Tworzenie REST API w oparciu o frameworki Ruby on Rails i Spring

Mateusz Czajka
Wydział Matematyki i Informatyki
Uniwersytet im. Adama Mickiewicza w Poznaniu

Praca inżynierska

Poznań 2022

Spis treści

Wstęp

Praca ma na celu przedstawienie czym jest oraz jak tworzyć REST API z wykorzystaniem protokołu HTTP. Na początku omówione zostaną podstawowe pojęcia związane z architekturą REST. Praca opisze czym jest protokół HTTP wraz z jego własnościami, metodami oraz kodami odpowiedzi. Następnie przedstawiona zostanie sama architektura REST oraz zasady według, których powinna być implementowana. W kolejnym rozdziale praca skupi się przede wszystkim na praktycznych aspektach tworzenia REST API. Omówione zostaną dobre praktyki, których warto się trzymać podczas programowania interfejsu sieciowego. W tej sekcji pokazane zostaną również częste problemy, które można napotkać podczas tworzenia REST API oraz sposoby na ich rozwiązanie. Dodatkowo, praca opisze pewne aspekty związane z bezpieczeństwem podczas projektowania API. Na sam koniec przedstawiony zostanie plan tworzenia serwisu RESTful. Wszystko to w oparciu o przykłady stworzone przy pomocy dwóch popularnych frameworków: Ruby on Rails oraz Spring.

Rozdział 1

HTTP, API, REST

1.1 HTTP

Protokół HTTP (Hypertext Transfer Protocol) jest fundamentem, na którym postawiona jest współczesna komunikacja w internecie. HTTP działa w architekturze klient-serwer co oznacza, że zapytania, inicjowane przez odbiorcę (najczęściej przeglądarkę internetową), wysyłane są do serwera, a ten zwraca określoną odpowiedź. W celu wyświetlenia strony internetowej przeglądarka wysyła żądanie pobrania dokumentu HTML, który reprezentuję stronę. Następnie przeglądarka analizuje otrzymany plik, wykonuje dodatkowe żądania w celu uzyskania informacji na temat układu strony (CSS) czy pobrania pod-zasobom zawartym na stronie (obrazki, filmy video). Przeglądarka łączy wszystkie te zasoby, aby zaprezentować kompletny dokument - stronę WWW. Skrypty wykonywane przez przeglądarkę mogą pobierać kolejne zasoby w późniejszych fazach, a ta na bieżąco aktualizuje stronę WWW [1].

1.1.1 Własności HTTP

1. niezależne od typów danych - każdy typ danych może być przesyłany za pomocą HTTP, o ile zarówno klient, jak i serwer wiedzą, jak je obsłużyć. Wymagane jest, aby zarówno klient jak i serwer określili typ zawartości za pomocą odpowiedniego typu MIME.

2. rozszerzalne - nagłówki HTTP pozwalają w łatwy sposób dodać nowe parametry do protokołu. Należy jedynie ustalić semantykę nowego nagłówka między klientem a serwerem. Niestandardowe nagłówki wykorzystuje się w celu przesyłania dodatkowych danych na temat żądań i odpowiedzi.

3. bezstanowe, ale nie bez sesji - serwer i klient są świadomi siebie nawzajem tylko podczas bieżącego żądania. Oznacza to, że nie ma powiązania między dwoma żądaniami wykonywanymi kolejno przez tego samego klienta. Dla użytkowników, którzy chcą w ciągły sposób wchodzić w interakcję z niektórymi stronami (np. poruszać się jako zalogowany użytkownik po aplikacji e-learningowej), może stanowić do duży problem. Jednym z rozwiązań tego problemu są ciasteczka HTTP. Te pozwalają na stworzenie sesji dla każdego zapytania w celu udostępnienie tego samego kontekstu lub stanu.

4. niezawodne - protokół powinien być niezawodny, przesyłane wiadomości nie mogą być tracone. Wśród dwóch najpopularniejszych protokołów służących do transportowania danych w internecie TCP jest niezawodny, a UDP nie gwarantuje dostarczenia wiadomości. Z tego powodu HTTP oparty jest na standardzie TCP.

1.1.2 HTTPS

Połączenia z wykorzystaniem protokołu HTTP nie są szyfrowane. Oznacza to, że dane wysyłane przez klienta nie są chronione, w szczególności poufne dane, takie jak hasła. Strony, które opierają swoją komunikację o HTTP, nie są bezpieczne dla użytkowników i odchodzi się od nich na rzecz HTTPS.

HTTPS (Hypertext Transfer Protocol Secure) to rozszerzenie HTTP, które wykorzystywane jest do bezpiecznej komunikacji w internecie. Protokół komunikacji jest szyfrowany przy pomocy Transport Layer Security (TLS) lub wcześniej Secure Sockets Layer (SSL) [6].

HTTPS tworzy bezpieczny kanał, który chroni przed atakami podsłuchującymi i atakami typu "man-in-the-middle", pod warunkiem, że stosowane są odpowiednie pakiety szyfrowania, a certyfikat serwera jest zaufany i zweryfikowany.

1.1.3 Metody HTTP

HTTP definiuje dziewięć metod, które określają jaka akcja powinna zostać wykonana dla określonego zasobu [2].

- GET - żąda reprezentacji określonego zasobu. Żądania używające metody GET powinny być wykorzystywane jedynie do pobierania danych [10].

- HEAD - żąda takiej samej odpowiedzi jak GET, ale bez treści odpowiedzi (response body).

