Praca Dyplomowa Inżynierska

Dawid Wijata
205006

Porównanie aplikacji frontendowych opartych na mikrofrontendach z tradycyjną architekturą monolityczną na przykładzie aplikacji do zarządzania finansami osobistymi

Comparison of Microfrontend Applications and Monolith Frontend Applications Based on the Example of Expense Tracker

Praca dyplomowa na kierunku:
Informatyka

Praca wykonana pod kierunkiem
dr inż. Piotra Wrzeciono
Instytut Informatyki Technicznej
Katedra Systemów Informacyjnych

Warszawa, rok 2023

SZKOŁA GŁÓWNA GOSPODARSTWA WIEJSKIEGO

Wydział Zastosowań Informatyki i Matematyki

Oświadczenie Promotora pracy

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem i stwierdzam, że spełnia ona warunki do przedstawienia tej pracy w postępowaniu o nadanie tytułu zawodowego.

Data . . . . . Podpis promotora . . . . . .

Oświadczenie autora pracy

Świadom/a odpowiedzialności prawnej, w tym odpowiedzialności karnej za złożenie fałszywego oświadczenia, oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami prawa, w szczególności z ustawą z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. 2019 poz. 1231 z późn. zm.)

Oświadczam, że przedstawiona praca nie była wcześniej podstawą żadnej procedury związanej z nadaniem dyplomu lub uzyskaniem tytułu zawodowego.

Oświadczam, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną. Przyjmuję do wiadomości, że praca dyplomowa poddana zostanie procedurze antyplagiatowej.

Data . . . . . Podpis autora pracy . . . . . .

Streszczenie

Porównanie aplikacji frontendowych opartych na mikrofrontendach z tradycyjną architekturą monolityczną na przykładzie aplikacji do zarządzania finansami osobistymi

Tematem niniejszej pracy była implementacja i poźniejsze porównanie frontendu dwóch wersji aplikacji do zarządzania finanasami osobistymi w dwóch architekturach - mikrofrontendowej i monolitycznej. Praca składa się z części teoretycznej wprowadzającej w pojęcie mikroserwisów i mikrofrontendów oraz części praktycznej porównującej obie implementacje pod względem metryk kodu, wydajności, kosztów obu rozwiązań oraz zarządzania projektem.

Słowa kluczowe – architektura rozproszona, mikroserwisy, mikrofrontendy, architektura oprogramowania

Summary

Comparison of Microfrontend Applications and Monolith Frontend Applications Based on the Example of Expense Tracker

The subject of this thesis was to implement and compare frontend parts of expense tracker web application written in two different architectural concepts - monolith and microfrontends. The first part is an introduction to microservice and microfrontend concepts. The second one consists of a comparison of both application versions by terms of code metrics, performance, financial costs of both solutions and project management.

Keywords – distributed architecture, microservices, microfrontends, software architecture

