Aplikacje WWW. Wykład #4

REST (Representensional State Transfer)

1. Czym jest REST?

Zaproponowany przez Roya T. Fieldinga w roku 2000 styl architektoniczny oprogramowania w artykule Principled Design of the Modern Web Architecture. Początkowo to rozwiązanie było projektowanie w kontekście WWW, ale z czasem zostało zaadoptowane do szerszego zakresu zastosowań (np. IoT, środowiska chmurowe, internet mobilny).

REST API (nazywany również RESTful API) to interfejs architektury REST służący do przesyłania informacji w architekturach klient-serwer oraz mikrousługach. Mówiąc inaczej jest to format wymiany danych, który rozumiany jest przez klienta oraz serwer.

Aby powiedzieć, że aplikacja jest aplikacją RESTful musi ona spełniać zasady zdefiniowane w specyfikacji REST. Są to:

1. **Jednolity interface (ang. Uniform interface)**

API udostępniane przez serwer musi być rozumiane przez wszystkie aplikacji komunikujące się z nim, a dane mają ten sam format i zakres. Możliwa jest obsługa wielu urządzeń i aplikacji.

2. **Architektura klient-serwer**

Aplikacje są rozdzielone na aplikację kliencką oraz aplikację serwerową, co pozwala na ich niezależny rozwój. Sprzyja to dobremu skalowaniu i przenośności. Ułatwia to również integrację z innymi usługami.

3. **Bezstanowość (ang. Stateless)**

Sama specyfikacja REST nie definiuje możliwości utrzymywania sesji pomiędzy klientem a serwerem. Po stronie serwera nie powinniśmy więc przechowywać żadnych informacji od klienta, które są potrzebne do poprawnej obsługi wysyłanego przez niego żądania. Klient powinien dostarczać w tym żądaniu wszystkich niezbędnych informacji do jego poprawnego przetworzenia.

Istnieje oczywiście możliwość realizacji mechanizmu weryfikującego czy użytkownik posiada uprawnienia do wykonania danego żądania i jest to realizowane poprzez przesyłanie w nagłówku informacji autoryzacyjnych za pośrednictwem tokena, np. JWT (ang. JSON Web Token).

4. **Pamięć podręczna (ang. Cacheability)**

Pamięć podręczna to mechanizm pozwalający na zapamiętanie pewnych informacji, zasobów, które dla kolejnych żądań mogą być takie same, więc można zrezygnować z przetworzenia żądania i zamiast tego odczytać je z magazynu pamięci podręcznej zmniejszając obciążenie sieciowe i sprzętowe.

Mechanizm pamięci podręcznej może być implementowany zarówno po stronie klienta jak i serwera.

Przeglądarka internetowa może obsługiwać przechowywanie pamięci podręcznej lokalnie dla żądań typu GET oraz POST. Do obsługi tego mechanizmu wykorzystywane są nagłówki protokołu HTTP, które mogą definiowac czy zasób ma zostać przechowany i kiedy może zostać odświeżóny lub usnunięty.

Przykładowa odpowiedź protokołu HTTP dla zapytania GET do serwera www.wp.pl

```
curl -I https://www.onet.pl

...
Cache-Control: s-maxage=0,stale-if-error=0,no-cache
ETag: "1489104-eec3cfe5d372e4e9"
...
# s-maxage określa jak długo te dane pozostają świeże we współdzielonej pamięci
podręcznej
# ta dyrektywa jest ignorowana w przypadku prywatnej pamięci podręcznej
# oraz nadpisuje dyrektywę max-age lub nagłówek Expires dla współdzielonej
# pamięci podręcznej jeżeli jest ona obecna
# ETag – unikalny identyfikator / token, bezpośrednio powiązany z
# zapamiętanym zasobem. Zmienia się w momencie aktualizacji wartości.

curl -I https://www.google.pl

...
Expires: Wed, 22 Oct 2025 06:42:30 GMT
Cache-Control: private
...
# Expires – data wygaśnięcia danych. Kiedy termin upłynie, zasób musi zostać
pobrany z serwera i może być ponownie zapamiętany.
# cache może być przechowywany tylko lokalnie, np. w przeglądarce
# Taki rodzaj pamięci podręcznej powinien być stosowany dla zawartości dostępnej
po zalogowaniu, gdyż może służyć
# do przechowywania wrażliwych danych, które w przypadku braku dyrektywy private
# mogą zostać wykorzystane przez innych użytkowników i doprowadzić do wycieku
danych
```