- POST - wysyła dane do serwera, najczęściej w celu stworzenia zasobu.
- PUT - wysyła dane do serwera w celu zasępienia zasobu nową reprezentacją.
- PATCH - wysyła dane do serwera w celu częściowej aktualizacji zasobu.
- DELETE - usuwa określony zasób.
- CONNECT - ustanawia tunel do serwera.
- OPTIONS - opisuje możliwości komunikacji.
- TRACE - wykonuje test zwrotny do zasobu docelowego.

1.1.4 Kody odpowiedzi HTTP

Kod odpowiedzi HTTP informuje użytkownika o statusie żądania HTTP. Kody są podzielone na pięć grup: [3]

1. kody informacyjne (100 - 199)
2. kody powodzenia (200 - 299)
3. kody przekierowania (300 - 399)
4. kody błędu aplikacji klienta (400 - 499)
5. kody błędu aplikacji serwera (500 - 599)

Powyższe kody statusów są zdefiniowane przez RFC 9110. Najczęściej używane kody statusów:

- 200 OK - zapytanie powiodło się.
- 201 Created - zapytanie powiodło się i powstał nowy zasób.
- 204 No Content - zapytanie powiodło się, ale nie ma dla niego treści odpowiedzi.
- 301 Moved Permanently - URL zasobu zmieniło się na stałe. Nowe URL jest podawane w odpowiedzi.
- 302 Found - zasób został odnaleziony, jednak nie w miejscu, w którym był oczekiwany.

- 400 Bad Request - serwer nie obsłuży zapytanie z powodu błędu w zapytaniu klienta.
- 401 Unauthorized - klient musi się uwierzytelnić, aby otrzymać odpowiedź od serwera.
- 403 Forbidden - klient nie ma uprawnień do zawartości.
- 404 Not Found - serwer nie może znaleźć żądanego zasobu. Oznacza to, że URL nie jest rozpoznane lub zasób nie istnieje.
- 405 Method Not Allowed - użyta została nieodpowiednia metoda HTTP.
- 415 Unsupported Media Type - rodzaj danych wysłanych w zapytaniu nie jest obsługiwany przez serwer.
- 500 Internal Server Error - serwer napotkał na sytuację, z która nie jest w stanie sobie poradzić.
- 502 Bad Gateway - serwer otrzymał nieprawidłową odpowiedź, kiedy próbował uzyskać odpowiedź potrzebną do obsłużenia zapytania.
- 503 Service Unavailable - serwer nie jest gotowy do obsłużenia zapytania.

1.2 API

API (Application Programming Interface) jest specyfikacją wytycznych opisujących, w jaki sposób programy komunikują się między sobą. Oprogramowanie upublicznia interfejs dla świata zewnętrznego (użytkowników sieci lub innych programów i serwisów). API jest tworzone, aby programy mogły komunikować się ze sobą, lecz muszą być zrozumiałe dla człowieka, który jest odpowiedzialny za kod aplikacji [8].

Interfejsy API są kluczowym komponentem, który umożliwia funkcjonowanie aplikacjom w internecie. API pozwala na zamówienie pizzy przez internet, zapłatę za subskrypcję VOD, obejrzenie filmu video i wiele więcej.

1.3 Architektura REST

Przed rokiem 2000 nie istniał standard, który mówił w jaki sposób tworzyć i używać API w sieci Internet. Integracje wymagały użycia protokołów takich jak SOAP,

które były trudne do implementacji, utrzymania i debugowania. Dopiero Roy Fielding w swojej pracy "Architectural Styles and the Design of Network-based Software Architectures"[9] stworzył REST (Representational state transfer), który ustandaryzował tworzenie i korzystanie z API. Celem pracy było stworzenie standardu, który pozwoli dwóm serwerom na komunikacje i wymianie informacji z dowolnego miejsca na świecie. Z tego powodu zaprojektowano szereg zasad, właściwości i ograniczeń, które nazwano REST. REST to architektura oparta na zasobach, jednolitości interfejsu, architekturze klient-serwer, bezstanowości, używająca cachu i wykorzystująca protokół do przesyłania danych (najczęściej HTTP).

1.3.1 REST API

REST to obecnie najpopularniejsza architektura wybierana przez programistów do tworzenia API [8]. API tworzone w oparciu o paradygmaty REST nazywane jest REST API (RESTful API).

Zapytanie wysyłane za pomocą interfejsu REST API przekazuje reprezentację stanu zasobu do odpowiedniego endpointu aplikacji. Dane są dostarczane przy pomocy HTTP w określonym formacie (najczęściej JSON/XML). Najczęściej używanym em jest JSON, ponieważ jest niezależny od języka i jest czytelny dla człowieka oraz programów. Ważnym elementem zapytań REST API są nagłówki i parametry HTTP. Mogą one zawierać ważne informacje identyfikacyjne, takie jak metadane żądania, uprawnienia, URI (Uniform Resource Identifier), ciasteczka i inne.

1.3.2 Zasady REST API

Kilka podstawowych zasad, których należy się trzymać w trakcie tworzenia REST API z wykorzystaniem HTTP:

- Zasoby są częścią URL (/companies)
- Dla każdego zasobu powinny istnieć 2 rodzaje URLi, jeden dla kolekcji (/companies) i drugi dla określonego elementu (/companies/10)
- Używa się rzeczowników zamiast czasowników dla zasobów (zamiast /getCompanyInfo/10 używa się /companies/10)
- Metody HTTP (GET, POST, PATCH, DELETE) informują serwer jaka akcja powinna zostać wykonana. Różne metody wykonują różne akcje na tym samym URL [7].

