Praca magisterska

Kierunek: Informatyka

Mikrousługi i Rozwiązania Monolityczne w Aplikacjach Internetowych

Krystian Skibiński

Publiczna wersja pracy bez danych osób trzecich i uniwersytetu.

Streszczenie

Praca dotyczy porównania dwóch popularnych podejść, mikrousługowego i monolitycznego, stosowanych przy tworzeniu serwisów internetowych. Jej celem jest przedstawienie podobieństw i różnic między tymi wzorcami, a także ich właściwości i wskazanie przypadków, dla których najlepiej sprawują się dane architektury. Na jej potrzeby stworzono dwa serwisy internetowe z wykorzystaniem omawianych podejść. Następnie przedstawiono konfigurację ich środowisk i potrzebnych narzędzi. Badania wykonane w ramach pracy obejmowały sprawdzenie aplikacji pod kątem wydajności wysyłanych przez serwer żądań, a także skalowalności poziomej i pionowej. Przeprowadzono również analizę implementacji aplikacji i zastosowanych technologii. W rezultacie pozwoliło to na wskazanie wad i zalet obu podejść, ocenie, kiedy najlepiej zastosować jedną z nich i zaproponowaniu możliwych ulepszeń w architekturze projektu.

Słowa kluczowe: praca dyplomowa, aplikacje internetowe, mikrousługi, monolit, architektura serwisów internetowych.

Spis treści

Spis rysunków

Spis tabel

Spis fragmentów kodu

Rozdział 1

Wstęp

Celem niniejszej pracy było zaprojektowanie aplikacji pod kątem architektury opartej o podejście monolityczne i mikrousługi. W jej ramach przygotowano dwa serwisy, a następnie starannie opisano proces ich implementacji i konfigurację. Szczególnie ważne było to, aby oba serwisy były napisane w ramach tych samych technologii i narzędzi, tak żeby można je było odpowiednio porównać. Szczegółowo przedstawiono również proces konteneryzacji projektu, tak aby, mógł on zostać w łatwy sposób zintegrowany z platformą dostarczającą rozwiązania internetowe.

W ramach porównania obu serwisów przeprowadzono testy wydajnościowe, sprawdzono możliwości skalowania aplikacji i wpływ tej technologii na ich sprawność, a następnie poddano analizie dostarczone rozwiązania architektoniczne, łatwość wdrożenia i wykorzystane technologię. Dzięki temu możliwe było opracowanie wniosków, których celem było wskazanie wad i zalet, a także przypadków dla których najlepiej sprawuje się dana architektura.

Tekst pracy oparto głownie o pozycję [1], gdzie autor również przygotował aplikację opartą o mikrousługi, opisał jej architekturę i przedstawił narzędzia potrzebne do jej implementacji, ale także inne pozycje takie jak [2], czy [3], traktujące o ideach stojących za oboma podejściami, ich charakterystykami, wadami i zaletami. We wszystkich książkach znajdowały się cenne rozdział poświęcone analizie przypadków, czy pomysłami na wdrożenie konkretnej architektury.

Przygotowana praca powinna pomóc w wybraniu odpowiedniej architektury, przedstawiono w niej proces przygotowania serwisu internetowego wraz z analizą załączonego kodu. Czytelnik po jej przeczytaniu powinien móc odtworzyć prostą aplikację opartą o oba podejścia, przygotować ich strukturę i integrację z środowiskiem produkcyjnym przy pomocy narzędzia do konteneryzacji. Następnie analizując badania i ich wyniki, a także wnioski autora, móc wybrać najlepszą z architektur dla jego własnego zastosowania.

Praca składa się z rozdziału 2., gdzie omówiono główne założenia aplikacji internetowych, a następnie wyjaśniono strukturę architektury monolitycznej i podejścia opartego o mikrousługi. Przedstawione zostało studium przypadku serwisu internetowego, gdzie wdrożony byłby model scentralizowany i rozwiązanie oparte o mniejsze usługi.

W rozdziale 3. omówiono projekt dwóch serwisów internetowych, opisano jego założenia, możliwe problemy które mogą zaistnieć w trakcie jego implementacji i dostępne rozwiązania. Następnie uszczegółowiono architektury poszczególnych aplika-

cji. Wyjaśniono relację, które wykorzystane będą w bazie danych, a także potrzebne biblioteki. Przeprowadzono również analizę wymagań funkcyjnych i niefunkcyjnych, a także opisano środowisko programistyczne, które będzie potrzebne do uruchomienia projektu i wspierane przeglądarki.

Natomiast rozdziały 4. i 5. skupiają się na szczegółowym opisie implementacji danej architektury, są one podzielone według poszczególnych warstw biznesowych aplikacji.

Problem integracji z platformami dostarczającymi rozwiązania w zakresie hostingu aplikacji opisano w rozdziale 6. Przedstawiono w nim proces konteneryzacji aplikacji, a także przygotowania jej środowiska deweloperskiego i produkcyjnego, tak, aby można było w łatwy sposób ją rozwijać, a także dostarczyć użytkownikom.

W rozdziale 7. na środowisku produkcyjnym przeprowadzono badania mające na celu sprawdzenie wydajności obu architektur. Przeanalizowano możliwości ich skalowania. Wpływ tego rozwiązania na sprawność systemu, a następnie analizie poddano cały projekt, tak, aby podsumować, która z architektur jest łatwiejsza do implementacji.

W ostatnim rozdziale 8. przedstawiono wnioski po implementacji i przetestowaniu obu aplikacji, następnie wskazano charakterystyki obu architektur, ich wady i zalety. Przedyskutowano osiągnięte rezultaty, a na końcu wskazano, dla jakiego przypadku najlepszym rozwiązaniem jest dane podejście.

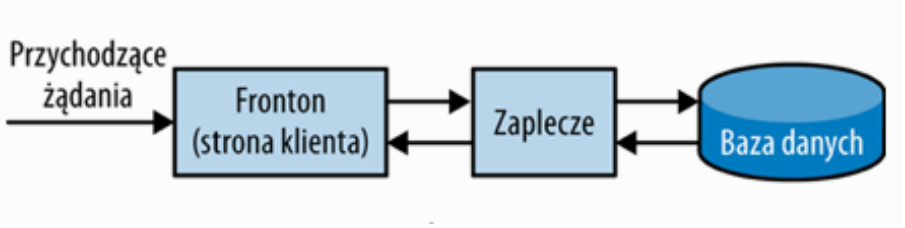
Rozdział 2

Rodzaje aplikacji internetowych

2.1. Aplikacje internetowe

Szybko rosnąca popularność internetu w latach 90. XX wieku sprawiła, że poza prostymi stronami *HTML*[1] zaczęto budować bardziej skomplikowane portale internetowe, a następnie ogromne serwisy. Potrzeba większych i lepszych systemów sprawiła, że ich twórcy musieli znajdować coraz to nowsze sposoby na ich budowanie[4]. Obecnie popularnością[2] cieszą się aplikacje internetowe. Są to programy komputerowe, które wykorzystują przeglądarkę jako *klienta*, przez którego wprowadzane i pobierane są dane. Inny komputer nazywany w tym przypadku *serwerem* je przechowuje, a w razie potrzeby udostępnia[5]. Prawie wszyscy obecni giganci internetowi korzystają z aplikacji internetowych[4, 1], a niezawodność działania ich usług stała się filarem ich sukcesu.

Podstawowe aplikacje internetowe mają warstwę prezentacji napisaną przy pomocy *HTML*, a także program uruchamiany na *serwerze*. Jego zadaniem jest obsługa baz danych, czyli zarządzanie trwałymi danymi, które są powiązane oprogramowaniem[6]. Do manipulacji nimi używa się języka SQL[3]. To rozróżnienie nazywamy *architekturą trójwarstwową*[3].



Rys. 2.1. Schemat aplikacji trójwarstwowej. Źródło [3]

Przykładowe zaplecze oparte o technologię *LAMP* (ang. *Linux-Apache-MySQL-Perl/PHP/Python)*[1] dla każdej akcji przesłanej przez formularz *HTML* generuje

[1]HTML(ang. *HyperText Markup Language*), hipertekstowy język znaczników powszechnie używany do tworzenia stron internetowych. Źródło `https://www.w3.org/TR/html4/`

[2]Według portalu *Google Trends*, fraza ta zaczęła zdobywać popularność od około 2009 roku. Źródło `https://trends.google.com/trends/explore?date=all&q=web%20app,android%20app,ios%20app`

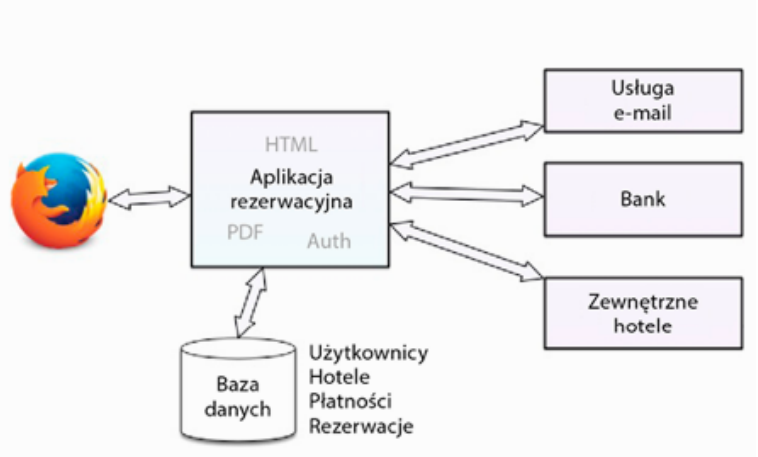
[3]SQL(ang. Structured Query Language) strukturalny język zapytań stosowany do tworzenia i zarządzania bazami danych[6].

dużą ilość zapytań *SQL* do bazy, a następnie odsyła nowo wygenerowaną stronę uaktualnioną o żądane informacje[6]. Twórcy takich serwisów zauważyli elementy stałe w działaniu dla każdego z nich. Dlatego naturalnym etapem rozwoju takich systemów było opracowanie ogólnodostępnych *frameworków*, czyli zestawu narzędzi i bibliotek służących do tworzenia podstawy aplikacji. Chcąc przewidzieć jakie problemy napotka programista stosowalni oni ujednolicone architektury, nie tylko jeżeli chodzi o wzorce programowania, ale też o cały system, począwszy od kontaktu z bazą danych, działaniem serwera, sposobu generowania warstwy prezentacji, aż po zarządzaniem wydawania aplikacji i jej serwowaniem użytkownikom końcowym. Obecnie standardem są dwa główne podejścia, monolityczne i te oparte o mikrousługi.

2.2. Podejście monolityczne

Monolit znaczy jednolity blok kamienny[4]. Słowo to idealnie oddaje charakter tego typu architektury. Większość takich aplikacji została napisana w oparciu o model trójwarstwowy. Z tym, że *fronton*[5] (rys. 2.1) i zaplecze mają wspólną bazę kodu. Umieszczone są we wspólnym repozytorium i uruchamiane za pomocą jednego pliku wykonywalnego[3].

W książce *Rozwijanie mikrousług w Pythonie*, *Tarek Ziadé* opisuje taką aplikacje na przykładzie agregatora hoteli[1]. Omawiana strona posiada napisaną warstwę prezentacji w statycznie generowanym *HTML-u*. Użytkownik wchodząc na stronę przesyła zapytania *SQL* do scentralizowanej aplikacji, która następnie prosi zewnętrzne usługi o listę hoteli. Podczas rejestrowania miejsca w hotelu kupujący otrzymuje stronę na którą musi się zalogować. Po przesłaniu danych o płatności serwer komunikuje się z usługą bankową zewnętrznego dostawcy, następnie informacje o zakończonej transakcji przekazuje do bazy danych. Aplikacja sprawdza, czy wpis o zakończeniu płatności jest już zapisana w usłudze *SQL*, jeśli tak, to na podstawie bazy danych generuje plik *PDF* i przy pomocy *usługi e-mail* przesyła potwierdzenie do użytkownika, sprawdzając jednocześnie informację o wybranym przez niego hotelu i również do niego kieruje powiadomienie email z potwierdzeniem transakcji.



Rys. 2.2. Schemat scentralizowanej usługi rezerwującej miejsca w hotelu. Źródło [1]

[4]Źródło: Słownik Języka Polskiego PWN `https://sjp.pwn.pl/sjp/monolit;2568347.html`
[5]Warstwa prezentacji, również nazywana *frontendem*.

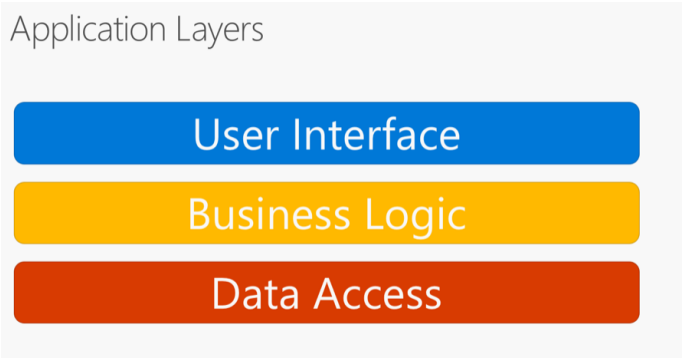
Z opisu jasno wynika, że każda interakcja użytkownika musi mieć swoją odpowiedź idącą z aplikacji zarządzającej rezerwacjami. Nawet jeśli potrzebuje ona informacji, które bezpośrednio można byłoby otrzymać z usług zewnętrznych. Tak samo dodatkowe serwisy, odsyłają informację zwrotną jedynie przez aplikacje centralną. Ten przykład dobrze przedstawia idee podejścia monolitycznego, centralnej usługi, która posiada w sobie wiele przeróżnych funkcji, takich jak autentykacja, generowanie statycznych stron, a nawet plików *PDF*. Wszystkie one są dostępne w ramach jednej aplikacji, jednolitego bloku kamienia.

2.2.1. Założenia aplikacji monolitycznej

W tego typu aplikacji cała logika zawarta jest w minimalnie jednym projekcie, który kompilowany jest pojedynczo, a następnie wdrażany jest jako całość. Oddzielanie poszczególnych problemów następuje poprzez grupowanie ich w osobnych folderach, które w miarę możliwości powinny izolować dane części aplikacji na *warstwy logiczne*[7]. Założeniem jest wyodrębnienie struktur, które można ponownie używać w ramach całej aplikacji. Wedle zasady *DRY*[6]. Dodatkowym atutem jest możliwość wymuszania ograniczeń w sposobie komunikowania się pomiędzy poszczególnymi warstwami, dzięki hermetyzacji kodu[7]. Podstawy podejścia *monolitycznego* są zbudowane na tych samych założeniach, co programowanie obiektowe[2]. Organizacja kodu następuje poprzez hierarchię klas i *interfejsów*[7]. Daje to możliwości tworzenia różnych sposobów komunikacji pomiędzy poszczególnymi warstwami i łatwiejszego wdrażania nowych funkcji.

2.2.2. Architektura aplikacji monolitycznej

Warstwy logiczne są powszechnie używane przy tworzeniu aplikacji monolitycznych, dlatego powstało wiele technik pozwalających na ich organizację[7]. Najpopularniejszą z nich jest podzielenie warstw aplikacji na te, które będą odpowiadać za dostęp do danych (*Data Access Layer*), ich zarządzeniem (*Business Logic Layer*) i interakcje z użytkownikiem (*User Interface*).



Rys. 2.3. Struktura warstw aplikacji internetowej. Źródło [7]

[6]DRY (ang. *Don't Repeat Yourself*) zasada zalecająca jak najmniejszą powtarzalność tych samych fragmentów kodu w różnych miejscach aplikacji[8].

[7]W programowaniu obiektowym *interfejsy* odpowiadają za definicję dla grup powiązanych funkcji, które klasa musi zaimplementować[7].

Dzięki tej architekturze izoluje się możliwości interakcji z poszczególnymi elementami aplikacji. Użytkownicy wchodzą w interakcje jedynie z warstwą *UI*, która następnie współdziedziczy pewne elementy jedynie z logiką biznesową, gdy zachodzi taka potrzeba, to ona komunikuje się z warstwą odpowiedzialną za dostęp do bazy danych. Daje to gwarancję, tego, że każda część aplikacji zna dobrze swoje kompetencje i ogranicza je jedynie do tych, za które jest odpowiedzialna[7].

Współczesne *frameworki* jeszcze bardziej rozdrabniają poszczególne warstwy, tak, aby udostępnić jedynie programistom *interfejsy* za pomocą, których będą mogli tworzyć swoje aplikacje. *Ruby on Rails* jest oparte o architekturę *MVC* (ang. *Model-View-Controller*), gdzie modele, są klasami odpowiedzialnymi za interakcję z dostarczonym przez *framework* mechanizmem kontaktowania się z bazą danych (*ORMem*[8]). Kontrolery, których zadaniem jest przyjmowanie danych od użytkownika, modyfikowanie modelu oraz odświeżanie widoków. Natomiast widoki, są odpowiedzialne za wygenerowanie szablonów *HTML* i przesłanie ich do klienta aplikacji (przeglądarki użytkownika)[9].

Jedną z odnóg architektury *MVC* zastosowano w *frameworkach* napisany w języku *Python*, takich jak *Django* i *Flask*. Ich twórcy stworzyli je w oparciu o wzorzec *MVT* (ang. *Model-View-Template*)[10], nie różni się on bardzo od pierwowzoru, z tą różnicą, że zamiast klas generujących pliki *HTML*, jego twórcy stworzyli moduł odpowiedzialny za przetwarzanie tego rodzaju plików posiadających specjalne znaczniki (na przykład *Jinja2*[9]). Dostarczają one dodatkowej logiki do interakcji z danymi przesłanymi przez widok.

```
1 {% extends "base_generic.html" %}
2
3 {% block title %}{{ section.title }}{% endblock %}
4
5 {% block content %}
6 <h1>{{ section.title }}</h1>
7
8 {% for story in story_list %}
9 <h2>
10   <a href="{{ story.get_absolute_url }}">
11     {{ story.headline|upper }}
12   </a>
13 </h2>
14 <p>{{ story.tease|truncatewords:"100" }}</p>
15 {% endfor %}
16 {% endblock %}
```

Listing 2.1. Przykład szablonu *Django*. Plik *index.html*. Źródło [10]

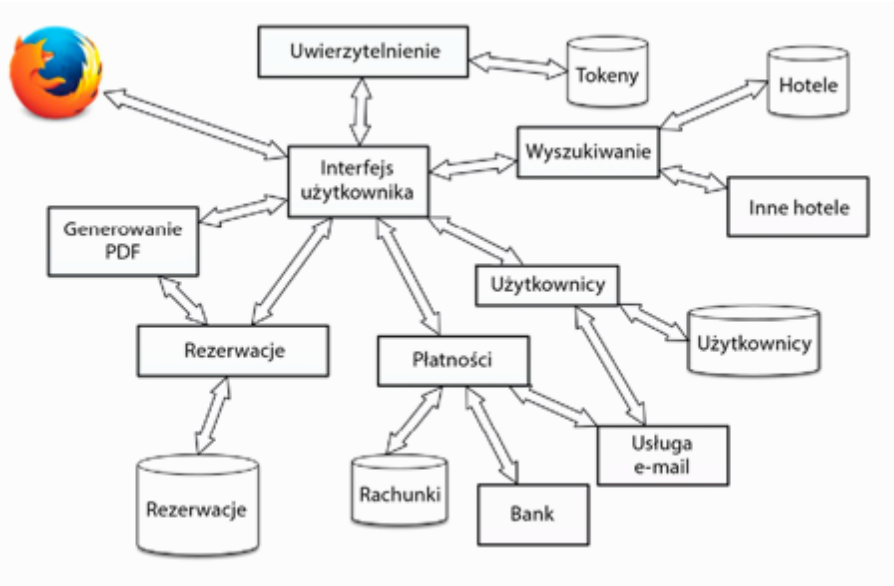
Istnieje jeszcze wiele innych podejść do projektowania monolitycznych aplikacji internetowych, te przedstawione tutaj są jednymi z najpopularniejszych. Jak widać struktury zaprezentowanych *frameworków* internetowych są mocno powiązane z wcześniej opisanym modelem trójwarstwowym (rys. 2.1).

[8]ORM (ang *Object-Relational Mapping*) system służący za odwzorowanie struktur bazodanowych na struktury programowania obiektowego, tak, aby uprościć tworzenie i dostęp do bazy danych.