Więcej o nagłowkach `Cache-Control` można przeczytać m.in. tu: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Cache-Control

W przypadku pamięci podręcznej po stronie serwera zazwyczaj jest to realizowane przez dedykowany cache serwer. Serwer ten sprawdza czy posiada zasów dla danego żądania i jeżeli tak oraz jego termin ważności nie wygasł to zwraca go do klienta bez ponownego przetwarzania API.

Popularnymi serwerami tego typu są:

- Redis
- Memcached
- Nginx
- Varnish
- Squid

5. **Odseparowane warstwy (ang. Layered system)**

Warstwy logiki oraz WWW są odseparowane. Przykładowy zbiór warstw aplikacji może wyglądać tak:

- **Żądanie klienta:** Żądanie dociera do warstwy klienta (np. przeglądarki internetowej).
- **Warstwy pośrednie:** np. CDN (Content Delivery Network) i load balancer.
- **API Gateway:** może służyć jako punkt wejścia dla żądań, zajmując się m.in. uwierzytelnianiem i autoryzacją.
- **Serwer aplikacji:** Żądanie trafia do serwera aplikacji, który przetwarza logikę biznesową.
- **Baza danych:** Serwer aplikacji komunikuje się z bazą danych w celu pobrania lub zapisu danych.
- **Odpowiedź:** Odpowiedź jest przesyłana z powrotem, przechodząc przez te same warstwy, co żądanie.

To pozwala na skalowanie aplikacji na każdej warstwie z osobna np. poprzez wykorzystanie mikroserwisów.

6. **Kod na żądanie (ang. Code on demand)**

Jest to jedyna opcjonalna cecha, która nie musi być wykorzystana w aplikacji, aby nazwać ją aplikacją RESTful. Ta funkcjonalność daje możliwość wysyłania wykonywalnego kodu do klienta (np. JavaScript, dynamiczne komponenty, aktualizacja aplikacji, aplety). Ma to swoje wady i zalety.

Zalety kodu na żądanie:

- Możliwe jest wysłanie kodu do klienta, który zapewni wykonanie części logiki biznesowej po jego stronie odciążając w ten sposób serwer (np. wstępna walidacja).
- Zmniejszenie ilości żądań do obsługi funkcjonalności co może poprawić wydajność aplikacji (UX (ang. User Experience) - wrażenia użytkownika).
- Łatwiejsze aktualizacje - możliwe jest wykoanie aktualizacji aplikacji po stronie klienta z wykorzystaniem takiego rozwiązania.

Wady takiego rozwiązania:

- Bezpieczeństwo: potrzebne jest zapewnienie mechanizmów, które uchronią aplikację kliencką przed wykonaniem złośliwego kodu. Wymagane jest dodanie zabezpieczeń przed atakami typu XSS (ang. Cross-Site-Scripting) oraz konfiguracja CORS (ang. Cross-Origin Resource Sharing).

2. Dostępne metody żądań protokołu HTTP poprzez API REST

Główne metody.

- `GET` – pobieranie zasobów,
- `POST` – dodawanie nowych zasobów,
- `PUT` – aktualizowanie istniejących danych,
- `PATCH` - częściowa aktualizacaj zasobów,
- `DELETE` – usuwanie zasobów.

Metody pomocnicze.

- `HEAD` - służy do pobierania metadanych zasobów, np. rozmiar,
- `OPTIONS` - umożliwia uzyskanie listy dostępnych metod dla zasobu,
- `TRACE` - może być wykorzystane jako narzędzie do debuggowania, może ujawniać wrażliwe dane, co powoduje, że jest podatny na ataki, np. atak Cross-Site Tracing (XST).

Podstawowe przykłady.

- **GET /orders** – pobranie listy wszystkich zamówień.