Operacja	Metoda HTTP	/companies	/companies/10
Create	POST	tworzy nową firmę	Nie stosuje się
Read	GET	zwraca wszystkie firmy	zwraca firmę 10
Update	PATCH/PUT	aktualizuje wiele firm	aktualizuje firmę 10
Delete	DELETE	usuwa wszystkie firmy	usuwa firmę 10

Tabela 1.1: CRUD (Create, Read, Update, Delete)

- Zasób, który istnieje tylko w innym zasobie (nested resources) powinien być przedstawiony jako zasób podrzędny, a nie jako zasób najwyższego poziomu w adresie URL. Dzięki temu programista korzystający z API klarownie widzi relację między zasobami.

Metoda HTTP	/companies/:company/notes	/companies/:company/notes/:note
POST	tworzy notatkę firmy	Nie stosuje się
GET	zwraca notatki firmy	zwraca notatkę firmy
PATCH/PUT	aktualizuje notatki firmy	aktualizuję notatkę firmy
DELETE	usuwa wszystkie notatki firmy	usuwa notatkę firmy

Tabela 1.2: zasób zagnieżdżony - CRUD

- Oprócz operacji CRUD, czasami potrzeba zastosować inną operację. W takim wypadku często wykorzystywane są następujące podejścia[8]:
  - Wskazanie na akcje poprzez przekazanie parametru w treści zapytania.

    `PATCH /companies/:company/notes/:note z parametrem {"archived": true}`

    archiwizuję notatkę firmy.

  - Traktowanie akcji jak podzasobu.

`POST /companies/:company/notes/:note/lock`

blokuję notatkę firmy

- Operacje takie jak szukanie mogą być jeszcze trudniejsze do zaimplementowania w architekturze REST. Typową praktyką w takim przypadku jest użycie czasownika wskazującego na akcje wraz z potrzebnymi parametrami w adresie URL.

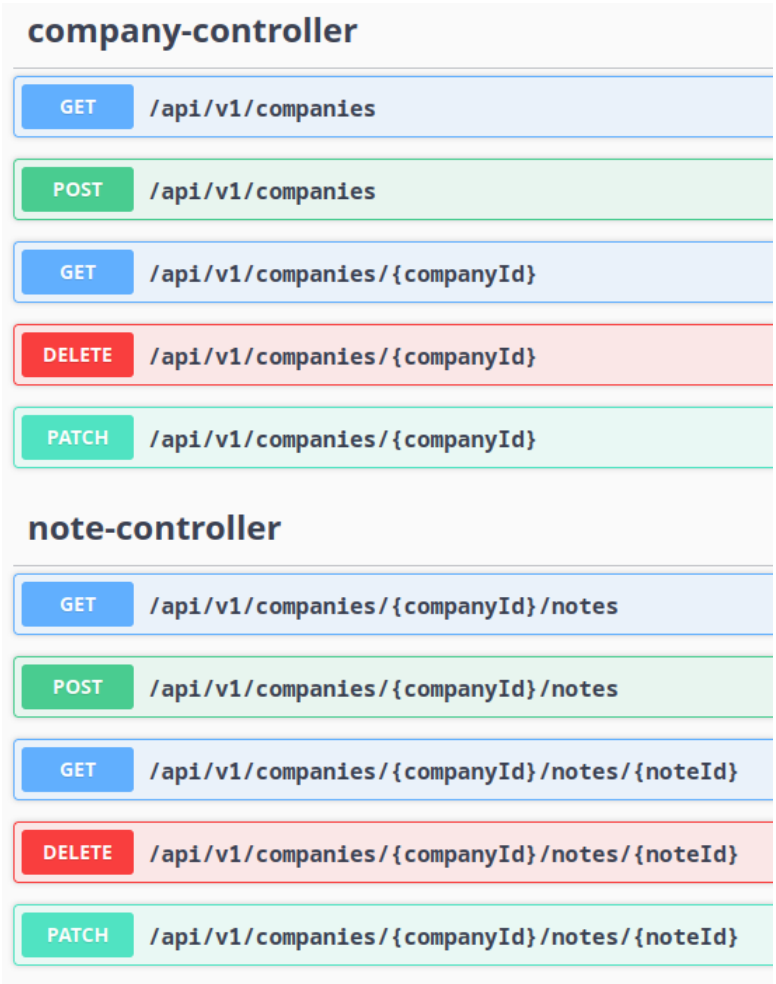
  `GET /companies/:company/notes/search?q="query"`

  wyszukuje odpowiednie notatki

- Statusy kodów HTTP są zwracane przez serwer w celu poinformowania co się stało z zapytaniem. Kody statusów 2XX informują o sukcesie, 3XX o tym, że zasób został przeniesiony, 4XX o błędzie po stronie klienta, a 5XX o błędzie po stronie serwera [10].

1.3.2.1 Spring

Endpointy do operacji CRUD przygotowane przy pomocy Springa. Company-controller udostępnia najpopularniejsze endpointy dla firm. Z kolei note-controller odpowiada za operacje dla zasobu zagnieżdżonego note, który istnieje tylko jako zasób podrzędny względem company.



