Technologie informacyjne
przed 11 wykładem

Andrzej Giniewicz

15.05.2024

W niniejszej porcji materiałów dowiemy się, jak połączyć naszą wiedzę dotyczącą HTML i CSS z wykładu 9 i 10, wiedzę o tym, jak działa Internet oraz wiedzę dotyczącą Pythona z pre-rekwizytów kursu, aby dodać do strony nieco interaktywności. Omówimy elementy składowe strony internetowej napisanej za pomocą biblioteki Flask.

1 Architektura strony w Pythonie

Komputery w Internecie identyfikowane są przez numery IP[1]. Najczęściej nie posługujemy się jednak bezpośrednio numerem, tylko domeną. Domeny tłumaczone są na numery IP przez serwery DNS. Niezależnie od tego, czy podamy numer IP czy domenę, gdy wpisujemy adres w przeglądarce, wysyłamy do serwera zapytanie HTTP. To zapytanie, zależnie od użytego protokołu[2], trafia na odpowiedni port na serwerze, na którym pewna aplikacja nasłuchuje przychodzący ruch i postępuje dalej zgodnie z zaprogramowanymi zasadami.

W przypadku aplikacji napisanych w Pythonie najczęściej pierwszym programem, z którym spotyka się zapytanie, jest serwer pełniący funkcję proxy. Takim serwerem w większości konfiguracji jest NGINX lub Apache[3]. Serwery takie najczęściej rozróżniają zapytania o dwa rodzaje zasobów — **statyczne**, których zawartość nie zależy od kontekstu, czyli na przykład od tego, który użytkownik jest zalogowany, oraz **dynamiczne**, których zawartość zależy od kontekstu, na przykład jest inna dla każdego zalogowanego użytkownika. Zapytania o zasoby statyczne, to na przykład obrazy, filmy, dokumenty PDF, pliki stylu w CSS oraz niektóre strony w HTML, takie jak strony błędów lub notka o firmie. Większość zapytań HTML jest

[1] Obecnie najczęściej jest to numer IPv4, jednak numery IP w standardzie w wersji 4 zaczęły się już wyczerpywać — RIPE, który jest odpowiedzialny za rozdzielanie numerów IP dla dostawców sieci, ogłosił w 2019 roku, że przydzielił ostatni wolny adres w swojej puli (źródło: `https://www.ripe.net/publications/news/about-ripe-ncc-and-ripe/the-ripe-ncc-has-run-out-of-ipv4-addresses`). Można się spodziewać, że z konieczności prace nad wdrożeniem standardu IPv6 przyśpieszą. Zapytanie o koordynację prac nad wdrożeniem IPv6 na terenie Unii Europejskiej zostało zgłoszone do Parlamentu Europejskiego w 2020 roku, niestety jednak nie podjęto konkretnych kroków, ze względu na niewielki poziom wdrożenia standardu IPv6 (źródło: `https://www.europarl.europa.eu/doceo/document/E-9-2020-000064_EN.html`).

[2] Obecnie około 80% stron internetowych korzysta z protokołu HTTPS, dla którego domyślnym portem jest 443 (źródło: `https://letsencrypt.org/stats/#percent-pageloads`).

[3] Patrz `https://www.nginx.com/`, `https://httpd.apache.org/` oraz `https://twitter.com/niebezpiecznik/status/1304532288752029702`.

jednak dynamiczna — lista produktów lub zawartość koszyka w sposób oczywisty musi zależeć od tego, jakich produktów szukaliśmy i jakie zakupiliśmy — nie może być zatem taka sama dla wszystkich. Pliki statyczne mogą być udostępnione przez serwer proxy od razu, bez uruchamiania Pythona, co stanowi bardzo szybkie rozwiązanie. Zapytania te mogą być również przekazane do sieci CDN (ang. Content Delivery Network), które na podstawie geolokalizacji źródła zapytania, wybierają najbliższy serwer, z którego pliki statyczne dotrą najszybciej do odbiorcy. Jeśli serwer proxy nie obsłuży zapytania jako zapytania do zasobów statycznych, musi przekazać je dalej, do programu napisanego w innym języku — w naszym przypadku w Pythonie.