[9]Więcej o składni *Jinja2* pod adresem `https://jinja.palletsprojects.com/en/2.11.x/`.

2.3. Mikrousługi

Opozycją do podejścia gdzie, jedna centralna aplikacja, komunikuje się z zewnętrznymi usługami jest stworzenie kilku mniejszych działających osobno programów. Ich kod można by było podzielić na kilka oddzielnie uruchamianych procesów, które mają odmienne mniejsze zadania do zrealizowania[1]. Wracając do przykładu agregatora hoteli podanego przez *Tarek Ziade*, zamiast jednej usługi odpowiedzialnej za komunikację, generowanie plików, obsługę banku i e-maili, można byłoby stworzyć wiele mniejszych *mikrousług*.



Rys. 2.4. Schemat scentralizowanej aplikacji z rys. 2.2 oparty o architekturę mikrousług. Źródło [1]

Autor książki *Rozwijanie mikrousług w Pythonie* z powyższego schematu (rys. 2.4), wyodrębnia 7 niezależnych komponentów[1], takich jak:

1. **Interfejs użytkownika**: frontalną usługę zajmująca się generowaniem *HTML-a* i komunikacją z innymi procesami.

2. **Wyszukiwanie hoteli**: program tworzący listę hoteli na podstawie danych z zewnętrznych usług.

3. **Użytkownicy**: usługa mająca na celu zarządzanie bazą użytkowników i wysyłaniem do nich wiadomości e-mail.

4. **Rezerwacje**: proces odpowiedzialny za zarządzeniu danymi na temat rezerwacji, a także przesyłaniem ich do generatora *PDFów*.

5. **Generator PDF**: program generujący na podstawie szablonów i danych odpowiednio sformatowane pliki *PDF*.

6. **Płatności**: moduł komunikujący się z zewnętrzną usługą banku, również zapisuje informacje o płatnościach i wysyła je użytkownikowi w wiadomości email.

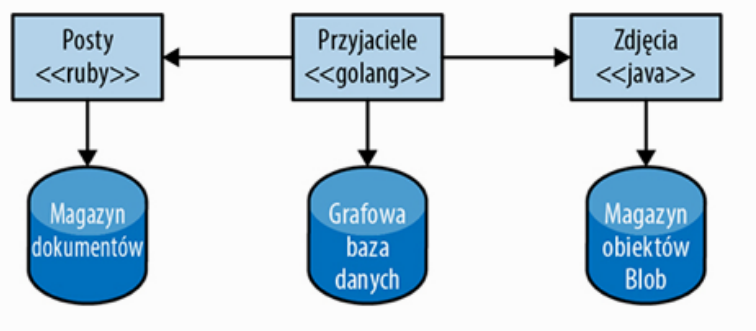
7. **Uwierzytelnianie**: usługa generująca *tokeny* stosowane przez inne procesy do uwierzytelniania użytkowników.

Powyższe mikrousługi mają tą samą funkcjonalność, co aplikacja monolityczna z rys. 2.2. Jednak do komunikacji nie wykorzystują elementów programowania obiektowego, a odpowiedni do tego protokół. W tym kontekście każda usługa stanowi osobną logiczną jednostkę odpowiedzialną za jedno ściśle określone zadanie[1]. Takie niezależne usługi nazywane są również komponentami.

2.3.1. Założenia mikrousług

Podejście z wykorzystaniem mikrousług oparte jest o niewielkie komponenty, które komunikują się z sobą w ten sam jednolity sposób bez żadnych przeszkód[2]. **Niezależnie od warstwy transportowej**, to znaczy, że dane dwie usługi nie muszą nawet wiedzieć o swoim istnieniu, aby jedna zastąpiła drugą w przenoszeniu informacji[2]. Wystarczy, że otrzymują i zwracają odpowiednie komunikaty. Dlatego ważne jest, aby wewnątrz nich znajdowały się informacje o tym jakie wartości powinny być w wiadomości zwrotnej, ta cecha to **dopasowanie do wzorca**. Daje ona możliwość dynamicznego definiowania z jakiej pomocy innej mikrousługi może skorzystać ta, która obecnie przetwarza informację. Najczęściej w tym celu stosuje się protokół *HTTP*[11] oparty o architekturę *REST* (ang. *Representational state transfer*), mającą za zadanie zunifikować odbierane i wysyłane żądania w zależności od ich treści[2]. W niezależnych od języka programowania formatach takich jak *JSON*, *XML*, czy *YAML*[1].

Osiągnięcie pełnej uniwersalności komunikatów nie jest możliwe. Wspomniane wcześniej wzorce muszą być zdefiniowane przez programistów w ramach architektury *REST*. Każda usługa powinna udostępniać *API* (ang. *Application Programming Interface*), czyli interfejsy, które pomogą innym programistom dostosować się do wzorca komunikatów dla danej usługi[1, 2]. To na zespole zajmującym się określony mikroserwisem spoczywa odpowiedzialność za dobrze udokumentowanie *API*, wówczas nie musi się on przejmować wyborem języka programowania, *frameworków*, czy bazy danych[1].



Rys. 2.5. Schemat prezentujący zastosowanie różnych technologii do budowania aplikacji opartych o mikroserwisy. Źródło [4]

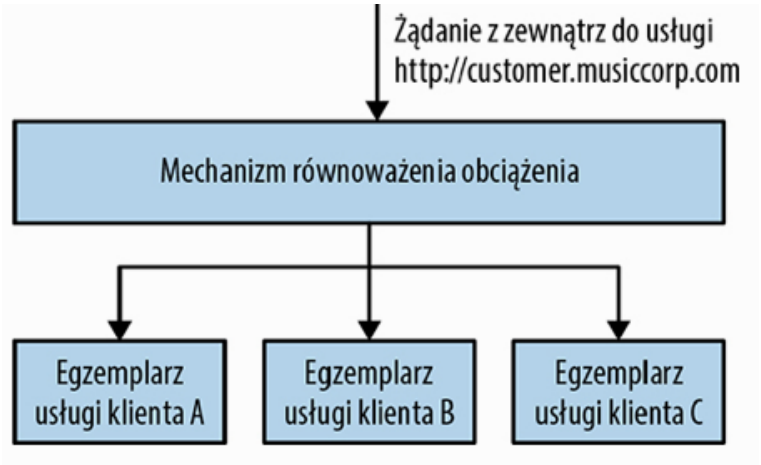
Z zasady każda usługa powinna być niewielka, ponieważ ma ona ściśle określone cele, a w razie, gdy będą potrzebne nowe funkcje dla całego systemu, to będzie

je można bez przeszkód wprowadzić dodając niezależną jednostkę. Ta charakterystyczna dla mikrousług cecha, to **addytywność**. Pomaga to w zapewnieniu dużego i bardzo skomplikowanego systemu (jako zbioru wszystkich mikrousług) bez większego długu technicznego[2]. W razie potrzeby dowolna jego część jest wymieniana i zastępowana, bez żadnego ryzyka niestabilnego działania całości aplikacji. Co w praktyce powinno przekładać się na dużą skalowalność projektu i jego łatwe wdrożenie[1].

2.3.2. Architektura mikrousług

Nie ma jednego wzorca pozwalającego określić budowę całego systemu opartego na mikrousługach. W przeciwieństwie do aplikacji monolitycznych, małe niezależne serwisy nie wymagają dogłębnego projektowania hierarchii klas, czy zależności pomiędzy poszczególnymi modułami. W tym wypadku najważniejsze jest spojrzenie na cały system od strony komunikatów, a nie z perspektywy poszczególnych komponentów. Łatwiej wówczas analizować sposoby interakcji pomiędzy nimi, znajdować odpowiednie wzorce i dobrze dostosować ich architekturę[2]. Taki pogląd na sytuacje uświadamia, że w przypadku mikrousług najważniejsze jest dobre zaprojektowanie warstwy transportowej. Wiedza, że poszczególna usługa, to tak naprawdę osobny mini serwer *HTTP* pozwala zwrócić uwagę na ważną cechę i zarazem niebezpieczeństwo takich jednostek, czyli potrzebę znajomości lokalizacji innej zależnej mikrousługi. W tym celu powinien zostać zaprojektowany proces pozwalający na odnajdowanie innych aplikacji. Rodzi to szereg problemów, ponieważ utrzymywanie takiej usługi nie jest proste, a napisanie dobrej implementacji jest zdaniem dość trudnym[2]. Kolejną sprawą jest to, że przy takim układzie, mikrousługi muszą przechowywać szczegółową wiedzę o konkretnych serwisach, a to prowadzi do ścisłych zależności. Właśnie tutaj widać fundamentalną różnicę między architekturą mikrousług, a monolitu. Żeby jednolita centralna aplikacja w którymś z modułów skorzystała z metody danej klasy ważne jest, aby posiadała do niej referencję. Z mikrousługami jest podobnie, z tą różnicą, że wszystko odbywa się przez sieć. Dana usługa korzysta z informacji o lokalizacji innego serwisu, a także odpowiedniego adresu *URL*, aby skorzystać z możliwości kolejnej. Odpowiednie zaprojektowanie tej komunikacji ogranicza potrzebę dostarczania dodatkowych usług i kodu, którego zadaniem byłoby współpracowanie z mechanizmem wykrywania poszczególnych modułów[2].

W najprostszej konfiguracji mikrousługi komunikują się z sobą bezpośrednio, ale warto rozszerzyć tą architekturę o moduły równoważenia obciążenia protokołu *HTTP*, tak zwane *load balancery*. Pozwala to na obsługiwanie ruchu dla różnych instancji mikrousług, a także zapewni proste skalowanie tych usług[2]. Niestety, zwiększy to również złożoność wdrażania takiego systemu.



Rys. 2.6. Schemat prezentujący zastosowanie mechanizmu do skalowania liczby instancji usługi klienta. Źródło [4]

Za jednym *load balancerem* mogą stać różne usługi, ogranicza to czas ich wdrożenia, koszty i zbędne konfiguracje, ale również może się okazać wąskim gardłem w działaniu systemu. To w jaki sposób zostanie wykorzystane narzędzie równoważenia obciążenia zależy od architekta aplikacji i ma duże znaczenie w jej działaniu. Ważne jest znalezienie optymalnej konfiguracji i pokrycie odpowiednich serwisów.

Rozdział 3

Projekt aplikacji

Do porównania budowy aplikacji stworzonych w oparciu o podejście monolityczne i mikrousługi wykorzystane będą dwa serwisy. Obie platformy będą miały taką samą funkcjonalność, tak, aby porównanie ich było jak najbardziej miarodajne. W tym celu zostaną również wykorzystane, gdzie tylko to możliwe, te same technologie i *frameworki*. Oczywiście, charakterystyka obu podejść będzie wymagała, żeby niektóre obszary aplikacji znaczenie się od siebie różniły, ale będą to jedynie elementy konieczne do realizacji założeń danej architektury.

3.1. Opis projektu

W ramach projektu stworzona zostanie prosta aplikacja do dodawania zadań. Po wejściu na stronę internetową użytkownik będzie mógł się do niej zarejestrować lub zalogować, a następnie ukaże mu się widok listy, służący do przeglądania aktualnych celów. Serwis pozwalać będzie na dodawania nowych zadań, ich aktualizację, a także usuwanie. Możliwe będzie również odznaczenie danego celu jako zrealizowany. Po skorzystaniu z aplikacji będzie można się z niej wylogować.

3.2. Założenia projektowe

Głównym założeniem projektu było stworzenie dwóch niewielkich aplikacji, które mogłyby w pełni zobrazować idee obu wspomnianych podjeść. Istotne jest przedstawienie procesu tworzenia architektury takich aplikacji i ich implementacji, a nie opracowanie skomplikowanych systemów posiadających ogromną ilość funkcji. To oczywiście jest celem biznesowym każdego realnego projektu informatycznego, ale założeniem pracy jest głównie porównanie takich usług, a nie ich całkowita realizacja i wdrożenie. Nie mniej jednak te aplikacje posiadać będą podstawowe elementy do ich przetestowania, takie jak mechanizm przesyłania stron internetowej do klienta, interfejs użytkownika, część odpowiedzialną za zarządzanie danymi, a także bazę danych, która będzie je przechowywać. Ważne z projektowego punktu widzenia będzie zaimplementowanie odpowiedniej komunikacji na poziomie poszczególnych komponentów aplikacji. W systemie monolitycznym będzie to zadbanie o dobrą strukturę projektu i stworzenie hierarchii klas, tak, aby zapewnić modułom odpowiedni poziom izolacji i ich logiczny podział. W przypadku mikrousług głównym założeniem

będzie stworzenie interfejsu komunikacji opartego na dostarczeniu *API* przez poszczególne komponenty, a także opracowanie sposobu pozwalającego na łączenie się między sobą dwóch usług. Ważne będzie ograniczenie działania tych jednostek do obsługi konkretnego zadania.

Do zarządzania tymi systemami wykorzystane zostanie narzędzie odpowiedzialne za konteneryzacje, tak aby zapewnić aplikacjom niezawodność działania niezależnie od platformy, a także ułatwić ich zarządzanie i wdrażanie. Szczególnie w systemach opartych o mikrousługi wykorzystanie takiego narzędzia usprawnia ich rozwój, ponieważ cały system jest konfigurowany raz, a pomimo tego działa w ten sam sposób na wielu hostach.

Następnym elementem projektu jest przedstawienie procesu testowania obu platform, porównania ich właściwości, a także przedstawienie możliwości integracji z usługami dostarczającymi infrastrukturę serwerową. Wskazane zostaną różnice pomiędzy dwoma tymi podejściami na każdym z wymienionych wcześniej etapów.

Ostatnim elementem projektu będzie analiza obu podejść, wskazanie ich wad i zalet. Znalezienie przypadków, dla których najlepiej sprawuje się dana architektura, tak, aby czytelnik, który w przyszłości tworzyłby serwis internetowych, był świadomy korzyści i konsekwencji płynących z wyboru jednego z opisanych podejść.

3.3. Opis problemu i dostępnych rozwiązań

Problemem przy tworzeniu dwóch aplikacji internetowych opartych o różne podejścia co do architektury całego sytemu, a zarazem muszących być pod względem technologicznym do siebie podobne, jest znalezienie uniwersalnego narzędzia do tworzenia takich usług. Istnieje wiele *frameworków* pozwalających na budowanie serwisów internetowych, ale tylko część z nich zapewnia odpowiedni poziom uniwersalności, tak, aby nie faworyzować jednej z dostępnych architektur. Na przykład *Django* zostało opracowane głównie tak, aby tworzyć serwisy monolityczne, przy generowaniu projektu powstaje już odpowiednia do tego struktura[1] z wygenerowanymi szablonami i panelem admina. Dzięki istnieniu bibliotek takich jak *Django Rest Framework*[2] można zmienić strukturę aplikacji i zaimplementować w niej *API* wysyłające komunikaty zgodnie z założeniami *REST*, ale polega to na wielu zmianach konfiguracji i doinstalowaniu kilku narzędzi. Szczególnie, że *Django* domyślnie posiada wiele modułów i jest stosunkowo dużym frameworkiem[3].

Z drugiej strony istnieją biblioteki głównie zorientowane i przystosowane na dostarczanie *REST API*. Są to minimalistyczne *frameworki* w których najważniejsza jest ich mała waga, brak wielu zależności i duża szybkość, takim przykładem jest *Falcon*[4].

Na szczęście istnieją narzędzia pozwalające na łatwą implementację zarówno jed-

[1] Opis tworzenia nowego projektu i generowania jego struktury został szczegółowo opisany w pierwszym rozdziale dokumentacji *frameworku*. Link do niej - `https://docs.djangoproject.com/en/3.0/intro/tutorial01/`.

[2] Link do projektu `https://www.django-rest-framework.org`.

[3] Wszystkie narzędzia i funkcje, które posiada *Django* są szczegółowo opisane w dokumentacji. Link `https://docs.djangoproject.com/en/3.0/`.

[4] Szczegółowo idea *Falcona* została przedstawiona w jego dokumentacji pod adresem `https://falcon.readthedocs.io/en/stable/index.html`.

nolitej aplikacji jaki i minimalistycznej usługi skierowanej na wykonanie poszczególnego zadania, serwującej *API* oparte o *REST*. Nie wymaga to usuwania dużej ilości zależności instalowanych wraz z generowaniem projektu, a także całkowitego przepisywania istniejących konfiguracji. Taką biblioteką jest *Flask*. Jego twórcy opisują go jako *mikroframework*, co oznacza, że całą aplikację można napisać w jednym pliku bez utraty jakichkolwiek funkcji, a rdzeń narzędzia jest utrzymywany jako maksymalnie mały i możliwy do rozbudowania przez korzystającego z niego programistę[12]. Biblioteka ta ma dużą bazę dodatków rozszerzających funkcję tworzonej w niej aplikacji, które w łatwy sposób można z nią zintegrować[12]. Rozszerzenia te pozwalają na dodanie takich narzędzi jak integracje z różnymi bazami danych, obsługa wysyłania plików, czy integracja z bibliotekami do autentykacji[5]. Framework pomimo swojej małej wielkości nadaje się do szerokiego zastosowania[12].

Ta biblioteka sprawdzi się również, w ramach opisywanego wcześniej problemu, do tworzenia dużych aplikacji. Posiada ona dodatki do obsługi różnych baz danych, system *ORM*[6] pozwalający na stworzenie klas *modeli*, narzędzia do zarządzania logiką aplikacji, nadawania im poszczególnych adresów (funkcje za które odpowiedzialny są widoki, (ang. *view*)) i wysyłania odpowiedzi w formacie *JSON* lub generowania szablonów *HTML* (*templates*) na podstawie przesłanych do nich danych przy pomocy składni *Jinja2*[12]. Z tego powodu *Flask* zapewnia dostateczne możliwości do tworzenia dużych monolitycznych aplikacji jaki i małych mikrousług.

Kolejnym problemem, który należałoby rozwiązać na etapie projektowania specyfikacji technicznej aplikacji jest warstwa prezentacji. W podejściu opartym o mikrousługi powinna to być osobna niezależna jednostka mogąca działać nawet bez konieczności uruchamiania innych usług. Istnieje rozwiązania oparte o *runtime* silnika *V8* przeglądarki *Google Chrome*, czyli narzędzie *Node.js*[7]. To pozwoliło na powstanie progresywnych *frameworków webowych* do budowania interfejsu użytkownika[13]. Wykorzystują one wcześniej wspomniany *runtime* do tworzenia *virtual DOMa*, czyli struktury *HTML'u* czytanej już bezpośrednio przez przeglądarkę klienta, ale zarządzanej przez język *JavaScript*, co ma przekładać się na łatwość obsługi jej poszczególnych elementów, aktualizowania lub sterowania ich stanem[13]. Dodatkowo takie *frameworki* posiadają skrypty do generowania gotowych projektów, gdzie dostarczają proste serwery, które pozwalają na przeładowywanie strony przy każdej zmianie pliku. Opisywaną bibliotekę można również dołączyć do statycznego pliku *HTML*, co przyda się przy tworzeniu aplikacji monolitycznej. Większość z tych narzędzi manipuluje kodem strony jedynie przez język *JavaScript*[13]. Natomiast biblioteka *Vue.js* pozwala na to za pomocą kodu *HTML* i specjalnej składni (podobnej do *Jinja2*), dzięki czemu o wiele łatwiej jest tworzyć taką aplikację wewnątrz pliku *HTML*. Z kolei nowoczesne przeglądarki posiadają narzędzia do pobierania i wysyłania zasobów takie jak *Fetch API*[14]. Pozwala to na obsługę zapytań *HTTP*[8] po stronie klienta już po przesłaniu mu wygenerowanego pliku *HTML*, jest to więc sposób na komunikację między usługą interfejsu użytkownika, a pozostałymi mikrousługami

[5]Więcej szczegółów zostało opisanych w dokumentacji. Link `https://flask.palletsprojects.com/en/1.1.x/foreword`.

[6]Link do jego dokumentacji `https://flask-sqlalchemy.palletsprojects.com/en/2.x/`.

[7]Link do dokumentacji *Node.js*, `https://nodejs.org/en/`.

[8]Więcej informacji o działaniu i posługiwaniu się *Fetch API* można znaleźć dokumentacji zasobów internetowych fundacji *Mozilla*. Link `https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch`.

działającymi po stronie serwera.