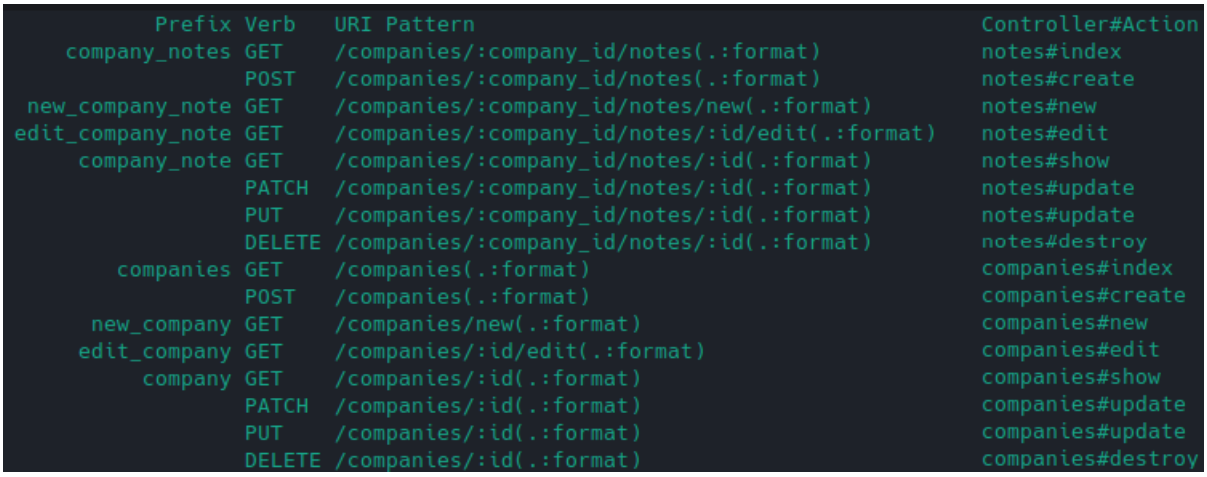
Rysunek 1.1: Spring - endpointy w Swagerze

1.3.2.2 Ruby on Rails

Endpointy wygenerowane przy pomocy komendy [4]

```
rails generate scaffold
```

oraz przygotowaniu zasobu note jako zasobu podrzędnego względem company:



```
           Prefix Verb   URI Pattern                                       Controller#Action
    company_notes GET    /companies/:company_id/notes(.:format)            notes#index
                  POST   /companies/:company_id/notes(.:format)            notes#create
 new_company_note GET    /companies/:company_id/notes/new(.:format)        notes#new
edit_company_note GET    /companies/:company_id/notes/:id/edit(.:format)   notes#edit
     company_note GET    /companies/:company_id/notes/:id(.:format)        notes#show
                  PATCH  /companies/:company_id/notes/:id(.:format)        notes#update
                  PUT    /companies/:company_id/notes/:id(.:format)        notes#update
                  DELETE /companies/:company_id/notes/:id(.:format)        notes#destroy
        companies GET    /companies(.:format)                              companies#index
                  POST   /companies(.:format)                              companies#create
      new_company GET    /companies/new(.:format)                          companies#new
     edit_company GET    /companies/:id/edit(.:format)                     companies#edit
          company GET    /companies/:id(.:format)                          companies#show
                  PATCH  /companies/:id(.:format)                          companies#update
                  PUT    /companies/:id(.:format)                          companies#update
                  DELETE /companies/:id(.:format)                          companies#destroy
```

Rysunek 1.2: Ruby on Rails - endpointy

Jak widać, oprócz podstawowych akcji CRUD Ruby on Rails wygenerował też dwa dodatkowe endpointy, które służą do zwrócenia widoków z formularzami do tworzenia (/companies/new) i akutalizacji (/companies/:id/edit) zasobu. Warto zauważyć, że Ruby on Rails dopuszcza metodę PATCH oraz PUT w celu edycji zasobu. Ponadto, do każdego endpointu dodany jest parametr :format, który wskazuje w jakim formacie danych zostanie zwrócona odpowiedź.

Na przykład, dla zapytania:

`http://localhost:3000/companies/?format=json`

zwrócone zostaną dane w formacie json:

Rysunek 1.3: Ruby on Rails - odpowiedź json

Z kolei dla ?format=html zwrócona zostanie zwykła strona html.

Rozdział 2

Tworzenie REST API

2.1 Dobre praktyki

Interfejsy REST API są obecnie jednym z najbardziej popularnych rodzajów usług w internecie. Pozwalają różnym klientom, w tym przeglądarką, na komunikację z serwisem. Bardzo ważne jest, aby prawidłowo zaprojektować REST API. Podczas tworzenia API należy wziąć pod uwagę bezpieczeństwo, wydajność oraz łatwość użycia interfejsu dla konsumentów. Z tego powodu warto poznać i stosować się do powszechnie przyjętych konwencji. W przeciwnym razie klienci oraz programiści utrzymujący projekt mogą poczuć się zdezorientowani podczas korzystania z API.

W tej sekcji opisanie zostanie kilka praktyk, które warto stosować przy tworzeniu REST API, aby to było proste w użyciu oraz utrzymaniu.

2.1.1 Dokumentacja

API jest tak dobre, jak jego dokumentacja. Dokumenty opisujące działanie API powinny być dostępne publicznie i łatwo wyszukiwalne. Większość programistów sprawdzi dokumentację, zanim podejmie się jakichkolwiek prób integracji, dlatego ukrywanie jej w plikach PDF czy udostępnianie tylko po wcześniejszym zalogowaniu może zniechęcić potencjalnych klientów do korzystania z API.

Dokumentacja powinna zawierać przykłady pełnych żądań i odpowiedzi. Najlepiej, aby przedstawić je w postaci możliwej do łatwego skopiowana (linki, curle lub całe zapytania, których można użyć np. w Postmanie).

Po publicznym udostępnieniu API nie należy wprowadzać do niego żadnych zmian, które sprawią, że programy klientów przestaną poprawnie się z nim komunikować. Dokumentacja musi zawierać informację o poprzednich wersjach API oraz zmianach, które zostały wprowadzone.

