dat po nějakou trvalejší dobu, např. v předchozím případě bychom museli pořád dávat routování na login a zadávat uživatelské jméno, což by bylo nepoužitelné

Data v session jsou uchovávána na serveru samotného webu a jsou též dočasné, tudíž při zrušení session se veškerá data v ní ztratí

Pro použití v session je nejdříve nutné ji naimportovat from flask import sesion

Data v session lze vnímat jako nějaký dictionary, tudíž hodnoty do ní se přiřazují jako session[„nazev promenne“]=vlozenahodnota

Pro následnou kontrolu, jestli ta session je vůbec spuštěná a jestli je např. uživatel přihlášený, se používá if „promenna“ in session:

Veškerá data v session jsou uschována v šifrované podobě na web serveru, což vyžaduje použití secret key, a to pomocí app.secret\_key=“hodnotaklice“, pozor app je ta instance Flask aplikace, ne univerzální název, hodnota klíče může být v zásadě cokoliv

Session lze i zrušit manuálně pomocí session.pop(„nazev promenne“,None), což odstraní proměnnou ze session, None je nutností

Lze vytvořit i tzv. permanent session, tedy session, která se nezruší při refreshnutí prohlížeče, což lze využít např. pro automatické odhlášení po nějaké době nečinnosti či něco podobonéo

Je pro to nutné naimportovat datetime, a to from datetime import timedelta

Na začátku kódu se pak uvede app.permanent\_session\_lifetime=timedelta(days=int,hours=int,minutes=int,…)

Tím to ale nekončí, protože musíme tu naši session definovat právě jako infinite, což se udělá uvnitř funkce, kde se pracuje s danou session pomocí uvedení session.permanent=True na začátek

# Krátkodobé zobrazení zpráv

Lze to využít např. při přihlášení uživatele, kdy při redirectu na úvodní stránku bude nahoře napsáno, že se úspěšně přihlásil

Je pro tento účel nutné naimportovat flash, tedy from flask import flash

Zpráva se poté flashne velice jednoduše zavoláním funkce flash(„Zpráva“,“kategorie zprávy“)

Kategorie je volitelný parametr

Tyto zprávy je ale nutné zobrazit v rámci požadované stranky pomocí jinja, a to {% width messages=get\_flashed\_messages() %}{% endwith %} statementu, do jehož obsahu lze vložit jakoukoliv manipulaci s těmito zprávami, přičemž obvykle je dobré zkontrolovat, jestli tam vůbec nějaké zprávy jsou pomocí {% if messages %}{% endif %}

# Propojení s databází SQLalchemy

SQLalchemy se stahuje pomocí pip install flask-sqlalchemy či pomocí jiného vhodného balíčkového managera

Na začátku python souboru s flaskem je tak nutné uvést from flask\_sqlalchemy import SQLAlchemy

Lze to použít např. pro zobrazení univerzální stránky o informacích uživatele

To lze udělat jako jednoduchou form, kde bude např. email

U těchto jednotlivých hodnot je dobré nastavit jednak jejich placeholder i value, kde placeholder bude nějaká výzva k zadání emailu a value bude obsahovat {{email if email}}, což způsobí, že se zobrazí hodnota emailu pouze, pokud je definována, což je pouze, když je uživatel přihlášen

V rámci logiky zobrazení stránek v python souboru je pak nutné rozšíření např. o nastavení základní hodnoty emailu a získání hodnoty emailu ze session v závislosti na request.method

Pro konkrétní práci s DB je nutné založit objekt třídy SQLAlchemy, tedy např. db=SQLAlchemy(app), to app tam musí být vždy a je to ta proměnná reprezentující název proměnné flask aplikace

App také musí být kompatibilní s sqlalchemy, proto je nutné opravit její config – app.config[‚SQLALCHEMY\_DATABASE\_URI‘]=‘sqlite:///users.sqlite3‘

Volitelná konfigurace pak zamezí projevení jakékoliv sebemenší změny v databázi – app.config[„SQLALCHEMY\_TRACK\_MODIFICATIONS“]=False

SQLAlchemy umožňuje veškerou práci s DB dělat v rámci pythonu a ne pomocí SQL dotazů

Jednotlivé tabulky v rámci databáze se zde tvoří jako třídy, které dědí od db.Model, tedy class users(db.Model) – db zde reprezentuje ten databázový objekt

Jednotlivé parametry se poté uvádí do této class, základem je id, které se zapisuje \_id=db.Column(„id“,db.Integer,primary\_key=True)

Mimochodem není nutné definovat jméno té proměnné, pokud to zůstane prázdné (tedy se neuvedou ani ty uvozovoky), tak to převezme jméno té proměnné

Je dobré podotknout, že column zde reprezentují jednotlivé parametry a row pak jednotlivé hodnoty

Poté se zde uvádí init metoda, která se použije při každé nové registraci

Def \_\_init\_\_(self, name, email, …):

Self.name=name….