Istnienie wielu powiązanych z sobą usług lub nawet jednej centralnej aplikacji wiąże się z dużą ilością potrzebnych do uruchomienia zależności, konfiguracji, a nawet warstwy sprzętowej[1]. Dlatego dobrym rozwiązaniem jest zastosowanie *maszyny wirtualnej*, programu symulującego środowisko danego sprzętu, wraz z jego systemem. Niektóre duże projekty zostały nawet przeniesione jako gotowe obrazy takich *maszyn wirtualnych*. Dzięki czemu udało się je spakować wraz z wypełnionymi bazami danych do uniwersalnych formatów, które bez żadnej konfiguracji można uruchomić za pomocą jednego polecenia[1]. Niestety, takie rozwiązania często są płatne i ze względu na potrzebę dużej mocy obliczeniowej trudno jest je stosować w środowisku produkcyjnym[1]. Rewolucją okazał się *Docker*, otwarta platforma do konteneryzacji. W odróżnieniu od wcześniej wspomnianych metod *Docker* nie stara się symulować całego sprzętu, a jedynie odizolować środowisko za pomocą wysokopoziomowych narzędzi wykorzystywanych w systemie *Linux*. Projekt zbudowany za pomocą tej platformy jest w stanie działać w każdym środowisku bez względu na system operacyjny z zachowaniem normalnej prędkości działania[1], a dzięki usłudze *Docker Compose* można dostarczyć do projektu plik z konfiguracją[15] całego procesu jego uruchomienia za pomocą narzędzia *Docker*, co w efekcie pozwoli na wystartowanie projektu wykorzystując jedną prostą komendę. Wystarczy mieć na komputerze zainstalowaną omawianą platformę[9].

Niezależnie od tego, czy obsługujemy duży połączony z sobą system wielu usług, czy też jedną centralną aplikację potrzebne będzie usługa *serwera WWW*. W przypadku monolitu obsługuje on zapytania *HTTP* na poziomie systemu, nasłuchując na zapytania zewnątrz i przekazując te odpowiedzi (serwer *Proxy*[16]) do i z serwera *uWSGI Flaska*, wówczas można byłoby zastosować usługę *Nginx*[12], lub też skorzystać z gotowych rozwiązań, takich jak chmura *AWS*, *Heorku*, *Azure*, *PythonAnywhere*[12]. W przypadku mikrousług, które będą wewnątrz kontenera to, potrzebny będzie proces odpowiedzialny za wewnętrzną komunikację, a także *load balancer*. Wszystkie te opcje również udostępnia *Nginx*[16].

3.4. Architektura systemu

Zgodnie z głównym założeniem pracy w jej ramach przedstawione zostaną dwie aplikacje oparte o różne podejścia, monolityczne i mikrousługi. Każda z nich wymaga odmiennego spojrzenia na ich budowę i architekturę. Odmiennego zaprojektowania hierarchii klas i warstwy transportowej dla przetwarzanych danych.

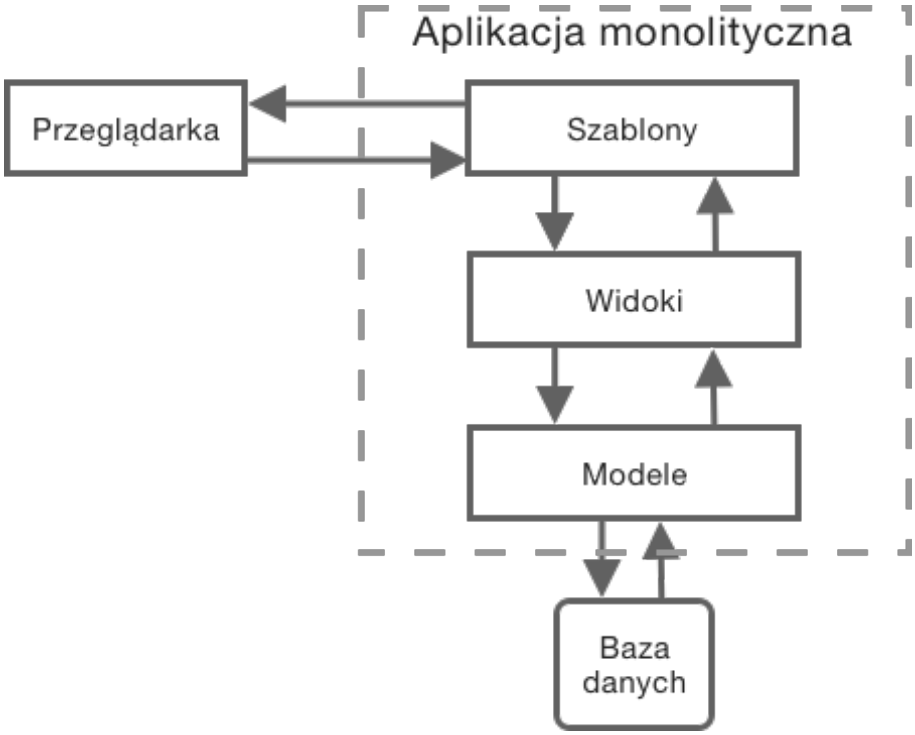
3.4.1. Podejście monolityczne

Architektura aplikacji monolitycznej została zorientowana o wcześniej opisany wzorzec *MVT*. Zakłada on wykorzystanie hierarchii obiektów dostarczony przez *framework* do zbudowania komponentu opartego o **modele**, czyli klasy odpowiadającej za dostarczenie odpowiednich struktur i relacji w bazie danych. Następnie za pomocą *ORMa*, który posiada zestaw narzędzi do tworzenia kwerend, dostarczone będą

[9]Link do instrukcji zainstalowania *Dockera* i *Docker-compose*, `https://docs.docker.com/get-docker/`.

dane przetwarzane w warstwie **widoków**. Ich rola nie tylko sprowadza się do interpretowania przesłanych informacji i odsyłania ich klientowi. Odbywa się w nich również dopasowywanie (*routing*) zapytań do funkcji i metod, które je przetwarzają i zwracają jako wygenerowane **szablony**.

Na schemacie poniżej (rys. 3.1) jest strzałka zwrotna z przeglądarki do szablonu. Użytkownik dostając wygenerowany kod *HTML* wykorzystuje zawarte w nim formularze do wysłania danych do widoku. Możliwe jest ich bezpośrednie wysłanie, ale polega to na odtworzeniu odpowiedzi serwera.



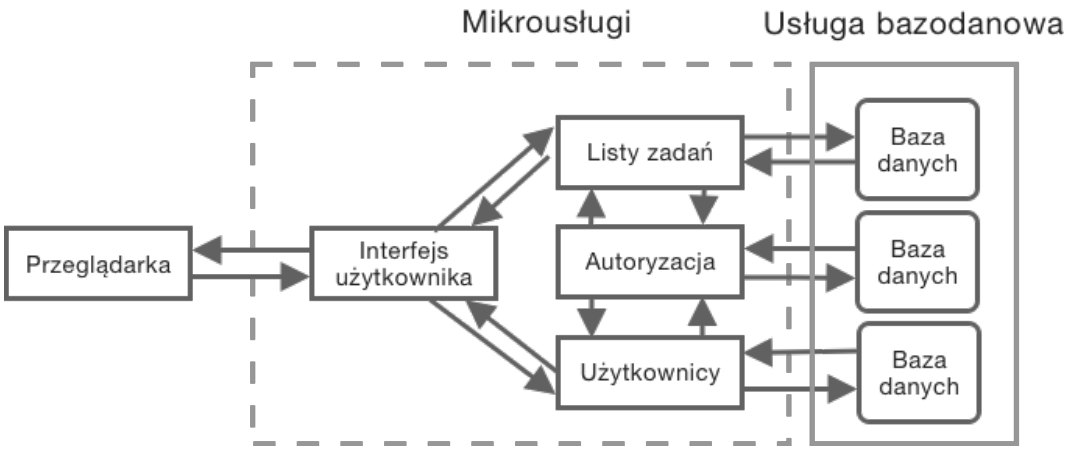
Rys. 3.1. Schemat architektury monolitycznej.

3.4.2. Mikrousługi

Architektura mikrousług będzie zakładała, że jedna centralna aplikacja zostanie rozbita na mniejsze działające osobno. W celu porównania obu serwisów baza danych nie będzie dzielona na mniejszą, tak aby to nie zapis i odczyt do niej był decydującym czynnikiem wskazującym na wyższość jednego z rozwiązań, ale będzie korzystać z różnych baz jednej usługi.

Aplikacja zarządzająca listami zadań posiadałaby cztery podstawowe moduły. Pierwszy z nich **interfejs użytkownika**, który za pomocą dostarczanego z przeglądarką *Fetch Api*[14] kontaktuje się z resztą aplikacji. Użytkownik po wejściu na stronę główną przechodzi do zakładki logowania. Następnie informacje wymagane do zalogowania wpisuje w formularz i przesyła żądanie do usługi **użytkownicy**, zwraca ona dane użytkownika z jego aktualnym *tokenem*. Interfejs użytkownika sprawdza następnie jego poprawność w jednostce *autentykacja*[17], gdy wszystko się zgadza klient dostaje ten sam *token* z powrotem. Gdyby dany token istniał, ale jego termin ważności wygasł, to ta usługa jest odpowiedzialna za wygenerowanie nowego. Uzyskany klucz od tej pory służy do otrzymywania informacji zarezerwowanych jedynie

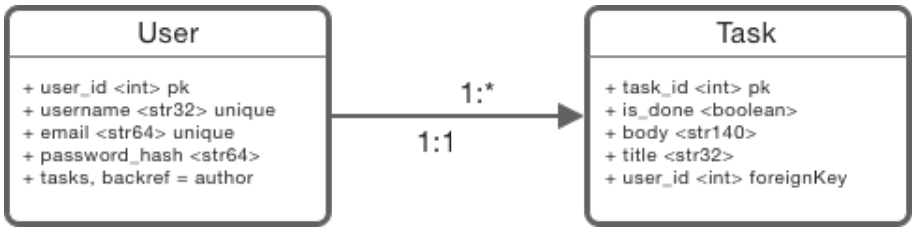
dla zalogowanych użytkowników. Inną usługą są listy zadań, komponent ten odpowiada za zwracanie, tworzenie i usuwanie, zadań dla poszczególnych ich właścicieli.



Rys. 3.2. Schemat architektury mikrousług.

3.5. Baza danych

Baza danych będzie posiadała dwie główne tabele: użytkownicy i listy zdań. Będą one połączone za pomocą relacji *one-to-many* w wielu *frameworkach* nazywanej również kluczem obcym. Oznacza to, że jeden użytkownik będzie mógł mieć wiele zadań przypisanych do siebie, natomiast jedno zadanie będzie przypisane tylko do jednego użytkownika. Struktura ta szczegółowo została pokazana na schemacie poniżej.



Rys. 3.3. Schemat bazy danych.

Jak można zauważyć na schemacie (rys.3.3), użytkownik posiada pole *tasks*, które odpowiada za relację *one-to-many* i tworzy referencję wsteczną[6] dla modelu zadań o nazwie *author*, dzięki temu korzystając z listy zadań nadal można mieć dostęp do informacji o kliencie. Doprowadza to do potrzeby stworzenia pola *user_id* (klucza obcego) i ustanowienia relacji *many-to-one* po stronie zadania.

Inną sprawą jest też, aby każdy użytkownik posiadał unikatowy *username* i *email*, tak żeby nie było sytuacji, w której wyciągając te informacje z bazy danych, a następnie nimi operując nie wysłać emaila z jakimiś wrażliwymi danymi do dwóch osób na raz. W bazie danych hasła użytkowników nie są zapisane zwykłym tekstem (ang. *plain text*), ale są zatajone za pomocą funkcji haszującej (ang. *hash function*). Domyślnie zestaw narzędzi *Werkzeug* posiada odpowiednie metody spełniające standard *PBKDF2* i wykorzystujące algorytm *SHA-1* do szyfrowania haseł[1].

3.6. Wykorzystane biblioteki

Ważne jest, aby aplikacja monolityczna, tak jaki i te mniejsze mikrousługi oparte były o ten sam zestaw technologii za nie odpowiedzialny. Dlatego większość wymienionych bibliotek będzie użytych w obu projektach. W liście zostały uwzględnione tylko te główne pakiety. Wiele z nich domyślnie instaluje mniejsze jako swoje zależności i wypisanie ich wszystkich nie jest konieczne, aby zrozumieć działanie aplikacji. Lista najistotniejszych znajduje się poniżej:

- *Vue.js*[13], biblioteka odpowiadająca za warstwę prezentacji.
- *Flask*[12], *mikroframework* odpowiedzialny za usługę serwera *HTTP*.
- *Nuxt.js*[18], framework do tworzenia progresywnych aplikacji oparty o *Vue.js*.
- *Flask SQLAlechmy*[19], system do zarządzania bazą danych w ramach aplikacji napisanych we *Flask*u.

3.7. Analiza wymagań

Tworzone na potrzebę pracy projekty powinny mieć zdefiniowane wymagania, tak aby jasne były konkretne ich cele do zrealizowania. Ma to zapobiec implementacji nadmiarowej ilości funkcji, a przy okazji wyszczególnić zestaw cech, które definiują projekt jako gotowy. W związku z tym, że celem pracy jest porównanie dwóch architektur, to sam projekt nie musi działać jak pełnoprawny produkt, a jedynie posiadać wszystkie istotne elementy potrzebne do realizacji danej architektury i możliwości porównania jej z drugą.

3.7.1. Wymagania funkcyjne

Korzystając z wspieranych przeglądarek internetowych (rozdział 3.8.2), użytkownik powinien móc wejść na wskazany adres i wczytać stronę główną projektu. Następnie klikając odnośnik *login* na górnej belce serwisu, następuje przekierowanie do strony logowania, gdzie będzie możliwy wybór jednego z dwóch formularzy, jeden do zalogowania już istniejącego użytkownika, a drugi do jego zarejestrowania. Po ich wypełnieniu i przesłaniu następuje autentykacja, dzięki czemu osoba korzystające z serwisu będzie mogła przeglądać utworzstone przez siebie zdania, tworzyć nowe i je usuwać. Dodatkowo możliwe będzie określenie czy zadanie zostało wykonane. Powyżej listy celów będzie znajdował się przełącznik do wyświetlania zadań, które nie zostały skończone, a także tych które są gotowe. Szata graficzna strony będzie minimalistyczna, wyróżniać się będą różne rodzaje szarości. Po skończeniu korzystania z strony użytkownik będzie mógł się z niej wylogować, wówczas straci dostęp do listy zadań i nie będzie mógł tworzyć nowych, ani edytować istniejących.

3.7.2. Wymagania niefunkcyjne

Istotą projektów jest sprawdzenie dwóch różnych podejść przy tworzeniu aplikacji internetowych. Ważne jest przetestowanie podstawowych funkcji, które oferują

serwisy, takich jak logowanie użytkownika, tworzenie formularzy, wysyłanie zapytań: *GET, POST, DELETE, UPDATE, PUT*[10] i komunikacja pomiędzy poszczególnymi komponentami serwera. Obie strony powinny posiadać również interfejs użytkownika, wraz z skryptami w języku *JavaScript* i arkuszami styli *CSS*.

Oba projekty będą wykorzystywać konteneryzację do spakowania ich i łatwiejszego uruchomienia na maszynie programisty, prywatnym serwerze wirtualnym lub usłudze do serwowania serwisów internetowych.

3.8. Środowisko programistyczne

Obie aplikacje będą dzielić się na stronę serwerową i warstwę prezentacji. Końcowy użytkownik do działania serwisu będzie potrzebował jedynie przeglądarki. Natomiast do uruchomienia serwera potrzebne będzie narzędzie odpowiadające za konteneryzację projektu.

3.8.1. Aplikacja serwera

Narzędziem wykorzystanym do konteneryzacji projektu jest *Docker*, dzięki tej usłudze żadne inne zależności ani programy nie są wymagane. Należy jedynie pobrać program z strony: `https://docs.docker.com/get-docker/` i w zależności od zainstalowanego na komputerze systemu operacyjnego, odpowiednio przejść wszystkie kroki w instalatorze.

Po instalacji wystarczy, że usługa ta zostanie uruchomiona[11] i w terminalu przejść do głównego katalogu projektu, a następnie wpisać `docker-compose up --build`. Komenda ta odpowiada za odnalezienie pliku *docker-compose.yml*, następnie zbudowanie zadeklarowanych w nim serwisów (przy pomocy plików *Dockerfile*), a na końcu ich uruchomieniu[15]. Wówczas w terminalu powinny pojawić się logi z działania serwera i odnośnikami *URL* ich adresów w przeglądarce.

3.8.2. Wspierane przeglądarki

Projekt nie skupia się na optymalizacji go pod poszczególne przeglądarki, w szczególności te, które nie wspierają podstawowych technologii *CSS*, takich jak *flexbox*[14]. W związku z tym wsparcie nie obejmuje *Windows Explorera* w wersji mniejszej niż 11, a także innych przeglądarek w starszych wersja. Aplikacja powinna działać prawidłowo na wszystkich nowych przeglądarkach takich jak *Firefox, Chrome, Opera, Microsoft Edge* i *IE 11* oraz aplikacjach dostępnych dla systemu *Android/iOS*.

[10] Podstawowe rodzaje zapytań dla standardu *HTTP REST*.

[11] Dla systemów z rodziny *Linux* konfiguracja jest nieco trudniejsza. Trzeba dodać jeszcze *Dockera* do specjalnej grupy, aby przy każdym uruchomieniu nie używać praw administratora, *sudo*. Opis procedury znajduje się na stronie `https://docs.docker.com/engine/install/debian/`. Dodatkowo należy stworzyć *demon*, żeby narzędzie uruchamiało się przy starcie systemu, link do opisu procedury `https://docs.docker.com/config/daemon/systemd/`.

Rozdział 4

Implementacja projektu - monolit

W tym rozdziale pokrótce zostanie opisana implementacja projektu monolitycznego. Został on podzielony według wcześniejszych założeń architektonicznych i obejmować będzie procedury potrzebne do napisania aplikacji serwerowej w *frameworku Flask* i szablonów *HTML* przy wykorzystaniu *Vue.js* i innych technologii webowych takich jak *Javascript*, *HTML* i *CSS*.

4.1. Backend

Sekcja ta odpowiada za opisanie implementacji od strony serwera, której zadaniem jest przyjmowanie zapytań, a następnie zwracanie odpowiadających im szablonów *HTML*. Za ich pomocą użytkownik może podejmować interakcję z stroną i używając wygenerowanych formularzy tworzyć kolejne żądania, na które program musi odpowiednio zareagować.

Pierwszym wykonanym krokiem było stworzenie nowego projektu o nazwie *mono*. Następnie wewnątrz niego utworzono katalog o tej samej nazwie i w jego środku dodano nowy folder *___init___.py*. Odpowiada on za import wszystkich innych modułów, a także stworzenie instancji dla głównej klasy z biblioteki *Flask*[12]. Ważne jest, zwrócenie uwagi na to, że silnik wykorzystany do generowania szablonów *Jinja2*, ma takie same elementy składni jak *Vue.js* i niemożliwe ich jednoczesne wykorzystanie, chyba, że zmienione zostaną ustawienie jednego z nich[13, 12]. W tym celu zaimplementowano klasę *VueFlask*, która dziedziczy po *Flask*, a następnie przypisano w niej skopiowany słownik[1] z ustawieniami *Jinja*. Korzystając z metody *update*[20] nadpisano klucze odpowiadające za deklarowanie zmiennych wewnątrz plików *HTML*. Kod klasy prezentuje się następująco:

[1]Struktura danych składająca się z klucza i jego wartości[20], dobrze nadaje się do zapisywania w niej różnego rodzaju ustawień.

```
from flask import Flask

class VueFlask(Flask):
    jinja_options = Flask.jinja_options.copy()
    jinja_options.update(dict(
        variable_start_string='%%',
        variable_end_string='%%',
    ))
app = VueFlask(__name__)
```

Listing 4.1. Zmiana opcji silnika *Jinja2* poprzez nadpisanie klasy *Flask*.

Klasa *VueFlask* przyjmuje argument *___name___*, jest to zmienna globalna zawierająca nazwę obecnego modułu i przekazanie jej jest wymagana przez dokumentacje *Flaska*[12].

W podobny sposób potrzebne jest stworzenie instancji dla biblioteki *Flask SQLAlchemy*, tak samo wymagane jest zaimportowanie klasy i przypisanie jej do zmiennej *db*, podając jako argument wcześniej utworzony obiekt *app*. Daje to możliwość zaprojektowanie modeli dla użytkowników i zadań.