2.1.1.1 Interaktywna dokumentacja

Interaktywna dokumentacja pozwala programistom pracować z API, dając im jasny obraz tego, jak API odpowiada na żądania z różnymi parametrami i opcjami. Zapewnia wszystko, czego deweloperzy potrzebują, aby korzystać z interfejsu przy niewielkim nakładzie czasu i wysiłku. Taka dokumentacja jest niezwykle przydatna, ponieważ umożliwia testowanie, eksperymentowanie i używanie API. Przykładami narzędzi, które umożliwiają dokumentację oraz korzystanie ze stworzonego API są Postman czy Swagger.

2.1.2 Wersjonowanie

API powinno mieć określoną wersję. Dzięki temu można wprowadzać duże zmiany w nowszych wersjach i utrzymywać stare wersje dla klientów, którzy nie są jeszcze gotowi do migracji ich aplikacji do nowego API. Wersja API jest najczęściej zawarte w adresie URL, aby zapewnić możliwość przeszukiwania zasobów w przeglądarce oraz ułatwia programistom pracę z API.

```
http://www.example.com/api/v2/companies, v2 - wersja API
```

Bardzo rzadko API będzie całkowicie stabilne. Ważne, aby w odpowiedni sposób informować użytkowników o zachodzących zmianach oraz utrzymywać starsze wersje API. Dobra dokumentacja i odpowiednie wersjonowanie pozwoli zapobiec wielu problemom i sprawi, że programiści będą w stanie w prosty sposób zintegrować się z API.

2.1.3 Sortowanie, filtrowanie i wyszukiwanie

Podstawowe adresy URL powinny być tak proste, jak to możliwe. Złożone filtry, żądania sortowania i zaawansowane wyszukiwanie (jeśli ograniczone do jednego zasobu) można zaimplementować jako parametry zapytania.

1. Filtrowanie
   Należy użyć unikalnego parametru zapytania dla każdego pola, które implementuje filtrowanie.

   ```
   GET /companies?state=active
   ```

   zwraca tylko aktywne firmy

2. Sortowanie
   Podobnie jak w przypadku filtrowania, można zastosować ogólny parametr (np. sort) do opisania reguł sortowania [12]. W celu bardziej złożonych operacji sortowania zezwala się na podanie listy pól oddzielonych przecinkami. Używa się '-', aby sortować malejąco względem pola.

   ```
   GET /companies?sort=name
   ```

   zwraca listę firm w porządku rosnącym względem nazwy.

   ```
   GET /companies?sort=-name
   ```

   zwraca listę firm w porządku malejącym względem nazwy.

   ```
   GET /companies?sort=-amount_employees,name
   ```

   zwraca listę firm w porządku malejącym względem liczby pracowników, W ramach tej samej liczby pracowników sortuje się firmy rosnąco względem nazwy.

3. Wyszukiwanie
   Czasami podstawowe filtry i parametry sortowania nie wystarczą i potrzebne są bardziej zaawansowane wyszukiwania. Jeśli wyszukiwanie jest używane jako mechanizm pobierania zasobów, można udostępnić metodę w API z parametrem zapytania (np. query) [12]. Łącząc razem wszystkie przedstawione powyżej sposoby pozyskiwania danych, można tworzyć zapytania:

   ```
   GET /companies?state=active&sort=name
   ```

   zwraca listę aktywnych firm posortowanych po nazwie.

   ```
   GET /companies/:company/notes?state=open&sort=updated_at&query="REST API"
   ```

   zwraca listę notatek, które są otwarte, posortowane rosnąco po dacie ich aktualizacji oraz zawierają tekst "REST API".

4. Aliasy
   Dobrym pomysłem jest stworzenie osobnych metod w API dla najczęściej używanych opcji wyszukiwania. Ułatwi i przyspieszy to korzystanie z API.

   `GET /companies/recently_created`

   zwraca listę niedawno stworzonych firm.

2.1.4 Zwracanie zasobów

Użycie w zapytaniu metod POST, PATCH i PUT często powoduje zmiany w polach zasobu. Za dobrą praktykę uznaje się zwracane zaktualizowanej reprezentacji zasobu w ramach odpowiedzi.

2.1.4.1 Spring

W tym celu warto wykorzystać klasę ResponseEntity wraz z pozytywnym kodem odpowiedzi HTTP oraz zasobem.

```
@PostMapping
public ResponseEntity<Company> create(@RequestBody String name) {
    Company company = companyService.createCompany(name);

    return ResponseEntity
            .status(HttpStatus.OK)
            .body(company);
}
```

Rysunek 2.1: Spring - tworzenie firmy

2.1.4.2 Ruby on Rails

W zależności od parametru "format"framework renderuje widok HTML z zasobem lub sam zasób w formacie json.

```ruby
# POST /companies or /companies.json
def create
  @company = Company.new(company_params)

  respond_to do |format|
    if @company.save
      format.html { redirect_to company_url(@company), notice: "Company was successfully created." }
      format.json { render :show, status: :created, location: @company }
    else
      format.html { render :new, status: :unprocessable_entity }
      format.json { render json: @company.errors, status: :unprocessable_entity }
    end
  end
end
```

Rysunek 2.2: Ruby on Rails - tworzenie firmy

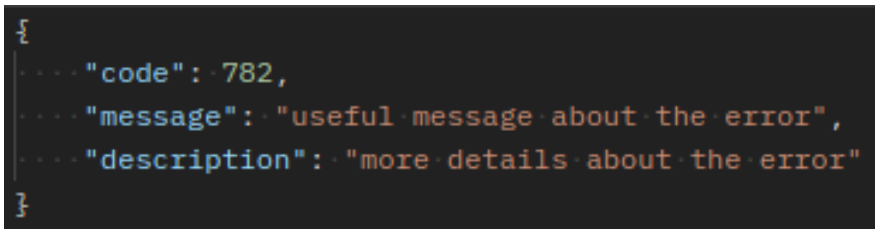
2.1.5 JSON czy XML?

