POLITECHNIKA POZNAŃSKA
Wydział Informatyki i Telekomunikacji

P R A C A D Y P L O M O W A
I N Ż Y N I E R S K A

Autor:
Rehina Pashkevych

Aplikacyjny interfejs programistyczny zgodny ze specyfikacją REST (RESTful API) – przykład dydaktyczny

Promotor:
prof. dr hab. inż. Grzegorz Danilewicz

Poznań, styczeń 2024

Spis treści

1. Wprowadzenie

Paradygmat architekturalny REST (ang. REpresentational State Transfer) został wprowadzony na początku XXI wieku jako abstrakcyjny model wzorca projektowania rozproszonych systemów informatycznych. Opiera się on na ogólnych założeniach i ograniczeniach, które po raz pierwszy sformułowano i skodyfikowano w ramach pracy badawczej dotyczącej nowatorskich rozwiązań w dziedzinie architektury sieciowych systemów programowych.

Tematem niniejszej pracy jest zaprojektowanie i implementacja aplikacyjnego interfejsu programistycznego (API) zgodnego ze specyfikacją REST. Celem pracy jest stworzenie dydaktycznego przykładu takiego interfejsu, aby zilustrować zasady jego projektowania oraz wykorzystania w budowie nowoczesnych systemów informatycznych opartych na architekturze mikroserwisów. Rozwiązania takie są współcześnie wykorzystywane na przykład w sieciach szkieletowych 5G.

W rozdziale 2 omówione zostaną ogólne zasady tworzenia mikroserwisów oraz architektura i zasady projektowania mikroserwisów. Rozdział 3 szczegółowo omówi interfejsy REST API – ich cele, zalety i dobre praktyki projektowania. Zostaną wyjaśnione podstawowe elementy, takie jak identyfikatory URI, operacje CRUD i odpowiednie kody odpowiedzi HTTP. W rozdziale 4 zaprezentowany zostanie projekt systemu będącego przedmiotem pracy – jego architektura, wybrane technologie implementacji oraz szczegółowy opis poszczególnych mikroserwisów i API.

Praca ma na celu kompleksowe omówienie i przykładową realizację procesu projektowania i implementacji nowoczesnego REST API w oparciu o aktualne standardy i dobre praktyki. Zaproponowane rozwiązania implementacyjne mogą być potencjalnie wykorzystane w realnych projektach informatycznych. Opracowany system dydaktyczny może posłużyć do nauczania studentów oraz specjalistów IT nowoczesnych technik programistycznych z zakresu tworzenia REST API. Materiał zawarty w pracy może więc mieć realne zastosowanie zarówno w aspekcie naukowo-badawczym, jak i praktycznym.

2. Mikroserwisy

2.1. Definicja i cechy

Pojęcie mikroserwisów zaczęło pojawiać się w branży informatycznej od lat 2000-nych [1] i [2], zyskując popularność jako innowacyjna architektura programistyczna. W tym czasie popularne było podejście budowania monolitycznych aplikacji, które wykonują wszystkie funkcje, będące w aplikacji, w jednym kodzie źródłowym. W przeciwieństwie do systemów monolitycznych (dalej monolity), mikroserwisy reprezentują akrchitekturę systemu opartą na niezależnych usługach. Kluczowym elementem tej architektury jest komunikacja między serwisami poprzez interfejsy programistyczne (ang. Application Programming Interface, dalej API).

Zastosowanie architektury mikroserwisowej w systemach informatycznych niesie za sobą szereg profitów dla różnych grup specjalistów zaangażowanych w cykl życia oprogramowania. Z perspektywy programistów istotne są możliwości modularnej konstrukcji systemu, pracy w niewielkich zespołach skoncentrowanych na poszczególnych komponentach oraz doboru techniki dedykowanych pod kątem wymagań danego modułu. Dla testerów oprogramowania kluczowe znaczenie odgrywa ograniczenie testów regresji jedynie do fragmentów systemu objętych zmianami. *Testy regresji* służą do ponownej weryfikacji poprawności działania systemu po wprowadzeniu modyfikacji i mają na celu wykrycie ewentualnych błędów w funkcjonalnościach, które wcześniej zostały przetestowane i działały poprawnie. Dzięki temu testerzy mogą skupić swoje wysiłki na sprawdzeniu zmienionych mikroserwisów zamiast ponownie testować cały system. Eliminuje to konieczność każdorazowego zaangażowania całości zasobów testowych. Właściciele firm i decydenci biznesowi akcentują wzrost elastyczności reakcji na zmienne wymagania rynku, możliwość szybszego dostarczania użytkownikom nowych funkcji systemu, a także obniżenie całkowitych kosztów utrzymania i rozwoju rozwiązania informatycznego. Administratorzy systemów podkreślają natomiast usprawnienie procesów wdrażania i zarządzania środowiskiem informatycznym, ułatwioną identyfikację wąskich gardeł i lokalizację źródeł problemów oraz poprawę stabilności i dostępności kluczowych funkcji biznesowych [3]. Według danych statystycznych, w przybliżeniu 1/3 współczesnych systemów informatycznych w różnorodnych branżach i sektorach gospodarki adaptuje w praktyce paradygmat architektoniczny oparty na mikroserwisach [4].

Główne cechy mikroserwisów obejmują [1], [3], [5], [6] i [7]:

- Rozdzielenie funkcji: każdy mikroserwis powinien obsługiwać jedną konkretną funkcję lub dziedzinę biznesową. To pozwala na łatwe skalowanie, rozwijaniei utrzymywanie poszczególnych części systemu niezależnie od siebie.
- Mały rozmiar i koncentrację: każdy mikroserwis powinien być traktowany jak odrębna aplikacja, posiadając własne repozytorium kodu i proces dostarczania. Koncentracja oznacza, że każdy mikroserwis powinien być skoncentrowany na realizacji jednej, konkretnej funkcji biznesowej lub jednego obszaru odpowiedzialności. Jest to zgodne z *zasadą pojedynczej odpowiedzialności* (ang.

single responsibility principle), co oznacza, że mikroserwis powinien pełnić tylko jedną funkcję i być odpowiedzialny za jedną część systemu. Pełniąc tylko jedną funkcję, rozmiar mikroserwisu nie powinien być duży. Odpowiednie rozmiary ułatwiają rewizję, przepisanie i utrzymanie mikroserwisu.

- Rozwijalność: każdy mikroserwis powinien być łatwo rozwijalny, umożliwiając dostosowywanie się do zmieniających się wymagań biznesowych. Nowe funkcje lub zmiany w istniejących mikroserwisach można wprowadzać niezależnie od reszty systemu.
- Neutralność językową i technologiczną różnorodność: używanie odpowiedniego narzędzia do odpowiedniego zadania jest istotne. Mikroserwisy powinny być tworzone przy użyciu języka programowania i technologii, które są najbardziej adekwatne do danego zadania. Mikroserwisy są łączące, tworząc złożoną aplikację, i nie muszą być napisane w tym samym języku programowania.
- Zarządzanie danymi: mikroserwisy powinny zarządzać swoimi danymi, co oznacza, że każdy serwis przechowuje dane związane z własną dziedziną biznesową. Pozwala to unikać zależności między serwisami w kwestii danych.
- Testowanie: każdy mikroserwis powinien być testowany niezależnie od reszty systemu, zarówno pod względem jednostkowym, jak i integracyjnym.
- Automatyzację i CI/CD: Procesy wdrażania (ang. Continuous Integration/Continuous Deployment) powinny być zautomatyzowane, umożliwiając szybkie i bezpieczne wprowadzanie zmian w kodzie każdego mikroserwisu.
- Bezpieczeństwo: bezpieczeństwo musi być priorytetem. Każdy mikroserwis powinien być odpowiednio zabezpieczony, a komunikacja między serwisami powinna być szyfrowana.
- Samodzielność: mikroserwisy są samodzielne, co oznacza, że mogą działać i być wdrażane niezależnie od reszty systemu.
- Komunikację przez API: mikroserwisy komunikują się ze sobą za pomocą interfejsów programistycznych, co umożliwia integrację między nimi.
- Odporność na awarie: jeżeli jeden mikroserwis nie działa poprawnie, nie wpływa to na pozostałe, co sprawia, że cały system jest bardziej odporny na awarie.
- Możliwość wielokrotnego wykorzystania: architektura mikroserwisów ułatwia powtórne wykorzystywania komponentów systemu poprzez ich skalowalność i niezależność [8].
- Ściśle zdefiniowane interfejsy: mikroserwisy komunikują się wyłącznie za pomocą wiadomości, których strukturę trzeba jasno zdefiniować. Specyfikacja komunikatów i sposób ich wysyłania stanowią interfejs mikroserwisu [8].

2.2. Zasady projektowania mikroserwisów

Mikroserwisy charakteryzują się wysokim stopniem niezależności pod względem wyboru architektury, techniki implementacji oraz procesu projektowania. Projektowanie mikroserwisów to złożony proces, który wymaga uwzględnienia wielu czynników. Można wydzielić pięć uogólnionych elementów projektowania [5]:

1. Usługa: stanowi podstawową jednostkę, a jej właściwe zaprojektowanie jest kluczowe.
2. Rozwiązanie: jest perspektywą makro, obejmująca koordynację wielu usług. To koncepcja, która koncentruje się na całościowym spojrzeniu na system mikroserwisów, uwzględniającym współpracę różnych usług w celu osiągnięcia zamierzonego rezultatu.
3. Procesy i narzędzia: obok samej architektury systemu, kluczowe znaczenie mają procedury i narzędzia wykorzystywane przez zespoły programistów do tworzenia, testowania, wdrażania i monitorowania mikroserwisów. Przykładowo, przestrzeganie zasad CI/CD, stosowanie kontenerów Docker czy monitorowanie metryk pracy serwisów za pomocą narzędzi telemetrycznych wpływa na elastyczność i niezawodność systemu.
4. Organizacja: odnosi się do struktury, nadzoru władzy, granulacji i kompozycji zespołów pracujących nad mikroserwisami.
5. Kultura: zestaw wartości, przekonań i ideałów, które są współdzielone przez pracowników organizacji. Kultura organizacyjna ma wpływ na decyzje podejmowane na wszystkich poziomach i kształtuje zachowania pracowników. W kontekście mikroserwisów, elastyczność, otwartość na zmiany i współpraca są często wartościami kultury organizacyjnej sprzyjającymi stosowaniu mikroserwisów.

Do zaprojektowania usługi wykorzystuje się *architukturę zorientowaną na usługę* (ang. Service-Oriented Architecture, dalej SOA) [3], [5], [7], [8] i [9]. Jest to modułowa, zestandaryzowana i zorientowana na integrację architektura systemów korporacyjnych. Polega ona na modularnym projektowaniu systemów informatycznych poprzez wyodrębnienie funkcjonalności biznesowych w formie niezależnych, luźno powiązanych usług komunikujących się za pośrednictwem zdefiniowanych interfejsów. Koncepcja mikroserwisów odwołuje się do podobnych założeń, dodatkowo precyzując, że pojedyncza usługa wykonuje wąski obszar funcjonalności. Podejście mikroserwisowe można więc traktować jako rozwinięcie i konkretyzację ogólnych założeń SOA.