4.1.1. Modele

Wszystkie modele będą umiejscowione w pliku *models.py* i zaprojektowane jako klasy dziedziczące po obiekcie *db.Model*[19]. Jako pierwszy został stworzony model użytkownika posiadający pola, *id*, unikalny identyfikator dla wszystkich użytkowników opisany wyłącznie przy pomocy liczb całkowitych. *Username*, kolumnę w bazie danych, zapisaną za pomocą ciągu znaków, mających długość *32* liter. *Email*, podobnie jak jego poprzednik, to również ciąg znaków długości *64* liter i *password*, mające maksymalna długość *128* znaków. Zgodnie z ustaleniami z rozdziału 3.3, hasła nie będą w bazie danych zapisane zwykłym tekstem, tylko szyfrowane. W związku z tym klasa posiada jeszcze dwie metody odpowiedzialne za ich utajnienie i sprawdzanie zgodności z haszem[1].

```
from werkzeug.security import generate_password_hash,
    check_password_hash
from datetime import datetime
from mono import db


class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(32), index=True, unique=True)
    email = db.Column(db.String(64), index=True, unique=True)
    password_hash = db.Column(db.String(128))

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)
```

Listing 4.2. Klasa odpowiedzialna za model użytkownika.

Kolejnym modelem jest ten odpowiedzialny za definiowanie zadań. Posiada on także pole *id* będące unikatową liczbą identyfikującą każde z nich. Dzięki polu *is_done*, które jest typu *prawda/fałsz*[2] użytkownik może oznaczyć zdanie jako skończone lub nie. Każdy zapisany w bazie danych cel będzie składał się z tytułu (*header*) i jego opisu (*body*). Obie własności są kolumnami typu *string* zawierających ich w kolejności *64* i *256*. Ostatnim polem w tym modelu jest stempel czasu (ang. *time stamp*) trzymającym dane o dacie i dokładnym czasie (zgodnym z formatem *utc*[20] zawierającym godziny, minuty, sekundy i milisekundy), jego wartość jest natychmiastowo przypisywana, przy utworzeniu zadania, na podstawie aktualnych informacji z systemu[20].

```
1 from datetime import datetime # wymagany import umieszczony na
    poczatku pliku
2
3
4 class Task(db.Model):
5     id = db.Column(db.Integer, primary_key=True)
6     is_done = db.Column(db.Boolean)
7     header = db.Column(db.String(68))
8     body = db.Column(db.String(256))
9     timestamp = db.Column(db.DateTime, index=True, default=datetime
    .utcnow)
```

Listing 4.3. Klasa odpowiedzialna za model zadania.

Brak jeszcze informacji o korelacji między użytkownikiem, a zadaniem. Zgodnie z schematem opisanym w rozdziale 3.5. Jedna osoba korzystająca z aplikacji może mieć wiele stworzonych przez siebie celi, natomiast jeden taki cel może mieć tylko jednego autora. Dlatego wewnątrz klasy *User* należy dodać pole *task*, tworząc powiązanie *db.relationship*, odpowiadające relacji *one-to-many*[19] z referencją wsteczną do zadania, o nazwie *author*. Oznacza to, że będzie można odnieść się od strony użytkownika do utworzonych przez niego zadań dzięki temu, że będą one miały w bazie danych tabelę *author* posiadającą informacje umożliwiając jego identyfikację.

Natomiast relacja pomiędzy zadaniem i jego właścicielem opisana jest jako *one-to-many*, nazywana również *Foreign Key*[1, 19] i do jej utworzenia należy po prostu zapisać w modelu *user_id* będącą kolumną numeryczna, która jako drugi argument przyjmuje *db.ForeignKey('user.id')* z odniesieniem właśnie do identyfikatora użytkownika.

Samo utworzenie modeli w języku *Python* nie wystarczy, aby baza danych przyjęła wymaganą strukturę. Najpierw trzeba skonfigurować połączenie z nią. W folderze głównym projektu utworzono plik *.env* przechowujący wszystkie dane na temat środowiska, są w nim opisane informacje o bazie danych, tym, czy aplikacja działa w trybie produkcyjnym, czy powinna wyświetlać informacje o błędach (tryb *debug*)[12]. Są one wczytywane przez plik *config.py* i mapowane na klasy dotyczące danego środowiska konfiguracyjnego[3], a na końcu za pomocą funkcji z biblioteki *Flask* o nazwie *config.from_object*[12], wczytywane do opisanej wcześniej instancji biblioteki, zmiennej *app*.

[2] Z angielskiego *Boolean*, trzymające informację binarną o stanie logicznym[19].

[3] Istnieją różne środowiska, w których aplikacja może zostać uruchomiona takie jak testowe, produkcyjne lub deweloperskie.

```
POSTGRES = {
    'user': os.environ.get('DB_USER'),
    'pw': os.environ.get('DB_PASS'),
    'db': os.environ.get('DB_NAME'),
    'host': os.environ.get('DB_ADDRESS'),
    'port': os.environ.get('DB_PORT'),
}


class Config:
    DEBUG = False
    SECRET_KEY = os.environ.get('SECRET_KEY')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SQLALCHEMY_DATABASE_URI = 'postgresql://%(user)s:\
        %(pw)s@%(host)s:%(port)s/%(db)s' % POSTGRES


class ConfigDevelopment(Config):
    DEBUG = True
```

Listing 4.4. Klasa odpowiedzialna za wczytywanie i zapisanie konfiguracji międzyinnymi bazy danych.

Aplikacja do poprawnego działania potrzebuje takich informacji jak nazwa użytkownika, login administratora bazy danych, jego hasło, nazwa bazy i adres, na którym znajduje się włączona usługa. Zmienne mające w nazwie *SQLALCHEMY* są wymagane do uzupełnienia przez bibliotekę *Flask SQLAlchemy* i potrzebne do jej działania. Proces konfiguracji już samej bazy danych *Postresql* zostany dokładniej opisany w rozdziale o konteneryzacji aplikacji (rozdz. 6).

Proces tworzenie struktur lub aktualizowania bazy danych o wcześniej zdefiniowane modele, to migracje. Powinno je się wykonać po każdej zmianie w kodzie *models.py* i uruchomieniu nowo stworzonej bazy danych[10]. Do ich obsługi w systemie *Flask* istnieje narzędzie o nazwie *Alemibic*[4] i jest ono częścią biblioteki *Flask Migration*[5].

Uruchomienie migracji nie będzie wiązało się z włączeniem aplikacji[21], ale będą one tworzone na żądanie osoby nią operującą, dlatego potrzebne będzie dodanie do zbioru komend *Flaska*[12] nowych, odpowiedzialnych za włączenie *Alembica*. Do tego potrzebna będzie jeszcze jedna biblioteka o nazwie *Flask Script*[6], a następnie dopisanie do pliku *mono/___init___.py* następujący fragmentów kodu:

```
migrate = Migrate(app, db)
manager = Manager(app)

manager.add_command('db', MigrateCommand)
```

Listing 4.5. Dodanie migracji do skryptu *Flaska* w pliku *mono/___init___.py*.

Wówczas możliwe będzie uruchomienie w konsoli polecenia `flask db init` (w głównym katalogu projektu) do inicjalizowania połączenia modeli z bazią danych, a

[4]Link do dokumentacji *Alembica*: `https://alembic.sqlalchemy.org/en/latest/`.

[5]Link do dokumentacji pakietu *Flask Migration*: `https://flask-migrate.readthedocs.io/en/latest/`.

[6]Link do dokumentacji biblioteki *Flask Script*: `https://flask-script.readthedocs.io/en/latest/`.

następnie `flask db migrate` do utworzenia plików migracyjnych. W katalogu powinien pojawić się nowy folder o nazwie *migrations*, w którym będą trzymane wszystkie informacje o zmianach w strukturze modeli, tak aby mogły one zostać zaaplikowane do bazy danych. Warto te pliki mieć w repozytorium projektu i gdy będzie tworzona jego nowa instancja, to wówczas całą strukturę bazy danych (bez samych danych) można przywrócić za pomocą komendy `flask db migrate`. Natomiast po każdorazowej zmianie w pliku *models.py* należy uruchomić komendę `flask db upgrade`. Po wykonaniu tych poleceń model danych dla aplikacji jest gotowy, następnym krokiem jest opracowanie widoków.

4.1.2. Widoki

Aplikacja napisana w *Flasku* odbiera wszystkie informację kierowane w jej stronę i w zależności od wytycznych programisty odpowiednio je przekształca. Wykorzystuje do tego mechanizm przetwarzania funkcji nazwany w języku *Python*, dekoratorem[20]. W zależności od adresu i metody *HTTP* wywoływana jest odpowiednia funkcja w widoku.

```
1 @app.route('/index')  # symbol 'at' - oznacza deklaracji dekoratora
2 def index:
3   return render_template('index.html')
```

Listing 4.6. Dekorator *@app.route('index')* zastosowany na funkcji index.

W obiekcie *app* znajduje się metoda o nazwie *route*, która przyjmuje argument *rule* opisujący za pomocą *wyrażeń regularnych*[20, 12] zasadę, która pozwoli dopasować żądanie do podanego wzorca. Framework zapisuje te dane w dostępnych adresach serwera, a funkcje na której wykorzystano dekorator przypisuje jako tą, która ma zostać bezpośrednio wykonana po otrzymaniu żądania. Istnieje jeszcze możliwość dodania opcji, tak aby, dana funkcja była wywoływana przy odpowiednich typach zapytań.

```
1 @app.route('/task/<int:task_id>', methods=['GET', 'POST', 'DELETE'
    ])
2 def task(task_id):
3   # ...
4   return render_template('task.html', task=task)
```

Listing 4.7. Dekorator *@app.route* z dodatkową opcja *methods*.

W przykładzie powyżej zasada dla zadania ma jeszcze dopisane *<int:task_id>* jest to *reguła dla zmiennej* (ang. *variable rule*[12]), która może być przekazana przez adres *URL* i przesłana do funkcji *task* jako argument. Na przykład zapytanie *https://przykładowa-strona.co/task/123* wywoła funkcję *task* z argumentem *task_id* o wartości *123*. Dzięki temu możliwe jest łatwe identyfikowanie zasobów podając już sam adres *URL*. W żądaniu *HTTP* poza *parametrem identyfikującym* dany zasób możliwe jest jeszcze jego określenie przy pomocy *parametrów zapytania* (ang. *query params*). Jest to struktura podawana w adresie *URL* po znaku zapytania (*?*). Dostęp do tych parametrów możliwy jest poprzez obiekt *request*[12].

```
from flask import request

# dla zapytania /task?done=true
@app.route('/task', methods=['GET'])
def task():
  # wszystkie przekazane parametry zapytania
  print(request.query_string)
  is_done = request.args.get('done') # zwroci True
  return render_template('task.html', task=task)
```

Listing 4.8. Dostęp do parametrów zapytania w frameworku *flask*.

W ten sposób realizowane jest w *Flasku* kierowanie zapytań. Dla potrzeb aplikacji monolitycznej stworzone zostały funkcje dla następujących wzorców:

- `\`, `\index` dla strony głównej.
- `\login`, *URL* przekierowujący do strony logowania.
- `\register` do strony rejestracji.
- `\logout` stosowany do wylogowania użytkownika.
- `\task`, *URL* wyświetlający wszystkie zadania, można do niego dopisać parametr zapytania taki jak *done*, aby wyświetlić tylko ukończone zadania `\tasks?done=True`.
- `\tasks\<int:task_id>` odnosi się do zadania o określonym identyfikatorze.

Innym zadaniem tej warstwy aplikacji jest przetwarzanie informacji z bazy danych. Rozszerzenie *Flask-SQLAlchemy* przejmuje wszystkie zapytania do niej, udostępniając przy tym obiekt sesji, który jest wykorzystywany przez widoki do przetwarzania informacji[1]. Przykładem może być aktualizacja zadania z odpowiednim numerem identyfikacyjnym. Użytkownik aplikacji podaje go w *URLu*, funkcja z widoku dostaje go w argumencie, a następnie musi dla tej wartości znaleźć odpowiedni post w bazie danych. W tym celu importuje obiekt *db Flask-SQLAlchemy* z *___init___.py* i model *Task* z *models.py*. Następnie przy pomocy metody *query* odnajduje go, nadpisuje i wykorzystując sesję zastępuje go w bazie danych.

```
from mono.models import Task

@app.route('\task\<int:task_id>', methods=['PUT'])
def update_task(task_id):
  form = UpdateTaskForm
  if form.validate_on_submit():
    task = Task.query.filter_by(id=task_id).first()
    task.body = form.task.data.body
    db.session.commit()
    flash('Task updated')
  return url_for('index')
```

Listing 4.9. Wykorzystanie sesji bazy danych w widoku do zaktualizowania zadania.

Warto zwrócić uwagę na konstrukcję odpowiadającą za kwerendę do bazy danych, bez wywołania *first()* lub *all()* jest to obiekt typu *zapytanie do bazy danych*[19], a nie dane, które można edytować.

Wśród rozszerzeń do *Flaska* istnieje, takie, które pomaga zarządzać formularzami, *FlaskWTF*[7]. Są one w aplikacji monolitycznej główną metodą na przesłanie danych z *frontendu* do aplikacji serwera i pozwalają użytkownikowi na interakcje z nimi. To w jaki sposób są one zarządzane przypomina tworzenie modeli.

```
1 from wtforms.validators import ValidationError, DataRequired, Email
    , EqualTo
2 from flask_wtf import FlaskForm
3 from wtforms import StringField, PasswordField, BooleanField,
    SubmitField, TextField, TextAreaField
4
5
6 class LoginForm(FlaskForm):
7     username = StringField('Username', validators=[DataRequired()])
8     password = PasswordField('Password', validators=[DataRequired()
    ])
9     remember_me = BooleanField('Remember Me')
10    submit = SubmitField('Sign In')
```

Listing 4.10. Tworzenie formularzy przy pomocy *FlaskWTF*. Przykład formularza do rejestracji użytkownika.

Następnie wystarczy przekazać instancję klasy formularza do funkcji *render_template('login.html', form=form)* w funkcji widoku. Przekazany obiekt widoczny jest w kodzie szablonu *HTML*.

```
1 <form action="" method="post" novalidate>
2     <p>
3       %% form.username.label %%<br>
4         %% form.username(size=32) %%<br>
5     </p>
6     <p>
7       %% form.password.label %%<br>
8         %% form.password(size=32) %%<br>
9     </p>
10    <p>%% form.submit() %%</p>
11 </form>
```

Listing 4.11. Uproszczony formularz *FlaskWTF* zaimplementowany w pliku *login.html*.

W razie kliknięcia przycisku *Submit* w kodzie funkcji widoku, znajduje się warunek pozwalający sprawdzić przesłane dane.

```
12 @app.route('/login', methods=['GET', 'POST'])
13 def login():
14     form = LoginForm()
15     if form.validate_on_submit():
16         user = User.query.filter_by(username=form.username.data).
    first()
17         if user is None or not \
18             user.check_password(form.password.data):
19             flash('Invalid username or password')
20             return redirect(url_for('login'))
21         login_user(user, remember=form.remember_me.data)
22     return render_template('login.html', form=form)
```

Listing 4.12. Sprawdzenie przesłanych danych z formularza logowania w widoku.

[7]Link do dokumentacji rozszerzenia: `https://flask-wtf.readthedocs.io/en/stable/`.

Ostatnim istotnym elementem widoków jest sposób przechowywania sesji użytkownika, zostało to zaimplementowane przy pomocy rozszerzenia *Flask-Login*[8]. Ważne jest utworzenie instancji *LoginManagera* i przekazanie jej do pliku, na przykład *models.py*, tak, aby utworzyć tam funkcję korzystającą z dekoratora *user_loader*, pozwalającego na uzyskanie informacji o użytkowniku z bazy, dzięki temu możliwe jest korzystanie w widokach z funkcji umożliwiającej sprawdzenie, czy jest on zalogowany lub nadania niektórym adresom restrykcji pozwalających na korzystanie z nich wyłącznie wtedy, gdy użytkownik jest zalogowany[21].

4.2. Frontend

Ostatnim elementem aplikacji serwerowej są szablony wysyłane do przeglądarki użytkownika. Następnie na ich podstawie generowana jest warstwa prezentacji, pozwalająca na interakcje z aplikacją. Wszystkie szablony trzymane są wewnątrz folderu *templates*. Uciążliwe byłoby tworzenie dla każdej strony takiej samej podstawowej warstwy graficznej, dlatego wewnątrz pliku *base.html* są importowane dodatkowe skrypty i arkusze stylów. Plik ten jest podstawą innych szablonów, dzięki silnikowi *Jinja2* można go eksportować w każdym kolejnym pliku przy pomocą składni *extend*[12].

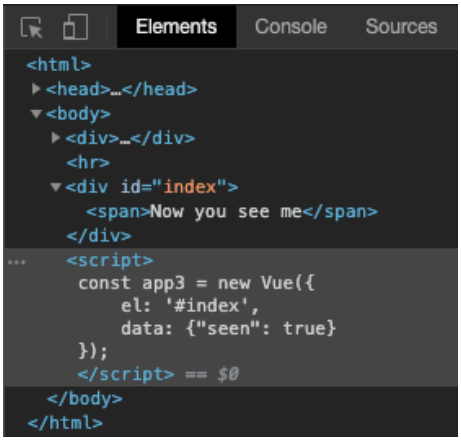
Frameworkiem, który został użyty do napisania warstwy prezentacji jest *Vue.js*, został on załączony w nagłówku *base.html* jako ścieżka do zminimalizowanego pliku *vue.min.js* pobranego z strony z dokumentacją[13] lub serwisu *github*[9]. Każdy szablon posiada nowo utworzoną instancje obiektu *Vue*, która posiada pole *el* w którym jest przypisany element *HTML*[13] do którego podpięta została aplikacja. Ważną rzeczą jest też przekazanie zmiennych z serwera do kodu *Javascript*, odbywa się to w przy wykorzystaniu filtra *tojson* składni *Jinja2*[12].

```
23 {% extends "base.html" %}
24
25 {% block content %}
26     <div id="index">
27       <span v-if="seen">Now you see me</span>
28     </div>
29  <script>
30  const appIndex = new Vue({
31      el: '#index',
32      data: %% data | tojson %%
33  });
34  </script>
35 {% endblock %}
```

Listing 4.13. Przykładowy kod strony głównej przy użyciu *Flaska* i *Jinja2*.

Jeżeli spojrzeć w kod źródłowy strony w przeglądarce wówczas można zauważyć, jak wyglądają już wygenerowane dane z serwera dla podanego przykładu i zauważyć, że metoda *tojson* zwraca dane przesłane w formie akceptowanej jako obiekt języka *Javascript*, idealnie wpasowując się w strukturę danych wymaganych przez pole *data* obiektu instancji *Vue*. Co sprawie, że można nimi operować bezpośrednio w

[8]Link do dokumentacji *Flask-Login*: `https://flask-login.readthedocs.io/en/latest/`.
[9]Link do repozytorium *Vue.js* w serwisie *Github*: `https://github.com/vuejs/vue`.



Rys. 4.1. Kod źródłowy strony wygenerowanej na podstawie listingu 4.13. Przeglądarka *Safari 13.1* pod systemy *MacOS*.

kodzie *HTMLa* poprzez odpowiednie dyrektywy[13] (w tym przypadku dyrektywę *v-if*) biblioteki *Vue.js*.

W prosty sposób osiągnięto komunikację między serwerem, a biblioteką odpowiadającą za warstwę prezentacji. Niepotrzebne byłoby żadne dodatkowe narzędzie wystarczyło tylko wykorzystać elementy dostępne w ramach obu tych frameworków.

Istnieją biblioteki pozwalające na szybkie tworzenie komponentów w *Vue*, nazywane *UI Component Frameworks*. Są to zbiory gotowych elementów, do których wystarczy przekazać odpowiednio sformatowane dane i ustawienia, a wygenerują one żądany fragment interfejsu użytkownika. W razie, gdy dostarczony komponent nie wygląda odpowiednio lub ma za mało funkcji to takie frameworki dostarczają proste *API* do edycji ich wyglądu lub zastosowania, ale w większości przypadków ich ustawienia i dostępne możliwości są na tyle bogate, że ich większe zmiany są sporadyczne. Zastosowaną biblioteką w projekcie jest *ElementUI*[10], można ją pobrać z strony dokumentacji lub serwisu *Github*[11] następnie przenieść do folderu `static/js` i odpowiednio dodać do referencję w pliku *base.html* do:

- skryptu języka *Javascript*: `js/element-2.13.2/lib/index.js`
- arkuszy stylów biblioteki: `js/element-2.13.2/lib/theme-chalk/index.css`

W folderze *static/css* znajduje się plik *main.css* odpowiada on za wszystkie style napisane w projekcie monolitycznym i podobnie do bibliotek w języku *Javascript* został on dołączony do pliku *base.html*, więcej o kaskadowych arkuszach stylów można przeczytać w dokumentacji fundacji *Mozilla*[14].