Spis treści

1 Wprowadzenie do mikroserwisów **9**

1.1 Architektura monolityczna 9

1.2 Architektura mikroserwisów 10

2 Mikrofrontendy **14**

2.1 Wstęp 14

2.2 Dodatkowe ograniczenia względem mikroserwisów 16

2.2.1 Ograniczone zasoby obliczeniowe 16

2.2.2 Zasoby współdzielone 17

2.2.3 Arkusze styli CSS 17

2.2.4 Komunikacja między mikrofrontendami 18

2.3 Sposoby implementacji technicznej mikrofrontendów 18

2.3.1 Konfiguracja serwera używająca routingu 19

2.3.2 Elementy <iframe> 20

2.3.3 Przechowywanie mikrofrontendów w skryptach JavaScript 21

2.3.4 Dynamiczne ładowanie modułów poprzez menedżer pakietów . . . 22

2.3.5 Single SPA 22

3 Funkcjonalność badanej aplikacji **24**

4 Opis backendu projektu **25**

4.1 Dobór technologii do projektu 25

4.2 Szablony projektów 26

4.3 Podział backendu na serwisy 26

4.3.1 Hosting plików - File Storage Service 27

4.3.2 Autoryzacja - Authorization Service 27

4.3.3 Zarządzanie użytkownikami - User Service 27

4.3.4 Zarządzanie rodzinami - Family Service 27

4.3.5 Logika domenowa - Transaction Service 27

5 Opis badanych frontendów **29**

5.1 Dobór technologii do projektów 29

5.2 Wersja monolityczna 29

5.3 Wersja mikroserwisowa 29

6 Porównanie projektów frontendowych **30**

6.1 Wersjonowanie kodu 30

6.2 Wydajność 30

6.3 Dostępność 30

6.4 Testowanie kodu 30

6.5 Zależności między modułami i projektami 30

6.6 Możliwości w zakresie zarządzania projektami 30

6.7 Skalowalność 30

6.8 Koszty ustawienia środowiska produkcyjnego 30

7 Podsumowanie **31**

8 Bibliografia **32**

Spis rysunków **33**

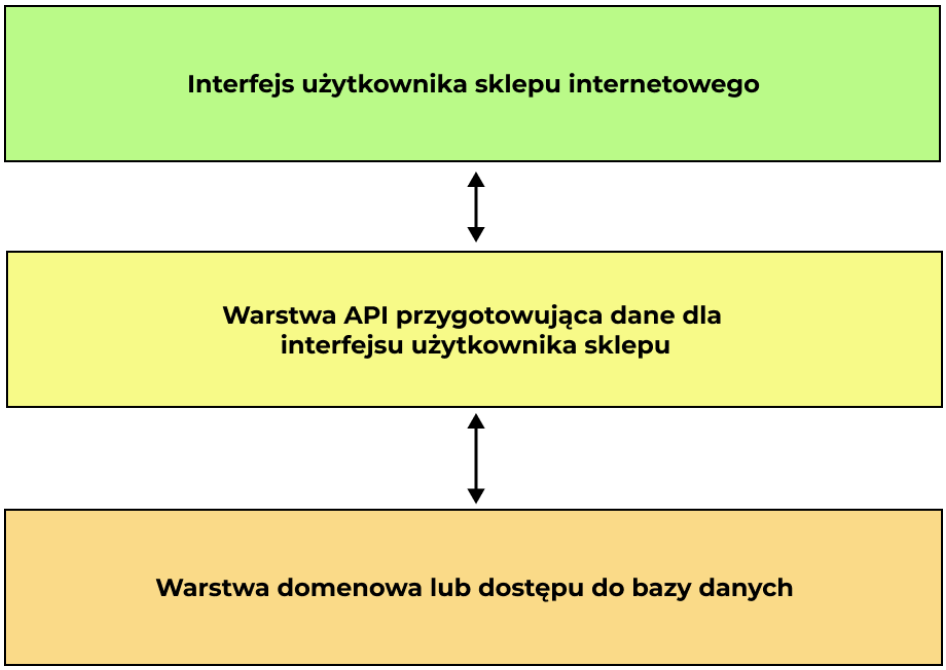
1 Wprowadzenie do mikroserwisów

1.1 Architektura monolityczna

Monolitem, bądź programem zaprojektowanym zgodnie z architekturą monolityczną nazywamy program uruchamiany w całości za pomocą jednego pliku wykonywalnego. W aplikacjach webowych taka architektura ma postać modelu trójwarstwowego. Na te warstwy składają się:

1. interfejs użytkownika składający się z dokumentów HTML, arkuszy styli CSS oraz skryptów JS definiujących zachowanie interfejsu użytkownika,
2. warstwa API przygotowująca dane podawane do interfejsu użytkownika w celu pokazania go użytkownikowi,
3. warstwa dostępu do danych.

Taką strukturę trójwarstwową możemy opisać rysunkiem 1



Rys. 1. Podział na warstwy dla przykładowego sklepu internetowego w architekturze monolitycznej

Warstwa druga jest czasami pomijana przez twórców oprogramowania pozwalając

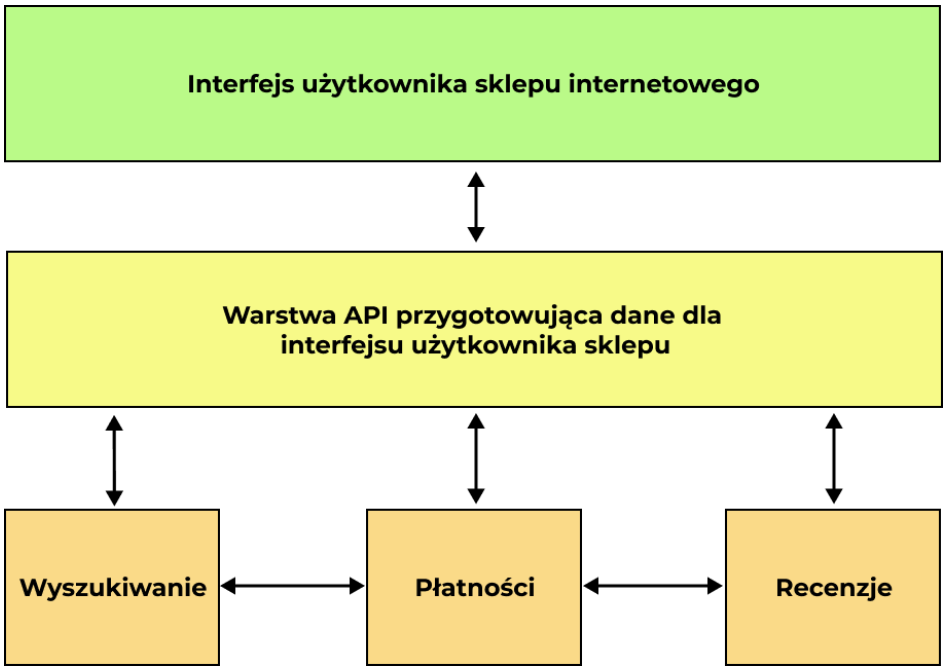
interfejsowi użytkownika na bezpośredni dostęp do danych. Różnicą między nimi jest brak istnienia warstwy oddzielającej logikę biznesową od interfejsu użytkownika. Wtedy możemy mówić o *modelu dwuwarstwowym*. Wraz z rozwojem aplikacji w modelu dwuwarstwowym i trójwarstwowym oraz ciągłemu dopisywaniu do nich nowych funkcjonalności, kod tych aplikacji stawał się coraz bardziej nieczytelny, a zależności między fragmentami kodu okazywały się być zbyt rozwlekłe. Ponadto każda zmiana w oprogramowaniu powoduje konieczność zbudowania oraz wystawienia na środowisko produkcyjne kolejnej wersji oprogramowania, co przy dużych aplikacjach potrafi zabierać znaczącą ilość czasu. Utrudnia to też proces testowania oprogramowania, zarówno manualnie (ze względu na rosnącą ilość scenariuszy testowych) jak i automatycznie (ze względu na rosnący czas uruchomienia siatki testów). Zgodnie z prawem Conwaya, takie trudności odbijają się na sposobie zarządzania organizacją, która jest producentem oprogramowania. Zgodnie z [2] pojęcia *monolit* w odniesieniu do oprogramowania pierwotnie używała społeczność programistów systemu operacyjnego Unix. Programiści Unixa określali tak systemy, które stają się zbyt duże, by móc je komfortowo utrzymywać.

1.2 Architektura mikroserwisów

Trudności z utrzymaniem oprogramowania, które w obrębie jednego projektu zawiera ciągle rozrastającą się logikę, spowodowały konieczność zastosowania podziału kodu na mniejsze i jednocześnie prostsze w utrzymaniu części. Zaczęto więc do większych aplikacji stosować podział aplikacji na części będące usługami niezależnymi od pozostałych elementów systemu. Taką pojedynczą usługę nazywamy mikroserwisem. Przykładowe rozmieszczenie modułów w projekcie mikroserwisowym możemy zaobserwować na rysunku 2

Komunikacja między mikroserwisami zachodzi przy użyciu protokołów sieciowych neutralnych w ujęciu technologicznym takich jak HTTP. Podział aplikacji na serwisy pozwala na wykonywanie prac nad konkretnymi funkcjonalnościami aplikacji bez potencjalnego ryzyka naruszenia kodu odpowiedzialnego za inne funkcjonalności. Innymi zaletami tego rozwiązania są:

1. dowolność w zastosowaniu technologii - pozwala na użycie do każdej funkcjonalności zasobów technologicznych najlepiej odpowiadających danemu zadaniu,



Rys. 2. Podział na moduły dla przykładowego sklepu internetowego w architekturze mikroserwisowej

2. stabilność - ewentualne problemy z aplikacją nie powodują zatrzymania całego systemu a jedynie konkretnego serwisu,
3. skalowalność - nowe funkcjonalności można dodawać poprzez dodanie nowych serwisów, a poszczególne serwisy można rozszerzać nie naruszając logiki innych serwisów,
4. wzrost dostępności usług - w razie, gdyby któryś z serwisów przestał działać, inne usługi nadal będą dostępne.