W celu osiądnięcia dobrego rozwiązania można wydzielić trzy główne zasady [1] i [7]: bazy danych w serwisie, luźne powiązanie (ang. loose coupling principle) oraz jednej odpowiedzialności. *Zasada bazy danych* na mikroserwis oznacza posiadanie, przez każdy mikroserwis swojej bazy danych, co ułatwia autonomiczne zarządzanie danymi i umożliwia niezależne funkcjonowanie mikroserwisów. *Zasada luźnego powiązania* jest związana z minimalizacją powiązań między serwisami. Oznacza to, że każda usługa jest niezależna, co ułtwia zmiany, aktualizacje i utrzymanie systemu.

Do punktu procesy i narzędzia też można zastosować różne podejścia projektowania. Takim podejściem jest *model 4+1* opisany przez Phillipa Krutchena [10], który definiuje cztery różne perspektywy systemu informatycznego, wraz z dodatkowym zestawem scenariuszy, które animują te perspektywy. Każda z tych perspektyw skupia się na konkretnej kwestii architektury i obejmuje określony zbiór elementów oprogramowania oraz relacji między nimi. Elementy tego modelu to [10]:

- Widok logiczny, który koncentruje się na klasach i pakietach, przedstawiając strukturę logiczną oprogramowania. Obejmuje relacje dziedziczenia, stowarzyszeń i zależności, oferując wysokopoziomową abstrakcję funkcji systemu.
- Widok implementacyjny, który opisuje wynik procesu kompilacji, skupiając się na modułach i komponentach. Ukazuje jak kod jest zorganizowany w jednostki gotowe do wdrożenia, oraz relacje zależności między nimi.
- Widok procesowy przedstawia komponenty w czasie rzeczywistym, traktując każdy z nich jako proces. Zawiera relacje komunikacji międzyprocesowej, ukazując interakcje pomiędzy różnymi procesami systemu.
- Widok wdrożeniowy ilustruje fizyczne lub wirtualne maszyny oraz procesy, pokazując, jak procesy są odwzorowywane w maszyny i jak komunikują się przez sieć. Zapewnia perspektywę dotyczące fizycznego wdrożenia systemu.

Scenariusze w kontekście modelu 4+1 stanowią narzędzie ożywiające opisane perspektywy, poprzez szczegółowe przedstawienie interakcji pomiędzy elementami systemu w odpowiedzi na konkretne żądania. Pełnią one rolę dynamicznego narzędzia, umożliwiając analizę i zrozumienie rzeczywistego działania systemu w różnych scenariuszach. Przez skoncentrowanie się na danych sytuacjach użycia, scenariusze ułatwiają identyfikację, zrozumienie i ocenę zachowań systemu w czasie rzeczywistym, co przyczynia się do pełniejszego zrozumienia jego architektury i funkcjonowania.

Przy projektowaniu mikroserwisów istotnym jest pamiętać o *projektowaniu zorientowanym na dziedzinę* (ang. Doman-Driven Design, dalej DDD). Autorem tego terminu jest Eric Evans [11]. Powyższa zasada jest powszechnie rekomendowana w licznych publikacjach branżowych [1], [2], [5], [6], [7], [8] i [12]. Idea DDD jest podejściem skoncentrowanym wokół domen biznesu, które polega na odzwierczedleniu w architekturze oprogramowania procesów występujących w rzeczywistej działalności organizacji lub wśród docelowych użytkowników systemu. Każdy mikroserwis reprezentuje wtedy określoną dziedzinę biznesową lub funkcjonalną, a słownictwo i reguły implementowane w jego ramach odpowiadają terminologii i zasadom danej domeny.

2.3. Definiowanie interfejsów API w architekturze mikroserwisów

Interfejsy API pełnią kluczową rolę w architekturze mikroserwisów, bo z ich pomocą istnieje możliwość wymiany informacji między poszczególnymi usługami. Interfejs odnosi się do sposobu implementacji komunikacji po stronie technicznej – określa protokoły, formaty danych, struktury komunikatów itp., wykorzystywane przez API.

Interfejs API definiuje zestaw publicznie dostępnych funkcji, które dany mikroserwis oferuje innym elementom systemu. Opisuje więc możliwości i sposób komunikacji z danym mikroserwisem [13].

Przykładowe API mikroserwisu do obsługi zamówienia może udostępniać funkcje takie jak pobieranie zamówień, dodawanie nowego zamówienia czy aktualizacja statusu realizacji. Interfejs tego API może wykorzystywać komunikaty JSON (ang. JavaScript Object Notation) przesyłane za pomocą protokołu HTTP (ang. HyperText Transfer Protocol) zgodnie ze standardem REST, który zostanie dokładniej omówiony w rozdziale

3. Popularnym formatem wymiany danych jest *JSON*, który został opisany w RFC 4627 [13], [14] i [15].

W swojej książce Sanjay Patni [15] wyróżnia dostępne strategie przy projektowaniu interfejsów API:

- *Strategia „Bolt-on*": ta strategia może być stosowana, gdy już istnieje aplikacja, a API jest dodawane po fakcie. Wykorzystuje istniejący kod i systemy, co może być korzystne, ale czasami wymaga dostosowania do istniejącej struktury.
- *Strategia „API-najpierw"* (ang. „Greenfield"): jeśli projekt rozpoczyna się od zera, strategia „API-najpierw" może być najbardziej efektywna. Pozwala to na swobodne korzystanie z nowoczesnych technologii i koncepcji, ponieważ nie ma istniejącej infrastruktury do uwzględnienia.
- *Strategia „Agile design"*: podejście zwinne zakłada możliwość rozpoczęcia pracy bez pełnego zestawu specyfikacji. Można dostosowywać i zmieniać specyfikacje w trakcie rozwoju, ucząc się w trakcie procesu. Jest to skuteczne, zwłaszcza na etapie projektowania, ale po opublikowaniu API zaleca się trzymać się ustalonych specyfikacji. Podejście zwinne ma 5 faz: analiza dziedziny lub opis API, projektowanie architektury, prototypowanie, budowanie API do produkcji oraz publikacja API.
- *Strategia „Facade"*: ta strategia stanowi rozwiązanie pośrednie między „bolt-on" a „greenfield". Pozwala na korzystanie z istniejących systemów biznesowych, dostosowując je do własnych preferencji i potrzeb. Daje to możliwość utrzymania działających systemów przy jednoczesnym doskonaleniu architektury podstawowej.

W analizie dziedziny lub opisu API w podejściu zwinnym chodzi o zrozumienie klientów API, sposobu komunikacji z innymi API oraz scenariuszy użycia. Ponadto, warto opisać wszystkie transakcje z operacjami CRUD (ang. Create, Read, Update, Delete) przy użyciu protokołu HTTP [15], które zostaną szczegółowo omówione w następnym rozdziale.

Podczas projektowania architektury warto rozważyć kwestie związane z wyborem protokołu, punktami końcowymi, projektowaniem URI (ang. Uniform Resource Identifier), a także bezpieczeństwem, dostępnością i wydajnością. Należy opisać zasoby, ich reprezentacje i typ, parametry i metody HTTP. Większość z wymienionych aspektów będzie omówiona w trzecim rozdziale niniejszego opracowania.

Prototypowanie stanowi istotny etap przedwdrożeniowy w inżynierii oprogramowania, mający na celu weryfikację założeń projektowych interfejsu aplikacji z perspektywy przyszłych użytkowników. Polega ono na implementacji wycinka docelowej funkcjonalności systemu w sposób zbliżony do produkcyjnego, lecz z pewnymi uproszczeniami. W prototypie realizuje się kluczowe ścieżki użycia aplikacji, zazwyczaj ograniczając złożoność implementacji części funkcjonalnej programu (ang. backend). Celem prototypowania jest stworzenie modelu systemu, który pozwoli przyszłym użytkownikom wyrobić sobie opinię na temat wygody i łatwości jego obsługi. Gotowy prototyp poddaje się testom akceptacyjnym z udziałem wewnętrznych interesariuszy w roli pilotowych użytkowników. Ich zadaniem jest zweryfikowanie zgodności zaprojektowanego interfejsu API z wymaganiami oraz oczekiwaniami kluczowych grup

docelowych przyszłych odbiorców rozwiązania. Uzyskane dzięki prototypowaniu informacje zwrotne pozwalają na korektę kierunku prac projektowych jeszcze przed etapem wdrożenia produkcyjnego API [15] i [16].

Dzisiaj można wydzielić dwa sposoby definiowania API: podejście zorientowane na wiadomości (ang. message-oriented) oraz metodyka oparta na hipermediach (ang. hypermedia-driven) [5].

W *podejściu zorientowanym na wiadomościach* interakcja polega na asynchonicznej wymianie wiadomości bez utrzymania sesji, często za pomocą JSON poprzez HTTP.

Przyjmując *podejście oparte na wiadomościach*, programiści mogą udostępniać ogólne punkty wejścia do komponentu (np. adres IP i numer portu) i jednocześnie odbierać komunikaty związane z konkretnym zadaniem. Z kolei metodyka *oparta na hipermediach* polega na komunikacji mikroserwisów poprzez udostępnianie zasobów, odwołując się do reprezentacji treści za pomocą adresów URI. Taki styl jest nazywany *hipermedia jako silnik stanu aplikacj* (ang. Hypermedia As The Engine Of Application State, dalej HATEOAS). Architektura REST obejmuje także styl HATEOAS. Przykładowo, jeśli klient otrzymuje dane dotyczące zasobu, to jednocześnie otrzymuje hipermedia z informacjami na temat tego, jakie akcje można na nim wykonać, jakie są dostępne powiązane zasoby itp. To sprawia, że klient nie musi polegać na wcześniej zdefiniowanych stałych adresach URL (ang. Uniform Resource Locator), co z kolei czyni interakcję z aplikacją bardziej elastyczną i odporną na zmiany w strukturze serwera [1] i [14].

Implementacja API musi być zgodna z jego specyfikacją oraz dostarczona możliwie szybko. API jest w pełni zintegrowane z systemem części funkcjonalnej programu i portfoliem API. Posiada wszystkie wymagane funkcjonalności i pożądane cechy takie jak wydajność, bezpieczeństwo i dostępność [15].

Opublikowanie API to ważny kamień milowy, oznaczający przekazanie odpowiedzialności za API z zespołu developerskiego do operacyjnego. Od tej pory zmiany w API wymagają tradycyjnego procesu zarządzania zmianą. Po opublikowaniu analizuje się statystyki wywołań API i luki w dokumentacji, które są obsługiwane przez zespół utrzymania [15].