[10]Link do dokumentacji *ElementUI*: `https://element.eleme.io/`.
[11]Link do repozytorium *ElementUI* w serwisie *github*: `https://github.com/ElemeFE/element`.

Rozdział 5

Implementacja projektu - mikrousługi

W tym rozdziale opisany zostanie proces implementacji czterech mniejszych usług składających się w jedną całą aplikację. Podstawową różnicą między tymi serwisami, a podejściem monolitycznym jest to, że każda z nich działać będzie jako osobny serwer zajmując inny port. Zamiast porozumiewania się między komponentami przy pomocy struktury klas, będzie odbywać się to poprzez żądania *HTTP* i stworzony w celu interfejs *API*.

5.1. Zadania

Pierwszą opisaną mikrousługą jest ta odpowiedzialna za zarządzanie zadaniami. Jest to osobna aplikacja napisana w bibliotece *Flask*. Podobnie jak w przypadku usługi monolitycznej ma ona połączenie z bazą danych przy pomocy *ORMu*. Model zadania jest identycznie stworzony jak w poprzednim projekcie. Korzystając z biblioteki *Flask-SQLAlchemy* tworzona jest instancja jej głównej klasy, do której przekazano obiekt *app*, a następnie szablon klasy *Task*. Różnicą jest to, że posiada ona metodę *to_dict*, której zadaniem jest zwracanie słownika z wszystkimi polami, które będą dostępne dla klienta interfejsu *API*. W związku z tym, że wspomniany interfejs powinien być zgodny z standardem *JSON*, to nazwy kluczy w słowniku są zapisane w inny sposób niż standardowy dla języka *Python*, a zgodny z tym w języku *Javascript*[1]. Pozwali to w przyszłości na łatwiejsze operowanie danymi wewnątrz usługi odpowiedzialnej za interfejs użytkownika.

W związku z tym, że każda z poszczególnych aplikacji korzysta z tej samej usługi bazodanowej ale innych baz, to należy w pliku *.env* dodać nową nazwę dla niej. Niestety nie zostanie ona stworzona automatycznie i należy wykorzystać komendę `psql -U postgres` i po uruchomieniu programu dodać nową bazę danych przy pomocy komendy `CREATE DATEABASE postgrestasks;`. Takie rozwiązanie tworzy też kolejny problem, nie jest możliwe korzystanie z relacji między poszczególnymi usłu-

[1]W języku *Python* obwiązujący standard zapisywania nazw zmiennych to *snake_case*[20], gdzie poszczególne wyrazy odróżnia się poprzez podkreślnik. Natomiast w języku *Javascript* odbywa się to wyróżniając pierwszą literę drugiego słowa jako dużą. Konwencja ta nazwana jest *camelCase*. Link do reszty informacji: `https://firefox-source-docs.mozilla.org/code-quality/coding-style/coding_style_js.html`.

gami. Zadania i użytkownicy nie mogą być z sobą powiązani przez klucz obcy. *Flask-SQLAlchemy* udostępnia mechanizm *binds*, który w ramach zewnętrznego systemu pozwala tworzyć takie relacje[19], ale odpowiednia implementacja tego mechanizmu jest dość trudna dla kilku mikrousług korzystających z różnych baz danych. Struktura aplikacji tego nie wymaga, odpowiednio przygotowując funkcję odpowiedzialną za tworzenie zadań można przypisywać do nich *identyfikator* użytkownika na podstawie przesłanych *tokenów*. Zapewnią one integralność i to, że te dane nie będą dostępne publicznie.

```
1  class Task(db.Model):
2      id = db.Column(db.Integer, primary_key=True)
3      is_done = db.Column(db.Boolean)
4      header = db.Column(db.String(68))
5      body = db.Column(db.String(256))
6      user_id = db.Column(db.Integer)
7      timestamp = db.Column(db.DateTime, index=True, default=datetime
       .utcnow)
8
9      def to_dict(self):
10         return {
11             'id': self.id,
12             'isDone': self.is_done,
13             'header': self.header,
14             'body': self.body,
15             'timestamp': self.timestamp
16         }
```

Listing 5.1. Model zadania w mikrousłudze *tasks* wraz z metodą *to_dict*.

Najważniejszą warstwą mikrousługi serwerowej jest *API*. Odpowiadają za nie funkcje, które podobnie jak w aplikacji monolitycznej otrzymują żądania i odsyłają odpowiedzi. Nie są to natomiast odpowiedzi w formie renderowanych szablonów, a obiektów typu *JSON*. Biblioteka *Flask* posiada metodę o nazwie *jsonify*, która pozwala w łatwy sposób na przesyłanie odpowiedzi w tej formie. Inną istotną sprawą jest utrzymanie spójnej struktury dla *URLów* takich zapytań. W projekcie przyjęta będzie konwencja, gdzie żądany obiekt w tym przypadku *task* będzie przesyłany jako lista, gdy klient wyśle zapytanie *GET* pod adres `\tasks` (tu możliwe jest podanie parametrów zapytań do ich odfiltrowania). Dla metody POST z podanymi odpowiednimi danymi możliwe będzie stworzenie nowego celu, gdy potrzebne będzie poszczególne zadanie, wówczas wystarczy odpytanie adresu `tasks/<int:id>`, gdzie `<int:id>` odpowiada jego identyfikatorowi. Ten adres w zależności od rodzaju zapytania będzie odpowiadać za usunięcie lub uaktualnienie podanego celu użytkownika. W tej usłudze zapytanie kierowane na adres \, `\index` zwracać będzie wiadomość powitalną informującą o rodzaju danych serwowanych przez aplikację.

```
1 @app.route('/', method=['GET'])
2 @app.route('/index', method=['GET'])
3 def index():
4     return jsonify(message="Welcome to task micro-service", success
      =True)
```

Listing 5.2. Funkcja *index* zwracająca obiekt *JSON*.

Nadal możliwe jest sprawdzenie zwróconych danych w przeglądarce, wystarczy uruchomić aplikację i w pasu wyszukiwania wpisać `http://localhost:5000`,

wówczas powinien zostać wyświetlony obiekt *JSON*. Dla prostych zapytań typu *GET* możliwe jest ich takie podejrzenie. Problemem jest przetestowanie innych metod *HTTP*, ale istnieją do tego narzędzia takie jak prosty program konsolowy *cURL* (`https://curl.haxx.se/`) lub okienkowa aplikacja *Postman* (`https://www.postman.com/`).

```
$user@home: curl -i -H "Content-Type: application/json" http://
    localhost:5001/tasks

HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 59
Server: Werkzeug/1.0.1 Python/3.8.3
Date: Sun, 07 Jun 2020 19:58:18 GMT

{"message":"Welcome to task micro-service","success":true}
```

Listing 5.3. Testowe zapytanie wykonane przy pomocy programu *cURL* w terminalu systemu *Unix*.

Każde zapytanie o jakiekolwiek zadanie wymaga autentykacji (więcej w rozdziale 5.3) jest to warunek konieczny do jego otrzymania. Założenia aplikacji wykluczają możliwość podglądania zadań innych użytkowników, dlatego każda funkcja widoku najpierw sprawdza, czy obiekt *request*[12] posiada nagłówek[11] *Authentication*, w którym jest token. Gdyby go nie było, to zwracana jest wiadomość o braku autoryzacji, jeśli jest, to aplikacja kontaktuje się z usługą odpowiedzialną za uwierzytelnianie i pod adres `/token-decode` wysyłany jest wspomniany wcześniej token. Następnie jeżeli wszystko jest w porządku, to w odpowiedzi usługa dostaje identyfikator użytkownika, który następnie jest wykorzystywany do tworzenia kwerend bazy danych. Dobrym przykładem może być widok odpowiedzialny za listowanie wszystkich zadań.

```
@app.route('/tasks', methods=['GET'])
def get_tasks():
    auth_header = request.headers.get('Authorization')
    user_id = auth_user(auth_header)
    if not isinstance(user_id, int):
        return jsonify(message='Authentication failed', success=
    False)

    tasks_list = Task.query.filter_by(user_id=user_id)
    return jsonify(tasks=[selected_task.to_dict() for selected_task
    in tasks_list], success=True)
```

Listing 5.4. Kod widoku odpowiedzialnego za listowanie zadań.

Po otrzymaniu identyfikatora użytkownika sprawdzane jest, czy *user_id* jest typu numerycznego, tak aby upewnić się, czy to co zwróciła kontaktująca się z serwisem od uwierzytelniania usługa jest informacją poprawną, którą można wykorzystać do wyszukiwania zadań. Korzystając z mechanizmu kwerend biblioteki *Flask-SQLAlchemy* wybierane są tylko cele należące do użytkownika wysyłającego zapytanie, a następnie przy pomocą mechanizmu przekształcania list (ang. *list comprehension*)[20] wszystkie zadania są zamieniane na słowniki[2], umieszczane w ta-

[2]Proces ten nazywany jest serializacją danych. W frameworkach takich jak *Django* istnieją gotowe mechanizmy odpowiedzialne za ten mechanizm[10].

blicy i wysyłane przy pomocy funkcji *jsonify*. Jest ona odpowiednikiem stworzenia nowego obiektu typu *Request* do którego przekazany jest obiekt *JSON*, stworzony przy wykorzystaniu standardowej biblioteki *json* i metody *dumps*[12].

Poza funkcjami widoku, które wykorzystują metodę *GET* są jeszcze adresy odpowiedzialne za tworzenie nowego zadania. Wówczas do parametru *methods* przekazywana jest tablica z ciągiem znaków, *POST*. Wewnątrz tej funkcji poza sprawdzeniem poprawności tokenu, przeglądany jest obiekt *request.json*, czy posiada on klucz *header*, który jest jedynym wymaganym polem do utworzenia nowego zadania. Następnie otrzymane informacje są wykorzystane do utworzenia zadania, a jego dane są zwracane w odpowiedzi.

```
1 $user@home: curl -i -H "Content-Type: application/json" -H "
    Authorization: Bearer token_uzytkownika "  -X POST -d '{"header
    ":"Read the docs"}' http://localhost:5001/tasks
2
3 HTTP/1.0 200 OK
4 Content-Type: application/json
5 Content-Length: 127
6 Server: Werkzeug/1.0.1 Python/3.8.3
7 Date: Sun, 07 Jun 2020 19:49:25 GMT
8
9 {"success":true,"task":{"body":"","header":"Read the docs","id":2,"
    isDone":false,"timestamp":"Sun, 07 Jun 2020 19:49:25 GMT"}}
```

Listing 5.5. Testowe zapytanie *POST* wykonane przy pomocy programu *cURL* w terminalu systemu *Unix*.

W usłudze zaimplementowany jest również mechanizm aktualizacji celów, wówczas należy pod adres `/tasks/<int:task_id>` wysłać żądanie *PUT*. Cała procedura jest podobna jak ta do tworzenia nowego zadania, ale w obiekcie *request.json* sprawdzane jest, czy zostały przesłane wszystkie edytowalne pola wykorzystywane przy jego tworzeniu.

Ostatnim typem żądania jest usunięcie zadania, wówczas wykorzystywaną metodą jest *DELETE*, a w parametrze potrzebny jest jego number. Po uwierzytelnieniu, tworzona jest kwerenda, która odnajduje zadanie o podanym identyfikatorze dla danej osoby. Możliwe jest to przy pomocy kwerendy *AND*, którą łatwo implementuje się podając następne parametry funkcji *filter_by*. Po znalezieniu zadania wykorzystywana jest funkcja obiektu *db.session* o nazwie *delete*, a po zatwierdzeniu operacji wysyłana jest informacja o usunięciu zadania wraz z jego danymi.

```
@app.route('/tasks/<int:task_id>', methods=['DELETE'])
def delete_task(task_id):
  # czesc odpowiedzialna za uwierzytelnianie tokenu
  # ...
    try:
        task_to_delete = Task.query.filter_by(
            id=task_id,
          user_id=user_id
        ).first()
        db.session.delete(task_to_delete)
        db.session.commit()
    except:
        return jsonify(
            message="Cannot deleted task " + str(task_id),
            sucess=False
          )
    return jsonify(
      message="Task deleted",
      task=task_to_delete.to_dict(),
      sucess=True
    )

```

Listing 5.6. Kod widoku odpowiedzialnego za usuwanie zadań.

W razie jakichkolwiek błędów wszystkie wspomniane funkcję zwracają odpowiednie komunikaty wraz z informacją o niepowodzeniu. Wykorzystując dekorator *errorhandler* można również stworzyć odpowiedzi dla wszelkiego rodzaju błędów, których nie można obsłużyć wewnątrz zaimplementowanych widoków, takich jak skorzystanie z nieistniejącego *URLa* przez użytkownika[12].

```
@app.errorhandler(404)
def wrong_entrypoint(error):
    return jsonify(
      error="API entrypoint not found",
      success=False
    ), 404
```

Listing 5.7. Wykorzystanie dekoratora *errorhander* do nadpisania kodu błędu *404*.

5.2. Użytkownicy

Kolejna potrzebną usługą jest ta odpowiedzialna za zarządzenie użytkownikami. Posiada ona także dostęp do bazy danych z wykorzystaniem modułu *Flask-SQLAlchemy*. Utworzone modele mają taką samą strukturę co w aplikacji monolitycznej, jednak nie posiadają relacji z zadaniami, ze względu na użycie wcześniej wspomnianego mechanizmów *tokenów* nie ma potrzeby przechowywania żadnych relacji.

Głównym zadaniem tej mikrousługi jest tworzenie nowych użytkowników, zarządzenie ich logowaniem, wysyłaniem odpowiednich informacji o nich. W tym celu zaimplementowano niżej określony schemat *REST API*:

- `/`, `/index` - zwraca metodą *GET* informację, że podany *URL* odnosi się do mikrousługi odpowiedzialnej za użytkowników.

- `/users` - dla metody *GET* jeżeli klient prześle poprawny *token*, to zwraca informację takie jak email, nazwa użytkownika. Natomiast żądanie *POST* tworzy nowego użytkownika i zwraca dla niego *token*, tak, aby mógł on mieć dzięki niemu dostęp do reszty funkcji aplikacji.

- `/login` - wysyłając pod ten adres aplikacji żądanie *POST* użytkownik może się zalogować. Po udanym procesie przesyłany jest nowy *token* do uwierzytelniania.

- `/logout` - pod tym adresem możliwe jest wylogowanie użytkownika, wówczas przesłany token wysyłany jest do usługi uwierzytelniającej i tam dodawany do czarnej listy, tak aby nigdy więcej nie można było go użyć do korzystania z funkcji dostępnych wyłącznie dla zalogowanej osoby.

Szczegółowa implementacja widoków jest podobna do tej z usługi odpowiedzialnej za zadania. W każdym z nich, gdzie wymagany jest *token* najpierw sprawdzana jest jego poprawność, w razie braku, zwracana jest odpowiednia wiadomość. Może zdarzyć się, że z jakiejś przyczyny token wygaśnie, wówczas aplikacja odpowiadająca za interfejs użytkownika automatycznie wyloguje go, tak aby użytkownik znów się zalogował przypisując mu nowo stworzony *token*. Następnie na podstawie danych z tokenu wybierany jest użytkownik z bazy danych, a następnie odpowiednie dane są zwracane. Istnieją również adresy, gdzie przesyłane są dane i sprawdzane jest, czy przyjęte do serwera żądanie posiada w sobie wymagane informacje, w przeciwnym razie zwracana jest wiadomość o błędzie. W funkcjach, gdzie tworzone są *tokeny* przesyłane jest zapytanie do serwera tej usługi pod adres `/token-encode` z identyfikatorem użytkownika jako parametrem. Wówczas tworzony jest ciąg binarny, kodowany później na klucz uwierzytelniający i przesyłany jako odpowiedź serwera. Następnie w odpowiednio sformatowanej informacji wysyłany jest on do klienta pod kluczem *tokenAuth*. Ostatnią istotną różnicą w widokach jest wylogowywanie, metoda odpowiedzialna za nie jest podobnie zbudowana do *get_user* z tą różnicą, że adresem pod który serwer wysyła token to `/token-blacklist`, który natomiast w informacji zwrotnej potwierdza wylogowanie, gdy posiada ona status *success* to przekazywane jest to klientowi, wówczas może on bezpiecznie usunąć nieaktualny token.

5.3. Autentykacja

Autentykacja jest usługą pośrednią między listą zadań, a użytkownikami. Nie posiada ona otwartego portu[3], aby była ona widoczna zewnątrz kontenera co sprawia, że kontakt z nią jest jedynie możliwy wewnątrz wspólnego klastra kontenerów, stanowi to dodatkową warstwę zabezpieczeń.

Inną istotną sprawą jest *secret key*[12], czyli ciąg znaków, który powinien być trzymany w tajemnicy i jest to ważny element tej usługi. Do generowania *tokenów*

[3]Wewnątrz kontenerów *Dockera* można wskazać adres z którego aplikacja będzie widoczna dla klienta[15].

wykorzystuje ona mechanizm *JWT*[4]. Do szyfrowania informacji i ich odszyfrowania korzysta się właśnie z *secret key*, to on stanowi gwarancję tego, że podane dane będą odszyfrowane i zaszyfrowane pomyślnie. Wyciek wspomnianego ciągu znaków stanowi poważne zagrożenie dla poufności danych całego serwisu i w tym przypadku powinien on być natychmiastowo zastąpiony innym. Dla bezpieczeństwa takie klucze powinny być bardzo długą sekwencją losowych znaków[12], których nie powinno się dać zapamietać[5]. Aplikacja wykorzystuje bibliotekę *PyJWT*[6] do tworzenia tokenów *JWT*.

Widok odpowiedzialny za szyfrowanie kluczy ma adres `/token_encode`, wewnątrz niego sprawdzane jest, czy otrzymał on żądanie z identyfikatorem użytkownika (dzięki temu, że usługa dostępna jest wewnątrz kontenera, to niemożliwe jest przesłanie fałszywego identyfikatora z zewnątrz), gdy metoda potwierdzi, że przyjęty format danych jest prawidłowy, to tworzony jest obiekt *payload*. Zawiera on pola *exp* z datą wygaśnięcia klucza, *iat*, pole posiadające datę jego utworzenia i *user_id*, identyfikator użytkownika[17]. Wszystkie te dane zostaną zaszyfrowane przy pomocy metody *jwt.encode*, gdzie poza *payload* przekazany jest również *secret key* i rodzaj algorytmu wykorzystanego do zabezpieczenia danych. W tym przypadku jest to *HS256*[7]. Generuje on ciąg bajtów, których nie da się przesłać przez protokół *HTTP*, dlatego przed wysłaniem powinno się na nim wywołać funkcję *decode*.

Do odszyfrowania *tokenu* wykorzystywany jest widok `/token-decode`, otrzymuje on zaszyfrowany klucz w obiekcie *JSON*, następnie sprawdza, czy obiekt znajduje się w bazie na czarnej liście (*ang. blacklist*), jeśli nie, to jest odszyfrowywany przy pomocy metody *jwt.decode*. Cała procedura zamknięta jest w bloku *try...except*, służy on do wyłapywania błędnych *wyjątków*, a następnie w kolejnym bloku są one odpowiednio obsługiwane[20]. Aplikacja oczekuje błędu *ExpiredSignatureError* i *InvalidTokenError*, gdy jeden z nich wystąpi, to odsyłane są odpowiednie komunikaty o błędzie, jeżeli nie, to odszyfrowane wiadomości z tokenu są wysyłane z powrotem.

Aplikacja posiada prosty model służący do przetrzymywania *tokenów*, tak aby w razie wylogowania użytkownika zachować ten klucz w bazie i w razie potrzeby sprawdzić, czy nie został już wykorzystany. Klasa ta posiada takie pola jak *id*, *token*, który jest ciągiem znaków o długości *500* i *blacklisted_on*, które zachowuje informacje o dacie, kiedy klucz został dodany do bazy danych. Wówczas klient wylogowując się przesyła obecny *token* pod adres `/token-blacklist`, gdzie następnie jest deszyfrowany, aby sprawdzić, czy dane w nim są prawidłowe i jeżeli są, to wtedy token zostaje zapisany, a usługa zarządzająca użytkownikami dostaje informacje o pomyślnym zakończeniu procesu.