Ponieważ pisanie całości aplikacji sieciowych od podstaw za każdym razem byłoby uciążliwe, w Pythonie wprowadzono standardowy protokół do aplikacji sieciowych. Standard ten nosi nazwę Web Server Gateway Interface, w skrócie WSGI i jest zdefiniowany przez PEP3333[4]. Dzięki temu możemy wybrać jeden z dostępnych serwerów WSGI napisanych w Pythonie i możemy połączyć go z serwerem proxy opisanym w poprzednim kroku za pośrednictwem tak zwanego gniazda (ang. socket, dokładnie Unix Domain Socket) lub innej formy komunikacji międzyprocesowej. Popularnym serwerem WSGI jest GUnicorn, którego nazwa pochodzi od green unicorn, czyli zielony jednorożec. Kolor zielony pochodzi od tak zwanych zielonych wątków, czyli sposobu uwielowątkowienia aplikacji za pomocą wątków obsługiwanych przez aplikację, a nie system operacyjny[5]. Architektura serwera GUnicorn jest wielowątkowa. Oznacza to, że uruchamia on wiele kopii naszej aplikacji, aby móc obsłużyć więcej zapytań jednocześnie, w sytuacji, gdy jest zbyt dużo zapytań dla jednej instancji aplikacji, która jest zajęta przetwarzaniem dotychczasowych zapytań.

Rolą twórcy strony jest zatem przygotowanie serwera proxy oraz serwera WSGI i napisanie aplikacji korzystającej z WSGI. Interfejs ten nie jest najbardziej wygodny, dlatego istnieje wiele bibliotek pomagających tworzyć aplikacje zgodne ze standardem WSGI. Podczas zajęć wykorzystamy jedną z nich — Flask[6]. Dzięki komponentom, które opisaliśmy, gdy ktoś wpisze adres w przeglądarce i zapytanie o zasób trafi do serwera proxy, następnie zostanie zidentyfikowane jako zapytanie o zasób dynamiczny i trafi do serwera WSGI, który przetłumaczy je na wywołanie odpowiedniej funkcji w Pythonie. Naszą rolą jest napisane tej funkcji i zwrócenie odpowiedzi jako napisu. Odpowiedź ta zostanie opakowana przez serwer WSGI do tak zwanej odpowiedzi HTTP i przez serwer proxy przekazana do użytkownika. Odpowiedź HTTP jest wyświetlana potem przez przeglądarkę. Jednym z ważniejszych elementów odpowiedzi jest jej kod. Kod 200 oznacza OK i jest informacją dla przeglądarki, że zapytanie się udało. Kody zaczynające się od 4 lub 5 oznaczają, że coś się nie udało. Zwykle, jeśli kod zaczyna się od 5, oznacza to błąd w serwerze proxy lub serwerze WSGI (w kontekście Pythona 502, Bad Gateway, oznacza, że serwer proxy działa, ale serwer WSGI nie odpowiedział), natomiast, jeśli zaczyna się od 4, oznacza błąd w aplikacji lub

[4]PEP3333, czyli Python Enhancement Proposal o numerze 3333 — `https://peps.python.org/pep-3333/`.

[5]Więcej na ten temat można przeczytać na stronie `https://greenlet.readthedocs.io/en/latest/greenlet.html`.

[6]`https://flask.palletsprojects.com/`.

niewłaściwie wpisany adres (404, Not Found, oznacza, że nie odnaleziono zasobu, czyli na przykład obrazka lub strony)[7].

Zwykle w architekturze aplikacji jest jeszcze warstwa poniżej aplikacji — warstwa danych. Najczęściej na tym poziomie wykorzystywane są pliki oraz bazy danych. Przykładowo, jeśli ktoś chce się zalogować na stronę i aplikacja WSGI otrzyma zapytanie z loginem i hasłem, musi sprawdzić w jakiejś bazie danych, czy jest to prawidłowe hasło. Elementem tym nie będziemy zajmować się podczas kursu z technologii informacyjnych, tematyką baz danych zajmuje się osobny kurs w programie studiów. Na potrzeby stron internetowych, jedną z popularniejszych baz danych jest PostgreSQL[8], który zwykle integrowany jest z aplikacją we Flasku za pomocą kilku bibliotek, w tym Flask-Migrate[9]. Jeśli ktoś jest zainteresowany szczegółami, zachęcam do zapoznania się z dokumentacją lub książką Flask Web Development polecaną do wykładu[10].