- **GET /orders/1** – pobranie informacji o konkretnym zamówieniu o ID 1.
- **GET /products?category=book** – pobranie listy produktów z kategorii "książki" (z wykorzystaniem parametru zapytania).
- **POST /orders** – utworzenie nowego zamówienia poprzez przesłanie jego danych w treści żądania (np. w formacie JSON).
- **PUT /orders/1** – zaktualizowanie całego rekordu zamówienia o ID 1, wysyłając nowe dane w treści żądania.
- **PATCH /orders/5** – częściowa aktualizacja zamówienia o ID 5 (np. zmiana statusu).
- **DELETE /orders/1** – usunięcie zamówienia o ID 1.

Dodatkowe elementy żądania.

- **URL:** Adres, pod który wysyłane jest żądanie, np. https://api.example.com/orders/1.
- **Metoda HTTP:** Określa akcję, która ma zostać wykonana, np. GET, POST.
- **Nagłówki (Headers):** Dodatkowe informacje o żądaniu, np. Content-Type: application/json lub nagłówek autoryzacji Authorization: Bearer .
- **Treść żądania (Body):** Dane przesyłane do serwera, najczęściej w przypadku metod POST i PUT, np. dane nowego zamówienia w formacie JSON.
- **Parametry zapytania (Query Parameters):** Dodatkowe parametry dodawane do URL po znaku ?, np. ?page=2&sort=desc

Przykładowe wywołania powyższych żądań z wykorzystaniem narzędzia `curl`.

```
# Pobranie listy wszystkich zamówień
curl -X GET https://api.example.com/orders

# Pobranie informacji o konkretnym zamówieniu o ID 1
curl -X GET https://api.example.com/orders/1

# Utworzenie nowego zamówienia
curl -X POST https://api.example.com/orders \
     -H "Content-Type: application/json" \
     -d '{"product_id": 123, "quantity": 2}'

# Zaktualizowanie całego rekordu zamówienia o ID 1
curl -X PUT https://api.example.com/orders/1 \
     -H "Content-Type: application/json" \
     -d '{"product_id": 123, "quantity": 3, "status": "shipped"}'

# Częściowa aktualizacja zamówienia o ID 5
curl -X PATCH https://api.example.com/orders/5 \
     -H "Content-Type: application/json" \
     -d '{"status": "delivered"}'

# Usunięcie zamówienia o ID 1
curl -X DELETE https://api.example.com/orders/1
```

Skoro jesteśmy juz przy żądaniach protokołu HTTP, warto wspomnieć o kodach odpowiedzi, które są zwracane przez serwer w odpowiedzi na żądania klienta. Najczęściej wykorzystywane kody to:

Pomyślne wykonanie:

- **200 OK:** Standardowa odpowiedź na pomyślne żądanie.
- **201 Created:** Używane, gdy nowy zasób został pomyślnie utworzony na serwerze.
- **204 No Content:** Odpowiedź, gdy żądanie zakończyło się sukcesem, ale nie ma nic do zwrócenia (np. po usunięciu zasobu). Błędy po stronie klienta:
- **400 Bad Request:** Ogólny błąd, gdy serwer nie może przetworzyć zapytania z powodu błędnego formatu lub danych.
- **401 Unauthorized:** Wskazuje, że żądanie wymaga uwierzytelnienia.
- **403 Forbidden:** Oznacza, że serwer zrozumiał żądanie, ale odmawia jego realizacji (np. brak uprawnień).
- **404 Not Found:** Zasób, o który zapytano, nie został znaleziony.
- **405 Method Not Allowed:** Użyta metoda HTTP jest niedozwolona dla danego zasobu.
- **409 Conflict:** Żądanie nie może zostać przetworzone z powodu konfliktu stanu z aktualnym stanem zasobu.
- **429 Too Many Requests:** Klient przekroczył limit zapytań. Błędy po stronie serwera:
- **500 Internal Server Error:** Ogólny błąd serwera.
- **503 Service Unavailable:** Serwer jest tymczasowo niedostępny (np. przeciążony).