3. REST API

3.1. Definicja i cechy

Architektura REST została zdefiniowana w 2000 roku [17], jej autorem jest Thomas Fielding. Architektura REST opisuje Internet jako rozproszoną aplikację hipermedialną, gdzie powiązane zasoby komunikują się, wymieniając reprezentacje ich stanu. Jest ona często wykorzystywana w projektowaniu interfejsów API dla aplikacji internetowych, dlatego często nazywa się REST API. Jeżeli usługa webowa posiada REST API, to jest określany jako RESTful [18]. Spośród zalet architektury REST wynieniane są następujące [15]:

- Wydajność – styl komunikacji proponowany przez REST ma być wydajny i prosty, co pozwala na poprawę wydajności systemów, które go adoptują.
- Skalowalność interakcji komponentów – cecha kluczowa w przypadku systemów rozproszonych, gdyż umożliwia elastyczne dostosowywanie architektury oraz zasobów systemu do zmieniającego się obciążenia lub wymagań biznesowych. Taka architektura ułatwia zatem skalowanie rozwiązania poprzez rozproszenie obciążenia na większą liczbę węzłów przy jednoczesnym zachowaniu spójności interfejsu, dzięki czemu system może z łatwością rosnąć wraz ze wzrostem zapotrzebowania.
- Prostota interfejsu – prosty interfejs umożliwia łatwiejszą interakcję między systemami, co z kolei może przynieść korzyści takie, jak wyżej wymienione.
- Modyfikowalność komponentów – rozproszona architektura systemu oraz separacja funkcjonalności zgodnie z założeniami REST umożliwia niezależne modyfikowanie poszczególnych komponentów przy zachowaniu minimalnych kosztów i ryzyka. Dekompozycja systemu na niezależne, słabo powiązane elementy ułatwia ich modernizację bez negatywnego wpływu na pozostałe moduły.
- Przenoszalność – REST nie zależy od technologii i języków programowania, co oznacza że może być implementowany i wykorzystywany przez dowolną technologię.
- Niezawodność – paradygmat bezstanowości w architekturze RESTful, eliminujący konieczność przechowywania stanu konwersacji na serwerze, znacząco podnosi niezawodność systemu poprzez umożliwienie serwerowi obsługi żądań niezależnie od innych, co ułatwia szybkie przywracanie funkcjonalności po awarii. Dzięki temu, system nie traci istotnych informacji potrzebnych do kontynuacji świadczenia usług, co wspiera stabilność i nieprzerwane działanie nawet w sytuacjach krytycznych.
- Widoczność – bezstanowość zapewnia lepszą diagnozę błędów, dzięki zawarciu pełnego stanu żądania w odpowiedzi.

Kluczowe zasady REST obejmują [15], [17], i [19]:

1. Architekturę klient-serwer: oddzielenie interfejsu użytkownika od dostępu do danych, dzięki czemu możliwe jest skalowanie i udoskonalanie każdej warstwy niezależnie.

2. Jednolity interfejs: wszystkie zasoby są dostępne za pomocą ujednoliconego interfejsu, bez względu na bieżące wykorzystanie. Ujednolicenie interfejsu ułatwia odpowiednie wykorzystanie komponentów.
3. Strukturę warstwową: architektura oparta na warstwach umożliwia skalowalność każdej warstwy niezależnie.
4. Pamięć podręczną (ang. cache): dane mogą być tymczasowo przechowywane w celu redukcji liczby iteracji pomiędzy klientami i serwerami.
5. Bezstanowość: brak sesji klienta po stronie serwera zwiększa skalowalność i niezawodność serwera.
6. Kod na żądanie: serwery mogą wysyłać wykonywalny program, który zostanie uruchomiony ze strony klienta. Pozwolenie na pobieranie funkcji po wdrożeniu systemu poprawia jego rozszerzalność. Jednakże, równocześnie ogranicza to widoczność, stąd jest to jedynie opcjonalne ograniczenie w ramach architektury REST [18]. Taka cecha pozwala na rozszerzanie funkcji klientów przez pobieranie i wykonywanie kodu w trakcie działania systemu.

Jednolity interfejs ma też swoje cechy charakterystyczne [17] i [18]:

1. Identyfikacja zasobów za pomocą unikalnych identyfikatorów URI.
2. Manipulowanie zasobami poprzez reprezentacje – oznacza to, że klient wysyła i odbiera dane w określonym formacie reprezentującym zasób, np. JSON lub XML. Jednak sama reprezentacja (format danych) jest niezależna od rzeczywistej treści i znaczenia przesyłanych informacji. Na przykład ten sam obiekt może być przesłany w formacie JSON lub XML, ale zawierać te same wartości atrybutów opisujące dany zasób. Zasada ta odnosi się więc tylko do technicznej reprezentacji danych, a nie do ich semantycznej interpretacji.
3. Samoopisujące się komunikaty przenoszące informacje o zasobach i żądaniach: wiadomości mogą mieć metadane, które mają charakter uzupełniający. Wiadomości HTTP mają nagłówki dla ujednolicenia metadanych.
4. HATEOAS – linki w reprezentacjach informujące o dostępnych akcjach i przejściach między zasobami.

3.2. URI

Standardem internetowym, który unikalnie definiuje zasoby w sieci, jest *Ujednolicony Identyfikator Zasobów*, znany również jako URI. Jest on opisany w dokumencie RFC 2396 z 1998 roku [20] i pełni czołową rolę w architekturze sieci WWW.

Zestawienie URI obejmuje dwie główne kategorie: URL oraz URN (ang. Uniform Resource Name). Jednocześnie URL i URN są typami URI, bo są identyfikatorami szczególnymi. *URL* określa dostęp do zasobu oraz jego lokalizację w sieci. Kategoria URL zawiera protokół dostępu (np. http, https, ftp, telnet), nazwę hosta, numer portu (opcjonalny), oraz ścieżkę do konkretnego zasobu. Czasami w URL mogą być podane hasło i nazwa użytkownika (np. w FTP), co nie jest zalecane gdyż dane wrażliwe są przesyłane w sposób jawny [20]. Takie nazewnictwo ma swój angielski termin „Server-based Naming Authority”. *URN* jest stosowany do nazywania źródeł i jest niezależny od

ich lokacji. Składa się z identyfikatora przestrzeni nazw (ang. namespace identifier) oraz części specyficznej dla tej przestrzeni nazw (ang. namespace specific string) [21].

Przykładowo URI *„https://datatracker.ietf.org/doc/html/rfc2141”* ma URL *„https://datatracker.ietf.org/doc/html/”*, który wskazuje na konktretny adres internetowy, gdzie znajduje się dokument, oraz URN *„ rfc2141”*, bo ta część już identyfikuje samą nazwę zasobu, która jest za podanym URL.

Adres URI musi mieć strukturę i logikę [15] i [22] , oraz być unikalny dla każdego zasobu [1] i [20].

Podczas budowania aplikacji przydatne są zapytania albo parametry URL, które umożliwiają podanie bardziej szczegółowej informacji. Pozwala to na filtrowanie, sortowanie czy rodzielenie informacji na strony [1] i [18]. Taka funkcja jest przydatna nawet w przypadku monitorowania czy logowania sieci podczas śledzenia skuteczności kampanii reklamowych [23].

Parametry adresu w adresie URL są podane po znaku zapytania „?” i rozdzielone znakiem ampersand „&” [1], [18] i [20].

Dany jest przykładowy URL „https://www.intel.com/content/www/us/en/search.html #q=%40title%3Dlaptops&sort=relevancy”, którego parametrami są wartości:

- q=%40title%3Dlaptops – parametr zapytania o nazwie "q" z wartością "%40title%3Dlaptops", co może być interpretowane jako zapytanie, które poszukuje tytułów zawierających "laptops". Część %40, to kodowanie procentowe znaku "@", a %3D to kodowanie procentowe znaku "=". Wartość @title może być interpretowane jako specyfikacja pola, w którym należy przeprowadzić wyszukiwanie. W tym przypadku, może to oznaczać, że użytkownik oczekuje wyników, w których pole "title" zawiera określone słowo kluczowe.
- &sort=relevancy – kolejny parametr zapytania o nazwie "sort" z wartością "relevancy".

3.3. Operacje CRUD i kody odpowiedzi HTTP

3.3.1. Definicja

Przy projektowaniu aplikacji RESTful niezbędna jest wiedza o podstawowych operacjach wykonanych przez serwisy. W skrócie, takie operacje nazywają się *CRUD*. W kontekście mikroserwisów RESTful, te operacje są realizowane poprzez standardowe metody HTTP: POST, GET, PUT/PATCH, i DELETE. W kontekście protokołu HTTP, operacje protokołowe są kierowane do precyzyjnie zdefiniowanych punktów końcowych, co determinuje interakcję między klientem a serwerem w celu skomunikowania żądania i uzyskania odpowiedzi. *Punkt końcowy* – to specyficzna część URL, która wskazuje na konkretną usługę lub operację w danym serwisie sieciowym. W kontekście RESTful API, punkt końcowy to konkretne miejsce, do którego wysyła się żądania HTTP, aby wykonać określoną operację. Punkt końcowy często jest końcowym fragmentem URL, który reprezentuje konkretne zasoby lub akcje.

W celu dwukierunkowej komunikacji między klientem a serwerem zostały wprowadzone *kody odpowiedzi HTTP*. Taki kod składa się z trzech cyfr i informuje klienta

o wyniku wykonania operacji z żądania HTTP. Poniżej przedstawiono kilka kategorii ogólnych [1], [18] i [22]:

- 1xy – informacyjne, używane tylko w negocjacjach z serwerem HTTP;
- 2xy – sukces, jest wysyłany w przypadku udanej operacji wykonanej na żądanie;
- 3xy – przekierowanie, ma na celu poinformowanie klienta o zmianie lokalizacji danych oraz konieczności wysłania nowego żądania GET pod inny, wskazany adres;
- 4xy – błąd klienta, wskazuje, że problem leży po stronie klienta, np. problem z uwierzytelnieniem, z formatem reprezentacji, lub z samą biblioteką HTTP;
- 5xy – błąd serwera, problem jest po stronie serwera. W większości przypadków oznaczają one, że serwer nie jest w stanie przetworzyć żądania klienta lub nawet sprawdzić, czy jest ono poprawne. Klient powinien ponowić swoje żądanie później. Czasami serwer może oszacować, po jakim czasie klient powinien ponowić żądanie, i umieścić tę informację w nagłówku odpowiedzi Retry-After.

W protokole HTTP metody te są przesyłane między punktami końcowymi.

3.3.2. Operacja Read (metoda GET)

Metoda GET odczytuje dane z istniejącego zasobu. Na przykład, aby pobrać listę wszystkich użytkowników systemu, wysyłamy żądanie GET do punktu końcowego użytkowników:

```
GET /api/users
```

Jeżeli wszystko działa prawidłowo, to otrzymamy dane w postaci JSON albo XML i odpowiedź HTTP 200, co oznacza OK. W przypadku, jeżeli za podanym URL nic nie ma albo punkt końcowy jest błędny, to będzie zwrócony kod 404 (nie znaleziono) albo 400 (nieprawidłowe zapytanie) [24].

3.3.3. Operacja Create (metoda POST)

Metoda POST jest używana do utworzenia nowego zasobu. Metoda POST nie jest ani bezpieczna, ani idempotentna. Idempotentność to cecha pewnych operacji, polegająca na tym, że wielokrotne zastosowanie takiej samej operacji nie zmienia wyniku końcowego w stosunku do zastosowania jej tylko raz [25]. Oznacza to, że wysłanie dwóch identycznych żądań POST najprawdopodobniej spowoduje utworzenie dwóch oddzielnych zasobów z taką samą zawartością. Dlatego POST zalecane jest dla operacji nietrywialnych, które powodują zmianę stanu na serwerze [19].

Na przykład, jeśli chcemy dodać nowego użytkownika, wysyłamy żądanie POST do punktu końcowego odpowiedzialnego za obsługę użytkowników:

```
POST /api/users
```

W przypadku pomyślnego utworzenia nowego zasobu, serwer powinien zwrócić status HTTP 201 (Utworzono) oraz nagłówek Location zawierający adres utworzonej encji [24].