[4]skrót od *JSON Web Tokens*, jest to metoda uwierzytelniania wykorzystująca obiekty *JSON* do bezpiecznej transmisji danych, więcej informacji o *JWT* można znaleźć na stronie: `https://jwt.io/introduction/`.

[5]Więcej informacji o generowaniu odpowiednich *secret key* można znaleźć w dokumentacji *Flaska*[12].

[6]Link do dokumentacji *PyJWT*: `https://pyjwt.readthedocs.io/en/latest/`.

[7]Więcej o algorytmach wykorzystywanych do szyfrowania *tokenów* pod adresem: `https://jwt.io/`.

5.4. Interfejs użytkownika

W przypadku interfejsu użytkownika, będą to statyczne pliki serwowane przez *Nginx*, ale w celu rozwijania takich aplikacji powstały zestawy narzędzi, które to ułatwiają. Istnieją pakiety pozwalające na wygenerowanie skonfigurowanych projektów. W przypadku platformy *Nuxt.js*, czyli biblioteki do statycznego renderowania stron w *Vue.js* jest to *nuxt-create-app*[18]. Z pomocą tej aplikacji została stworzona stosowna struktura projektu i wybrany zestaw programów wykorzystywanych w ramach niego, takich jak odpowiedni preprocesor do plików *CSS*, biblioteka *ElementUI*. W celu zapewnienia dodatkowych opcji doinstalowano przy pomocy narzędzia *Yarn*[8] takie moduły jak *@nuxt/http* i *@nuxtjs/proxy*. Nazwy tych narzędzi należało podać w pliku *nuxt.config.js* w sekcji *modules*. Moduł *Proxy* pozwala na skonfigurowanie w tym samym pliku, gdzie jest informacja o jego wykorzystaniu, poprzez dodanie obiektu *proxy*, prostego przekierowania adresów, wówczas gdy zostanie odpytany serwer developerski o *URL* podany w konfiguracji, to automatycznie przekieruje on żądanie do jednej z aplikacji *Flaska*.

```
1  /*
2  ** Nuxt.js modules
3  */
4  modules: [
5    '@nuxt/http',
6    '@nuxtjs/proxy',
7  ],
8  proxy: {
9    '/tasks-api': {
10     target: 'http://tasks:5000/',
11     pathRewrite: {
12       '^/tasks-api': '/'
13     },
14   },
15   '/users-api': {
16     target: 'http://users:5000/',
17     pathRewrite: {
18       '^/tasks-users' : '/'
19     }
20   },
21 },
```

Listing 5.8. Fragment pliku *nuxt.config.js* z konfiguracją obiektu *proxy*.

Nuxt.js wykorzystuje mechanizm renderowania plików po stronie serwera (ang. *Server Side Rendering*), co w rezultacie odpowiada za to, że każda ścieżka w aplikacji będzie oddzielnym plikiem *HTML*[18]. Pliki odpowiedzialne za generowanie takich ścieżek są w projekcie umiejscowione w katalogu `pages/`. W przypadku list zadań potrzebne jest jednak dynamiczne generowanie adresów. *Nuxt.js* również posiada taki mechanizm[18]. Wówczas należy stworzyć wewnątrz omówionego folderu kolejny o nazwie odpowiadającej pożądanej ścieżce i wewnątrz umieścić plik *index.html*, to on będzie renderowany, gdy użytkownik poda w pasku przeglądarki `/tasks/` (dla listy zdań). Gdy będzie potrzebował celu o określony identyfikatorze, wtedy należy utworzyć wewnątrz tego folderu plik o nazwie *_id.vue*. Taką strukturę w nazwie aplikacja *Nuxt.js* identyfikuje jako szablon, który należy wczytać przy podaniu ad-

[8]Link do dokumentacji narzędzia *Yarn*: `https://yarnpkg.com`.

resu `/tasks/1`. Argument przekazany w adresie *URL* framework rzutuje na zmienną o nazwie *id* we wczytanym pliku *HTML*[18].

Do zarządzania ścieżkami, aby nie były one dostępne dla niezalogowanego użytkownika wykorzystano mechanizm *middleware*[18]. Jest to zestaw funkcji, które można bezpośrednio wywołać przed przejściem użytkownika na inny adres. Wewnątrz nich sprawdzane będzie, czy klient ma zapisany w globalnej zmiennej przeglądarki *localStorage*, która przechowuje informacje z danej witryny aż do zamknięcia sesji przeglądarki[14], *token*, jeżeli tak to korzystając modułu z *@nuxt/http* do argumentu obiektu globalnego aplikacji *Nuxt.js*, *$http* jest on dołączany przy pomocy metody *setToken*.

Po dodaniu do nagłówka sekcji uwierzytelnienia możliwe jest pobieranie danych z zabezpieczonych adresów aplikacji serwerowej. W przypadku braku możliwości pozyskania tokenów lub, gdy serwer zwróci błąd przy jego odszyfrowaniu, to przy wykorzystaniu argumentu funkcji *middleware*, *context* i metody *redirect* należy użytkownika przekierować do strony logowania, tak aby zalogował się on jeszcze raz i pobrał nowy *token*.

Biblioteka *@nuxt/http* pozwala w prosty sposób na pobieranie i wysyłanie żądań do aplikacji serwerowych, korzystając z funkcji *fetch* wewnątrz komponentu[18]. Posiada ona również zestaw metod, dzięki którym można sprawdzić stan zapytania i w zależności od jego wyniku wysłać odpowiednie komunikaty.

```
1  <template>
2    <div>
3      <p v-if="$fetchState.pending">Fetching posts...</p>
4      <p v-else-if="$fetchState.error">
5        Error while fetching posts: {{ $fetchState.error.message }}
6      </p>
7      <ul v-else>
8         <li v-for="task of tasks" :key="task.id">
9          <n-link :to="`/tasks/${task.id}`">
10           {{ task.header }} {{ task.body }} {{ task.isDone }} </n-
     link>
11        </li>
12     </ul>
13   </div>
14 </template>
15 <script>
16 //...
17 export default {
18   data () {
19     return {
20       tasks: []
21     }
22   },
23   async fetch () {
24     const request = await this.$http.$get('tasks-api/tasks')
25     if (request.success) {
26       this.tasks = request.tasks
27     }
28   }
29 }
30 </script>
```

Listing 5.9. Przykład komponentu wczytującego listę zadań.

Wykorzystanie prostego serwera *proxy* sprawiło, że nie jest potrzebne podawanie całego adresu, a wystarczy jedynie, aby aplikacja wysłała żądanie pod odpowiedni *URL* w swojej domenie. Mechanizm odpowiedzialny za przekierowywanie wyśle to zapytanie do jednej z aplikacji serwerowych.

Rozdział 6

Konteneryzacja

Konteneryzacja aplikacji przy pomocy platformy *Docker* zapewnia działanie systemu niezależnie od sprzętu programisty[1], ale również może być wykorzystana do stworzenia środowiska produkcyjnego[1] i integracji z wybraną infrastrukturą serwerową. Serwisy takie jak *Amazon Web Services* posiadają gotowe rozwiązania do publikacji aplikacji wewnątrz kontenerów[22].

W tym rozdziale przedstawione zostanie przygotowanie konfiguracji środowiska rozwojowego (deweloperskiego) i produkcyjnego przy wykorzystaniu narzędzi *Docker*, *Docker Compose*, aplikacji *Gunicorn* i *Nginx*[23].

Fundamentem działania platformy są kontenery, czyli procesy systemu *Linux* z dodatkowymi funkcjami pozwalającymi na ich izolację od maszyny użytkownika. Jednym z aspektów, który pozwala na odseparowanie każdego z kontenerów jest to, że posiadają one własny system plików dostarczany przez **obrazy Dockera** (ang. *docker images*), które mają w sobie wszystkie potrzebne zależności, pliki binarne i kody źródłowe[15]. Ich ważną właściwością jest to, że łatwo możne je przenosić, dzięki czemu twórcy platformy *Docker* stworzyli serwis *Docker Hub* w którym są one przechowywane[15]. Każdy użytkownik może stworzyć własny obraz, a następnie go udostępnić innym, tak, aby bez problemu można było go wykorzystać w tworzonej aplikacji. W efekcie każdy język programowania, wiele bibliotek i narzędzi posiada swoje odpowiedniki w formie obrazów *Dockera*. Nie jest potrzebne instalowanie interpretera języka *Python*, wystarczy, że zastanie pobrany odpowiedni obraz, z którego będzie można korzystać jak z normalnie zainstalowanej wersji na fizycznej maszynie.

```
1  $user@home: docker run  python:3.8.3-buster
2  Unable to find image 'python:3.8.3-buster' locally
3  3.8.3-buster: Pulling from library/python
4  Status: Downloaded newer image for python:3.8.3-buster
5  $user@home: docker image ls
6  python              3.8.3-buster        659f826fabf4        3 weeks
       ago         934MB
7  $user@home: docker run --name python -it python:3.8.3-buster
8  Python 3.8.3 (default, May 16 2020, 07:08:28)
9  [GCC 8.3.0] on linux
10 Type "help", "copyright", "credits" or "license" for more
       information.
11 >>>
```

Listing 6.1. Proces pobrania i wykorzystanie przykładowego obrazu *Dockera*.

Każdy obraz, przechowywany na platformie *Docker Hub*, poza nazwą posiada jeszcze, po dwukropku, numer wersji, a także, po myślniku, nazwę systemu, który został wykorzystany przy konteneryzacji. W przypadku listingu 6.1 jest to dystrybucji systemu *Debian* o nazwie *Buster*[1].

Aplikacja *Docker* może automatycznie tworzyć obrazy odczytując instrukcję z pliku *Dockerfile*. Jest to dokument tekstowy wykorzystujący specjalną składnie, dzięki której można zarządzać systemem znajdującym się wewnątrz kontenera, tak aby odtworzył on wymagane środowisko. W projekcie serwisu monolitycznego znajdują się dwa takie pliki pozwalające na zbudowanie obrazu, który uruchomi aplikację opartą o bibliotekę *Flask*. Pierwszy odpowiada za przygotowanie środowiska deweloperskiego, drugi produkcyjnego. *Flask* posiada wbudowany serwer (*flask run*), ale nie jest on przystosowany do bycia efektywnym, stabilnym i bezpiecznym[12]. Wiele bibliotek ma podobne problemy i także oferuje inne narzędzia do obsługi zapytań w konfiguracji produkcyjnej[10]. Aplikacje uruchomione przy wykorzystaniu narzędzi dostarczonych przez platformę są przystosowane do rozwijania, przekazywania błędów programiście i prostego debugowania, ale nie są one zalecane do wykorzystania przy wdrożeniu, dlatego potrzebne są przynajmniej dwa pliki *Dockerfile*.

Pierwszy z nich odpowiedzialny jest za środowisko deweloperskie. Najpierw pobiera on obraz `python:3.8-alpine`, a następnie ustanawia obecny katalog jako `/code`. Następnie kopiuje plik *requirements.txt*, umieszczone są w nim wszystkie wymagane biblioteki wraz z ich wersjami. Kolejną czynnością jest zainstalowanie wymaganych przez system *Linux* paczek, tak aby móc później zbudować potrzebne biblioteki języka *Python*. Na końcu kopiowana jest zawartość całego folderu w którym znajduje się plik *Dockerfile* do systemu plików kontenera i wcześniej ustawionego folderu *WORKDIR*.

```
1 FROM python:3.8-alpine
2
3 WORKDIR /code
4
5 COPY requirements.txt .
6
7 RUN \
8  apk add --no-cache postgresql-libs && \
9  apk add --no-cache --virtual .build-deps gcc musl-dev postgresql-
     dev && \
10  python3 -m pip install -r requirements.txt --no-cache-dir && \
11  apk --purge del .build-deps
12
13 COPY . .
```

Listing 6.2. Plik *Dockerfile* wykorzystany do zbudowania obrazu aplikacji monolitycznej.

Dockerfile z listingu 6.2 działa dla wszystkich aplikacji napisanych w *Flasku*, czyli również tych stworzonych na potrzeby rozwiązań mikrousługowych. Należało utworzyć je osobno dla każdej z usług.

Uruchamianie osobno kilku kontenerów, które z sobą współpracują może być uciążliwe, a dodatkowo wymaga wpisywania wszystkich parametrów w linii poleceń, dlatego twórcy platformy *Docker* przygotowali narzędzie *Docker Compose*. Pozwala

[1]Informację o wersjach systemu operacyjnego *Debian* można znaleźć pod adresem: `https://www.debian.org/releases/`.

ono na definiowanie i uruchamianie kilku kontenerowych aplikacji, którymi można zarządzać poprzez odpowiednią konfigurację plików o formacie *YAML*, a następnie przy pomocy jednej komendy włączać wszystkie serwisy[15]. Będą one wówczas działać w odizolowanym środowisku, wewnątrz którego kontenery mogą się między sobą porozumiewać[15].

Atutem takiego rozwiązania jest uproszczona integracja usługi bazodanowej z resztą aplikacji. Wystarczy stworzyć prosty serwis i podać odpowiedni obraz, a następnie udostępnić port, który zostanie wykorzystany do łączenia się między kontenerami. Niestety nie jest możliwe ustalenie, która usługa powinna uruchomić się najpierw, co w rezultacie sprawia, że baza danych może być niedostępna przez pierwsze kilka sekund po starcie serwisu webowego. Nie jest to wadą tego rozwiązania, ale warto o tym pamiętać. Istniej parę skryptów dostępnych w serwisie *Github*, które sprawiają, że można ten błąd naprawić, jeden z nich został wykorzystany przy implementacji środowiska produkcyjnego[2].

```
1  version: '3'
2  services:
3    db:
4      image: postgres:12
5      env_file: .env
6      volumes:
7        - .pgdata:/var/lib/postgresql/data
8
9    web:
10     env_file: .env
11     build: .
12     depends_on:
13       - db
14     command: python -m flask run --host 0.0.0.0
15     ports:
16       - "5000:5000"
17     volumes:
18       - ./mono:/code/mono
19       - ./bin:/code/bin
20       - ./.env:/code/.env
21       - ./main.py:/code/main.py
22       - ./config.py:/code/config.py
23       - ./migrations:/code/migrations
```

Listing 6.3. Plik *docker-compose.yml* wykorzystany do uruchomienia aplikacji monolitycznej.

Każda konfiguracja *Docker Compose* wymaga podania numeru wersji, tak aby program mógł odpowiednio rozpoznać strukturę pliku. Następnie wyróżniona jest specyfikacja dla serwisów (można też skonfigurować sieci lub wolumeny[15]), ich nazwa, a później poszczególne właściwości, takie jak pliki konfiguracyjne z których mają korzystać, miejsce gdzie znajduje się *Dockerfile* (klucz *build*), komenda uruchamiająca aplikację, porty, które mają zostać udostępnione. Na końcu mapowany jest system plików, tak aby byłby one dostępne wewnątrz kontenera i narzędzie mogło z nich skorzystać w celu uruchomienia aplikacji.

Podany `env_file`, to plik tekstowy przechowujący zmienne systemu *Linux*[15],

[2]Wspomniany skrypt został udostępniony w artykule [23] na podstawie, którego została przygotowana konfiguracja produkcyjna.

takie jak *secret key* aplikacji, dane o nazwie, użytkowniku i haśle do bazy danych, dlatego nie powinien on być dostępny publicznie, w szczególności zapisywany w repozytorium. Zwyczajem jest tworzenie szablonu takiego pliku, tak aby programista mógł go skopiować, a następnie tylko wypełnić odpowiednimi informacjami.

W przypadku aplikacji opartej o mikroserwisy, plik z listingu 6.3 posiada zamiast jednego serwisu *web* trzy: *users*, *tasks*, *tokens* o bardzo podobnej konfiguracji, z tą różnicą, że wartości portów muszą być dla nich inne, tak, aby dwie aplikacje nie działały na jednym. Usługa *tokens* w ogóle nie udostępnia portu[3].

Innym istotnym faktem, jest to, że aplikacje napisane w bibliotece *Flask*, aby były dostępne zewnątrz kontenera muszą być uruchomione na hoście `0.0.0.0`, jest to konwencja pozwalająca na przekazanie systemowi operacyjnemu, aby nasłuchiwał żądań serwera dla wszystkich publicznych adresów *IP*[12]. Sprawia to, że można się z aplikacją połączyć w ramach fizycznej maszyny, na której działa, wykorzystując na przykład przeglądarkę i wpisując w niej adres `http://localhost:5000`.

Do uruchomienia aplikacji z wykorzystaniem narzędzia *Docker Compose* należy w folderze, gdzie znajduje się plik z konfiguracją wpisać w terminalu `docker-compose up`, wówczas wszystkie kontenery najpierw się zbudują, a następnie uruchomią. Proces budowy można wymusić przy pomocy komendy `docker-compose build`. W celu wyłączenia kontenerów można spróbować przerwać proces wciskając kombinację klawiszy *control+c*, gdy to się nie powiedzie, to należy wpisać komendę `docker-compose down` i poczekać na informację o zakończeniu ich działania[15].

W aplikacji opartej o mikrousługi jest jeszcze konfiguracja serwisu odpowiedzialnego za dostarczenie interfejsu użytkownika. Wykorzystuje ona obraz `node:14.1.0-strech` w którym zainstalowane jest narzędzie *Yarn* (jego opis znajduje się w rozdziale 5.4), które instaluje wszystkie wymagane moduły w folderze `/tmp/node_modules`, a następnie kopiuje je do projektu w systemie plików kontenera. W pliku *docker-compose.yml* serwis nazywa się *frontend* i jest uruchamiany komendą `yarn dev`. Ma on ustawione przekierowanie na porty *8000:8000* i również w tym przypadku należy ustawić zmienną host na `0.0.0.0`, aby kontener był dostępny z zewnątrz. Zmiana tej właściwości nie jest możliwa poprzez konfiguracje *docker-compose.yml* ale dodanie odpowiedniej opcji w pliku *nuxt.config.js*[18].

Konfiguracja produkcyjna przechowywana jest w oddzielnych plikach *Dockerfile.prod* i *docker-compose.prod.yml*. W przypadku aplikacji opartych o bibliotekę *Flask*. Proces budowy obrazu posiada dwie fazy[23]. Pierwsza ma za zadanie zbudowanie jej do pakietu *wheel* przy pomocy komendy `pip wheel`. W drugiej natomiast pakiet ten jest instalowany, a następnie serwowany przy pomocy aplikacji *Gunicorn*, która odpowiada za obsługę żądań. W przypadku plików statycznych są one dzielone między serwisy *web*, a *nginx* przy wykorzystaniu wspólnych *wolumenów*[15]. Tworzone są one z użyciem *Dockerfile*'a w katalogu `./nginx`, wewnątrz którego usuwany jest domyślny plik z ustawieniami (*default.conf*), a kopiowany jest ten z dysku maszyny fizycznej(*nginx.conf*). W nim również znajduje się konfiguracja *load balancera*[16]. Ustawiony jest serwer nadrzędny (ang. *upstream*), który wykorzystuje aplikację *Gunicorn*, a następnie w sekcji serwer podawana jest jego lokalizacja jako *proxy_pass*, dzięki czemu każde zapytanie jest odpowiednio do niego przydzielane. Ostatnią czynnością jest podanie ścieżki do plików statycznych, tak aby były one dostępne dla ser-

[3]Zgodnie z założeniami usługi do uwierzytelniania (rozdz. 5.3) nie może być ona dostępna zewnątrz klastra kontenerów.

wisu pod adresem `/static/`. W przypadku aplikacji opartej o mikrousługi podane są dwa serwery nadrzędne, a lokalizacja \ odpowiada ścieżce do zbudowanych[4] plików projektu *Nuxt.js*[18]. W tym miejscu zamiast serwera *proxy* wykorzystana jest opcja *try_files*[16]. Po zakończeniu konfiguracji wszystkie włączone usługi powinny być dostępne pod jednym adresem `http://localhost:80/`. Podczas uruchamiania narzędzia *Docker Compose* należy pamiętać o wpisaniu polecanie uwzględniającego konfigurację produkcyjną: `docker-compose -f docker-compose.prod.yml up`.