Współcześnie, uważa się, że XML nie jest najlepszym formatem danych w przypadku REST API. Jest zbyt szczegółowy, trudny do analizy i odczytu. Zdecydowana większość współczesnych API jest oparta o JSON. Format ten pożycza niektóre z dobrych stron JavaScriptu i korzysta z bezproblemowej integracji z natywnym środowiskiem przeglądarki [11]. Jeśli nie istnieje jeszcze standardowy format dla danego typu zasobu (np. image/jpeg dla obrazów skompresowanych w JPEG), interfejs REST API powinien używać formatu JSON do strukturyzacji swoich informacji.

2.1.6 Błędy

API, kiedy zajdzie taka potrzeba, powinno dostarczać użytkownikowi użyteczny komunikat o błędzie w znanym formacie. Odpowiedź z błędem nie powinna różnić się strukturą od standardowych odpowiedzi (np. z aktualizowanym zasobem) [12]. API w każdej odpowiedzi powinno zwracać kod statusu HTTP, aby poinformować programistę o stanie, w jakim znalazło się zapytania.
Ciało błędów JSON powinno dostarczyć deweloperom:

- przydatny komunikat o błędzie
- kod błędu (który jest dokładnie opisany w dokumentacji)
- być może szczegółowy opis



Rysunek 2.3: Odpowiedź z wiadomością o błędzie

Błędy walidacji dla zapytań POST, PATCH i PUT wymagają dodatkowej informacji o polach zasobu. Dobrym sposobem na poradzenie sobie z tym problemem jest użycie kodu błędu najwyższego poziomu dla błędów walidacji i dostarczenie szczegółowych błędów dla poszczególnych pól.

Rysunek 2.4: Odpowiedź z wiadomością o błędzie z opisem pól

2.1.6.1 Ruby on Rails

Wygenerowany kod przy pomocy komendy

```
rails generate scaffold
```

służący do stworzenia nowej firmy jest w stanie obsłużyć błędy. Metoda zwraca odpowiednią wiadomość o błędzie w zależności od przekazanego w zapytaniu typu danych (html/json).

```ruby
# POST /companies or /companies.json
def create
  @company = Company.new(company_params)

  respond_to do |format|
    if @company.save
      format.html { redirect_to company_url(@company), notice: "Company was successfully created." }
      format.json { render :show, status: :created, location: @company }
    else
      format.html { render :new, status: :unprocessable_entity }
      format.json { render json: @company.errors, status: :unprocessable_entity }
    end
  end
end
```

Rysunek 2.5: Ruby on Rails - obsługa błędów

Dla zapytania, w którym nazwa firmy jest już zajęta np.

```
http://localhost:3000/companies/?format=json
z response body {"company": {"name": "name in use"}}
```

Aplikacja zwróci kod błędu 422 (Unprocessable Entity) wraz z informacją:

```
{
    "name": [
        "has already been taken"
    ]
}
```

Rysunek 2.6: Ruby on Rails - odpowiedź z wiadomością o błędzie

2.1.6.2 Spring

W przypadku Springa nie możemy liczyć na automatycznej wygenerowany kod. Można, tak jak w przypadku metody stworzonej przy pomocy komendy w Ruby on Rails obsługiwać błędy pojedynczo, ale popularnym rozwiązaniem jest zastosowanie klasy, która przechwytuje wyjątki (na poziomie kontrolera lub globalnie), a następnie zwraca

odpowiednią wiadomość dla klienta [5]. Obsłużenie pojedynczego wyjątku może wyglądać w następujący sposób:

```
@ExceptionHandler({BadCredentialsException.class})
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ResponseBody
public ApiError handleException(BadCredentialsException e)
{
    return logError( message: "Could not log in user, bad credentials.", e);
}
```

Rysunek 2.7: Spring - obsługa wyjątku

Z kolei aplikacja kliencka otrzyma odpowiedni kod błędu wraz z wiadomością:

```
{
    "type": "BadCredentialsException",
    "message": "Could not log in user, bad credentials."
}
```

Rysunek 2.8: Spring - odpowiedź z wiadomością o błędzie

2.1.7 Testowanie

Warstwa API każdej aplikacji jest jednym z najbardziej kluczowych komponentów oprogramowania. Jest to kanał, który łączy klienta z serwerem (lub serwis z innym serwisem), napędza procesy biznesowe i zapewnia usługi, które dają namacalną wartość użytkownikom.

Publiczny interfejs API skierowany do klienta, który jest wystawiony na działanie użytkowników końcowych, staje się produktem samym w sobie. Popsucie się go zagraża nie tylko pojedynczej aplikacji, ale całemu łańcuchowi procesów biznesowych zbudowanych wokół niej.

Testy API są szybkie, upraszczają walidację logiki biznesowej, bezpieczeństwa, zgodności i innych aspektów aplikacji. W przypadku, gdy API jest publiczne i zapewnia użytkownikom końcowym programowy dostęp do aplikacji lub usług, testy API efektywnie stają się testami end-to-end i powinny obejmować pełen scenariusz użytkowania funkcjonalności.