3.3.4. Operacja Update (metody PUT/PATCH)

W mikroserwisach RESTful, metoda PUT lub PATCH jest używana do aktualizacji danych zasobu. Metoda PUT jest często używana do całkowitej aktualizacji zasobu, podczas gdy metoda PATCH jest używana do aktualizacji tylko wybranych pól.

Jednak PUT może również służyć do tworzenia zasobu, gdy identyfikator zasobu jest wybrany przez klienta, a nie przez serwer – jeśli PUT wysłany jest pod URI zawierający identyfikator nieistniejącego zasobu. W treści żądania znajduje się reprezentacja zasobu. Ta metoda tworzenia powinna być używana sporadycznie, ponieważ narusza ideę REST API, w której serwer kontroluje identyfikatory zasobów.

Wysyłając prawidłowe żądanie `PUT /api/users/{id}` z istniejącym identyfikatorem, mogą być otrzymane kody 200 lub 204 (strona otrzymała poprawny status, jednak nie znaleziono na niej jakiejkolwiek treści). Przy tworzeniu nowego zasobu zostanie zwrócony kod 201 [24].

W procesie projektowania zasobów architektury interfejsu API, powinno rozważyć się następujące wytyczne dotyczące stosowania metod PUT oraz POST [19]:

- Metoda POST jest zalecana jako podstawowa metoda tworzenia nowych zasobów. Umożliwia serwerowi lub usłudze API kontrolę nad nadawaniem unikalnych identyfikatorów URI tworzonym obiektom. Dzięki czemu można zagwarantować ich globalną unikalność i skalowalność architektury.
- Metoda PUT powinna być stosowana w przypadku, gdy klient w świadomy sposób sam decyduje o identyfikatorze URI, pod którym ma zostać utworzony nowy obiekt zasobu. Dotyczy to np. aktualizacji stanu istniejącego już zasobu.
- Aby utrzymać idempotentność oraz bezpieczeństwo interfejsu, metoda PUT nie powinna wywoływać operacji modyfikujących wewnętrzny stan zasobu, takich jak inkrementacja liczników. Tego typu logika aplikacyjna powinna być implementowana za pomocą metody POST.

3.3.5. Operacja Delete (metoda DELETE)

Metoda DELETE jest metodą usunięcia istniejącego zasobu. Na przykład, aby usunąć użytkownika, wysyłamy żądanie DELETE do odpowiedniego punktu końcowego: `DELETE /api/users/{id}.` Klient otrzyma status HTTP 200 (OK) wraz z treścią odpowiedzi, na przykład reprezentacją usuniętego elementu (często wymaga zbyt dużej przepustowości) lub opakowaną odpowiedzią. Kod statusu 204 zostanie zwrócony, jeśli działanie zostało wykonane i nie będą dostarczane żadne dodatkowe informacje. W przypadku, gdy żądanie DELETE zostało zaakceptowane, ale usunięcie jeszcze się nie dokonało, wyświetli się status 202 (przyjęty) [24].

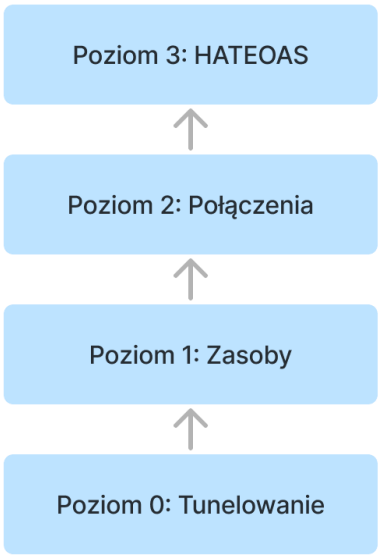
Pod względem specyfikacji HTTP, operacje DELETE są idempotentne. Wielokrotne wywołanie DELETE na tym samym zasobie kończy się tak samo – zasób zostaje usunięty. Drugie żądanie DELETE spowoduje błąd 404, ponieważ dany zasób już został usunięty fizycznie z bazy danych i system nie może go znaleźć, żeby usunąć go ponownie [19].

3.4. Model Dojrzałości Richardsona

3.4.1. Definicja

Model Dojrzałości Richardsona (ang. Richardson Maturity Model) to framework, który opisuje ewolucję stylu architektury RESTful, wprowadzając stopniowe poziomy

dojrzałości (patrz rysunek 1). Model ten został przedstawiony przez Leonarda Richardsona w 2008 roku i prezentuje cztery poziomy [26], [27] i [28].



Rysunek 1. Poziomy modelu Dojrzałości Richardsona

3.4.2. Poziom 0: Tunelowanie (ang. The Swamp of POX)

Ten etap charakteryzuje się prostotą i ograniczeniem w zakresie korzystania z protokołu HTTP. Na tym poziomie komunikacja między klientem a serwerem opiera się na pojedynczym punkcie końcowym (URL), a wszystkie operacje są wykonywane za pomocą żądań HTTP POST. Brak wykorzystania innych metod, takich jak GET, PUT czy DELETE, ogranicza różnorodność operacji i może prowadzić do mniej zrozumiałego interfejsu [26] i [27].

W rezultacie poziom 0 uważany jest za niski poziom dojrzałości w architekturze RESTful. Komunikacja oparta na pojedynczym punkcie końcowym i jednej metodzie HTTP utrudnia skalowalność, zwiększa złożoność interfejsu oraz nie wykorzystuje pełni możliwości, jakie oferuje protokół HTTP. W miarę awansowania na wyższe poziomy modelu, architektura staje się bardziej elastyczna, zrozumiała i zgodna z zasadami REST.

3.4.3. Poziom 1: Zasoby (ang. Resources)

Na poziomie 1 w Modelu Dojrzałości Richardsona pojawia się koncepcja zasobów. W przeciwieństwie do Poziomu 0, gdzie wszystkie akcje są realizowane poprzez jedno URL, na poziomie 1 zaczyna się wprowadzać rozróżnienie między punktami końcowymi [26] i [27].

Przejście na poziom 1 oznacza pewną organizację w strukturze API poprzez wprowadzenie zasobów, ale interakcja klienta z serwerem nadal ogranicza się do jednej metody HTTP.

3.4.4. Poziom 2: Połączenia (ang. HTTP Verbs)

Główną zmianą na tym poziomie jest użycie konkretnych metod HTTP w zależności od rodzaju operacji, co przyczynia się do zwiększenia przejrzystości i zgodności z zasadami RESTful. Wykorzystywane są także parametry operacji za pomocą różnych części żądania. Dzięki użyciu właściwych metod HTTP, serwisy na poziomie 2 mogą korzystać z dobrodziejstw infrastruktury sieciowej, takich jak pamięć podręczna w przypadku operacji GET. Wsparcie dla standardowych metod HTTP ułatwia korzystanie z istniejących narzędzi i technik sieciowych [26] i [27].

3.4.5. Poziom 3: HATEOAS

Na poziomie 3 Modelu Dojrzałości Richardsona implementuje się zasadę HATEOAS. Stąd odpowiedź na żądanie GET zwraca nie tylko same dane zasobu, ale także odnośniki (linki) do dostępnych operacji w tym zasobie. Na poziomie 3 brak jest konieczności kodowania URL-ów w kodzie klienta, co oznacza, że klient nie musi być świadomy konkretnych URL-ów, do których powinien wysyłać żądania. Zamiast tego, klient korzysta z dostarczonych linków, co sprawia, że interfejs API staje się bardziej elastyczny i bardziej odporny na zmiany w strukturze zasobów. Dlatego klient dynamicznie "odkrywa" dostępne akcje, przeglądając dostarczone linki. Dzięki temu podejściu, interfejs API staje się bardziej elastyczny, a klient nie musi być świadomy wszystkich szczegółów implementacyjnych [26] i [27].

W rezultacie poziom 3 reprezentuje najwyższy stopień dojrzałości w modelu Richardsona, co oznacza pełne wykorzystanie potencjału RESTful w architekturze API. HATEOAS przyczynia się do zwiększenia zrozumiałości i skalowalności interfejsu API, umożliwiając dynamiczne dostosowanie się do zmian w systemie [19].

4. Projekt systemu

4.1. Opis projektu

Praca prezentuje system służący do weryfikacji obecności osób na wykładach lub innych spotkaniach grupowych. System bazuje na architekturze mikroserwisów i składa się z dwóch niezależnych komponentów komunikujących się ze sobą. Celem projektu jest zgłębienie aspektów projektowania, implementacji i zarządzania mikroserwisami. W trakcie pracy zostaną omówione kluczowe kwestie, takie jak architektura mikroserwisowa, komunikacja międzyserwisowa, monitorowanie, testowanie oraz aspekty związane z bazą danych.

Pierwszy mikroserwis generuje losowe tokeny identyfikacyjne, które są przypisane do konkretnego spotkania. Umożliwia też opcjonalne zdefiniowanie dodatkowego hasła dostępu dla danego spotkania.

Drugi mikroserwis umożliwia rejestrację uczestników spotkania poprzez podanie ich danych osobowych (imię, nazwisko, grupa), nazwy przedmiotu, tokenu oraz hasła wygenerowanego przez pierwszy serwis. Przed dokonaniem rejestracji, drugi mikroserwis wysyła zapytanie do pierwszego serwisu o weryfikację poprawności tokenu i hasła. Jeśli token i hasło zgadzają się, uczestnik zostaje dodany do bazy danych obecności.

Takie rozwiązanie umożliwia szybką rejestrację osób uczestniczących w różnego rodzaju wykładach, szkoleniach, czy zebrań firmowych. System mógłby z powodzeniem posłużyć do potwierdzania obecności studentów na zajęciach dydaktycznych lub pracowników w miejscu pracy.

Komunikacja pomiędzy serwisami odbywa się w oparciu o protokół HTTP, gdzie drugi serwis wysyła zapytania o weryfikację tokenu i hasła do pierwszego, a w odpowiedzi otrzymuje dane w formacie JSON informujące o poprawności uwierzytelnienia.

4.2. Narzędzia i techniki

4.2.1. Flask jako framework do tworzenia mikroserwisów

Wybór frameworku ma kluczowe znaczenie dla efektywnej implementacji mikroserwisów. Flask to lekki i elastyczny framework do tworzenia aplikacji internetowych w języku Python. Został on wybrany ze względu na swoją lekkość, prostotę oraz elastyczność. Jego minimalistyczna struktura umożliwia skoncentrowanie się na implementacji konkretnych funkcjonalności, co jest szczególnie istotne w kontekście mikroserwisów. Dlatego Flask nazywa się mikroframeworkiem [13] i [29]. Dodatkowo, Flask oferuje bogatą dokumentację i aktywną społeczność, co ułatwia rozwiązanie ewentualnych problemów i szybkie wprowadzenie nowych członków zespołu do projektu [30], [31] i [32].

4.2.2. Baza danych na MySQL z użyciem XAMPP

Baza danych (ang. database) to struktura organizacji i przechowywania danych w sposób umożliwiający efektywne wyszukiwanie, aktualizację i zarządzanie nimi. Jest to

zbiór danych, zazwyczaj powiązanych ze sobą, przechowywanych w sposób uporządkowany, aby ułatwić dostęp do informacji [29] i [33].