Przedstawione zalety są znaczące nie tylko z punktu widzenia projektowania i tworzenia oprogramowania, ale też z punktu zarządzania projektami. Jasny podział projektu na niezależne od siebie części pozwala też na przyporządkowanie ludzi do zespołów mniejszych, ale ściśle skoncentrowanych na konkretnym wycinku wiedzy potrzebnym do realizacji danej funkcjonalności. Autonomiczność takich zespołów pozwala na zmniejszenie ilości potrzebnych kontaktów między zespołami w celu ustalenia rozwiązań problemów i dalszego przebiegu projektu. Pomaga to między innymi zmniejszyć ilość spotkań oraz skrócić te krytycznie potrzebne do realizacji projektu.

Pomimo wielu zalet, architektura mikroserwisowa nie jest wolna od wad. Pierwszą z nich są koszty infrastruktury potrzebnej do wdrożenia oprogramowania opartego na mikroserwisach. Aby spełnić postulat niezależności mikroserwisów od siebie, każdy

oddzielny mikroserwis musi być postawiony na osobnym serwerze. W wyniku tego, zamiast zapłacić koszty utrzymania jednego serwera z lepszymi parametrami, zmuszeni jesteśmy zreplikować poniesione koszty na każdy z istniejących mikroserwisów.

Rozrośnięcie się infrastruktury przy użyciu mikroserwisów zwiększa też zapotrzebowanie na ludzi zajmujących się na wsparciu developerów w zakresie konfiguracji infrastruktury (tzw. *inżynierowie DevOps*), a w przypadku zastosowania technologii chmurowych, również specjalistów w zakresie technologii chmurowych takich jak *Microsoft Azure*, *Google Cloud* oraz *Amazon Web Services*. Taka sytuacja również przyczynia się do zwiększenia kosztów związanych z oprogramowaniem w architekturze mikroserwisowej. Można jednak uznać ją za czynnik zmuszający firmy do rozwoju w zakresie R&D w celu ustalenia własnych standardów architektonicznych, które będą optymalne dla ich potrzeb biznesowych w zakresie realizacji mikroserwisów. Mimo to, czynniki kosztowe powodują, że wybór architektury mikroserwisowej do projektu może nie być optymalny dla małych i średnich projektów.

Mikroserwisy powodują też konieczność uzgodnienia konwencji między projektami w sprawach takich jak:

- postać wyjątków rzucanych przez serwisy,
- opis obiektów, które są częścią innych serwisów,
- sposób łączenia się między serwisami,
- autoryzacja i uwierzytelnianie,
- standard podziału kodu na projekty lub moduły.

Brak zgody między projektami w powyższych kwestiach może doprowadzić do nieporządku jeszcze większego, niż gdyby projekt był realizowany jako monolit. Ponadto, uporządkowana struktura projektu mikroserwisowego pozwala na replikowanie schematu na dowolną skalę, a co za tym idzie automatyzację procesu tworzenia nowych serwisów. Można to zrealizować na przykład za pomocą szablonów projektu aplikowanych do środowiska programistycznego, obrazów środowiska *Docker* lub idąc jeszcze dalej - odpowiednio ustawionym klastrze *Kubernetes*. Jednak przy sprostaniu tym wyzwaniom architektonicznym na etapie projektowania oprogramowania zyskamy dzięki mikroserwisom możliwość zarządzania projektem o dowolnej wielkości w sposób ściśle uporządkowany, jednocześnie zachowując dobrą jakość kodu źródłowego oraz łatwe dostosowanie projektu do potrzeb biznesowych i metodologii zwinnych zarządzania

projektami.

2 Mikrofrontendy

2.1 Wstęp

Pomimo zastosowania mikroserwisów przy projektowaniu logiki backendowej części aplikacji, frontendowa część projektu pozostawała w przeszłości mniej rozbudowana. Głównym powodem był fakt, że logika obliczeniowa i biznesowa występowała po stronie backendu. Odciążało to część frontendową, co pozwalało na mniejsze jej rozbudowanie oraz ograniczenie rozwarstwienia frontendu w postaci monolitu. Jednak wraz z rozwojem technologicznym w zakresie webowych interfejsów użytkownika, objętość kodu frontendowego w widoczny sposób wzrosła. Na taki stan rzeczy, złożyło się wiele czynników takich jak:

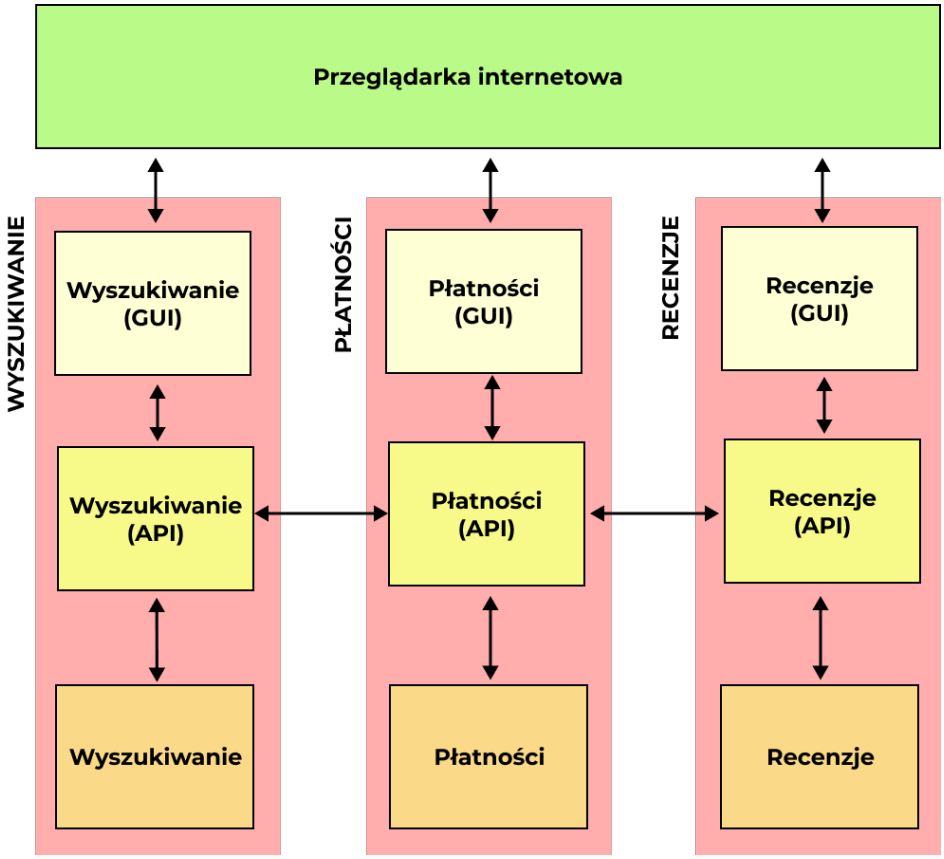
- powstawanie coraz bardziej kreatywnych i szczegółowych efektów wizualnych,
- optymalizacja działania skryptów JavaScript,
- zwiększająca się dojrzałość działalność programistów frontendu,
- rozwój frameworków strukturyzujących projekty frontendowe takich jak *Angular*, *React*, *Vue* (produkują one dziesiątki tysięcy linii kodu wynikowego).