3. Serializacja danych

> Serializacja – w programowaniu proces przekształcania obiektów, tj. instancji określonych klas, do postaci szeregowej, czyli w strumień bajtów lub postać tekstową (np. XML, JSON) z zachowaniem aktualnego stanu obiektu. Serializowany obiekt może zostać utrwalony w pliku dyskowym, przesłany do innego procesu lub innego komputera poprzez sieć. Procesem odwrotnym do serializacji jest deserializacja. Proces ten polega na odczytaniu wcześniej zapisanego strumienia danych i odtworzeniu na tej podstawie obiektu klasy wraz z jego stanem bezpośrednio sprzed serializacji.
>
> *źródło*: Wikipedia

Najbardziej typowymi zastosowaniami serializacji jest chęć zapisania stanu aplikacji w bazie lub pliku lub przesłanie danych/stanu przez sieć.

W przypadku formatów binarnych problemem jest brak kompatybilności między rozwiązaniami co ogranicza znacznie ich zastosowanie. W przypadku języka Python serializację binarną możemy wykonać np. za pomocą modułu `pickle`. W przypadku takich rozwiązań jak REST API dużo lepszym rozwiązaniem jest wykorzystanie powszechnie dostępnego formatu zapisu danych, który umożliwi szersze jego wykorzystanie i integrację z innymi systemami, które mogą nie być jednorodne, ale "dogadują" się tym samym językiem (formatem) danych. Najczęściej wykorzystywanymi w praktyce formatami są `JSON` oraz `XML`.

Format JSON

JavaScript Object Notation (JSON) to otwarty, niezależny od języka programowania, tekstowy format danych pozwalający na definiowanie par wartości klucz-wartość oraz tablic. Tu warto przytoczyć nazwisko Dougkasa Crockforda, który zdefiniował ten format na początku lat 2000.

Specyfikację formatu (która pojawiła się dopiero w roku 2013) można zaleźć pod adresem https://www.rfc-editor.org/rfc/rfc7158.

Dopuszczalne kodowanie znaków plików `.json` to UTF-8, UTF-16 oraz UTF-32, przy czym ten pierwszy jest formatem domyślnym.

Przykład zapisu w formacie JSON
źródło: https://en.wikipedia.org/wiki/JSON

```
{
  "first_name": "John",
  "last_name": "Smith",
  "is_alive": true,
  "age": 27,
  "address": {
    "street_address": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postal_code": "10021-3100"
  },
  "phone_numbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [
    "Catherine",
    "Thomas",
    "Trevor"
  ],
  "spouse": null
}
```

Odpowiadający powyższej strukturze zapis w formacie XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<person>
    <first_name>John</first_name>
    <last_name>Smith</last_name>
    <is_alive>true</is_alive>
    <age>27</age>
    <address>
        <street_address>21 2nd Street</street_address>
        <city>New York</city>
        <state>NY</state>
```

```
        <postal_code>10021-3100</postal_code>
    </address>
    <phone_numbers>
        <phone>
            <type>home</type>
            <number>212 555-1234</number>
        </phone>
        <phone>
            <type>office</type>
            <number>646 555-4567</number>
        </phone>
    </phone_numbers>
    <children>
        <child>Catherine</child>
        <child>Thomas</child>
        <child>Trevor</child>
    </children>
    <spouse xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
</person>
```

Przykład wykonania serializacji oraz deserializacji danych w formacie JSON w języku Python z wykorzystaniem wbudowanego modułu `json`.

```
import json
# Przykładowe dane do serializacji
data = {
    "first_name": "John",
    "last_name": "Smith",
    "is_alive": True,
    "age": 27,
    "address": {
        "street_address": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postal_code": "10021-3100"
    },
    "phone_numbers": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "office",
            "number": "646 555-4567"
        }
    ],
    "children": [
        "Catherine",
        "Thomas",
        "Trevor"
    ],
```

```
    "spouse": None
}
# Serializacja do formatu JSON
json_data = json.dumps(data, indent=4)
print("Zserializowane dane JSON:")
print(json_data)