Architektura tu opisana jest typową architekturą dla niedużej aplikacji w Pythonie. Mówimy tutaj o strukturze warstwowej: serwer proxy, serwer WSGI, aplikacja WSGI, baza danych. W przypadku kursu z technologii będziemy zajmować się głównie pisaniem aplikacji w WSGI za pomocą biblioteki Flask. Gdy instalujemy bibliotekę Flask, instalujemy również komendę `flask`, która pozwala wykonać różne operacje związane z tworzeniem strony. Jedną z nich jest `flask run`, która uruchamia prosty serwer WSGI działający również jako serwer proxy, które przekierowuje wszystkie zapytania (statyczne i dynamiczne) do aplikacji. Serwer ten posiada też tryb debugowania, który pozwala na poszukiwanie błędów. Tryb ten integruje się z narzędziami do debugowania w Visual Studio Code, zatem mocno ułatwia to pisanie kodu. Niestety, serwer ten nie jest wystarczający, ani dostatecznie bezpieczny, aby udostępnić aplikację w Internecie. W praktyce musimy skonfigurować odpowiedni serwer WSGI. Jednym z dostępnych rozwiązań jest serwer GUnicorn. Aplikacja działająca lokalnie z serwerem do debugowania `flask run` oraz aplikacja wgrana na serwer używająca GUnicorna ma identyczny kod.

Osoby zainteresowane architekturą dla dużych aplikacji, zachęcam do zapoznania się z książką „Bootstrapping Microservices with Docker, Kubernetes, and Terraform”[11]. Materiał ten choć ciekawy, znacząco wykracza poza możliwości czasowe na kursie technologii informacyjnych. Informacje zawarte w tej książce są bezcenne, gdy aplikacja, którą przygotowujemy, przestaje „mieścić się” na jednym serwerze i zaczynamy korzystać z chmury.

2 Hello World we Flasku

Standardowe „Hello World” we Flasku można znaleźć w dokumentacji na stronie `https://flask.palletsprojects.com/en/3.0.x/quickstart/`. Wygląda ono następująco.

[7] Aby dowiedzieć się więcej o kodach i czym różni się 403 od 404, zachęcam do przeczytania dokumentacji na stronie `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`, natomiast by dowiedzieć się więcej o zapytaniach i odpowiedziach HTTP, zachęcam do przeczytania `https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages`.

[8] `https://www.postgresql.org/`.

[9] `https://flask-migrate.readthedocs.io/`.

[10] `https://www.oreilly.com/library/view/flask-web-development/9781491991725/`.

[11] `https://learning.oreilly.com/library/view/bootstrapping-microservices-with/9781617297212/`.

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

Na początek importujemy klasę `Flask` z biblioteki `flask` i wykorzystujemy ją do stworzenia aplikacji w zmiennej `app`. Jest to domyślna nazwa, pod którą standardowy serwer WSGI uruchamiany przez komendą `flask run` jej szuka — nazwa głównego pliku również domyślnie powinna brzmieć `app.py` — jeśli jest inna, należy ją ustawić w zmiennej środowiskowej `FLASK_APP`.

Klasie `Flask` do konstruktora przekazujemy zmienną `__name__`, czyli nazwę modułu[12]. Nazwa ta jest wykorzystywana, aby znaleźć pliki szablonów i inne zasoby w odpowiednich katalogach. Można powiedzieć, że każda aplikacja we Flasku zaczyna się od podobnych linii. Ostatnie trzy linie przykładu definiują jedną trasę.

Trasy deklarujemy za pomocą dekoratora `route` zdefiniowanego w aplikacji. Dekorator ten przyjmuje różne parametry, przy czym pierwszy z nich to ścieżka do zasobu. Jeśli strona działa pod adresem `https://git.giniewicz.it/`, to strona główna ma ścieżkę do zasobu "/". Jeśli ktoś wchodzi na tę stronę, serwer WSGI wywołuje funkcję Pythona, która jest udekorowana przez `@app.route("/")`. Gdy ktoś wchodzi na stronę `https://git.giniewicz.it/login`, serwer WSGI uruchamia funkcję Pythona, która jest udekorowana przez `@app.route("/login")`. Dzięki temu można w pewnym sensie łączyć funkcje Pythona z adresami w pasku adresu przeglądarki[13].