2.1.7.1 Plan testowania

Znaczenie testów API jest oczywiste, ale co i jak powinno się testować w przypadku API? Proces testowania należy rozpocząć od testów funkcjonalnych, aby zapewnić, że interfejs działa poprawnie. Głównymi celami takiego testowania jest:

- zapewnienie, że implementacja działa poprawnie - bez błędów.
- zapewnienie, że implementacja działa zgodnie ze specyfikacją wymagań.

Każdy test powinien składać się z kilku punktów, które musi spełnić, aby zakończył się powodzeniem:

1. zweryfikowanie poprawności kodu odpowiedzi HTTP. Na przykład, niedozwolone żądanie powinno zwrócić 403 (FORBIDDEN).
2. zweryfikowanie ciała odpowiedzi, poprawności nazw pól, typów i wartości - również w odpowiedziach o błędach.
3. zweryfikowanie nagłówków odpowiedzi, mają one wpływ zarówno na bezpieczeństwo jak i wydajność.
4. opcjonalnie, można sprawdzić poprawność stanu aplikacji jeśli testy są wykonywane manualnie lub dostęp do UI jest prosty
5. zweryfikowanie podstawowej wydajności, jeśli operacja zakończyła się pomyślnie, ale zajęła nieuzasadnioną ilość czasu, test powinien zakończyć się niepowodzeniem.

2.1.7.2 Kategorie scenariuszy testowych

Testy można podzielić na następujące, ogólne grupy scenariuszy testowych:

1. podstawowe testy pozytywne (sprawdzają podstawową funkcjonalność i kryteria akceptacji API).
2. rozszerzone testy pozytywne z opcjonalnymi parametrami (sprawdzają opcjonalne parametry i dodatkową funkcjonalność).
3. negatywne testy z ważnymi danymi wejściowymi (sprawdzają czy aplikacja obsłuży scenariusze problemowe z ważnymi danymi np. rejestracja użytkownika z e-mailem, który nie jest unikalny).

4. negatywne testy z nieważnymi danymi wejściowymi (sprawdzają czy aplikacja obsłuży scenariusze problemowe z mniej ważnymi danymi np. zmiana wieku użytkownika na wartość tekstową).

5. testy destrukcyjne (testy, które polegają na celowym zepsuciu API w celu sprawdzenia jego stabilności).

6. testy bezpieczeństwa, autoryzacji i uprawnień.

2.2 Uwierzytelnianie i autoryzacja

Podczas obsługi niektórych żądań aplikacja musi wiedzieć, który użytkownik jest za nie odpowiedzialny. Uwierzytelnianie to problem powiązania żądania z użytkownikiem. Z kolei autoryzacja to problem określenia, czy użytkownik ma prawo wykonać daną akcje. Jeśli klient A chce usunąć swoje konto trzeba mieć absolutną pewność, że to właśnie klient A jest odpowiedzialny za wywołanie akcji (jeśli ustalimy, że tylko właściciel konta może zarządzać swoim kontem).

Kiedy klient API wysyła żądanie HTTP, może zawrzeć pewne dane uwierzytelniające w nagłówku HTTP Authorization. Usługa sprawdza poświadczenie i decyduje, czy poprawnie identyfikują one klienta jako konkretnego użytkownika (uwierzytelnianie), oraz czy ten użytkownik jest rzeczywiście upoważniony do zrobienia tego o co prosi (autoryzacja). Jeśli oba warunki są spełnione, serwer realizuje żądanie [10]. Jeżeli brakuje danych uwierzytelniających, bądź są one niepoprawne, aby zapewnić autoryzację, serwer wysyła komunikat o braku danych uwierzytelniających 401 (Unauthorized). W przypadku kiedy klient uwierzytelnił się poprawnie, ale nie ma prawa na wykonanie akcji, serwer zwraca komunikat 403 (Forbidden).

2.2.1 Uwierzytelnianie

Najpopularniejszymi sposobami na implementację uwierzytelniania są:

- Ciasteczka sesyjne
  Po pomyślnym zalogowaniu się użytkownika:

  1. serwer generuje token, który jednoznacznie identyfikuje sesję użytkownika.
  2. serwer podpisuje token sekretnym kluczem.
  3. serwer wiąże token z użytkownikiem i zapisuje go w bazie.
  4. serwer wysyła ciasteczko z token do przeglądarki klienta.

5. przeglądarka otrzymuje ciasteczko i zapisuje je lokalnie.
6. następnie przeglądarka dołącza ciasteczko do każdego zapytania wysyłanego na serwer.
7. dla każdego żądania serwer pobiera token z ciasteczka i porównuje go z tym, które zostało zapisane w bazie i jeśli tokeny są takie same obsługuje żądanie.

W podejściu opartym na ciasteczkach sesyjnych token nie zawiera żadnych informacji na temat użytkownika, jest jedynie losowym łańcuchem znaków wygenerowanym i podpisanym przez tajny klucz. Token jest zapisywany w bazie danych po stronie serwera w relacji One-to-One z użytkownikiem w celu późniejszego zidentyfikowania go na podstawie danych w tokenie.

- JSON Web Token (JWT)
  Po pomyślnym zalogowaniu się użytkownika:

  1. serwer generuje JWT token i szyfruje go.
  2. serwer wysyła token do przeglądarki klienta.
  3. przeglądarka otrzymuje token i zapisuje go lokalnie.
  4. następnie przeglądarka dołącza token do każdego zapytania wysyłanego na serwer przy pomocy nagłówka Authorization.
  5. dla każdego żądania serwer pobiera token z nagłówka Authorization, odszyfrowuje go i wydobywa dane o użytkowniku i jego uprawnieniach. Opierając się wyłącznie na danych z tokena jest w stanie zaakceptować lub odrzucić żądanie klienta.