Wszystkie te czynniki spowodowały potrzebę podzielenia kodu na pewnego rodzaju moduły. Z samym skutkiem w obrębie ilości i struktury kodu, najpopularniejsze frameworki radzą sobie wystarczająco dobrze, by używać tych rozwiązań w projektach komercyjnych. Najlepiej z tym problemem radzi sobie framework Angular, który proponuje programistom systemowy podział na moduły, oraz kilka propozycji na podziały niższego stopnia na struktury takie jak:

- component,
- service,
- guard,
- pipe
- i inne, rzadziej używane.

Jednak struktura organizacyjna całego projektu oprogramowania różni się znacząco po

stronie backendu, jak i frontendu. Może to powodować różne nieścisłości organizacyjne oraz zwiększać liczbę zapytań wykonanych między członkami zespołu w celu uzgodnienia szczegółów projektu. Zaczęto się więc rozglądać za sposobem organizacji pracy zespołu frontendowego takiego, żeby móc podzielić osoby pracujące nad projektem na zespoły w pełni skoncentrowane na konkretnych funkcjonalnościach. Takie podejście zwiększa autonomiczność zepsołów. Takim sposobem okazało się być dostosowanie frontendowych części projektu do architektury mikroserwisowej. Zgodnie z rysunkiem 3 można podzielić zespół programistów frontendu na funkcjonalne grupy dostosowane do wcześniej utworzonych podzespołów wśród innych specjalistów w zespole tak, aby wynikowe podzespoły były już w pełni skoncentrowane wobec wydzielonej partii wartości biznesowej według potrzeb biznesu.



Rys. 3. Podział na moduły dla przykładowego sklepu internetowego w architekturze mikrofrontendowej

Pojedynczą jednostkę funkcjonalną takiego podziału na frontendzie nazywamy *mikrofrontendem*, a w ogólności architekturę używającą takiego podziału nazywamy architekturą mikrofrontendową. Zgodnie z wynikami ankiety [4], 24.6% programistów biorących udział w ankiecie pracowało w 2021 roku w zespołach wykorzystujących

w projekcie architekturę mikrofrontendową. Ponadto, według tej ankiety, 37.2% programistów biorących udział w ankiecie twierdzi, że w przeciągu dwóch następnych lat popularność koncepcji mikrofrontendów wzrośnie. Na podstawie tych statystyk można wnioskować, że pomysł rozszerzenia mikroserwisów na część frontendową przyjął się w dużej części organizacji oraz przede wszystkim, w świadomości programistów.

2.2 Dodatkowe ograniczenia względem mikroserwisów

2.2.1 Ograniczone zasoby obliczeniowe

W przypadku prac nad backendowymi częściami projektu, coraz mniej zwraca się uwagę na optymalizację działania algorytmów używanych w oprogramowaniu. Pamięć operacyjna oraz masowa stały się dużo tańsze niż kilka lat temu. Dla klienta stało się tańsze zainwestowanie maksymalnie kilku tysięcy złotych na dołożenie pamięci do serwera udostępniającego żądaną usługę sieciową niż zainwestowanie wielokrotności tej kwoty w czas pracy programistów, którzy zoptymalizują wadliwy algorytmicznie kod. Nie ma to jednak zastosowania przy pracy nad frontendem. Przeglądarka, w której koniec końców efekty pracy są pokazane użytkownikowi końcowemu, oferuje jedynie swoje zasoby pamięci - nie można użyć tak dużego odsetka zasobów komputera, jak w przypadku backendu. Poza tym, minimalne odstępstwa od optymalizacji są zauważalne dla użytkownika końcowego. Każde przycięcie się elementów interfejsu, brak płynności powodują irytację użytkownika, a co dalej za tym idzie, spadek zadowolenia z używania wyprodukowanego oprogramowania. W przypadku zastosowania mikrofrontendów problem narasta. De facto ładujemy do przeglądarki jednocześnie kilka pełnoprawnych aplikacji i wymagamy od nich, aby one wszystkie podzieliły się niewielkimi zasobami pamięci i procesora zaalokowanymi przez przeglądarkę w taki sposób, aby każda z nich działała w sposób komfortowy dla użytkownika. Na szczęście rozwiązanie problemu jest konceptualnie bardzo proste. Wystarczy ładować do przeglądarki tylko te mikrofrontendy, które w danym momencie są użytkownikowi krytycznie potrzebne. Najczęściej stosowanym sposobem implementacji takiego rozwiązania jest ładowanie konkretnych mikrofrontendów po zmianie adresu URL w przeglądarce. Gotowe rozwiązania implementujące mikrofrontendy, takie jak *Single SPA* optymalizują ładowanie zasobów jeszcze bardziej poprzez zastosowanie wzorca *Lazy loading*. Dzięki tej prostej sztuczce,

można ściśle kontrolować ładowanie się mikrofrontendów do przeglądarki.