Relacyjne bazy danych to rodzaj systemów zarządzania bazami danych, które opierają się na modelu relacyjnym zaproponowanym przez Edgara F. Codda w latach 70. XX wieku. Model ten opisuje, jak dane są zorganizowane w tabelach, a relacje między nimi są definiowane za pomocą kluczy. Taki rodzaj baz danych będzie wykorzystywany w niniejszej pracy. Podstawowe cechy relacyjnych baz danych są następujące: struktura, relacje, operacje, integrowanie i udostępnienie danych, bezpieczeństwo i skalowalność [33].

We Flasku nie istnieje mechanizm obsługi baz danych, dlatego tę funkcję spełniają pakiety Pythona [29].

W celu skutecznego przechowywania danych w mikroserwisach zdecydowano się na wykorzystanie systemu zarządzania bazą danych MySQL. Aby ułatwić proces tworzenia, zarządzania oraz monitorowania bazy danych, zastosowano platformę XAMPP. XAMPP dostarcza środowisko, w którym można uruchomić serwer MySQL, umożliwiając jednocześnie korzystanie z innych komponentów, takich jak Apache, PHP czy Perl [34].

4.2.3. Testowanie z użyciem Postman

Do testowania funkcji oraz interfejsu API mikroserwisów zdecydowano się na wykorzystanie narzędzia Postman [35]. Postman umożliwia łatwe tworzenie i wysyłanie zapytań HTTP, co jest niezbędne do efektywnego sprawdzania poprawności implementacji punktów koncowych RESTful. Narzędzie to dostarcza również funkcjonalności do automatyzacji testów, co przyspiesza proces weryfikacji poprawności działania poszczególnych komponentów systemu.

4.2.4. Środowisko programistyczne: IntelliJ IDEA i Visual Studio Code

W trakcie implementacji mikroserwisów korzystano z dwóch popularnych środowisk programistycznych: IntelliJ IDEA [36] oraz Visual Studio Code [37]. IntelliJ IDEA charakteryzuje się bogatymi funkcjonalnościami, integracją z systemami kontroli wersji oraz obszerną pomocą w refaktoryzacji kodu. Z drugiej strony, Visual Studio Code jest lekki, łatwy w konfiguracji i dostępny dla wielu platform. Oba środowiska zapewniają narzędzia ułatwiające pracę z Pythonem i ułatwiają rozwój mikroserwisów.

4.2.5. Dodatkowe narzędzia i techniki

Oprócz wymienionych głównych narzędzi i technik, projekt wykorzystuje dodatkowe elementy wspierające, takie jak:

- Git i GitHub: do kontroli wersji i współpracy nad kodem [38];
- Swagger: do generowania interaktywnej dokumentacji API [39].

Swagger jest zestawem narzędzi i standardów służących do projektowania, budowy, dokumentowania interfejsów API. Głównym celem Swaggera jest ułatwienie zarządzania API i zwiększenie jego użyteczności poprzez dostarczenie jednolitego, interaktywnego i łatwego do zrozumienia interfejsu dokumentacyjnego.

Oto kilka głównych funkcji Swaggera [39]:

- Dokumentacja API: Swagger umożliwia tworzenie czytelnej dokumentacji API. Dokumentacja ta zawiera szczegółowe informacje na temat dostępnych punktów końcowych, parametrów, typów danych, metod HTTP, a także przykłady użycia. Dzięki temu deweloperzy mogą szybko zrozumieć, jak korzystać z API.
- Interaktywna przeglądarka API: Swagger dostarcza interaktywną przeglądarkę API, która umożliwia deweloperom eksplorację API bez konieczności korzystania ze zewnętrznych narzędzi. Dzięki temu można przetestować różne punkty końcowe, dostosować zapytania i sprawdzić odpowiedzi bezpośrednio w przeglądarce.
- Generowanie klientów API: Na podstawie specyfikacji Swagger można generować klienty API w różnych językach programowania. To ułatwia integrację z API, ponieważ deweloperzy mogą korzystać z automatycznie generowanego kodu.
- Walidacja zapytań i odpowiedzi: Swagger umożliwia walidację zapytań i odpowiedzi zgodnie z zdefiniowaną specyfikacją. Dzięki temu można uniknąć wielu błędów związanych z nieprawidłowymi danymi przesyłanymi między klientem a serwerem.
- Standard OpenAPI: Swagger jest często używany jako synonim dla standardu OpenAPI, który jest formalnym standardem branżowym dla opisu interfejsów API. OpenAPI definiuje format specyfikacji API w formie pliku JSON lub YAML (ang. YAML Ain't Markup Language). Format *YAML* służy do serializacji danych, czyli zapisywania danych w postaci tekstowej w sposób zorganizowany i hierarchiczny.

Całość powyższych narzędzi i technik tworzy spójne środowisko wspomagające procesy projektowania, implementacji, testowania i utrzymania mikroserwisów opartych na frameworku Flask oraz bazie danych MySQL.

4.3. Punkty końcowe

Projektowane dwa mikroserwisy oferują różnorodne punkty końcowe, umożliwiające komunikację i obsługę różnych aspektów aplikacji za pomocą CRUD. Pierwszy mikroserwis (Student) udostępnia punkty końcowe do rejestracji studentów w pokojach, pobierania listy studentów oraz szczegółowe informacje o konkretnych studentach (patrz Tablica 1). Ponadto, umożliwia on sprawdzanie poprawności tokenów i haseł poprzez komunikację z drugim mikroserwisem.

Punkt końcowy `/register-to-room` korzysta z dwóch metod HTTP (GET i POST) aby rozdzielić żądanie pobrania formularza rejestracji od wysłania danych z wypełnionego formularza.

Metoda GET służy do pobrania samego formularza rejestracji, który zostanie wyświetlony użytkownikowi w celu wypełnienia. Odbywa się to poprzez wyrenderowanie szablonu HTML index.html przy użyciu funkcji `render_template`.

Natomiast metoda POST służy do wysłania wypełnionego formularza rejestracji do części funkcjonalnej programu. Dane formularza są dostępne w zmiennej request. Dane te są następnie sprawdzane i walidowane – najpierw poprzez wysłanie zapytania do pierwszego mikroserwisu, a następnie poprzez dodanie użytkownika do bazy danych przy pomocy funkcji `insert_data_into_db`.

Punkt końcowy `/students` ma cztery opcjonalne parametry (patrz Tablica 2).

Tablica 1. Punkty końcowe mikroserwisu Student

Metoda HTTP	Punkt końcowy API	Opis
GET, POST	/register-to-room	Obsługuje żądania rejestracji studentów w pokoju. Został zaprojektowany do obsługi żądań POST dotyczących rejestracji studentów.
GET, DELETE	/student/<int:id>	Pobiera lub usuwa informacje o studencie o określonym identyfikatorze (id).
POST	/student/<int:id>	Aktualizuje informacje o studencie o określonym identyfikatorze (id). Wartości pól są przekazywane w ciele żądania.
GET	/students	Pobiera informacje o wszystkich studentach zapisanych w systemie.

Tablica 2. Parametry punktu końcowego \students

Parametr	Opis	Przykład
sort_by	Określa, według którego pola należy posortować wyniki.	`/students?sort_by=name` posortuje wyniki alfabetycznie według nazwiska.
token	Filtruje wyniki na podstawie przypisanego tokena studenta.	`/students?token=xyz123` zwróci wyniki tylko dla studentów z tokenem "xyz123"
filter_by	Oznacza kolumnę, według której zostanie przeprowadzona operacja filtrowania.	`/students?filter_by=name&search_value=John` filtrowanie wyników, aby zawierały tylko studentów o imieniu "John".
search_value	Wartość, według której filtrowane są wyniki w określonej kolumnie (używane z parametrem filter_by).	`/students?filter_by=group&search_value=A1` Pozwala na znalezienie wszystkich studentów, których grupa zawiera literę "A".

Istotną cechą jest to, że wyszukiwanie za pomocą parametru search_value odbywa się w trybie "zawiera fragment" co oznacza, że parametr search_value może stanowić początek, koniec lub dowolny fragment szukanego ciągu znaków. Dodatkowo wyszukiwanie nie jest wrażliwe na wielkość liter, tak więc np. "a1" i "A1" dadzą ten sam wynik. Taka funkcjonalność jest zimplementowana poprzez wykorzystywanie operacji LIKE w kodzie SQL. Ona umożliwia elastyczne dopasowywanie ciągów znaków spełniających określony wzorzec, wykorzystując następujące wieloznaczniki:

- % – zastępuje dowolny ciąg znaków (może być pusty);
- _ – zastępuje dokładnie jeden znak.

Podanie `filter_by=surname` oraz `search_value=Nowak%` zwraca wszystkie nazwiska zaczynające się od "Nowak" (np. Nowakowski, Nowakiewicz); `search_value= %ski` zwraca wszystkie nazwiska kończące się na "ski" (np. Kowalski, Maczkowski); `search_value= _o%'` – zwraca nazwiska, w których po pierwszej literze występuje litera "o" (np. Kowalski, Wojciechowski).

Drugi mikroserwis (Room) skupia się na zarządzaniu tokenami, umożliwia generowanie losowych tokenów, ich usuwanie oraz pobieranie informacji na temat konkretnego tokenu, co wspomaga proces autentykacji i autoryzacji studentów (patrz Tablica 3). Dla przedstawionego mikroserwisu opracowano dokumentację interfejsu programistycznego obejmującą punkty końcowe `/tokens`, `/token` oraz `/get-data`, z którymi skorelowane są trasy zaczynające się od prefiksu `/api`, na przykład `/api/tokens`. Dokumentacja zostawała storzona za pomocą Swaggera i jest dostępna pod adresem „/”. Takie punkty końcowe mogą być wykorzystywane przez inne mikroserwisy, bo zwracają dane w postaci JSON, w przeciwieństwie do punktów końcowych bez prefiksu do wizualizacji danych w przyjemny dla użytkownika sposób.

Tablica 3. Punkty końcowe mikroserwisu Room

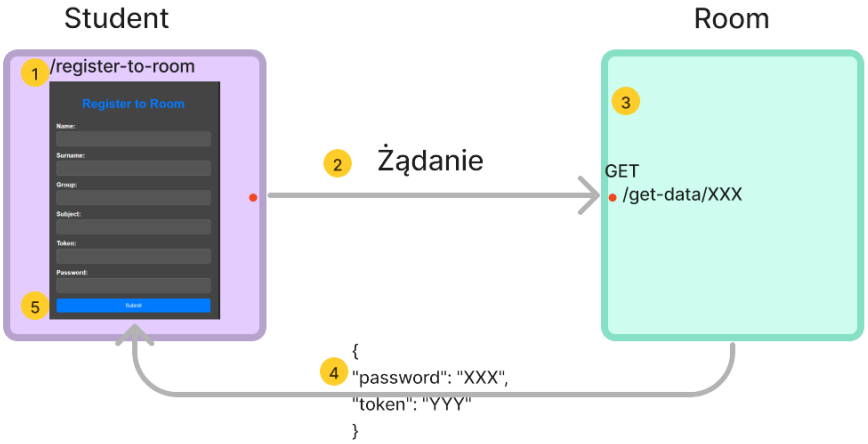
Metoda HTTP	Punkt końcowy API	Opis
GET	/generate-token	Generuje losowy token, opcjonalnie hasło, zapisuje je do bazy danych i zwraca widok z wygenerowanym tokenem i hasłem
POST	/generate-token	Zapisuje token i opcjonalne hasło do bazy danych
GET, PUT, POST, DELETE	/token/<int:id>	Pobiera, aktulizuje, dodaje, usuwa informacje o tokenie na podstawie id w bazie danych
GET	/tokens	Pobiera informacje o wszystkich tokenach. Ma dwa opcjonalne parametry: filter_by i search_value, które działają w analogiczny sposób jak w mikroserwisie „student”.
GET	/get-data/<string:token>	Pobiera dane o określonym tokenie poprzez zapytanie GET, a uzyskane informacje są zwracane w formie JSON, obejmując dane takie jak token i hasło, o ile token istnieje w systemie.