Domyślnie dekorator `route` obsługuje metodę `GET`. Metoda `GET` to sposób wysyłania zapytań, w którym ewentualnie przesłane dane są widoczne w adresie. Z tego powodu przy budowie aplikacji, zaleca się, aby przesłać dane metodą `POST`, szczególnie jeśli są to dane logowania lub inne dane poufne. Dodatkowo zapytania `GET` nie powinny nic zmieniać w stanie serwera, ewentualnie przesłane niewrażliwe dane powinny co najwyżej służyć do wyboru właściwego zasobu, na przykład wskazywać na konkretną stronę wyników w wyszukiwaniu[14]. Metody obsługiwane przez trasę możemy zmienić za pomocą opcji `methods` do dekoratora. Dekorator

```
@app.route("/login")
```

ma domyślnie ustawiony parametr `methods=['GET']`. Jeśli chcemy, możemy zmienić go na `POST`.

[12] `https://docs.python.org/3/reference/import.html#__name__`.

[13] Tak, serwer do kartkówek jest napisany we Flasku i korzysta z tych samych technologii, które tu opisujemy — NGINX, GUnicorn, Flask i PostgreSQL.

[14] Opis metod HTTP oraz ich przeznaczenie możemy sprawdzić pod adresem `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods`.

```
@app.route("/login", methods=['POST'])
```

Metody te możemy połączyć, aby obsłużyć oba rodzaje zapytań.

```
@app.route("/login", methods=['GET', 'POST'])
```

Aby wewnątrz funkcji obsługującej dwie metody rozpoznać rodzaj zapytania, możemy skorzystać z obiektu `request`

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        # tutaj ktoś wypełnił formularz, więc sprawdzamy, czy dobrze
    else:
        # tutaj ktoś wszedł na stronę logowania, wyświetlamy mu formularz
```

Więcej możliwości tworzenia tras za pomocą dekoratora `route` można znaleźć w dokumentacji na stronie `https://flask.palletsprojects.com/en/3.0.x/quickstart/#routing`. Opis tego, co można uzyskać z obiektu `request` znajdziemy na stronie `https://flask.palletsprojects.com/en/3.0.x/quickstart/#the-request-object`, natomiast sposób przygotowania formularza, aby można było z niego odczytać wartości, na stronie `https://www.w3schools.com/html/html_forms.asp`.

Jeśli dopuszczamy wypełnianie formularzy, musimy pamiętać, że niektóre znaki mają specjalne znaczenie. Gdybyśmy dopuścili możliwość podania dowolnego imienia, ktoś mógłby wpisać jako imię `<script>...</script>`, co spowodowałoby, że każdej osobie, której wyświetla się to imię, uruchamia się skrypt w ECMAScript. Skrypt ten może robić różne rzeczy, ale w szczególności mógłby powodować próby wyłudzenia hasła lub innych nielegalnych rzeczy. Aby temu zapobiec, Flask dostarcza funkcję `escape`, która powoduje, że w takim przypadku wszystkie znaki HTML, które ktoś wprowadzi, będą wyświetlone jako tekst, a nie jako kod HTML. W szczególności oznacza to, że wszyscy zobaczyliby imię składające się z kodu źródłowego `<script>...</script>`, zamiast zobaczyć efekt działania skryptu. Więcej na temat funkcji `escape` można przeczytać w dokumentacji pod adresem `https://flask.palletsprojects.com/en/3.0.x/quickstart/#html-escaping`.

Pliki statyczne, takie jak pliki `css`, umieszczamy domyślnie w katalogu `static`. Aby móc się do nich odwołać, używa się funkcji `url_for`.

```
from flask import url_for

@app.route('/obrazek')
def pokazujemy():
    adres = url_for('static', filename='obrazek.png')
    return f"<img src='{adres}' alt='Zdjęcie autora'>"
```

Dzięki funkcji url_for nie musimy zmieniać kodu strony w zależności od tego, jak uruchomimy kod aplikacji — jeśli uruchamiamy ją lokalnie, zapewne będzie to http://localhost:5000/static/obrazek.png. Detalami zajmie się Flask, my możemy myśleć, że w atrybut src wstawiamy „url for", czyli „adres dla" pliku statycznego o nazwie obrazek.png. Więcej o plikach statycznych można przeczytać pod adresem https://flask.palletsprojects.com/en/3.0.x/quickstart/#static-files. Nie wszystkie zdjęcia muszą być statyczne. Istnieje możliwość dynamicznego generowania zdjęć, na przykład w Matplotlib. Opis jak wykorzystać Matplotlib we Flasku znajduje się na stronie https://matplotlib.org/stable/gallery/user_interfaces/web_application_server_sgskip.html.

Tworzenie kodu HTML jako napisu wewnątrz funkcji bywa uciążliwe. Aby pomóc w tworzeniu poprawnego HTML bez konieczności zabawy w nawiasy, cudzysłowy i dodawanie napisów, powstał język szablonów Jinja. Plik szablonu umieszczamy w katalogu templates i zwracamy szablon opakowany w funkcję render_template.