# Deserializacja z formatu JSON
deserialized_data = json.loads(json_data)
print("\nDeserializowane dane:")
print(deserialized_data)
```

Oraz przykład serializacji z wykorzystaniem modułu `pickle` do formatu binarnego.

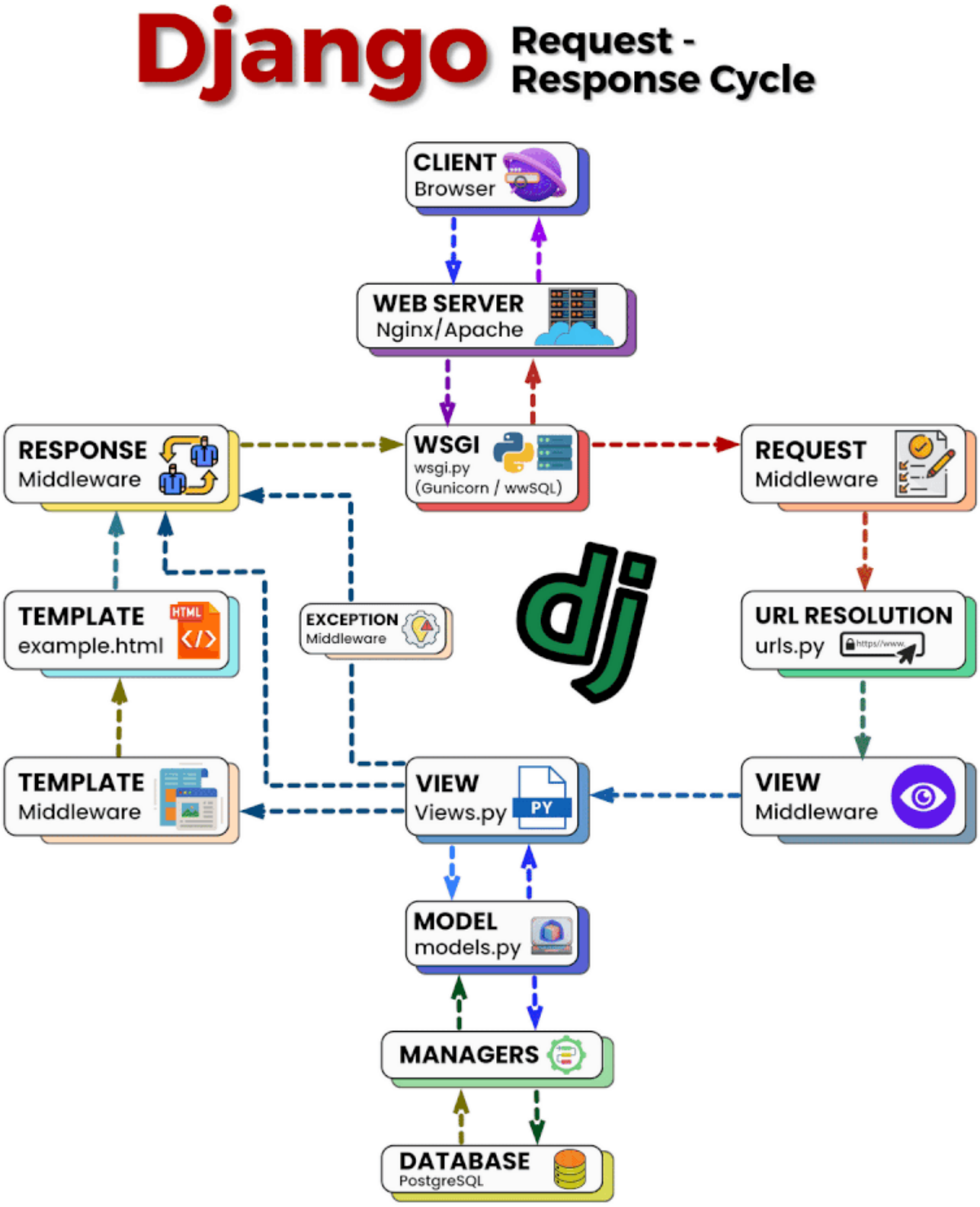
```
import pickle
# Przykładowe dane do serializacji
data = {
    "first_name": "John",
    "last_name": "Smith",
    "is_alive": True,
    "age": 27,
    "address": {
        "street_address": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postal_code": "10021-3100"
    },
    "phone_numbers": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "office",
            "number": "646 555-4567"
        }
    ],
    "children": [
        "Catherine",
        "Thomas",
        "Trevor"
    ],
    "spouse": None
}
# Serializacja do formatu binarnego
binary_data = pickle.dumps(data)