2.2.2 Zasoby współdzielone

W przypadku, gdy na potrzeby jednego z mikrofrontendów tworzony jest reużywalny komponent (przykładowo tabela o odpowiednim wyglądzie i zachowaniu), może zaistnieć potrzeba zastosowania takiego komponentu w innym mikrofrontendzie. Wtedy pierwszym z pomysłów, jakie mogą powstać, jest przekopiowanie kodu z jednego do drugiego mikrofrontendu. Jednak to rozwiązanie tworzy nowe problemy.

Po pierwsze, każda zmiana w komponencie, w celu uzgodnienia wersji wymaga zmian w tym samym komponencie we wszystkich mikrofrontendach używających tego projektu. W przypadku kilku mikrofrontendów jest to możliwe, ale w przypadku, gdy mamy takich mikrofrontendów setki lub tysiące, takie cykliczne zmiany są wręcz niemożliwe do wykonania i zabierałyby one duży odsetek czasu całościowo poświęconego na projekt. Rozwiązaniem problemu okazuje się być umieszczeniu tego kodu jeden raz w zewnętrznym repozytorium (technicznie mogą to być prywatne repozytoria NPM lub inne rozwiązanie) i zainstalowanie przez użycie systemu zarządzania pakietami. We frontendowych częściach projektu najczęściej są to NPM i Yarn.

Po drugie, komponent skopiowany wprost z jednego mikrofrontendu na drugi, powoduje konieczność powstawania oddzielnych scenariuszy testowych na ten sam komponent dla każdego mikrofrontendu. W praktyce, już przy kilkunastu mikrofrontendach taka ilość testów staje się nie do utrzymania w takim stanie, aby scenariusze odpowiadały aktualnemu stanowi funkcjonalności.

2.2.3 Arkusze styli CSS

Arkusze styli z poszczególnych mikrofrontendów są ładowane razem do przeglądarki, na której uruchomiony jest każdy z pożądanych mikrofrontendów. Występuje w takiej sytuacji ryzyko replikacji tych samych styli CSS oraz nadpisywania się styli CSS między mikrofrontendami. Drugi z przypadków jest szczególnie kłopotliwy, gdyż wygląd niektórych elementów na stronie może zależeć wyłącznie od tego, który z mikrofrontendów załaduje się do przeglądarki jako pierwszy. Jest na to kilka rozwiązań:

- stosowanie ogólnej konwencji nazewniczej dla klas CSS, takich jak BEM i OOCSS

- wstrzykiwanie styli CSS bezpośrednio przez kod JavaScript (tzw. CSS-in-JS)
- zastosowanie Shadow DOM

Każde z tych podejść wciąż wymaga dyskusji między programistami pracującymi nad projektem oraz ich ostrożności i uważności na aspekt nadpisywania się bądź replikacji styli CSS. Dlatego też, mimo opisanych ścieżek rozwiązania, najważniejsze stają się zdrowy rozsądek osób decydujących o przebiegu projektu oraz doświadczenie zespołu pracującego nad projektem.

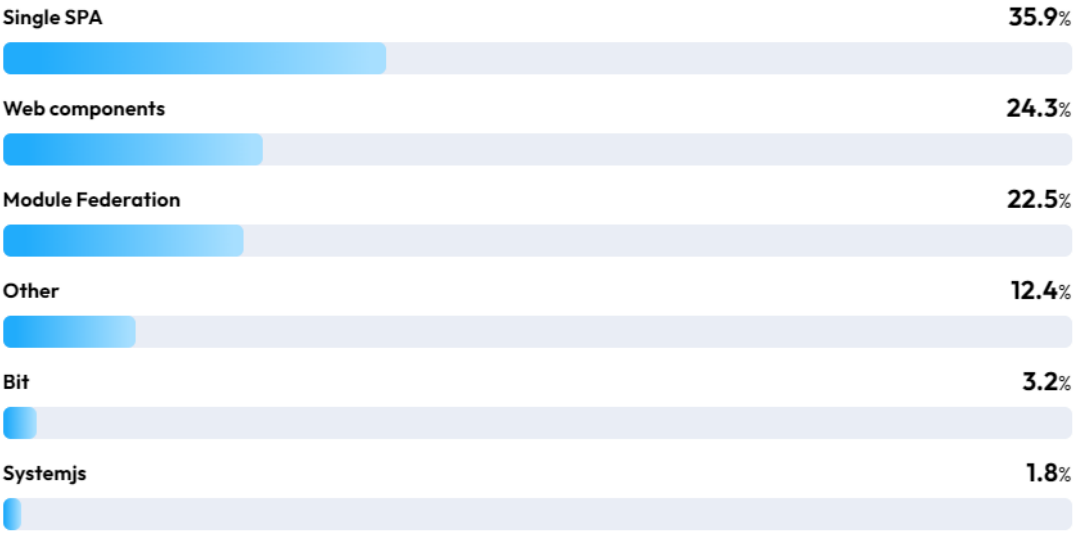
2.2.4 Komunikacja między mikrofrontendami

W projektach zdarzają się sytuacje, gdy istnieje potrzeba ukazania na widoku frontendu informacji z kilku wycinków domenowych projektu. Dla sytuacji z rysunku 3 zaistniałoby to wtedy, gdyby do listy recenzji należało dodać informację ile recenzujący użytkownik kupił sztuk produktu za jaką kwotę. Określając to modułami z tego przykładu - do modułu mikrofrontendu realizującego recenzje produktów należałoby dodać komunikację z mikrofrontendem obsługującym płatności. Rekomendowanym zachowaniem w tej sytuacji jest nie stosować komunikacji między aplikacjami na frontendzie. Odrobinę lepszym wyjściem jest zastosować taką komunikację po stronie backendu i na warstwie API aplikacji stworzyć endpoint, który podaje do frontendu potrzebne informacje. Martin Fowler w swoim artykule [1] wspomina, że możliwe jest tutaj zastosowanie wzorca architektonicznego BFF (*Backends For Frontends*). Jednak przy wystąpieniu takiej sytuacji wciąż należy się zastanowić, czy zastosowany sposób podziału projektu na serwisy jest wystarczająco dobry, aby spełnić potrzeby biznesowe w architektonicznie poprawny sposób.