```
from flask import render_template

@app.route('/hello/')
def hello():
    name = None
    # sprawdzamy, czy ktoś jest zalogowany, jeśli tak ustawiamy name na jego dane
    return render_template('hello.html', who=name)
```

Natomiast wewnątrz pliku templates/hello.html umieszczamy zawartość

```
<!doctype html>
<html lang="pl">
<head>
    <meta charset="utf-8">
    <title>Witaj w szablonie!</title>
</head>
<body>
{% if who %}
    <h1>Witaj {{ who }}!</h1>
{% else %}
    <h1>Witaj nieznajomy!</h1>
{% endif %}
</body>
</html>
```

Fragmenty zaczynające się od klamer stanowią elementy składni szablonu. W miejscu {{ who }} Python wstawia wartość zmiennej who przekazanej do funkcji render_template jako parametr opcjonalny. W {% if who %} sprawdza, czy wartość ta jest zdefiniowana. Język szablonów nazywa się Jinja. Posiada między innymi instrukcje warunkowe, pętle,

zmienne oraz dziedziczenie, pozwalające na lekkie modyfikowanie istniejących szablonów. Dodatkowo do szablonu możemy przekazać nie tylko napis, ale również listy, słowniki lub inne obiekty Pythona. Posiada on możliwość rozpoznawania elementów programowania obiektowego, zatem składnia wyciągania atrybutów z obiektów za pomocą kropki jest wspierana. Dzięki temu można na przykład przygotować pętlę po liście produktów w sklepie i stworzyć listę odpowiedniej długości.

```
<ul>
{% for item in cart %}
    <li>
        <a href="{{ item.product_page }}">
           <img src="{{ item.image_url }}" alt="{{ item.name }}">
           {{ item.name }}
        </a>
    </li>
{% endfor %}
</ul>
```

Więcej o języku Jinja i tworzeniu szablonów można przeczytać na stronie `https://jinja.palletsprojects.com/en/3.1.x/templates/` oraz `https://flask.palletsprojects.com/en/3.0.x/quickstart/#rendering-templates`.

Choć baza danych jest standardowym sposobem przechowywania informacji po stronie serwera, częste odpytywanie bazy o to samo nie jest wydajne. Najczęściej po zalogowaniu, zapamiętujemy informacje o użytkowniku w ciasteczku, czyli niewielkim pliku przechowywanym w przeglądarce. Przeglądarki dodają ciasteczko do wszystkich zapytań na stronie i na podstawie tej wiedzy, serwer wie, który użytkownik się zapisał. Informacje takie są przechowywane w tak zwanej sesji. Sesje we Flasku są zaimplementowane przez podpisane cyfrowo ciasteczka, zatem użytkownik może zobaczyć swoje ciasteczko, ale nie może go zmienić. Sesje implementowane są za pomocą obiektu `session`. Obiekt ten działa podobnie do słownika. Możemy sprawdzić, czy coś jest ustawione w sesji

```
'user' in session
```

ustawić wartość

```
session['user'] = "admin"
```

pobrać wartość

```
name = session['user']
```

lub ją usunąć (tu składnia nieco inna niż dla słowników)

```
session.pop('user', None)
```

Aby korzystać z sesji musimy zaimportować obiekt sesji oraz ustawić tajny klucz.

```
from flask import session

app.secret_key = b'juhr39r7"y2omfi/342#'
```

Tajny klucz powinien być tajny — każda osoba, która ma dostęp do klucza, może podpisać ciasteczko, zatem sfałszować dane użytkownika w naszym systemie. Więcej o sesjach możemy przeczytać w `https://flask.palletsprojects.com/en/3.0.x/quickstart/#sessions`.