Oba mikroserwisy integrują się, tworząc kompleksową infrastrukturę wspierającą rejestrowanie i zarządzanie studentami w systemie. Punkty końcowe zostały zaimplementowane zgodnie z architekturą REST.

4.4. Komunikacja między mikroserwisami

Na rysunku 2 pokazano w jaki sposób odbywa się interakcja między dwoma mikroserwisami. Dzięki modularnej architekturze opartej na mikroserwisach, poszczególne komponenty systemu są luźno powiązane, co ułatwia ich niezależny rozwój i skalowanie. Proces wymiany danych pomiędzy mikroserwisami w celu walidacji i rejestracji użytkownika składa się z następujących etapów:

1. Użytkownik w interfejsie graficznym aplikacji webowej wprowadza wymagane dane identyfikacyjne w formularzu rejestracji i inicjuje żądanie poprzez akcję wciśnięcia przycisku "Zarejestruj".
2. Mikroserwis odpowiedzialny za rejestrację użytkowników (Student) wysyła żądanie HTTP GET do punktu końcowego `/get-data/<token>` mikroserwisu Room, podając otrzymany od użytkownika token jako zmienną ścieżki.
3. Mikoserwis Room sprawdza otrzymany token w lokalnej bazie danych pod kątem istnienia rekordu z tym tokenem.
4. Następnie zwraca odpowiedź HTTP zawierającą dane w formacie JSON – token i powiązane z nim hasło.
5. Mikroserwis Student prawdza kod odpowiedzi – gdy jest równy 200 OK, oznacza to udane wykonanie żądania. Następnie z rozkodowanego strumienia treści odpowiedzi w formacie JSON pobierany jest słownik zawierający token i hasło przesłane przez Room. Mikroserwis Student porównuje otrzymane w odpowiedzi dane z tymi przekazanymi przez użytkownika. W przypadku zgodności, rejestracja kończy się powodzeniem a dane użytkownika zostają zapisane w dedykowanej bazie danych.



Rysunek 2. Komukikacja między mikroserwisami

4.5. Implementacja mikroserwisów

Oba mikroserwisy zostały zbudowane w podobny sposób. Mikroserwisy korzystają z zewnętrznych bibliotek takich jak `requests (2.31.0) [40]` do wykonywania zapytań

HTTP oraz `mysql-connector-python (8.1.0) [41]` do obsługi operacji w bazie danych MySQL. Mikroserwis Room dodatkowo korzysta z biblioteki `random` dla generacji losowych tokenów.

Konfiguracja bazy danych jest określona za pomocą słownika `db_config`, który zawiera informacje niezbędne do nawiązania połączenia z bazą danych MySQL. W tym słowniku są takie parametry konfiguracyjne jak: host, user, password i database. Te dane są umieszczone w pliku głównym `main.py`. Komunikacja z bazą danych w bibliotece `mysql.connector` odbywa się w kilku krokach:

1. Utworzenie połączenia z bazą danych, wykorzystując wcześniej zdefiniowany słownik konfiguracyjny `db_config`:

```
conn = mysql.connector.connect(**db_config)
```

2. Utworzenie kursora **(**służy do wykonywania poleceń SQL w bazie danych**):**

```
cursor = conn.cursor()
```

3. Wykonanie polecenie, np. DELETE:

```
cursor.execute("DELETE FROM `students-bd` WHERE id = %s", (id,))
```

4. Zatwierdzenia transakcji:

```
conn.commit()
```

5. Zamknięcie kursora i połączenia:

```
cursor.close()
conn.close()
```

Z powodu częstych powtórzeń operacji otwarcia i zamknięcia kursora i połączenia do bazy danych zdecydowano się stworzyć funkcje `connect_to_database()` oraz `close_database_connection(cursor, conn).`

We wszystkich punktach końcowych mikroserwisów została zaimplementowana obsługa wyjątków za pomocą konstrukcji `try – except`.

Definiowanie punktów końcowych we Flasku odbywa się za pomocą dekoratora @app.route(). Dekorator ten informuje Flask, który widok (funkcja) powinien obsługiwać żądania HTTP na danym adresie URL. Przykładowe wykorzystywanie dekoratora dla mikroserwisu Student:

```
@app.route('/register-to-room', methods=['GET', 'POST'])
```

W nawiasach są podane URL oraz metody HTTP, które są obsługiwane przez dany punkt końcowy. Dalej jest definicja funkcji obsługującej dany punkt końcowy. Funkcja ta przyjmuje na siebie rolę kontrolera, obsługując logikę związaną z danym punktem końcowym.

Każdy mikroserwis ma stworzene 3 pliki HTML *(*ang. HyperText Markup Language, hipertekstowy język znaczników) [42], które są przechowywane w katalogu `templates`. Katalog `templates` ułatwia organizację plików szablonów, a Flask automatycznie znajduje i używa tych plików do renderowania stron HTML w trakcie obsługi zapytań HTTP [13] i [29] . Mikroserwis Room ma plik `index.html`, który jest wyświetlany do generowania tokenów; plik `token_info.html` prezentujący szczegóły pojedynczego tokenu (ciąg znaków, powiązane hasło, data utworzenia) na podstawie zadanego identyfikatora tokenu oraz plik `tokens.html` zawierający tabelaryczne zestawienie

wszystkich tokenów przechowywanych w bazie danych z ich podstawowymi parametrami w poszczególnych wierszach. W podobny sposób mikroserwis Student ma `index.html` z formularzem do rejestrowania użytkownika; plik `student_info.html` prezentujący szczegóły pojedynczego zarejestrowanego studenta oraz tabelaryczny widok `students.html` ze wszystkimi zarejestrowanymi użytkownikami systemu.

We Flasku, renderowanie i zwracanie plików HTML odbywa się przy użyciu szablonów. Flask korzysta z języka szablonów, który umożliwia dynamiczne generowanie treści HTML na podstawie danych dostarczonych z aplikacji. Do tego celu najczęściej używany jest moduł render_template, gdzie także można dostarczyć dane z kodu do szablonu jak to zostało zrobiono w mikroserwisie Student:

```
return render_template('student_info.html', student=student)
```

Jak już wspomniano w poprzednim podpunkcie, mikroserwis Student komunikuje się z mikroserwisem Room poprzez wysłanie żądania HTTP GET do odpowiedniego punktu końcowego w mikroserwisie Room. Punkt końcowy `/get-data/<string:token>` wewnętrznie wywołuje funkcję `retrieve_token_info_by_token(token)`, aby uzyskać dane na temat tokena z bazy danych. Funkcja ta nawiązuje połączenie z bazą danych i wykonuje zapytanie, aby pobrać informacje o tokenie (token i hasło) na podstawie przekazanego tokena. Zwraca ona te informacje lub None, jeśli token nie został znaleziony.

Mikroserwis Student ma zaimplementowaną funkcję `validate_token_and_password()`, która służy do walidacji danych tokenu przesłanych z innego mikroserwisu i komunikacji z nim. Mikroserwis Student używa modułu `requests`, aby wysłać żądanie GET do punktu końcowego `/get-data/<string:token>` w Room mikroserwisie, przekazując token jako część ścieżki:

```
requests.get(f'{first_microservice_url}/get-data/{token}')
```

Odpowiedź od mikroserwisu Room jest przetwarzana w funkcji `get_json(response)`, a następnie sprawdzane są odpowiednie pola (token i hasło). Jeśli dane są zgodne, funkcja zwraca True, co oznacza pomyślną walidację tokena i hasła. W przypadku błędów, takich jak brak połączenia lub błąd odpowiedzi, funkcja zwraca False.

Mikroserwis Room korzysta z biblioteki `flask_restx (1.3.0) [43]`, która oferuje funkcje Swaggera. Dla konfiguracji Swaggera został stworzony plik `swagger_config.py`, w którym jest inicjalizacja Swaggera, parsera tokenu i definicje modeli opisujących strukturę danych reprezentujących tokeny. Model token_model zawiera atrybuty token oraz password, wykorzystywane w komunikacji pomiędzy mikroserwisami. Model token_model_full rozszerza go o dodatkowe informacje takie jak identyfikator, czas utworzenia oraz zagnieżdżony model links_model, który reprezentuje adresy URL umożliwiające operacje na danym tokenie (GET, PUT, DELETE). Wykorzystywanie hipermediów jest przydatne dla renderowania strony `/tokens,` gdzie każdy token ma przyciski przekierowujące na poszczególne strony tych przycisków (Details) i możliwość usuwania tych tokenów (Delete).

Punkty końcowe Swagera i Flasku są stworzone w podobny sposób, ale:

- Użycie klasy Resource przez Swaggera: punkt końcowy jest reprezentowany jako klasa dziedzicząca po `Resource` z Flask-RESTful. Każda metoda HTTP jest reprezentowana jako metoda w tej klasie (np. get, post, delete).
- Deklaracja punktów końcowych z użyciem dekoratora klasy: Flask i Swagger mają rożne klasy, dlatego i deklaracja odwoluje się do różnych klas.
- Obsługa parametrów zapytania: w przypadku metody GET, parametry zapytania są pobierane we Flasku bezpośrednio z URL-a. W kodzie „Swaggera" dla metody `get` klasy `TokenResource`, id jest przekazywane jako argument metody.
- Walidacja danych wejściowych przez Swaggera: używane jest narzędzie do walidacji danych wejściowych (`token_parser`). Dzięki temu możemy zdefiniować, jakie dane wejściowe są oczekiwane, a Swagger generuje odpowiednią dokumentację.
- Obsługa błędów z użyciem `api.abort`: ta funkcja jest używana w bibliotece Flask-RESTx do generowania odpowiedzi błędów w przypadku problemów z zapytaniami API. Jej rola polega na przerwaniu obecnego przetwarzania i zwróceniu odpowiedzi błędu z określonym kodem HTTP i komunikatem błędu.

W celach edukacyjnych został przygotowany dodatkowy projekt przedstawiający implementację omówionego systemu mikroserwisów w środowisku Jupyter Notebook. Pozwoliło to na szczegółowe omówienie i przedstawienie kolejnych kroków tworzenia aplikacji wraz z dodatkowymi objaśnieniami.

Ze względu na specyfikę i ograniczenia platformy Jupyter, gdzie każdy notebook stanowi oddzielną całość, konieczne było umieszczenie fragmentów konfiguracji Swaggera bezpośrednio w głównym pliku notebooka zamiast wyodrębnienia ich do dedykowanego modułu swagger_config.py, jak ma to miejsce w pełnej wersji systemu. Dzięki temu możliwe było uruchomienie i przetestowanie działania aplikacji bezpośrednio w środowisku Jupytera.

Projekt ten stanowi zatem kompletny materiał edukacyjny, pozwalający krok po kroku prześledzić proces powstawania systemu mikroserwisów z dodatkowymi objaśnieniami, w formie dostosowanej do specyfiki platformy Jupyter Notebook.