  W podejsciu JWT sam token zawiera zaszyfrowany sekret, więc token nie jest przechowywany w żadnej bazie danych po stronie serwera. Wszystkie potrzebne informacje o użytkowniku są przechowywane w tokenie JWT.

2.2.1.1 OAuth

OAuth jest otwartym protokołem, który pozwala użytkownikom zweryfikować się za pomocą zaufanego podmiotu (Google, Microsoft Azure czy Facebook) poprzez wymianę tokenów, aby uzyskać dostęp do aplikacji. Najnowsza wersja protokołu, OAuth 2.0 jest uważana za standard w branży i wykorzystywana przez największe firmy.

Największą zaletą korzystania z OAuth jest przerzucenie odpowiedzialności za zarządzanie hasłami na podmioty zewnętrzne. Zmniejsza to ilość przechowywanych danych użytkowników oraz przyśpiesza proces tworzenia aplikacji. Dodatkowo, użytkownicy nie potrzebują nowego konta i hasła w celu korzystania z usługi, ponieważ posługują się już istniejącym kontem.

2.2.2 Autoryzacja

Celem autoryzacji jest sprawdzenie, czy użytkownik jest uprawniony do korzystania z określonego zasobu (np. tylko administrator może wyświetlić stronę z listą wszystkich użytkowników serwisu). Autoryzację najczęściej implementuje się za pomocą ról. Oznacza to, że każdemu użytkownikowi jest przypisana rola (np ADMIN, EMPLOYEE, CLIENT), którą przechowuje się w bazie danych. Następnie, dla każdego żądania, które wymaga konkretnych uprawnień serwer sprawdza czy klient jest upoważniony do wykonania akcji.

2.3 plan tworzenia serwisu Restful [10]

Przed rozpoczęciem programowania REST API warto stworzyć plan, który ułatwi i ustandaryzuje proces tworzenia oprogramowania. Poniższa procedura zawiera wszystkie standardowe kroki, które należy zrealizować, aby dodać nową funkcjonalność do REST API. Trzymanie się takiego planu ułatwi deweloperom pracę z API oraz pozwoli uniknąć wielu błędów.

1. Ustal zestaw danych
2. Podziel zestaw danych na zasoby

   Dla każdego zasobu:

3. Nazwij zasób przy pomocy URI (Uniform Resource Identifier - sekwencja znaków, która identyfikuje zasób np. example.com/api/v1/companies/:companyID [11]).
4. Udostępnij endpointy potrzebne do korzystania z zasobu.
5. Zaprojektuj reprezentacje zasobu przesyłane do serwera przez klienta.
6. Zaprojektuj reprezentacje zasobu udostępnianie dla klienta.

7. Zintegruj nowy zasób z istniejącymi zasobami.
8. Rozważ typowe flow związane z zasobem (co powinno się z nim stać?).
9. Rozważ, w jakich sytuacjach mogą wystąpić błędy (co może pójść nie tak?).

Zakończenie

Praca przedstawiła podstawowe koncepty związane z tworzeniem REST API. Na początku pokazany i opisany został protokół HTTP oraz jego rozszerzenie - HTTPS. W pracy zawarte zostały podstawowe informacje na temat metod HTTP oraz popularnych kodów odpowiedzi które należy wykorzystywać w architekturze REST. W kolejnej części praca skupiła się na istocie REST API, a więc samej architekturze REST oraz zasadach, które należy przestrzegać, by określić serwis RESTowym. Rozdział drugi, "Tworzenie REST API"położył nacisk na praktyczne aspekty programowania interfejsu sieciowego. Rozpoczął się od sekcji dobrych praktyk, gdzie opisany został szereg istotnych elementów przy tworzeniu REST API, od dokumentacji i wersjonowania oprogramowania po obsługę błędów oraz testowanie. Następnie praca przedstawiła dwa mechanizmy na implementację uwierzytelniania użytkowników, opisała czym jest OAuth oraz wytłumaczyła na czym polega autoryzacja. W ostatniej sekcji zademonstrowany został plan, który warto znać podczas projektowania REST API.

W dobie ciągłego przepływu informacji w sieci Internet, należy tworzyć REST API, które są przyjemne w użytkowaniu oraz zdatne do utrzymania w celu zmaksymalizowania swojej atrakcyjności dla potencjalnych klientów. Współcześnie, wykrystalizowały się pewnie branżowe standardy, których warto trzymać się przy implementacji REST API.

W tej pracy wytłumaczono czym właściwie jest REST API oraz przedstawiono wiele praktyk, które warto wdrożyć podczas programowania interfejsów sieciowych, aby dostarczyć użytkownikom oprogramowania produkt wysokiej jakości.

Spis tabel i rysunków

Spis tabel

Spis rysunków

Bibliografia

[1] Developer mozilla - an overview of http. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

[2] Developer mozilla - http request methods. https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods.

[3] Developer mozilla - http response status codes. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status.

[4] Ruby on rails documentation. https://guides.rubyonrails.org/.

[5] Spring documentation. https://docs.spring.io/spring-framework/docs/current/reference/html/.

[6] Wikipedia - htps. https://en.wikipedia.org/wiki/HTTPS.

[7] Subbu Allamaraju. *RESTful Web Services Cookbook*.

[8] Brenda Jin Amir Shevat and Saurabh Sahni. *Designing Web APIs: Building APIs That Developers Love*.

[9] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures, 2000.

[10] Sam Ruby Leonard Richardson. *RESTful Web Services*.

[11] Mark Masse. *REST API Design Rulebook*.

[12] Vinay Sahni. Best practices for designing a pragmatic restful api. https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api/.