deserialized_data = pickle.loads(binary_data)
# typ deserializowanych danych
print(type(deserialized_data))  # <class 'dict'>
print("\nDeserializowane dane:")
print(deserialized_data)
```

4. Django Rest Framework

Django Rest Framework (DRF) to potężny i elastyczny zestaw narzędzi do budowania interfejsów API w Django. Umożliwia szybkie tworzenie API RESTful, oferując wiele funkcji, takich jak serializacja danych, uwierzytelnianie, uprawnienia i obsługa żądań HTTP.

Cykl żądanie - odpowiedź w Django

Poniżej znajduje się schemat obrazujący cykl żądanie-odpowiedź w Django (bez DRF):



źródło: https://medium.com/@praseeshprasee/django-request-response-cycle-explained-e3d707eed99c

Cykl żądanie - odpowiedź w Django Rest Framework (DRF)

Krok 1: Inicjalizacja żądania

Metoda `dispatch()` opakowuje standardowe żądanie `HttpRequest` Django w specjalny obiekt żądania DRF. Wybierane są parsery i autentykatory (poprzez `parser_classes` lub `DEFAULT_PARSER_CLASSES` oraz odpowiednio `authentication_classes` lub `DEFAULT_AUTHENTICATION_CLASSES`).

Krok 2: Uwierzytelnianie użytkownika

Metoda `dispatch()` próbuje uwierzytelnić użytkownika, wywołując metodę `perform_authentication()`. Wywoływane są tutaj klasy uwierzytelniające, które próbują zidentyfikować użytkownika. Jeśli żadna z nich nie powiedzie się, użytkownik jest ustawiany jako instancja `AnonymousUser`.

Krok 3: Sprawdzanie uprawnień

Metoda `dispatch()` sprawdza uprawnienia użytkownika, wywołując metodę `check_permissions()`.

Jest to miejsce, w którym oceniane są klasy `permission_classes` zadeklarowane w widoku lub `DEFAULT_PERMISSION_CLASSES` zdefiniowane w ustawieniach. Tutaj zazwyczaj klasy uprawnień sprawdzają metodę żądania, tj. `GET`, `POST`, `DELETE` itp., i określają, czy użytkownik ma prawo do jego żądania. Zwracają odpowiednio `True` lub `False`.

Jeżeli którakolwiek z klas uprawnień zwróci `False`, metoda `check_permissions()` zgłasza jedno z dwóch wyjątków:

- `NotAuthenticated` exception (401) dla użytkowników anonimowych
- `PermissionDenied` exception (403) w pozostałych przypadkach

Krok 4: Wybór handlera

Metoda `dispatch()` szuka odpowiedniego handlera dla metody żądania. To tutaj wywoływane są handlery `list()`, `create()`, `retrieve()`, `update()` i `destroy()`, tj. te, które najczęściej nadpisujesz. Jeżeli nie zostanie znaleziony odpowiedni handler, `dispatch()` zgłasza wyjątek `MethodNotAllowed` (błąd HTTP o kodzie 405).

Krok 5: Pobranie querysetu lub obiektu

Metoda handlera wywołuje metody `get_queryset()` i/lub `get_object()` w celu pobrania żądanych obiektów modelu. Zwróć uwagę, że żadna z tych metod nie jest wywoływana w `create()`, ponieważ nie są one potrzebne do tworzenia nowych obiektów. Warto również zauważyć, że metoda `get_object()` jest wywoływana tylko w trasach szczegółowych, takich jak `retrieve()`, `update()` i `destroy()`, gdzie podany jest parametr `PK`.

Krok 6: Inicjalizacja serializera