4.6. Bazy danych

Mikroserwisy korzystają z dwóch oddzielnych baz danych: room-db służącej celom mikroserwisu Room oraz students-db obsługującej mikroserwis Students. Szczególny opis kolumn baz danych znajduje się w Tablicy 4 oraz Tablicy 5.

Tablica 4. Kolumny tabeli room-db

Kolumna	SQL polecenie	Opis
id	`INT(100) AUTO_INCREMENT PRIMARY KEY`	Unikalny identyfikator dla każdego pokoju (automatycznie inkrementowany).
token	`VARCHAR(255) NOT NULL`	Token powiązany z pokojem (wydarzeniem).
password	`VARCHAR(255) NULL`	Hasło pokoju.
creation_time	`TIMESTAMP DEFAULT CURRENT_TIMESTAMP`	Znacznik czasu wskazujący, kiedy wpis o pokoju został utworzony.

Tablica 5. Kolumny tabeli student-db

Kolumna	SQL polecenie	Opis
id	`INT(100) AUTO_INCREMENT PRIMARY KEY`	Unikalny identyfikator dla każdego studenta (automatycznie inkrementowany).
name	`VARCHAR(255) NOT NULL`	Imię studenta.
surname	`VARCHAR(50) NOT NULL`	Nazwisko studenta
group	`VARCHAR(50) NOT NULL`	Grupa, do której należy student.
subject	`VARCHAR(50) NOT NULL`	Przedmiot związany ze studentem
token	`VARCHAR(10) NOT NULL`	Token powiązany z pokojem.
password	`VARCHAR(50) NULL`	Hasło pokoju.
registration_time	`TIMESTAMP DEFAULT CURRENT_TIMESTAMP`	Czas rejestracji użytkownika

4.7. Wnioski

Dane mikroserwisy spełniają główne zasady RESTful aplikacji:

- Punkty końcowe opisują zasoby, które one oferują;
- Operacje na zasobach i używanie metod HTTP: dla każdego zasobu zdefiniowano różne operacje HTTP, takie jak GET, POST, DELETE;
- Używanie odpowiednich kodów statusu HTTP w odpowiedziach;
- Używanie parametrów URL;
- Reprezentacja zasobów: dane zwracane przez serwis są zazwyczaj w formie reprezentacji zasobów, najczęściej w formie JSON. W danym przypadku, zarówno przy użyciu Flask jak i Flask-RESTx, odpowiedzi zazwyczaj zawierają dane w formacie JSON;
- Bezstanowość – brak przechowywania stanu po stronie serwer.

Mikroserwis Room wdraża model Richardsona na poziomie trzecim, co wynika z jego skupienia na obsłudze podstawowych aspektów architektury RESTful, implementując HATEOAS. Oznacza to, że API serwisu Room zwraca reprezentacje zasobów zawierające linki umożliwiające odnalezienie powiązanych zasobów i wywołanie akcji na nich. Dzięki temu klient może dynamicznie odkrywać możliwe operacje bez z góry zdefiniowanej

wiedzy o API. Z kolei mikroserwis Student został zaimplementowany na poziomie drugim, ponieważ ogranicza się do udostępniania punków końcowych API operujących na zasobach studenci i grupach studenckich. Nie zwraca on reprezentacji z linkami nawigacyjnymi, co pozwoliłoby go zaklasyfikować do poziomu trzeciego wdrażania REST.

Proponowana architektura pozwoli na elastyczny rozwój systemu o kolejne funkcjonalności, bez zakłócania istniejących usług. Podejście oparte na mikroserwisach i API REST ułatwi również integrację z innymi systemami. System został zbudowany w oparciu o architekturę mikroserwisów, składającą się z niezależnych, luźno powiązanych komponentów realizujących określone funkcje.

5. Podsumowanie

Celem pracy było zaprojektowanie i implementacja interfejsu API REST, zgodnie z obowiązującymi standardami i dobrymi praktykami. Praca ma charakter dydaktyczny i stanowi kompleksowe omówienie procesu tworzenia nowoczesnego API wykorzystującego paradygmat REST.

W pracy przedstawiono ogólne informacje na temat architektury mikroserwisów, omówiono podstawowe zasady projektowania interfejsów API REST, zaprezentowano studium przypadku w postaci systemu rejestracji studentów zaimplementowanego w oparciu o REST API.

System składa się z dwóch głównych mikroserwisów – serwisu odpowiedzialnego za generowanie tokenów dostępu dla studentów oraz serwisu rejestracji studentów. Serwisy komunikują się za pomocą zdefiniowanych interfejsów API REST.

W pracy szczegółowo omówiono poszczególne elementy zaprojektowanego REST API – identyfikatory URI, operacje CRUD, formaty danych, odpowiednie kody HTTP. Interfejsy zostały zaimplementowane w języku Python z wykorzystaniem frameworka Flask.

Opracowany system stanowi kompletny przykład ilustrujący proces tworzenia nowoczesnego interfejsu REST API. Może być wykorzystany zarówno w celach edukacyjnych jak i stanowić punkt wyjścia do tworzenia rzeczywistych systemów informatycznych.

Literatura

[1] J. H. Peralta, Microservice APIs, NY: Manning Publications Co. , 2023.

[2] S. Newman, Monolith to Microservices, Sebastopol: O'Reilly Media, 2019.

[3] S. Daya, N. V. Duy, K. Eati, C. M. Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampkin, M. Martins, S. Narain i R. Vennam, Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach, IBM Redbooks, 2015.

[4] IBM, „Microservices in the enterprise, 2021: Real benefits, worth the challenges," March 2021. [Online]. Available: https://www.ibm.com/downloads/cas/OQG4AJAM.

[5] I. Nadareishvili, R. Mitra, M. McLarty i M. Amundsen, Microservice Architecture: Aligning Principles, Practices, and Culture, Sebastopol: O'Reilly Media, 2016.

[6] C. Richardson, Microservices Patterns, NY: Manning, 2019.

[7] S. Newman, Building Microservices, Sebastopol: O'Reilly Media, 2015.

[8] R. Rodger, The Tao of Microservices, Manning Publications, 2017.

[9] B. Götz , D. Schel, D. Bauer, C. Henkel, P. Einberger i T. Bauernhansl, „Challenges of Production Microservices," *Procedia CIRP,* tom 67, pp. 167-172, 2018.

[10] P. Krutchen, „Architectural Blueprints—The "4+1" View Model of Software Architecture," *IEEE Software,* tom 12, nr 6, pp. 42-50, November 1995.

[11] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2003.

[12] j. e. I. Y. j.-p. i m. , „Design a DDD-oriented microservice," Microsoft, 2022-13-04. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice.

[13] R. Haider, Web API Development With Python, 2021.

[14] L. Richardson, M. Amundsen i S. Ruby, RESTful Web APIs, Sebastopol: O'Reilly Media, 2013.

[15] S. Patni, Pro RESTful APIs with Micronaut, Santa Clara: Apress Media, 2023.

[16] API-University, „API Design – Prototyping," [Online]. Available: https://api-university.com/api-lifecycle/api-design/api-design-prototyping/. [Data uzyskania dostępu: 12 12 2023].

[17] R. T. Fielding, „Architectural Styles and the Design of Network-based Software Architectures," Irvine, 2000.

[18] M. Massé, „REST API Design Rulebook," O'Reilly Media, Sebastopol, 2012.

[19] T. Fredrich, „REST API Tutorial," 2013-02-08. [Online]. Available: https://github.com/tfredrich/RestApiTutorial.com/raw/master/media/RESTful%20Best%20Practices-v1_2.pdf. [Data uzyskania dostępu: 2023-12-12].

[20] T. Berners-Lee, L. M Masinter i R. T. Fielding, „Uniform Resource Identifiers (URI):

Generic Syntax," August 1998. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2396.

[21] R. Moats, „URN Syntax," Network Working Group , May 1997. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2141.

[22] L. Richardson i S. Ruby, RESTful Web Services, Sebastopol: O'Reilly Media, 2007.

[23] Google, „Parametry adresu URL," [Online]. Available: https://support.google.com/google-ads/answer/6277564?hl=pl.

[24] J. R. R. Fielding, „RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content," June 2014. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7231. [Data uzyskania dostępu: 2023-12-12].

[25] Wikipedia, „Idempotentność," [Online]. Available: https://pl.wikipedia.org/wiki/Idempotentność. [Data uzyskania dostępu: 2023-23-12].

[26] J. Gough, D. Bryant i M. Auburn, Mastering API Architecture, Sebastopol: O'Reilly Media, 2023.

[27] J. Webber, S. Parastatidis i . I. Robinson, REST in Practice, Sebastopol: O'Reilly Media, 2010.

[28] M. Fowler, „Richardson Maturity Model," 18 March 2010. [Online]. Available: https://martinfowler.com/articles/richardsonMaturityModel.html#TheMeaningOfTheLevels. [Data uzyskania dostępu: 2023-12-14].

[29] M. Grinberg, Flask Web Development, Sebastopol: O'Reilly Media, 2014.

[30] Full Stack Python, „Flask," [Online]. Available: https://www.fullstackpython.com/flask.html. [Data uzyskania dostępu: 2024-01-04].

[31] Pallets, „Flask," [Online]. Available: https://flask.palletsprojects.com/en/3.0.x/. [Data uzyskania dostępu: 2024-01-04].

[32] V. Podoba, „Best Python Frameworks for Web Development in 2023," SOFTFORMANCE, 2023-14-10. [Online]. Available: https://www.softformance.com/blog/python-web-frameworks/. [Data uzyskania dostępu: 2023-01-04].

[33] R. Elmasri i S. B. Navathe, Fundamentals Of Database Systems, Pearson Higher Education, 2016.

[34] Wikipedia, „XAMPP," [Online]. Available: https://pl.wikipedia.org/wiki/XAMPP. [Data uzyskania dostępu: 2023-23-12].

[35] Postman, „Postman," [Online]. Available: https://www.postman.com. [Data uzyskania dostępu: 2023-23-12].

[36] IDEA, „IntelliJ IDEA – the Leading Java and Kotlin IDE," [Online]. Available: https://www.jetbrains.com/idea/. [Data uzyskania dostępu: 2023-23-12].

[37] Visual Studio Code, „Visual Studio Code," [Online]. Available: https://code.visualstudio.com. [Data uzyskania dostępu: 23-12-2023].

[38] Wikipedia, „GitHub," [Online]. Available: https://pl.wikipedia.org/wiki/GitHub. [Data uzyskania dostępu: 2023-23-12].

[39] swagger, „API Development for Everyone,” [Online]. Available: https://swagger.io. [Data uzyskania dostępu: 2023-23-12].

[40] PyPi, „requests,” [Online]. Available: https://pypi.org/project/requests/. [Data uzyskania dostępu: 2023-29-12].

[41] MySQL, „MySQL Connector/Python Developer Guide,” [Online]. Available: https://dev.mysql.com/doc/connector-python/en/. [Data uzyskania dostępu: 2023-29-12].

[42] Wikipedia, „HTML,” [Online]. Available: https://pl.wikipedia.org/wiki/HTML. [Data uzyskania dostępu: 2023-12-16].

[43] PyPi, „flask-restx,” [Online]. Available: https://pypi.org/project/flask-restx/. [Data uzyskania dostępu: 2023-29-12].