W przypadku integracji z dowolna platformą dostawcy usług internetowych wystarczy, że będzie ona posiadać zainstalowaną usługę *Docker* (lub będzie można ją doinstalować), następnie wystarczy zmodyfikować konfigurację serwera, tak, aby przekierowała ona wszystkie połączenia do serwera nadrzędnego, który będzie ustawiony pod adresem `http://localhost:80/`. Dla platformy *AWS* jest to odpowiednie nadpisanie zmiennych globalnych narzędzia *Amazon Elastic Container Service*[22]. Natomiast w przypadku prywatnych wirtualnych serwerów *VPS* należy zainstalować na nich usługę *Nginx* i w podobny sposób jak przy konfiguracji kontenera dodać serwer nadrzędny i wykorzystać opcję *proxy_pass* do przekierowania żądań do aplikacji w kontenerze[5].

[4] W celu zbudowania projektu wykorzystano komendę *yarn generate*, która tworzy aplikację jako strukturę statycznych plików *HTML* w katalogu *dist*.

[5] Link do artykuły wyjaśniającego konfigurację na platformie *DigitalOcean*: `https://www.digitalocean.com/community/tutorials/docker-explained-how-to-containerize-and-use-nginx-as-a-proxy` (autor: O.S Tezer, data publikacji: 16.12.2013).

Rozdział 7

Porównanie i analiza aplikacji

Poprzednie rozdziały przedstawiły proces tworzenia i wdrażania aplikacji monolitycznej, a także jej odpowiednika opartego o mikrousługi. Po opracowaniu środowiska produkcyjnego możliwe jest przeprowadzenie badań mających na celu porównanie obu tych systemów. Istotne w nich będą takie parametry jak wydajność, skalowalność i łatwość implementacji.

7.1. Cel i zakres badań

Celem badań jest przeprowadzenie porównania wydajności obu serwisów w środowiskach produkcyjnych. Przebadane zostanie to jak podział aplikacji na mniejsze usługi wpływa na ogólną sprawność serwisu w porównaniu do jednej scentralizowanej aplikacji. Następnie sprawdzone zostanie, jak zmieniają się osiągi aplikacji przy ich skalowaniu, czyli zwiększaniu ilości zasobów sprzętowych i liczby kontenerów, które mogą obsługiwać żądania. Analizie zostanie poddana implementacja poszczególnych serwisów, tak, aby porównać otrzymane wyniki do zastosowanych technologii, skali projektu, trudności w przygotowaniu architektury i zastosowania danych rozwiązań technologicznych. Celem badań jest wskazanie podobieństw i właściwości poszczególnych podejść. Znalezienie ich atutów i wad, a następnie wskazanie przypadków, gdzie wykorzystanie danej architektury byłoby korzystniejsze.

Do testów wydajności wykorzystane będzie narzędzie *Boom*. Jest to biblioteka napisana w języku Python przez autora książki *Rozwijanie mikrousług w Pythonie*[1], której zadaniem jest przeprowadzenie testów obciążeniowych serwera. Przesyła ona do niego zadaną ilość żądań symulując ruch w danym serwisie, a następnie dla każdego z nich mierzy czas. Wyniki tych operacji uśrednia podając całkowity czas przeprowadzonego testu, najszybciej i najdłużej wykonane żądanie, a także ich średni czas, amplitudę, ilość żądań na sekundę.

```
$user@home: boom http://localhost:80 -c 10 -n 100
Server Software: nginx/1.17.10
Running GET http://127.0.0.1:80
  Host: localhost
Running 100 queries - concurrency 10
[=================================================================>]
    100% Done

-------- Results --------
Successful calls    100
Total time            0.8767 s
Average               0.0839 s
Fastest               0.0317 s
Slowest               0.1165 s
Amplitude             0.0848 s
Standard deviation    0.016234
RPS                   114
BSI                 Pretty good

-------- Status codes --------
Code 200              100 times.

-------- Legend --------
RPS: Request Per Second
BSI: Boom Speed Index
```

Listing 7.1. Przykładowy test przy wykorzystaniu narzędzia *Boom*.

Natomiast do przeprowadzenia analizy skalowalności również zostanie wykorzystane narzędzie *Boom*, aby porównać jak zwiększenie ilości serwisów wpływa na wydajność aplikacji. Samo skalowanie aplikacji może zostać przeprowadzone przy wykorzystaniu platformy *Docker*, posiada ona funkcję zwiększenia ilości zasobów takich jak ilość rdzeni, pamięci *RAM*. Proces zmiany tych wartości nazywany jest *skalowaniem w pionie*[3].

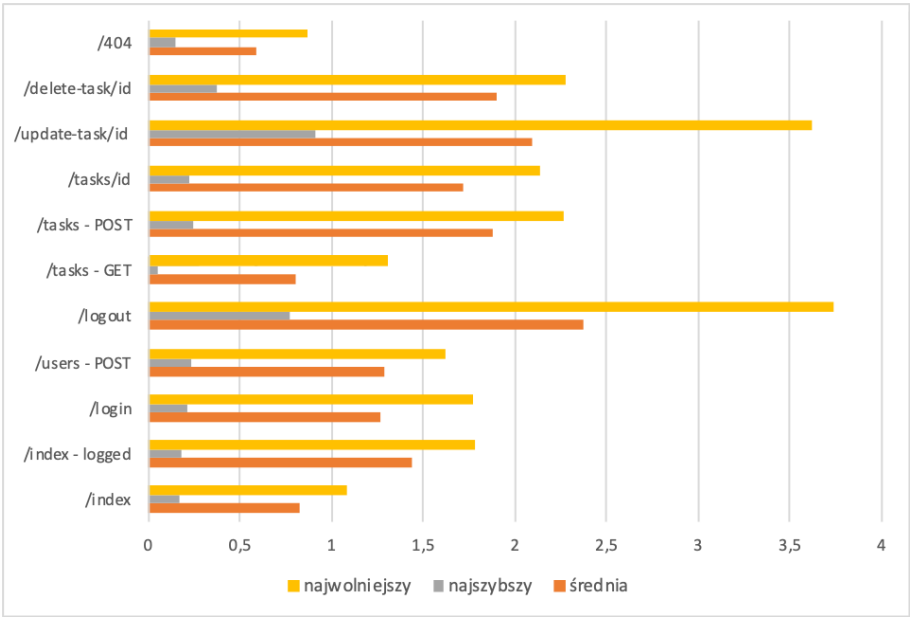
Do *skalowania w poziomie*[3] zostanie wykorzystana możliwość zwiększenia ilości kontenerów dla danego serwisu w ramach narzędzia *Docker Compose*. Przy pomocy polecenia `docker-compose up --scale nazwa_serwisu=ilosc_kontenerow`, można przypisać ile kontenerów powinna mieć dana usługa i dzięki temu zwiększyć przepustowość żądań aplikacji[15].

W celach sprawdzenia łatwości implementacji danego rozwiązania przeprowadzone zostanie analiza poszczególnych fragmentów obu serwisów, ich rozbudowania, wymaganych zależności i ocena zastosowanych rozwiązań architektonicznych.

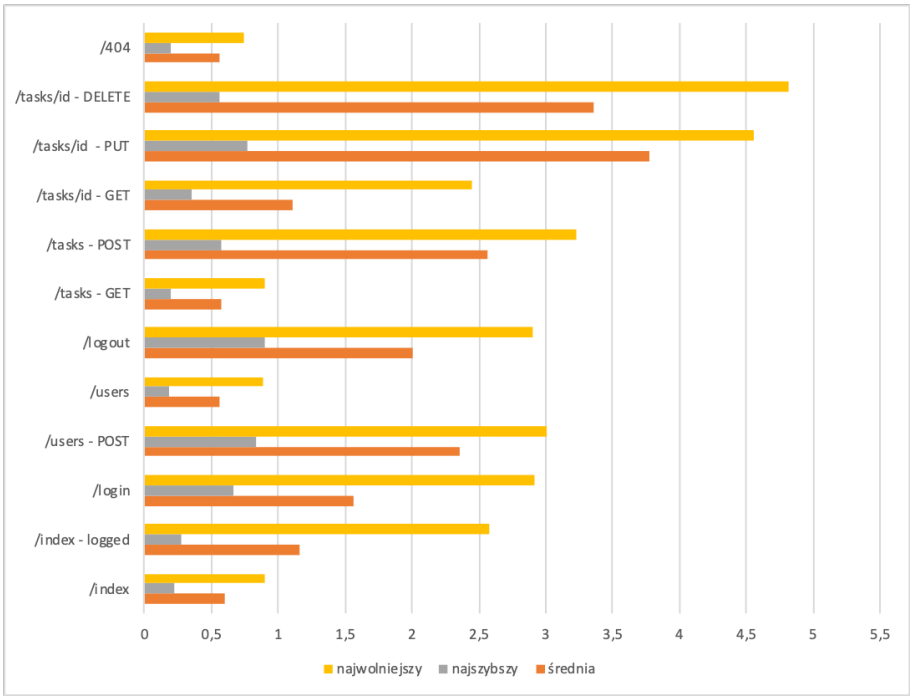
7.2. Wydajność

W każdym badaniu przeprowadzonym przy pomocy programu *Boom* wykorzystano takie same parametry, symulując ruch dla *100* użytkowników, którzy wysyłają maksymalnie *1000* żądań. Na początku przeprowadzono pomiar dla wszystkich dostępnych adresów w obu serwisach. Aplikacja monolityczna obsługuje informację o użytkowniku korzystając z zmiennych globalnych, które są dostępne w każdym z widoków, dlatego nie potrzebuje ona zwracać informacji o nim, w przeciwieństwie

do aplikacji opartej o mikrousługi, która na wykresie 7.3 będzie miała, z tego powodu, jeden adres więcej. Dla zapytania `\index` wyróżniono jeszcze przypadek, gdy użytkownik jest zalogowany (*logged*). Wyniki przedstawiono na poniżej wykresach:



Rys. 7.1. Czasy żądań (ms) dla każdego z adresu w aplikacji monolitycznej.



Rys. 7.2. Czasy żądania (ms) dla każdego z adresu w aplikacji mikrousługowej.

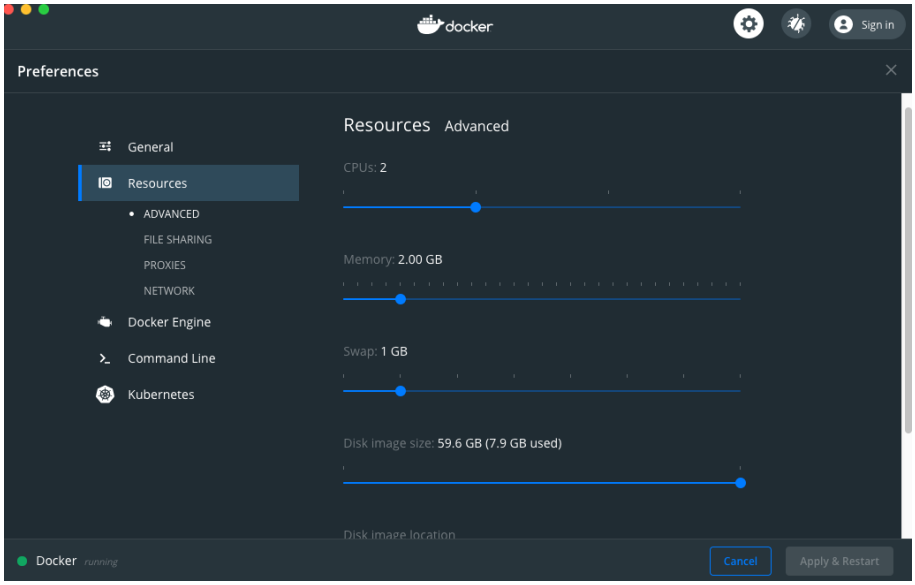
Z wykresów (7.3, 7.1) wynika, że dla prostszych informacji do zwrócenia takich jak te z adresów: *index*, strona *404*, obie aplikacje mają podobne wyniki i zabierają im one najmniej czasu. Więcej zajmuje natomiast zwrócenie stron przy użyciu zapytania *GET*, w tym przypadku widać, że nie ma reguły i średni czas dla nich

jest różny. Najdłużej w aplikacji monolitycznej wykonywane są żądania odpowiadające za wylogowanie użytkownika, a także aktualizację zadania. Z kolei w serwisie opartym o mikrousługi najdłuższe jest usunięcie, a także jak w przypadku pierwszej aplikacji zaktualizowanie zadania użytkownika. Wynikać to może z tego, że te żądania wymagają zweryfikowania wszystkich danych przesłanych w zapytaniu, a następnie nadpisania lub usunięcia danego obiektu z bazy danych.

Aplikacja oparta o mikrousługi ma ogólnie gorsze średnie czasy i dłuższe odpowiedzi, mimo, że dane zwracane przez serwer to proste obiekty *JSON*, a nie całe struktury *HTML* jak w serwisie monolitycznym. Widocznie, przy niewielkich rozmiarach serwisów rodzaj przesyłanych danych nie wpływa na szybkość działania usług, a może go mieć ich *serializacja* (mikrousługi) do formatu *JSON*.

7.3. Skalowalność

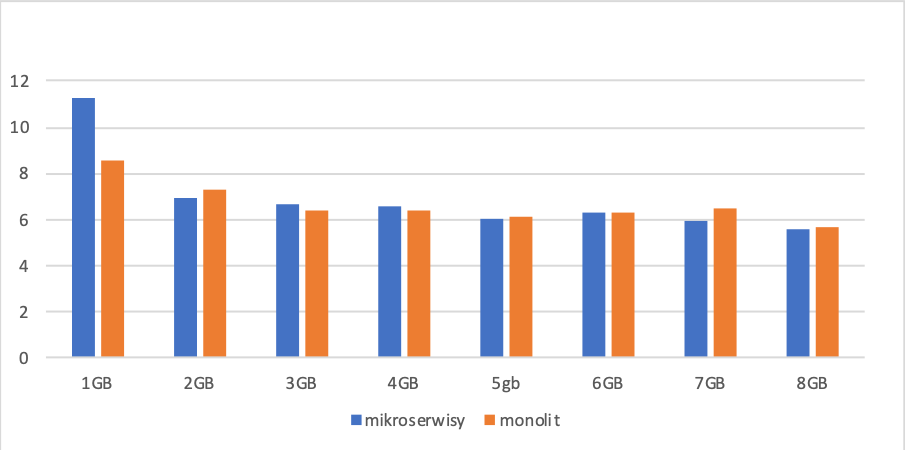
W panelu ustawień platformy *Docker* są funkcje odpowiedzialne za zarządzanie zasobami. Wśród nich możliwość zmiany ilości rdzeni procesora i *RAMu*, sterując tymi zależnościami zbadany będzie ich wpływ na szybkość zapytań.



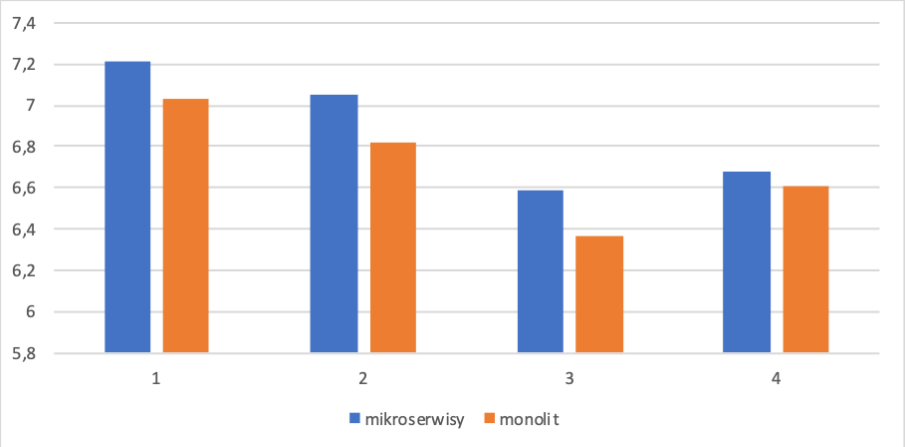
Rys. 7.3. Zakładka do zarządzania zasobami w ustawieniach platformy *Docker*.

Badania obejmą sprawdzenie zmiany szybkości odpowiedzi serwera w zależności od zasobów. Do ich przeprowadzania również wykorzystano narzędzie *Boom*.

Na wykresie 7.4 widać, że przy niewielkiej ilości pamięci *RAM* ma on wpływ na działanie aplikacji szczególnie, gdy jest go niewiele, wówczas czasy żądań się wydłużają. Prawdopodobnie jest to związane z tym, że serwisy wymagają minimalnej jego ilości do stabilnego działania, gdy ta zostanie osiągnięta to rola pamięci *RAM* w szybkości obsługiwanych zapytań jest już mniej istotna i czasy te pozostają na podobnym poziomie. Inną zależność można zauważyć w przypadku liczby rdzeni na wykresie 7.5, ich ilość nie wypływa na szybkość żądań, dopóki nie będzie ich więcej niż dwa. Każdy użytkownik to asynchroniczne zapytanie. W taki sposób program *Boom* symuluje ruch na serwerze[1]. Prawdopodobnie dwa rdzenie są wykorzystywane przez inne usługi i zwiększenie ich ilości powoduje, że aplikacja lepiej sobie



Rys. 7.4. Średnie czasy(ms) żądań w zależności od liczby posiadanej pamięci *RAM*.



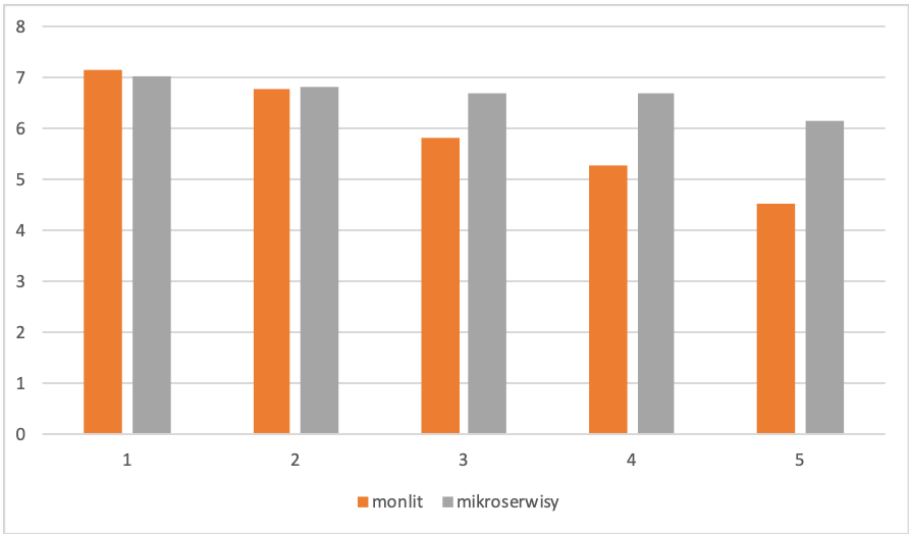
Rys. 7.5. Średnie czasy żądania(ms) w zależności od liczby rdzeni w procesorze (*CPU*).

radzi z obsługą zapytań, co może tłumaczyć spadek średniej czasów żądań przy trzech rdzeniach.

Następnym przeprowadzonym badaniem jest sprawdzenie skalowalności poziomej dla obu aplikacji, czyli tego jak zachowają się serwisy, gdy zwiększona będzie ilość ich instancji. *Load balancer* zaimplementowany w każdym z nich powinien kierować ruchem tak, aby nie obciążać tylko jednego z kontenerów, co powinno przyczynić się do szybszych czasów żądań[2].

W przypadku badania skalowalności na wykresie 7.6 widać tendencję spadkową, która jest zauważalnie większa przy aplikacji monolitycznej. Dla tego serwisu można również dostrzec, że przy liczbie kontenerów *1*, radził on sobie nieznaczenie gorzej od mikroserwisów. Następnie dla dwóch kontenerów oba serwisy miały zbliżone czasy. Dopiero przy kolejnych pomiarach scentralizowana usługa uzyskała przewagę. Posiada ona jeden serwis napisany w bibliotece *Flask*, co sprawia, że ich skalowanie mniej obciąża maszynę. W aplikacji opartej o mikrousługi potrzebne było stworzenie wielu instancji dla kilku usług, co w ostateczności mogło mocniej obciążyć jednostkę, na której testowano serwisy i zwiększyć czasy żądań aplikacji.