2.3 Sposoby implementacji technicznej mikrofrontendów

Społeczność programistów wypracowała kilka funkcjonalnych rozwiązań w zakresie implementacji mikrofrontendów. Procentowy odsetek popularności każdego z nich wśród programistów badanych w ankiecie [4], możemy zaobserwować na wykresie 4

Dla każdego z nich istnieje zestaw zastosowań biznesowych dla których rozwiązanie nie będzie właściwe, ale będą istnieć też takie, dla których będą one właściwe. Każde z nich zostanie omówione dokładniej omówione w poniższych podsekcjach.



Rys. 4. Wykres procentowy popularności rozwiązań dla implementacji mikrofrontendów w 2022 roku (źródło: [4])

2.3.1 Konfiguracja serwera używająca routingu

Konfigurowanie projektu, aby podawał do przeglądarki kod konkretnego mikrofrontendu w zależności od adresu URL można w prosty sposób wykonać poprzez odpowiednią konfigurację serwera. Taki sposób konfiguracji mkrofrontendów zalicza się do rozwiązań integrujących mikrofrontendy w czasie budowania projektu. To rozwiązanie jest wspierane przez najpopularniejsze oprogramowanie serwerowe takie jak IIS, Apache, NGINX. Konfiguracja serwera działa w bardzo prosty sposób - mapuje adresy URL na nazwy plików istniejących na serwerze oraz w zależności od tego jaki aktualnie jest adres URL, przypisuje do zmiennej odpowiedni plik HTML. Odnosząc się do przykładu na listingu 1, taką zmienną jest `$PAGE` z linii nr 8.

Listing 1 Przykładowa implementacja mikrofrontendu (przykład pochodzi z [1])

```
1 <html lang="en" dir="ltr">
2   <head>
3     <meta charset="utf-8">
4     <title>Feed me</title>
5   </head>
6   <body>
7     <h1>Feed me</h1>
8     <!--# include file="$PAGE.html" -->
```

```
  </body>
</html>
```

2.3.2 Elementy <iframe>

Najprostszym rozwiązaniem, obecnym już od dawna w standardzie języka znaczników HTML, jest element `<iframe>`. Zgodnie z teorią, takie rozwiązanie spełnia założenia mikrofrontendów. W wygodny sposób izolują poszczególne mikrofrontendy od siebie, nie pozwalają na nadpisanie się selektorów CSS oraz ewentualnych zmiennych globalnych w skrpytach JavaScript. Dzieje się tak ze względu na to, że elementy `<iframe>`posiadają swój osobny zakres dla styli CSS i kodu JavaScript. Mimo to mają one też kilka wad. Po pierwsze, pełna izolacja kontekstu w elemencie `<iframe>` powoduje, że treść tego elementu strony nie będzie reagowała na zmiany w routingu. Ponadto, operowanie na historii przeglądania, adresie URL jest znacznie utrudnione. Taki koncept sprawia też trudności w przypadku, gdy znaczenie ma responsywność strony.

Listing 2 Przykładowa implementacja mikrofrontendu za pomocą `<iframe>` (przykład pochodzi z [1])

```
<html>
  <head>
    <title>Feed me!</title>
  </head>
  <body>
    <h1>Welcome to Feed me!</h1>

    <iframe id="micro-frontend-container"></iframe>

    <script type="text/javascript">
      const microFrontendsByRoute = {
        '/': 'https://browse.example.com/index.html',
        '/order-food': 'https://order.example.com/index.html',
        '/user-profile': 'https://profile.example.com/index.html',
      };
```

```
17        const iframe = document.getElementById('
              micro-frontend-container');
18        iframe.src = microFrontendsByRoute[window.location.pathname
              ];
19      </script>
20    </body>
21  </html>
```

2.3.3 Przechowywanie mikrofrontendów w skryptach JavaScript

Idea ładowania mikrofrontendów przez podpinanie skryptów JavaScript jest bardzo prosta. Zgodnie z przykładem implementacji podanym na listingu 3, na początku sekcji `<body>` podpiete są trzy skrypty reprezentujące trzy mikrofrontendy. Obecny jest także element `<div id="micro-frontend-root">`, do którego będą ładowane mikrofrontendy. W skrypcie podanym niżej znajduje się przyporządkowanie tras w routingu do mikrofrontendów o odpowiednich nazwach oraz referencja na element `<div id="micro-frontend-root">`

Listing 3 Przykładowa implementacja mikrofrontendu za pomocą skrpytów JavaScript (przykład pochodzi z [1])

```
1  <html>
2    <head>
3      <title>Feed me!</title>
4    </head>
5    <body>
6      <h1>Welcome to Feed me!</h1>

8      <script src="https://browse.example.com/bundle.js"></script>
9      <script src="https://order.example.com/bundle.js"></script>
10     <script src="https://profile.example.com/bundle.js"></script>

12     <div id="micro-frontend-root"></div>
```

```
14      <script type="text/javascript">
15        const microFrontendsByRoute = {
16          '/': window.renderBrowseRestaurants,
17          '/order-food': window.renderOrderFood,
18          '/user-profile': window.renderUserProfile,
19        };
20        const renderFunction = microFrontendsByRoute[
              window.location.pathname];

22        renderFunction('micro-frontend-root');
23      </script>
24    </body>
25  </html>
```

2.3.4 Dynamiczne ładowanie modułów poprzez menedżer pakietów

Istnieje również grupa rozwiązań oparta na dynamicznym ładowaniu modułów. Do grupy rozwiązań tego typu należą wspomniane na wykresie procentowym z rysunku 4 Module Federation, Bit oraz Systemjs. Pierwsze z rozwiązań, Module Federation, jest wtyczką do transpilatora Webpack. Wszystkie trzy, pomimo pewnych niewielkich różnic w zakresie składni poleceń, instalacji i procesu konfiguracji, działają one bardzo podobnie.

2.3.5 Single SPA

Single SPA jest frameworkiem, za pomocą którego można zrealizować koncepcję mikrofrontendów. Ten framework czerpie swój sposób działania z rozwiązań opartych na dynamicznym ładowaniu modułów. Istnieje też więc możliwość używania Single SPA z każdym popularnym frameworkiem stosowanym przez programistów frontendu. Innowacją, którą proponuje Single SPA są predefiniowane konfiguracje, które automatyzują tworzenie mikrofrontedów. Do celów automatyzacji ustawiania infrastruktury mikrofrontendów twórcy udostępniają narzędzie CLI o nazwie *create-single-spa*. Single SPA narzuca też sam z siebie określony podział konkrentych mikrofrontendów ze względu na zastosowania.