V rámci konfigurace se také ve spodní části programu, kde se spouští aplikace, uvádí db.create\_all(), což zajistí vytvoření databáze, pokud ještě neexistuje při každém spuštění programu

Je to ale nutné provést s with app.app\_context(), bez toho to z nějakého důvodu nefunguje

# Vytváření uživatelů

Při registraci se konkrétní záznam s uživatelským jménem apod. vytvoří zavedením objektu třídy users či prostě té, kde je definována tabulka uživatelů s požadovanými vstupními parametry

Do databáze se poté tento objekt přidá pomocí db.session.add(promenna)

Tyto změny je poté nutné commitnout pomocí db.session.commit()

Toto se ale vykoná pouze, pokud uživatel neexistuje

Pro zjištění, jestli uživatel už je v databázi, se používá found\_user=users.query.filter(name=user).first(), tedy SQL dotaz, který vybere záznamy, kde jméno je rovno hodnotě user

Poté se to již ověří podmínkou if found\_user, protože navrací hodnotu něčeho či None

Pokud user již existuje, lze např. nastavit session[„email“] na atribut email toho uživatele, tedy session[„email“]=found\_user.email

Found\_user ale obsahuje atributy, které když jsou změněny, tak se ta změna projeví i v databázi a nejen v aktuální session

Např. found\_user.email=session[„email“]

Poté je samozřejmě nutné tyto změny commitnout db.session.commit()

Pro výpis všech uživatelů a jejich hodnot se používá users.query.all(), což lze použít např. při vytvoření stránky pro už. profil

Uživatelé je také nutné z databáze odstraňovat

Ten zápis je podobný hledání uživatelů, a to users.query.filter\_by(name=user).delete(), což odstraní všechny uživatele, jejichž jméno v databázi je rovno zadanému parametru

Kdybychom chtěl odstranit více uživatelů najednou, museli bychom loopnout přes původní query pro výpis uživatelů a k dané proměnné, přes kterou se provádí for cyklus připsat .delete()

# Struktura složek

Standartně je každý python soubor v kořenovém adresáři daného projektu, dále musí být založen adresář templates pro html soubory a dále static, kde je uloženo vše, co se nemá v průběhu běhu programu měnit – css, javascript.

U css je ale rozdíl při jeho použití v html souboru –

    <link rel="stylesheet" href="{{ url\_for('static', filename='style.css') }}">

Podobně se přidávají obrázky, pro které se také ve static vyhradí speciální adresář

# Blue prints, rozdělení programu na více .py souborů

Kód si lze rozdělit na více souborů, což je vhodné především při jejich opakovaném používání

Další soubor lze vytvořit v kořenovém adresáři projektu, ale v jeho obsahu je nutné uvést from flask import Blueprint

Konkrétní instance aplikace se v tomto souboru vytvoří pomocí second=Blueprint(„jmeno“, \_\_name\_\_, static\_folder=“cesta ke static adresáři“, templates\_folder=“cesta k templates adresáři“)

Veškeré route se v tomto rozšiřujícím souboru tvoří pomocí @second.route(„“), což závisí na proměnné reprezentující instanci aplikace

Ten rozšiřující soubor je ale nutné zviditelnit v rámci toho hlavního .py souboru, tudíž v hlavním souboru musí být nahoře uvedeno from second import second, název opět závisí na pojmenování proměnné na začátku

Následně se v hlavním python souboru musí uvést app.register\_blueprint(second, url\_prefix=““)

Url\_prefix zde souvisí s tím, že v obou souborech může být route s naprosto stejným názvem, v důsledku čehož by mohlo dojít ke konfliktům, tudíž se do prefixu uvede slovo (většinou s lomenem, aby to fungovalo s routováním), které má být naspáno hned po localhost, pokud chce člověk přistoupit k tomu rozšiřujícímu souboru

Někdy je vhodné udělat ve flasku celou komponentu, což lze zajistit vytvořením individuální složky, která může obsahovat vlastní templates, vlastní static apod., ale v tom naprosto prvním .py souboru je nutné v rámci import místo from second uvést from adresar.second