Metoda handlera tworzy instancję klasy `ModelSerializer` i przekazuje do niej dane żądania i/lub obiekt modelu. W tym momencie nie zachodzi jeszcze walidacja ani serializacja, ponieważ DRF wykonuje te operacje leniwie.

Krok 7: Walidacja

Metoda handlera wywołuje metodę `is_valid(raise_exception=True)` na serializatorze. Zauważ, że jest ona wywoływana tylko w `create()`, `update()` i `partial_update()`, ponieważ inne akcje nie przyjmują żadnych danych wejściowych.

Wewnątrz metody `is_valid()` dzieje się kilka rzeczy:

- surowe dane są konwertowane na typy danych Pythona za pomocą metody `to_internal_value()`, która również uruchamia walidację modelu i pól. Jeśli wystąpią błędy na polu (np. wymagane pole jest

puste), zostanie zgłoszony wyjątek i nie zostanie uruchomiona jego niestandardowa walidacja.
- logika sprawdza, czy istnieje niestandardowa metoda walidacji i uruchamia ją, jeśli tak. Tutaj uruchamiane są też niestandardowe metody walidacji definiowane przez programistę.
- na koniec logika uruchamia ogólną niestandardową metodę walidacji `validate()`.
- jeżeli walidacja się nie powiedzie, a handler przekazał `raise_exception=True` do `is_valid()`, zostanie zgłoszony wyjątek `ValidationError` (400).
- metoda `is_valid()` uruchamia walidację na wszystkich polach przed zgłoszeniem wyjątku, dzięki czemu użytkownik może otrzymać wiele komunikatów o błędach naraz.

Krok 8: Wykonanie akcji żądania

Metoda handlera wywołuje `perform_create()`, `perform_update()` lub `perform_destroy()` w zależności od typu metody żądania. To tutaj zachodzi zapytanie do bazy danych.

Krok 9: Generowanie odpowiedzi

Na koniec metoda handlera tworzy instancję odpowiedzi DRF.

Wszystkie handlery, z wyjątkiem `destroy()`, zwracają reprezentację obiektu(ów) modelu. Oznacza to, że obiekt(ów) muszą być zserializowane, przekształcone w format JSON lub inne.

W tym celu handler uzyskuje dostęp do właściwości `.data` na serializatorze. Z kolei wywołuje metodę `to_representation()`, która przekształca obiekt(y) modelu w kolekcję Pythona (tj. dict lub list).

Później w cyklu ta kolekcja Pythona zostanie przekazana do klas renderujących, które zserializują ją do wymaganego formatu.

Wewnątrz obiektu odpowiedzi DRF następuje konwersja kolekcji Pythona na format(y) zdefiniowane przez `renderer_classes` w widoku lub `DEFAULT_RENDERER_CLASSES` zdefiniowane w ustawieniach. Najczęściej jest to JSON.

Następnie odpowiedź jest przesyłana w górę strumienia i przechodzi przez te same kroki, które miały miejsce na początku, ale w odwrotnej kolejności. To znaczy:

Django przesyła odpowiedź przez middleware, a następnie wraca do serwera aplikacji. Serwer aplikacji (np. Gunicorn) przekazuje ją do serwera WWW (np. Apache lub Nginx). Serwer WWW wysyła odpowiedź z powrotem do użytkownika

Przykład serializatora w DRF

Zakładając, że mamy model `Product` zdefiniowany w pliku `models.py` naszej aplikacji Django:

```
from django.db import models
from django.contrib.auth.models import User


class Category(models.Model):
    name = models.CharField(max_length=30)
    description = models.TextField(null=True, blank=True)
    subcategory = models.ManyToManyField("self", symmetrical=False)
```

```
    class Meta:
        verbose_name_plural = 'categories'

    def __str__(self):
        return f"{self.name}"


class Product(models.Model):
    name = models.CharField(max_length=200)
    description = models.TextField(null=True, blank=True)
    price = models.DecimalField(max_digits=6, decimal_places=2)
    category = models.ForeignKey(Category, on_delete=models.CASCADE)

    def __str__(self):
        return self.name
```

Stworzenie serializatora od podstaw dla tego modelu w pliku `serializers.py` może wyglądać następująco:

```
from rest_framework import serializers
from .models import Product

class ProductSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    name = serializers.CharField(max_length=100)
    description = serializers.CharField(max_length=100)
    price = serializers.DecimalField(max_digits=10, decimal_places=2)
    category = serializers.PrimaryKeyRelatedField(queryset=Category.objects.all())

    def create(self, validated_data):
        return Product.objects.create(**validated_data)

    def update(self, instance, validated_data):
        instance.name = validated_data.get('name', instance.name)
        instance.description = validated_data.get('description',
instance.description)
        instance.price = validated_data.get('price', instance.price)
        instance.category = validated_data.get('category', instance.category)
        instance.save()
        return instance
```

5. Frameworki do tworzenia API REST w Pythonie

- Django REST Framework
- Flask RESTful
- FastAPI
- Pyramid
- Falcon
- Bottle
- Eve

- Sanic
- Tornado
- Hug

Każdy z tych frameworków oferuje różne funkcje i możliwości, więc wybór odpowiedniego zależy od specyficznych potrzeb projektu.

Wady i zalety wybranych z nich.

Django + Django REST Framework (DRF)

- Zalety:
    - Bogaty zestaw funkcji, w tym serializacja, uwierzytelnianie i uprawnienia.
    - Świetna dokumentacja i duża społeczność.
    - Integracja z Django ORM.
- Wady:
    - Może być zbyt ciężki dla prostych aplikacji.
    - Wymaga dość dużej wiedzy o Django.
    - Czasami może być wolniejszy niż lżejsze frameworki.
    - Monolityczność

Flask RESTful

- Zalety:
    - Dokumentacja
    - Skalowalność
    - Lekkość
- Wady:
    - Wysokie koszty utrzymania i wdrożenia
    - Złożony stos technologiczny
    - Ryzyka związane z bezpieczeństwem

FastAPI

- Zalety:
    - Nowoczesny i szybki
    - Wbudowana walidacja danych
    - Obsługa asynchroniczności
    - Szybkość kodowania
    - Łatwe tworzenie wtyczek
- Wady:
    - Wsparcie społeczności
    - Używa Pydantic do walidacji żądań, co czasami jest trudne do zrozumienia i wymaga napisania własnego walidatora.

Pyramid

- Zalety:

    - Wykorzystanie żądań Ajax
    - Elastyczność i łatwość konfiguracji
    - Dobre dla projektów opartych na SQL

- Wady:

    - Braki w dokumentacji
    - Mniejsza społeczność w porównaniu do innych frameworków
    - Trudności w konfiguracji szablonów
    - Brak wbudowanej funkcjonalności zarządzania bazą danych

Kilka wybranych stron, które wykorzystują Django jako backend:

- Instagram: https://www.instagram.com/
- Pinterest: https://www.pinterest.com/
- Disqus: https://disqus.com/
- Mozilla: https://www.mozilla.org/
- The Washington Post: https://www.washingtonpost.com/
- Bitbucket: https://bitbucket.org/
- Prezi: https://prezi.com/
- National Geographic: https://www.nationalgeographic.com/
- Udemy: https://www.udemy.com/
- Spotify for Artists: https://artists.spotify.com/
- YouTube for Artists: https://artists.youtube.com/
- Dropbox: https://www.dropbox.com/
- Quora: https://www.quora.com/
- Reddit: https://www.reddit.com/

6. Dodatkowe materiały, linki, ciekawostki

1. Cykl żądanie-odpowiedź w Django (video): https://www.youtube.com/watch?v=P5gQmlrwLjc
2. Dość obszerne porównanie frameworków do tworzenia API REST w Pythonie: https://www.speakeasy.com/blog/choosing-your-framework-python
3. Porówanie frameworków pod kątem mikroserwisów: https://www.planeks.net/best-python-microservices-framework/
4. Dokumentacja Django REST Framework: https://www.django-rest-framework.org/
5. Jeden z przeglądów najbardziej popularnych frameworków backendowych: https://codeop.tech/key-backend-programming-languages-for-full-stack-devs-in-2024/
6. A tu przegląd technologii z ankiety Stack Overflow 2025: https://survey.stackoverflow.co/2025/technology/
7. Film przedstawiający historię powstania Django: https://www.youtube.com/watch?v=YE_SXB7xkSw

Ciekawostki:

Nazwa frameworka Django pochodzi od piosenkarza jazzowego Django Reinhardta, a to za sprawą Adriana Holovaty'ego, który był jego wielkim fanem i również grał na gitarze.

Co roku organizowana jest konferencja DjangoCon, która gromadzi społeczność programistów Django z całego świata, aby dzielić się wiedzą i doświadczeniami związanymi z tym frameworkiem. Odbywa się ona w

dwóch głównych edycjach: DjangoCon US i DjangoCon Europe. Więcej informacji można znaleźć na stronie: https://www.djangocon.org/