Składnikami tego podziału są:

- *root-config* - mikrofrontend mający zastosowanie głównie przy agregacji innych mikrofrontendów,
- *application* - domyślny typ mikrofrontendu bez specjalnych zadań projektowych,
- *parcel* - reużywalny komponent, który jest niezależny w konstrukcji od frameworków używanych w projekcie; w przypadku użycia tylko jednego frameworka w całym projekcie, twórcy rekomendują [3] poleganie na systemie komponentyzacji tego frameworka.

Oprócz wymienionych możliwości, framework Single SPA udostępnia swój interfejs do testów jednostkowych i testów E2E oraz wspiera renderowanie stron za pomocą serwera. Dokumentacja frameworka Single SPA zawiera bardzo dokładne instrukcje postawienia środowiska oraz wdrożenia produkcyjnego pod każdy z popularnych frameworków, co znacząco obniża próg wejścia w samą koncepcję mikrofrontendów.

3 Funkcjonalność badanej aplikacji

Na potrzeby porównania obu architektur stworzona została aplikacja do zarządzania finansami osobistymi o roboczej nazwie Midas. W założeniu odbiorcami aplikacji mają być pojedyncze osoby lub gospodarstwa domowe, które chcą zadbać o kontrolę nad swoim budżetem domowym. Głównymi funkcjonalnościami aplikacji są:

- możliwość wykonywania operacji CRUD w zakresie informacji o wydatkach i przychodach poszczególnych użytkowników,
- system autoryzacji i uwierzytelniania spełniający aktualne normy w zakresie bezpieczeństwa aplikacji sieciowych,
- system uprawnień w obrębie rodziny (przykładowo użytkownik o roli rodzica może edytować transakcje na kontach dzieci, a dzieci nie mogą podglądać transakcji rodziców)
- przechowywanie paragonów i faktur w wersji elektronicznej i możliwość przypisania ich do konkretnej transakcji.

Podany zestaw funkcjonalności umożliwi wydzielenie kilku mikroserwisów, co zapewni wystarczającą bazę do symulowania połączeń między serwisami. Wykonanie jej w architekturze ściśle mikroserwisowej postanowi jednocześnie dobudować do tych mikroserwisów poszczególne mikrofrontendy tak, aby stanowiły one razem pełnoprawne aplikacje komunikujące się między sobą. Z drugiej strony pozwoli to też dobudować monolityczny frontend, który będzie komunikował się ze wszystkimi serwisami. Dzięki takiemu posunięciu dla obu badanych projektów - monolitycznego oraz mikrofrotendowego, zapewnione będą jednolite warunki w zakresie komunikacji z backendem, co będzie stanowiło bazę do porównania obu koncepcji architektonicznych w zakresie wydajności i dostępności.

4 Opis backendu projektu

4.1 Dobór technologii do projektu

Do zrealizowania tej części projektu został wykorzystany język C# oraz środowisko .NET. Użyto środowiska .NET w wersji 6.0, która jest jednocześnie najnowszą dostępną wersją LTS (*Long Term Support*). Wspomniane technologie są dobrym wyborem do realizacji mikroserwisów ze względu na:

- użycie paradygmatu programowania obiektowego i będący jego częścią polimorfizm, który sprzyja replikacji pojedynczego wzorca mikroserwisu,
- będący częścią środowiska .NET framework ASP.NET realizujący architekturę REST przy niewielkim narzucie w ilości kodu,
- istnienie wielu gotowych klas i modeli realizujących podstawowe funkcje sieciowe takie jak autoryzacja i autentykacja, komunikacja z bazą za pomocą Entity Framework,
- generator NSwag służący do generowania klas w językach C# oraz TypeScript reprezentujących metody kontrolerów poszczególnych mikroserwisów wykonanych w ASP.NET.

Środowiska mikroserwisowe są de facto oddzielnymi serwisami działającymi jednocześnie i pobierającymi dostępne zasoby. Do celów testów lokalnych na komputerze programisty zachodzi więc potrzeba zastosowania rozwiązania, które zminimalizuje użycie zasobów komputera tak, aby jednocześnie jak zachować wierność odwzorowania środowiska z wieloma serwerami. W celu osiągnięcia takiego efektu, zostało użyte oprogramowanie Docker służące do konteneryzacji środowisk. Za jego pomocą, używając specjalnych plików nazywanych obrazami można tworzyć kontenery, którym Docker przydziela zasoby w czasie rzeczywistym tak, aby zoptymalizować ich użycie. Razem z narzędziem Docker, użyto też narzędzia *docker-compose*, które umożliwia uruchomienie wielu kontenerów Dockera za pomocą pojedynczego skryptu.

Opisane technologie mogą też być komfortowo używane w połączeniu z oprogramowaniem Docker służącym do konteneryzacji. Producent środowiska .NET,

Microsoft udostępnił w serwisie Docker Hub obraz środowiska .NET z ustawionym frameworkiem ASP.NET oraz połączeniem z bazą SQL Server. Pozwala to na skorzystanie z gotowego środowiska, które jest konfigurowalne za pomocą pliku konfiguracyjnego o nazwie Dockerfile.

4.2 Szablony projektów

W celu zapewnienia skalowalności oraz łatwego tworzenia nowych serwisów, na potrzeby projektu opracowano dwa szablony projektu - zawierający wstępną autentykację użytkownika poprzez sprawdzanie zawartości nagłówka HTTP oraz taki, który jej nie zawiera.

Najważniejszą cechą tych szablonów jest możliwość szybkiego ustawienia nowego rozwiązania Visual Studio zawierającego ustawienia dla testów jednostkowych i integracyjnych aplikacji, ustawienia plików Dockerfile i *docker-compose.yml*, oraz ściśle określonego rozłożenia projektów w rozwiązaniu. Zapewnia to spójność kodu w zakresie całej aplikacji. W przypadku, gdyby nad kodem pojedynczego serwisu pracowało kilku programistów, mają oni ściśle narzuconą przez szablon strukturę kodu i podstawowa jego struktura zostanie zachowana. To z kolei powoduje, że wynikowo mimo tego, że nad poszczególnymi serwisami pracują różne osoby, ich struktura jest w dużym stopniu podobna. Ma to wiele zalet w zakresie zarządzania projektami, przykładowo:

- pozwala na szybszą aklimatyzację programistów przenoszonych między projektami, bądź takich, których rola w zespole pomaga na wsparciu istniejących projektów,
- przyspiesza czas tworzenia nowych funkcjonalności oprogramowania,
- czas poświęcany na odtwórcze powtarzanie realizacji wzorca można poświęcić na ważniejsze czynności takie jak redukcja długu technologicznego, zwiększanie pokrycia testami, itd.