Przeglądając otrzymane wyniki można zauważyć, że serwis mikrousługowy w każdym z porównań wypada gorzej, mimo, że to jego architektura zorientowana jest



Rys. 7.6. Średnie czasy żądania(ms) w zależności od liczby powielonych kontenerów na jeden serwis.

na łatwą skalowalność[2], to monolit wykorzystujący hierarchię klas do komunikacji, ma lepszą wydajność. Prawdopodobnie ilość potrzebnych serwisów w aplikacji mikrousługowej jest niewspółmierna do obsługiwanych danych, rozmiarów projektu i w badanych warunkach zastosowane rozwiązania nie przynoszą żadnych dodatkowych korzyści w polepszeniu wydajności aplikacji, a nawet wpływają na nią negatywnie.

7.4. Analiza implementacji projektu

Wskaźniki wydajności są istotne, ponieważ to one mogą być podstawą do rozważenia przejścia na daną architekturę. Inną decydującą cechą jest łatwość implementacji. To jakie rozwiązania należy zastosować, ich konfiguracja i wykorzystywane technologie.

Aplikacja monolityczna głównie opiera się o bibliotekę *Flask* i proste szablony *HTML*, których konfiguracja jest podstawowa. Wszystkie dodatkowe opcje, takie jak formularze są dostarczane przy pomocą zewnętrznego modułu *Flask WTF*, a ich renderowanie odbywa się automatycznie przy pomocy metody *render_template*. Do zarządzania sesją użytkownika również wykorzystywano dodatkową bibliotekę, która sprawia, że konfiguracja jest automatyczna, a dodatkowo wprowadza ona możliwość sprawdzenia danych klienta nawet z szablonów, oferując globalny obiekt *current_user*. Nie jest potrzebne pobieranie danych z bazy, a wylogowywanie użytkownika sprowadza się wyłącznie do wykorzystania funkcji *logout_user()*. Struktura projektu monolitycznego jest logiczna, wszystkie elementy aplikacji, które nie współgrają bezpośrednio są od siebie odseparowane, a konfiguracja aplikacji opiera się na przekazaniu niewielkiej ilości zmiennych globalnych i podłączeniu z bazą danych. Nie musi on być uruchamiany wewnątrz kontenera, aby komunikować się z innymi komponentami, a projekt mógłby być bez problemu uruchomiony i skonfigurowany nie wykorzystując platformy *Docker*.

W przypadku aplikacji opartej na mikrousługach jej decentralizacja sprawia, że każdy z serwisów jest osobno konfigurowany. Muszą one korzystać z różnych bazy

danych, co tworzy wiele utrudnień takich jak potrzeba powielania plików z ustawieniami i wielokrotne instalowania tych samych zależności. Struktura pomniejszych usług przypomina małą aplikację monolityczną, co sprawia, że w każdej z nich należy zaimplementować takie same mechanizmy obsługi żądań i komunikacji z usługą bazodanową. Zarządzenie sesjami użytkownika opiera się o zewnętrzny mechanizm, z którym usługi muszą łączyć się przy pomocy protokołu *HTTP* w celu autoryzacji. Baza danych tej usługi musi przechowywać wszystkie stworzone *tokeny* i sprawdzać ich poprawność, co jest rozwiązaniem bardziej skomplikowanym niż zautomatyzowany proces uwierzytelniania z aplikacji monolitycznej. Dodatkowo należy dbać o integralność danych, sprawdzać w każdej z usług, nawet interfejsie użytkownika, czy żądanie posiada *token*, a następnie przesyłać go dalej w celu jego odszyfrowania.

Kilka baz danych to też problem z ich zarządzaniem, potrzebne jest wdrażanie trzech osobnych migracji i tworzenie tabel. Natomiast po stronie klienta, w przeglądarce internetowej potrzebne było wykorzystanie jej pamięci wewnętrznej (*localStorage*) do przechowywania informacji o użytkowniku i wygenerowanych tokenów.

Interfejs użytkownika nie jest zbiorem kilku szablonów w języku *HTML*, ale osobną aplikacją zbudowaną przy pomocy programu generującego, który tworzy projekt z wszystkimi strukturami i konfiguracjami. Do swobodnego korzystania z wszystkich bibliotek w nim zawartych i dowolnego ich konfigurowania, należy poznać wiele modułów frameworka *Nuxt.js*. Są to znaczne ilości dodatkowych warstw abstrakcji i ustawień w projekcie.

Poszczególne aplikacje mogą działać osobno, ale tylko razem są możliwe do użytkowania, a do tego potrzebne jest stworzenie wielu plików konfiguracyjnych, niejednokrotnie z powielonymi informacjami. Kontenery muszą wykorzystywać mechanizmy platformy *Docker* do porozumiewania się, inaczej trzeba by było tworzyć fizyczne struktury sieciowe, a moduł do uwierzytelniania musiałby być widoczny jednie dla pozostałych serwisów wewnątrz prywatnej sieci, aby mógł bezpiecznie pobierać i przekazywać dane.

Analizując oba podejścia widoczne jest, że aplikacja oparta o mikroserwisy jest o wiele trudniejsza do wdrożenia, czas jej stworzenia był dłuższy, wymagała ona skrupulatnego obmyślenia całej architektury, warstw komunikacji pomiędzy poszczególnymi elementami i ręcznego wdrożenia wielu mechanizmów takich jak serializacja danych, proces uwierzytelniania użytkowników i struktura zapytań serwera. Do jej szybkiej konfiguracji niezbędna jest platforma *Docker*. Natomiast aplikacja monolityczna wszystkie te mechanizmy ma zaimplementowane w ramach dodatkowych bibliotek frameworku *Flask*, a sama struktura plików jest przejrzysta i dobrze zorganizowana na warstwy logiczne, gdzie żadna z nich się nie powiela.

Rozdział 8

Dyskusja rezultatów i wnioski końcowe

W ramach pracy udało się przygotować architekturę dla dwóch aplikacji, jednej opartej o podejście monolityczne, drugiej o mikrousługi, a następnie je zaimplementować. Szczegółowo przedstawiono wszystkie założenia i problemy, które mogą się pojawić podczas tworzenia aplikacji, a także środowisko programistyczne potrzebne do ich uruchomienia. Opisano strukturę bazy danych, wykorzystane biblioteki, wymagania funkcyjne i niefunkcyjne aplikacji. Następnie omówiono proces tworzenia serwisów, zaprezentowano przykłady wykorzystanego kodu, tłumacząc jego działanie i zwracając uwagę na jego istotne kwestię. Przeprowadzono proces konteneryzacji projektu, tak aby można było go zintegrować z dowolną platformą oferującą hosting serwisów internetowych. Przygotowano i przeprowadzono badania mające sprawdzić różnicę pomiędzy wydajnością aplikacji, ich skalowalnością i to jak te czynniki wypływają na czasy odpowiedzi żądań. Platformy przeanalizowano pod kątem ich implementacji, tego jak trudne były do osiągnięcia zakładane architektury, jakie technologie były wykorzystane. Uzyskane rezultaty pozwalają na ocenę obu podejść. *Richard Rodger* w książce *Tao mikrousług. Projektowanie i wdrażanie*, napisał, że:

> Mikrousługi nie są cudownym panaceum, dzięki któremu możesz z dnia na dzień rozwiązać wszystkie swoje problemy związane z rozwojem oprogramowania[2].

Wyniki badań jasno przedstawiły, że powyższe stwierdzenie jest prawdą. Mikrousługi posiadają szereg atutów, takich jak to, że ich implementacja nie musi sprowadzać się do jednego projektu, w danym języku, a dzięki uniwersalnemu sposobowi komunikacji pomiędzy komponentami możliwości ich rozwoju jest wiele. Każda z poszczególnych usług ma swój zakres odpowiedzialności, który jest łatwy do przestrzegania, natomiast trudniejsze do osiągnięcia jest to, aby dane dwie mikrousługi powielały go. Istotne jest to, że każda z usług to osobny projekt, który nie musi być tworzony przez jeden zespół, tylko kilka równolegle. Co w zasadzie sprawia, że sama aplikacja posiada więcej możliwości, jeżeli chodzi o jej skalowanie, można tworzyć instancję dla poszczególnych usług, które są najbardziej obciążone i im przekazywać najwięcej zasobów, aby działały wydajniej. Niestety przy tak niewielkich aplikacjach jak te stworzone w ramach niniejszej pracy, rozwiazanie oparte o mikrousługi okazało się powodować, że podział architektury był zbyt skomplikowany. Stworzenie osobnych serwisów, które musiały się z sobą porozumiewać w celu uwierzytelniania tokenów,

a także w warstwie interfejsu użytkownika sprawiło, że liczba interakcji wewnątrz sieci niepotrzebnie wzrosła, dodatkowo ją obciążając. Innym niekorzystnym zjawiskiem było powielanie kodu i danych, implementowanie tych samych mechanizmów do obsługi bazy danych dla trzech aplikacji, wykorzystywanie identycznych plików konfiguracyjnych. Wnioski z tych obserwacji potwierdzają przeprowadzone badania, gdzie aplikacja monolityczna o wiele lepiej radzi sobie z wydajnością, a nawet skalowalnością. Mimo, że to jedna centralna aplikacja to i tako możliwe było stworzenie jej kilku instancji, których ruch był obsługiwany przez *load balancer*, dzięki czemu w tych testach wypadła lepiej od serwisu opartego o mikrousługi. Nie mniej jednak także ta architektura posiada wady, takie jak potrzeba utrzymywania jednego projektu, opartego o hierarchię klas, gdzie obsługa ich zakresów i uprawień spoczywa na programiście. Poszczególne komponenty nie są już tak odizolowane jak w aplikacji mikrousługowej, co może doprowadzić do zawiłości i mieszania się odpowiedzialności. To w rezultacie sprawia, że trudniej jest utrzymać porządek w kodzie. Gdy jedna mikrousługa będzie wymagała usprawnienia, to wystarczą zmiany wyłącznie w jej ramach, bez zmian w całym systemie, co jest niemożliwe w aplikacji monolitycznej. Także możliwości skalowania serwisu są ograniczone, można uruchomić kilka instancji, ale nie jest to tak elastyczne rozwiazanie jak w przypadku mikrousług. Analiza wykazała, że wady te nie są istotne przy niewielkich projektach, w przypadku aplikacji monolitycznych o wiele łatwiej jest rozpocząć projekt, wymaga on mniej konfiguracji. Centralna baza danych jest o wiele prostsza w utrzymaniu i porządkowaniu danych. Architektura centralna jest także, tą przedstawianą w dokumentacjach projektów[12] i jej odwzorowanie, a także zaprojektowanie jest o wiele prostsze. Dodatkowo istniej wiele zewnętrznych modułów pozwalających na znaczne uproszczenie poszczególnych mechanizmów, takich jak uwierzytelnianie, przesyłanie formularzy, czy obsługa danych.

Przeprowadzone badania wskazują na lepszą wydajność serwisu monolitycznego, ale należy zwrócić uwagę, że stworzona aplikacja jest niewielka, generowane przez nią szablony *HTML* były małej wielkości. Mikrousługi pomimo, że przesyłają jedynie obiekty *JSON*, to muszą je jeszcze serializować, co mogło obciążyć aplikację bez uwidaczniania zalety tego rozwiązania. Dodatkowo usługi uruchomione byłby wewnątrz kontenerów, które mimo tego, że mogą być stosowane w rozwiązaniach produkcyjnych, to wymagają więcej zasobów. Uruchamianie większej ilości kontenerów w przypadku aplikacji zdecentralizowanej również mogło wpłynąć na czasy ich odpowiedzi. Innym czynnikiem, który mógł wpłynąć na badania były ograniczenia sprzętowe fizycznej maszyny. Nie była to jednostka przygotowana z myślą wyłącznie o rozwiązaniach serwerowych i posiadła ona ograniczoną liczbę pamięci *RAM*, procesor starszej generacji i niewielką przestrzeń dyskową. Niemniej jednak przeprowadzone badania miały wyłącznie cel porównawczy, a ich wyniki świadczyć mogą, że zgodnie z tezą *Richarda Rodgera*, mikrousługi nie wszędzie sprawdzą się jako rozwiazanie najbardziej optymalne[2].

Ostatecznie można przejść do wniosku, że wybór pomiędzy architekturą monolityczną, a mikrousługami powinien być kierowany głównie ze względu na charakter projektu. Jeżeli ma on być tworzony w ramach większego zespołu znającego kilka technologii, rozsądne byłoby podzielenie go, tak aby każdy z nich pracował nad osobną usługą, wówczas przy większych serwisach, możliwe będzie, że wady rozwiązania opartego o mikrousługi będą mniej istotne. Większa organizacja będzie mogła zainwestować pieniądze w infrastrukturę sieciową, tak aby odpowiednio przystosować ją

do rozwiązań zdecentralizowanych.

Architektura monolityczna to nadal dobry pomysł na mniejsze aplikacje, gdzie zespół nie składa się z dużej liczy osób znających kilka technologii. Rozpoczęcie projektu jest wówczas o wiele prostsze, a wdrożenie takiej aplikacji szybsze. W razie, gdyby urósł on do większych rozmiarów istnieje opcja przepisania go na mniejsze aplikacje i zmiana architektury na tą opartą o mikrousługi.

Każdy przypadek aplikacji powinien być indywidualnie przemyślany. Zespół musi być świadomy wszystkich wad i zalet danych architektur, przewidzieć przyszłe koszty infrastruktury, a także zaplanować czas na niezbędne konfigurację wybranych rozwiązań. Realizacja projektu w ramach niniejszej pracy pokazała jak istotnym elementem aplikacji internetowych jest ich konfiguracja, odpowiednie zaplanowanie systemu komunikacji między danymi komponentami i opracowanie architektury tak, aby nie powielała ona niepotrzebnie poszczególnych modułów. W ramach usprawnienia obecnej implementacji projektu mikrousługowego można byłoby inaczej zaplanować odpowiedzialność poszczególnych serwisów, zrezygnować z osobnego modułu uwierzytelniania i wielu baz danych. Podzielić cały system jedynie na dwie usługi, jedną odpowiedzialną za przetwarzanie i przechowywanie wszystkich informacji, drugą za interfejs użytkownika. Byłby to idealny kompromis między architekturą monolityczną i mikrousługami.

Dodatek A

Tabele wyników wykorzystanych do stworzenia wykresów

W ramach przeprowadzonych badań przygotowano następujące tabele, które następnie wykorzystano do stworzenia wykresów opracowanych w głównej części pracy:

Tab. A.1. Czasy żądań (ms) dla każdego z adresów w aplikacji monolitycznej.

Adres	**średnia**	**najszybszy**	**najwolniejszy**
/index	0,8216	0,1702	1,0862
/index - logged	1,4356	0,1791	1,7834
/login	1,2667	0,2167	1,7762
/users - POST	1,2843	0,2329	1,6195
/logout	2,3735	0,7693	3,7333
/tasks - GET	0,8025	0,0479	0,2177
/tasks - POST	1,8744	0,2447	2,2621
/tasks/id.	1,7152	0,2233	2,1407
/update-task/id	2,0891	0,9121	3,6156
/delete-task/id	1,9022	0,3753	2,2752
/404	0,5871	0,1475	0,8700

Tab. A.2. Średnie czasy żądań (*ms*) w zależności od liczby posiadanej pamięci *RAM* (*GB*).

Serwis/ *RAM*	**1GB**	**2GB**	**3GB**	**4GB**	**GB**	**6GB**	**7GB**	**8GB**
mikrousługi	11,2469	6,8961	6,7101	6,5419	6,0812	6,3221	5,991	5,5612
monolit	8,6012	7,3012	6,4191	6,3814	6,1328	6,3112	6,5312	5,7131

Badania przeprowadzono na maszynie wyposażonej w procesor *2.7 GHz Intel Core i5* o pamięci RAM *8 GB 1867 MHz DDR3*. Do ich zrobienia wykorzystano narzędzie *Boom* w konfiguracji *100* użytkowników, którzy wysyłali maksymalnie *1000* żądań.

Tab. A.3. Czasy żądań (ms) dla każdego z adresów w aplikacji opartej o mikrousługi.

Adres	**średnia**	**najszybszy**	**najwolniejszy**
/index	0,5957	0,2217	0,9042
/index - logged	1,1632	0,2763	2,5775
/login	1,5587	0,6577	2,9107
/users - POST	2,3541	0,8331	3,0011
/users.	0,5594	0,1804	0,8861
/logout	2,0031	0,9011	2,8993
/tasks - GET	0,5754	0,1902	0,8965
/tasks - POST	2,5665	0,5664	3,2221
/tasks/id - GET	1,1123	0,3561	2,4412
/tasks/id - PUT	4,5511	0,7656	3,7732
/tasks/id - DELETE	4,8133	0,5589	3,5581
/404	0,5605	0,1984	0,74620

Tab. A.4. Średnie czasy żądań(ms) w zależności od liczby rdzeni w procesorze (*CPU*).

Serwis/ *CPU* (rdzenie)	**1**	**2**	**3**	**4**
mikrousługi	7,2156	7,0517	6,5919	6,6790
monolit	7,0357	6,8230	6,3658	6,6058

Tab. A.5. Średnie czasy żądań(ms) w zależności od liczby powielonych kontenerów na jeden serwis.

Serwis/ Liczba kontenerów	**1**	**2**	**3**	**4**	**5**
mikrousługi	7,0349	6,8001	6,6881	6,6766	6,1601
monlit	7,1432	6,7775	5,8308	5,2573	4,5396

Bibliografia

[1] Tarek Ziadé. *Rozwijanie mikrousług w Pythonie. Budowa, testowanie, instalacja i skalowanie*. Helion, 2018.

[2] Richard Rodger. *Tao Mikrousług. Projektowanie i wdrażanie*. Helion, 2019.

[3] Susan J. Fowler. *Mikrousługi. Wdrażanie i standaryzacja systemów w organizacji inżynierskiej*. Helion, 2019.

[4] Sam Newman. *Budowanie mikrousług*. Helion, 2019.

[5] Daniel Nations. What exactly is a web application? *Lifewire*, 2019. `https://www.lifewire.com/what-is-a-web-application-3486637`.

[6] John Colby Paul Wilton. *SQL. Od podstaw*. Helion, 2005.

[7] © Microsoft 2020. *Dokumentacja Microsoft Build*. `https://docs.microsoft.com/pl-pl/`.

[8] David Thomas Andrew Hunt. *Pragmatyczny programista. Od czeladnika do mistrza*. Helion, 2011.

[9] "Ruby on Rails" David Heinemeier Hansson. All rights reserved "Rails". *Dokumentacja Ruby on Rails*. `https://guides.rubyonrails.org`.

[10] © 2005-2020 Django Software Foundation and individual contributors. *Dokumentacja Django w wersji 3.0*. `https://docs.djangoproject.com/en/3.0/`.

[11] Copyright © 1996-2003 W3C®. *Oficjalna specyfikacja najnowszej wersji protokołu HTTP*. `https://www.w3.org/Protocols/`.

[12] © Copyright 2010 Pallets. *Dokumentacja Flaska w wersji 1.1.x*. `https://flask.palletsprojects.com/en/1.1.x/`.

[13] Copyright © 2014-2020 Evan You. *Dokumentacja Vue.JS w wersji 2.x*. `https://vuejs.org/v2/guide/`.

[14] © 2005-2020 Mozilla and individual contributors. *MDN web docs*. `https://developer.mozilla.org/en-US/`.

[15] Copyright © 2013-2020 Docker Inc. *Dokumentacja Dockera*. `https://docs.docker.com/`.

[16] Inc Copyright © F5. *Dokumentacja Nginxa*. `https://docs.nginx.com`.

[17] Real Python (Michael Herman). Token-based authentication with flask. *Real Python*, 2017. https://realpython.com/token-based-authentication-with-flask/.

[18] Brak informacji o autorze (wydane na podstawie licencji MIT). *Dokumentacja frameworku NuxtJS*. https://nuxtjs.org/guide.

[19] © Copyright 2010 Pallets. *Dokumentacja biblioteki Flask-SQLAlchemy w wersji 2.x*. https://flask-sqlalchemy.palletsprojects.com/en/2.x/.

[20] Python Software Foundation. © Copyright 2001-2020. *Dokumentacja języka Python 3.x*. https://docs.python.org/3/.

[21] Miguel Grinberg. The *Flask Mega-Tutorial*. *miguelgrinberg.com*, 2017. https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world.

[22] Inc. or its affiliates. © 2020, Amazon Web Services. *Dokumentacja Amazon Web Services*. https://docs.aws.amazon.com.

[23] Michael Herman. Dockerizing *Flask* with *Postgres*, *Gunicorn*, and *Nginx*. *TestDriven Labs*, 2020. https://testdriven.io/blog/dockerizing-flask-with-postgres-gunicorn-and-nginx/.