[44] Swagger, „Swagger,” [Online]. Available: https://swagger.io. [Data uzyskania dostępu: 2023-12-17].

Załącznik

Main.py:

```
from datetime import datetime
from flask_restx import Resource
from swagger_config import api, token_parser, token_model, token_model_full
from flask import Flask, render_template, request, jsonify
import random
import string
import mysql.connector

app = Flask(__name__)
api.init_app(app)  # init object Api
api = api.namespace('', description='HTTP Operations related to tokens')

# MySQL Configuration
db_config = {
    'host': 'localhost',
    'user': 'root',
    'password': '',
    'database': 'test'
}


# Function to generate a random 5-character token
@app.route('/generate-token', methods=['GET', 'POST'])
def generate_and_insert_token():
    password = ""  # Initialize the password variable
    token = ""

    if request.method == 'POST':
        password = request.form.get('password')  # Retrieve the password from the form
        token = generate_token()
        insert_data_into_db(token, password)

    return render_template('index.html', token=token, password=password)


@app.route('/token/<int:id>', methods=['GET', 'PUT', 'POST', 'DELETE'])
def display_or_delete_or_update_token_info(id):
    method = request.args.get('_method', None)
    if request.method == 'GET' and method != 'DELETE':
        # Retrieve and display token information
        token, password, creation_time = retrieve_token_info(id)
        if token is not None:
            return render_template('token_info.html', id=id, token=token, password=password,
                                   creation_time=creation_time)
        else:
```

```
            return "Row with ID {} not found.".format(id), 404

    elif request.method == 'PUT':
        # Update the password and the token for the specified token ID
        new_token = request.form.get('token')
        new_password = request.form.get('password')
        if update_token_info(id, new_token, new_password):
            return "Information for ID {} has been updated.".format(id)
        else:
            return "Row with ID {} not found.".format(id), 404

    elif request.method == 'POST':
        new_token = request.form.get('token')
        new_password = request.form.get('password')
        if new_token:
            insert_data_into_db(new_token, new_password, id)
            return "New token added successfully."
        else:
            return "Missing 'token' parameter in the request.", 400

    elif request.method == 'DELETE' or method == 'DELETE':
        # Delete the row with the specified id
        if delete_token(id):
            return "Row with ID {} has been deleted.".format(id)
        else:
            return "Row with ID {} not found.".format(id), 404


@api.route('/api/token/<int:id>')
class TokenResource(Resource):
    def get(self, id):
        token, password, creation_time = retrieve_token_info(id)
        if token is not None:
            # Convert date to string before returning response
            creation_time_str = creation_time.strftime("%Y-%m-%d %H:%M:%S") if creation_time else None
            return {'id': id, 'token': token, 'password': password, 'creation_time': creation_time_str}
        else:
            api.abort(404, f"Row with ID {id} not found")

    @api.expect(token_parser)
    def post(self, id):
        args = token_parser.parse_args()
        new_token = args['token']
        new_password = args['password']
        if new_token:
            insert_data_into_db(new_token, new_password, id)
            return f"The row for ID {id} has been added."
        else:
            api.abort(400, "Missing 'token' parameter in the request.")
```

```
    @api.expect(token_parser)
    def put(self, id):
        args = token_parser.parse_args()
        new_token = args['token']
        new_password = args['password']
        if update_token_info(id, new_token, new_password):
            return f"Information for ID {id} has been updated."
        else:
            api.abort(404, f"Row with ID {id} not found")

    def delete(self, id):
        if delete_token(id):
            return f"Row with ID {id} has been deleted."
        else:
            api.abort(404, f"Row with ID {id} not found")


@api.route('/api/tokens')
class TokenList(Resource):
    @api.doc(responses={200: 'OK'})
    @api.marshal_with(token_model_full, as_list=True)
    def get(self):
        try:
            conn, cursor = connect_to_database()

            # Default query to retrieve all data
            query = "SELECT id, token, password, creation_time FROM `room-db`"

            # Check if filter_by and search_value parameters are provided
            filter_by = request.args.get('filter_by')
            search_value = request.args.get('search_value')

            # If both filter_by and search_value are provided, add a WHERE clause to the query
            if filter_by and search_value:
                query += f" WHERE `{filter_by}` LIKE '%{search_value}%'"

            # Execute the query
            cursor.execute(query)

            # Fetch the results after executing the query
            tokens = cursor.fetchall()

            # Convert the creation_time to ISO format for each token
            results = []
            for token in tokens:
                token_data = {
                    'id': token['id'],
                    'token': token['token'],
```

```
                'password': token['password'],
                'creation_time': token['creation_time'].isoformat(),
                '_links': {
                    'self': f"http://localhost:5000/token/{token['id']}",
                    'update': f"http://localhost:5000/token/{token['id']}",
                    'delete': f"http://localhost:5000/token/{token['id']}",
                }
            }
            results.append(token_data)

        close_database_connection(cursor, conn)
        return results, 200

    except mysql.connector.Error as e:
        print("Error:", e)
        return {"error": "An error occurred while fetching token data."}, 500


@app.route('/tokens', methods=['GET'])
def display_filtered_tokens():
    try:
        conn, cursor = connect_to_database()

        # Default query to retrieve all data
        query = "SELECT id, token, password, creation_time FROM `room-db`"

        # Check if filter_by and search_value parameters are provided
        filter_by = request.args.get('filter_by')
        search_value = request.args.get('search_value')

        # If both filter_by and search_value are provided, add a WHERE clause to the query
        if filter_by and search_value:
            query += f" WHERE `{filter_by}` LIKE '%{search_value}%'"

        # Execute the query
        cursor.execute(query)
        data = cursor.fetchall()

        # Format each token data with links
        formatted_data = []
        for token in data:
            formatted_token = {
                'id': token['id'],
                'token': token['token'],
                'password': token['password'],
                'creation_time': token['creation_time'].isoformat(),
                '_links': {
                    'self': f"http://localhost:5000/token/{token['id']}",
                    'update': f"http://localhost:5000/token/{token['id']}",
```

```
                'delete': f"http://localhost:5000/token/{token['id']}",
            }
        }
        formatted_data.append(formatted_token)

    return render_template('tokens.html', data=formatted_data)

  except mysql.connector.Error as e:
    print("Error:", e)
  finally:
    close_database_connection(cursor, conn)

  return "An error occurred while fetching token data."


@app.route('/get-data/<string:token>', methods=['GET'])
def get_token_data(token):
  data = retrieve_token_info_by_token(token)
  if data:
    response_data = {
      'token': data[0],  # Assuming data[0] contains the token
      'password': data[1]  # Assuming data[1] contains the password
    }
    return jsonify(response_data)
  else:
    return jsonify({'error': 'Token not found'}), 404


@api.route('/api/get-data/<string:token>')
class TokenResource(Resource):
  @api.marshal_with(token_model)
  def get(self, token):
    data = retrieve_token_info_by_token(token)
    if data:
      response_data = {
        'token': data[0],  # Assuming data[0] contains the token
        'password': data[1]  # Assuming data[1] contains the password
      }
      return response_data
    else:
      api.abort(404, 'Token not found')


def retrieve_token_info_by_token(token):
  try:
    conn, cursor = connect_to_database()

    # Query the database to retrieve the token, password, and creation_time for the specified token
    cursor.execute("SELECT token, password FROM `room-db` WHERE token = %s", (token,))
```

```
        row = cursor.fetchone()

        if row:
            token = row['token']
            password = row['password']
            return token, password

    except mysql.connector.Error as e:
        print("Error:", e)
    finally:
        close_database_connection(cursor, conn)

    return None, None  # Return None if the row is not found or an error occurs


def generate_token():
    token = ''.join(random.choice(string.ascii_letters + string.digits) for _ in range(5))
    return token


# Function to insert the token and password into the MySQL database
def insert_data_into_db(token, password, token_id=None):
    try:
        if token:
            conn, cursor = connect_to_database()

            if token_id is not None and token_id != 0:

                cursor.execute("SELECT COUNT(*) FROM `room-db` WHERE id = %s", (token_id,))
                result = cursor.fetchone()

                if result and result['COUNT(*)'] > 0:
                    raise ValueError(f"Token with ID {token_id} already exists")

            if token_id is not None and token_id != 0:
                insert_query = """
                    INSERT INTO `room-db` (`id`, `token`, `password`)
                    VALUES (%s, %s, %s)
                """
                cursor.execute(insert_query, (token_id, token, password))
            else:
                insert_query = """
                    INSERT INTO `room-db` (`token`, `password`)
                    VALUES (%s, %s)
                """
                cursor.execute(insert_query, (token, password))

            conn.commit()
            close_database_connection(cursor, conn)
```

```
        else:
            raise ValueError("Error: 'token' is required")

    except mysql.connector.Error as e:
        print("Error:", e)


def retrieve_token_info(id):
    try:
        conn, cursor = connect_to_database()

        # Query the database to retrieve the token, password, and creation_time for the specified id
        cursor.execute("SELECT token, password, creation_time FROM `room-db` WHERE id = %s", (id,))
        row = cursor.fetchone()

        if row:
            token = row['token']
            password = row['password']
            creation_time = row['creation_time']
            return token, password, creation_time

    except mysql.connector.Error as e:
        print("Error:", e)
    finally:
        close_database_connection(cursor, conn)

    return None, None, None  # Return None if the row is not found or an error occurs


def delete_token(id):
    try:
        conn, cursor = connect_to_database()

        # Execute a DELETE query to remove the row with the specified id
        cursor.execute("DELETE FROM `room-db` WHERE id = %s", (id,))

        # Check if any rows were affected
        if cursor.rowcount > 0:
            conn.commit()
            close_database_connection(cursor, conn)
            return True
        else:
            # No rows were affected, indicating the ID was not found
            close_database_connection(cursor, conn)
            return False

    except mysql.connector.Error as e:
        print("Error:", e)
```

```
        return False


def retrieve_all_tokens():
    try:
        conn, cursor = connect_to_database()

        # Query the database to retrieve all data
        cursor.execute("SELECT id, token, password, creation_time FROM `room-db`")
        data = cursor.fetchall()

        return data

    except mysql.connector.Error as e:
        print("Error:", e)
    finally:
        close_database_connection(cursor, conn)

    return []  # Return an empty list if an error occurs or no data is found


# Function to update the password for a given token ID
def update_token_info(id, new_token, new_password, new_time=None):
    try:
        conn, cursor = connect_to_database()

        # Determine the new creation time
        if new_time is None:
            new_time = datetime.now()

        # Execute an UPDATE query to change the token, password, and time for the specified ID
        cursor.execute("UPDATE `room-db` SET token = %s, password = %s, creation_time = %s WHERE id =
%s",
                       (new_token, new_password, new_time, id))

        # Check if any rows were affected
        if cursor.rowcount > 0:
            conn.commit()
            close_database_connection(cursor, conn)
            return True
        else:
            # No rows were affected, indicating the ID was not found
            close_database_connection(cursor, conn)
            return False

    except mysql.connector.Error as e:
        print("Error:", e)
        return False
```

```
def connect_to_database():
    conn = mysql.connector.connect(**db_config)
    cursor = conn.cursor(dictionary=True)
    return conn, cursor


def close_database_connection(cursor, conn):
    cursor.close()
    conn.close()


if __name__ == '__main__':
    app.run(debug=True)
```