4.3 Podział backendu na serwisy

Logika backendowa aplikacji została podzielona według funkcjonalności na serwisy opisane w poniższych podsekcjach.

4.3.1 Hosting plików - File Storage Service

Serwis z hostingiem plików odpowiada za przechowywanie wszystkich plików przekazanych aplikacji przez użytkownika. Są to między innymi zdjęcia profilowe użytkowników oraz pliki z dowodami wykonania transakcji przypisanych do konkretnych transakcji w ramach logiki domenowej. Serwis hostingu plików pozwala też na pobranie samego pliku o znanym identyfikatorze UUID oraz pobranie informacji o umieszczeniu pliku oraz konkretnych pobrań pliku.

4.3.2 Autoryzacja - Authorization Service

Serwis odpowiada za autoryzację użytkowników oraz operacje związane z użytkownikami, które wymagają zachowania ostrożności pod kątem bezpieczeństwa aplikacji - tworzenie konta, logowanie do konta w aplikacji Midas, zmiana hasła. W tym serwisie przy logowaniu powstaje token JWT, który zapisany w ciasteczkach krąży po innych serwisach przekazywany przez nagłówek *Authorization* w zapytaniu HTTP.

4.3.3 Zarządzanie użytkownikami - User Service

Serwis odpowiada za przechowywanie oraz umożliwienie dostępu do informacji o użytkownikach. Obsługuje on też logikę związaną z kontami użytkowników, które nie wymagają zachowania szczególnej ostrożności w zakresie bezpieczeństwa aplikacji. Przykładem takiej operacji może być zmiana zdjęcia profilowego użytkownika.

4.3.4 Zarządzanie rodzinami - Family Service

Serwis odpowiada za przypisanie użytkowników do rodzin, których są członkami. Odpowiada też częściowo za uwierzytelnianie użytkownika - z tego serwisu pochodzą dane na temat tego, czy dany użytkownik ma wystarczające uprawnienia do edytowania danych dla swojej rodziny (ze względu na rolę przypisaną w systemie).

4.3.5 Logika domenowa - Transaction Service

Serwis ma za zadanie realizację operacji CRUD na transakcjach finansowych. Takimi operacjami może być wpisanie nowego wydatku do listy, usunięcie go z listy, pobranie listy

wydatków w zależności do konkretnych parametrów (np. kategoria, czas, osoba z rodziny), dodanie pliku z dowodem wykonania transakcji.

5 Opis badanych frontendów

5.1 Dobór technologii do projektów

Do realizacji mikrofrontendów użyto biblioteki *Single SPA*. Wspiera ona wszystkie najpopularniejsze frameworki do tworzenia aplikacji frontendowych. W badanych projektach użyto frameworka Angular ze względu na to, że najlepiej nadaje się do tworzenia dużych skalowalnych aplikacji. Ta cecha pozwoli utrzymać w ryzach potencjalnie rozwlekłą strukturę monolitycznej wersji projektu. W celu zapewnienia jak najmniejszych różnic technologicznych w projektach o różnej architekturze, wersja mikrofrontendowa będzie również używać frameworka Angular.

Do komunikacji z serwisami backendowymi, dla obu projektów użyto klas serwisów wygenerowanych przez generator NSwag. Będzie to działało tak jak w przypadku klas w języku C# generowanych na potrzeby komunikacji między serwisami backendowymi. Różnica będzie tu jedynie taka, że te klasy będą gotowymi klasami w języku TypeScript dostosowanymi do użycia we frameworku Angular.

Ze względu na to, że walory estetyczne nie są istotne w zakresie rozważań nad mikrofrontendami, użyto gotowej biblioteki z elementami wizualnymi o nazwie *angular-material*.

5.2 Wersja monolityczna

5.3 Wersja mikroserwisowa

6 Porównanie projektów frontendowych

6.1 Wersjonowanie kodu

6.2 Wydajność

6.3 Dostępność

6.4 Testowanie kodu

6.5 Zależności między modułami i projektami

6.6 Możliwości w zakresie zarządzania projektami

6.7 Skalowalność

6.8 Koszty ustawienia środowiska produkcyjnego

7 Podsumowanie

8 Bibliografia

[1] Martin Fowler. *Micro frontends*. Czerwiec 2019.
URL: https://martinfowler.com/articles/micro-frontends.html.

[2] Martin Fowler. *Microservices*.
URL: https://martinfowler.com/articles/microservices.html.

[3] *Single SPA Parcels*.
URL: https://single-spa.js.org/docs/parcels-overview.

[4] *The State of Frontend 2022*. Maj 2022. URL:
https://tsh.io/state-of-frontend/?fbclid=IwAR1NoxwAAX5qTprLKiJm4k-LBFWaQny3j3F7XIGTgWZRBpyYjQDb7bnq5DQ#report.

Spis rysunków

Rys. 1 Podział na warstwy dla przykładowego sklepu internetowego w architekturze monolitycznej 9
Rys. 2 Podział na moduły dla przykładowego sklepu internetowego w architekturze mikroserwisowej 11
Rys. 3 Podział na moduły dla przykładowego sklepu internetowego w architekturze mikrofrontendowej 15
Rys. 4 Wykres procentowy popularności rozwiązań dla implementacji mikrofrontendów w 2022 roku (źródło: [4]) 19

Spis listingów

1 Przykładowa implementacja mikrofrontendu (przykład pochodzi z [1]) . . . 19
2 Przykładowa implementacja mikrofrontendu za pomocą `<iframe>` (przykład pochodzi z [1]) . . . 20
3 Przykładowa implementacja mikrofrontendu za pomocą skrpytów JavaScript (przykład pochodzi z [1]) . . . 21

Wyrażam zgodę na udostępnienie mojej pracy w czytelniach Biblioteki SGGW w tym w Archiwum Prac Dyplomowych SGGW.

. . . . . .
(czytelny podpis autora pracy)